Dpto. Sistemas Informáticos y Computación
Escuela Técnica Superior de Ingeniería Informática
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

INTELLIGENT SYSTEMS

LAB WORK 1

DESIGN, IMPLEMENTATION AND EVALUATION OF A RULE-BASED SYSTEM

The 8-puzzle problem

INDEX

1. Implementing the 8-puzzle problem as a RBS 3
   1.1 Patterns........ 3
   1.2 Rules........ 4
   1.3 Initial data and goal state........ 6
   1.4 Solvable puzzle configurations 8
2. Breadth/Depth search strategies........ 8
3. Execution of the RBS 12
4. Evaluation........ 12

1. Implementing the 8-puzzle problem as a RBS

In this section, we will see the implementation of the 8-puzzle problem as a Rule-Based System (RBS). The elements of the problem are:

- **Initial Working Memory or Fact Base**: initial puzzle configuration
- **Goal state**: final puzzle configuration
- **Rules**: set of legal movements (movements of the empty space or blank tile)
- **Control strategy**: breadth, depth and heuristic search strategy

1.1 Patterns

We choose an ordered pattern to represent the tiles in each row of a puzzle state. A fact associated to this pattern represents a puzzle configuration. For example, the puzzle configuration:

2	**8**	**3**
1	**6**	**4**
7		**5**

would be represented with the following fact:

```
(puzzle 2 8 3 1 6 4 7 0 5)
```

The three first numeric values denote the tiles in the upper row; the next three values denote the tiles in the middle row and the last three values denote the tiles in the bottom row (blank is represented with the numeric value 0).

We will also include three additional fields in the fact defined above:

- a field (***level***) to show the depth level of the fact in the search tree. This field is used in two ways:
  - on the one hand, we use this field to control the maximum depth level of the search tree
  - on the other hand, it is used to return the depth level at which the solution is found

- a field (***movement***) to store the last movement (right, left, up, down, null) to reach our current state (fact). This fact is used to avoid direct loops in the problem, that is, to avoid firing the rule that performs the reverse movement to the last one.

- a field (***fact***) that stores the fact (fact index) to which the last movement was applied to generate the current fact; i.e. it stores the parent fact of the current fact. This field is used to retrieve the solution path.

An example of an initial fact in the Working Memory (WM) would be:

```
(puzzle 2 8 3 1 6 4 7 0 5 level 0 movement null fact 0)
```

Our goal is to reach the final state which can be represented, for instance, by the fact:

```
(puzzle 1 2 3 8 0 4 7 6 5 level 5 movement right fact <Fact-28>)
```

The final structure of our pattern will be:

(puzzle $x1^s$ $x2^s$ $x3^s$ $x4^s$ $x5^s$ $x6^s$ $x7^s$ $x8^s$ $x9^s$ level y^s movement z^s fact w^s))

where:

- **puzzle**, **level** and **movement** and **fact** are four constant symbols
- fields with a superscript 's' denote single-valued fields; i.e., fields which must take on a single value. Particularly:
    - $xi^s \in [0\text{-}8]$ / $\forall i,j$ $xi \neq xj$
    - $y^s \in$ Positive integer and zero
    - $z^s \in$ [up, down, right, left, null]
    - w^s fact-index

1.2 Rules

Rules represent legal movements in the puzzle problem. Since every movement in the puzzle involves the blank tile, rules are encoded as movements of the blank tile (right, left, up, down).

A movement to the right is applicable whenever the blank tile is at one of the positions shown in Fig. 1 (valid positions are denoted by black dots):

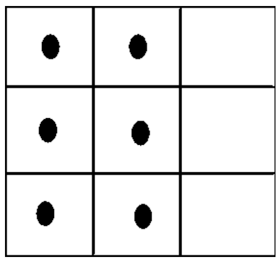


Fig. 1 Legal positions for the blank tile (denoted by black dots) to perform a movement to the right

Therefore, each movement of the blank tile is applicable from six different positions in the board. Rules implement the four possible movements of the blank tile. The Left-Hand Side (LHS) of a rule matches the appropriate puzzle configuration in order to perform the corresponding movement. For example, the rule `right` shown in Figure 2 encodes any of the movements of the blank tile to the right. The code of the rule `right` is as follows:

```
(defrule right
  ?f<-(puzzle $?x 0 ?y $?z level ?level movement ?mov fact ?)
  (max-depth ?depth)
  (test (and (<> (length$ $?x) 2) (<> (length$ $?x) 5)))
  (test (neq ?mov left))
  (test (< ?level ?depth))
=>
  (assert (puzzle $?x ?y 0 $?z level (+ ?level 1) movement right fact ?f))
  (bind ?*gen-nod* (+ ?*gen-nod* 1)))
```

Fig. 2 Rule `right`

The LHS of the rule `right` consists of two patterns and three tests:

1. facts in the WM match the pattern in the rule `right`; the blank tile (represented by 0) is matched against the constant 0 and the tile on its right is matched against the variable `?y`

2. the second pattern instantiates the fact that represents the maximum depth that can be reached in the problem

3. the first test checks whether the position of the blank is one of the six valid positions to make a movement to the right. In order to do so, we check the length of variable `$?x`, which stores all the numbers on the left of the blank tile position (0). For example, if the blank tile is located in cell [1,1] (upper left cell) then the length of `$?x` is 0. In this rule, we only have to check that the length of the variable `$?x` is neither 2 nor 5, in which case the blank would be in positions [1,3] or [2,3], respectively (forbidden positions for the blank to move to the right). It is not necessary to check the length of `$?x` is different from 8 (in which case the blank is at [3,3]) because this is not possible within the pattern variables.

4. the second test checks that the fact which matches the pattern has not been generated through a movement to the left (rule `left`); this test is used to avoid repeated states

5. the last test filters out the facts whose depth level is greater than the maximum depth level set by the user

The Right-Hand Side (RHS) of the rule inserts the new fact (state) which results from moving the blank to the right. Consequently, the position of the blank (0) and the tile on its right (`?y`) are switched in the new fact.

For rules `up` and `down`, we must take into account that these are vertical movements to be applied on a lineal structure. Thus, we have to find the position of the tile to be switched with the blank. The code of rule `down` is as follows:

```
(defrule down
   ?f<-(puzzle $?x 0 ?a ?b ?c $?z level ?level movement ?mov fact ?)
  (max-depth ?depth)
  (test (neq ?mov up))
  (test (< ?level ?depth))
=>
  (assert (puzzle $?x ?c ?a ?b 0 $?z level (+ ?level 1) movement down fact ?f))
  (bind ?*gen-nod* (+ ?*gen-nod* 1)))
```

Fig. 3 Rule `down`

We switch the blank with the tile which takes up the position represented in variable `?c`. We can observe that the three single-valued variables after 0 (`?a ?b ?c`) denote that there must necessarily be three values after 0, which would prevent us from doing illegal movements; that is, moving down when the blank is already at the bottom row.

1.3 Initial data and goal state

The **goal state** for the 8-puzzle problem is:

1	**2**	**3**
8		**4**
7	**6**	**5**

The goal state is directly represented in the rule `goal` described below to detect when the objective or goal state is reached:

```
(defrule goal
     (declare (salience 100))
   ?f <-(puzzle 1 2 3 8 0 4 7 6 5 level ?n movement ?mov fact ?)
  =>
    (printout t "SOLUTION FOUND AT LEVEL " ?n crlf)
    (printout t "NUMBER OF EXPANDED NODES OR TRIGGERED RULES " ?*gen-nod* crlf)
    (printout t "FACT GOAL " ?f crlf)
    (halt))
```

Fig. 4 Rule `goal`

This rule has a higher priority (`salience 100`) than the other rules because this is the rule which checks whether the goal state has been reached, in which case the inferential process stops. When a fact in the WM matches the pattern of the rule `goal`, the rule will be fired before any of the pending activations in the Agenda. The command (`halt`) stops the Inference Engine.

We also define a rule named `no_solution` whose priority (`salience`) is smaller than the rest of rules. This rule is used to identify the cases in which the search process does not find a solution to the problem.

```
(defrule no_solution
    (declare (salience -99))
    (puzzle $? level ?n $?)
=>
    (printout t "SOLUTION NOT FOUND" crlf)
    (printout t "GENERATED NODES: " ?*gen-nod* crlf)
    (halt))
```

Fig. 5 Rule `no_solution`

Additionally, we have a function `start` that requests the user the maximum depth level and the search strategy to use in the problem-solving process (breadth or depth so far). Depending on the user's response, the corresponding Agenda strategy is activated. Breadth and Depth search strategies are implemented by choosing the equivalent Agenda strategy. The function `start` also inserts the initial puzzle state and a static fact that denotes the maximum depth level to be reached in the problem.

```
(deffunction start ()
 (reset)
 (printout t "Maximum depth:= " )
 (bind ?depth (read))
 (printout t "Search strategy " crlf "1.- Breadth" crlf
             "2.- Depth" crlf )
 (bind ?a (read))
 (if (= ?a 1)
      then (set-strategy breadth)
      else (set-strategy depth)))
  (assert (puzzle 2 8 3 1 6 4 7 0 5 level 0 movement null fact 0))
  (assert (max-depth ?depth))
)
```

Fig. 6 Function `start`

Finally, we have a function `path` that shows the movement sequence of the solution on the screen. The function `path` is invoked as `(path <fact-index>)`, where `fact-index` is the index of the fact 'Fact goal' displayed by the rule `goal` .

Let's assume that the following output is shown after executing a particular puzzle configuration (see rule `goal` in Figure 4):

```
SOLUTION FOUND AT LEVEL 5
NUMBER OF EXPANDED NODES OR TRIGGERED RULES 36
FACT GOAL <Fact-39>
```

In order to retrieve the 5-step solution path, we must simply execute `(path 39)` .

1.4 Solvable puzzle configurations

In the 8-puzzle problem, there can be configurations for which there is no solution; i.e. they are unsolvable configurations. For instance, given the initial state:

2	**1**	**3**
8		**4**
7	**6**	**5**

there is no combination of movements right-left-up-down that reaches the final state described in section 1.3. In this case, we say the configuration is **unsolvable**.

We provide a function implemented in CLIPS to check whether a given puzzle configuration is solvable or not, and, consequently, whether it is possible to find a solution for the problem or not. The function name is `check_conf` and can be found in the file `auxiliar.clp`. The input parameter of `check_conf` is a list that represents the puzzle configuration. The function returns `TRUE` in case the configuration is solvable and `FALSE` otherwise.

For example:

```
CLIPS> (check_conf (create$ 2 8 3 1 6 4 7 0 5))
TRUE

CLIPS> (check_conf (create$ 2 1 3 8 0 4 7 6 5))
FALSE
```

NOTE: Before executing the function `check_conf`, it is necessary to load first the file `auxiliar.clp`.

2. Breadth/Depth search strategies

We implement a tree search by using breadth or depth search strategies. Some issues about the control of repeated states should be mentioned here:

1. In general, CLIPS performs an automatic control of the repeated states because, by default, it does not allow for fact duplication. This means that CLIPS will prevent a fact identical to one already existing in the WM from being asserted again in the WM.

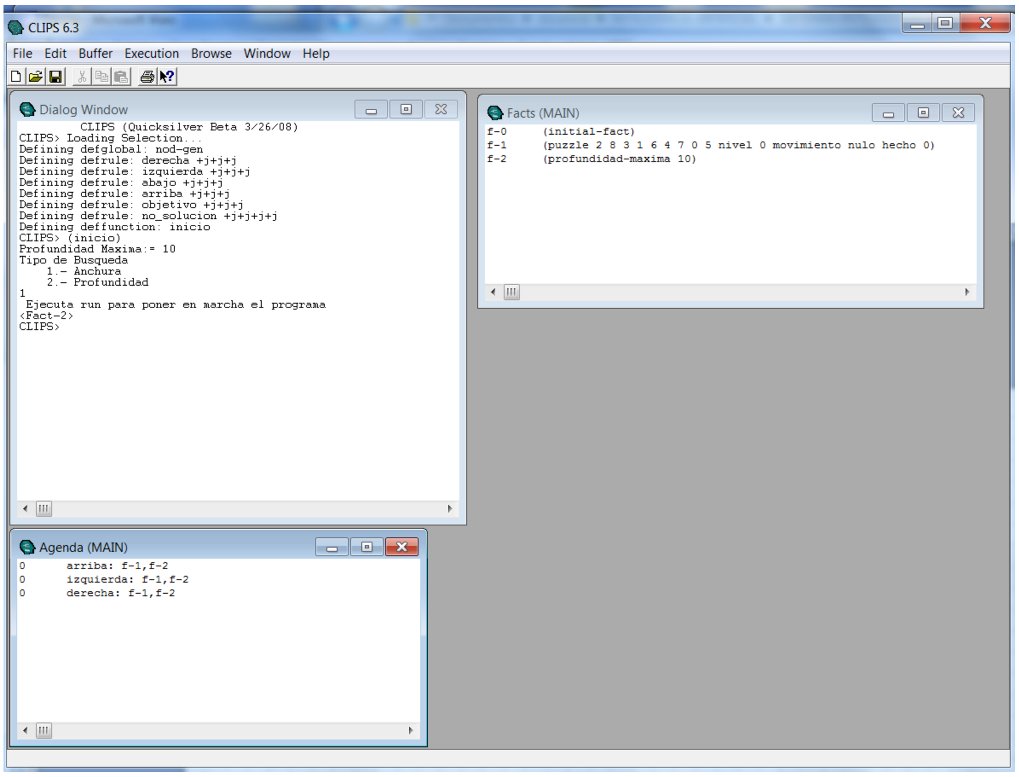
2. However, in our model we use the extra field ***level*** to store the depth level of node. This means that a node which is at level 3 and represents the same state as another node at level 5 are not actually denoted by the same fact because the depth levels are different. Therefore, CLIPS will assert both facts in the WM. Additionally, our fact representation also includes the extra field ***fact*** to be able to retrieve the solution path, which implies a distinguishing factor between two completely identical puzzle configurations.

3. In practice, for the reason mentioned in point 2, there is no actual control of repeated states in CLIPS, except for the direct loops that are avoided through the information in the field ***movement***.

4. Nevertheless, if we did not define the fields ***level***, ***movement*** and ***fact*** in the puzzle pattern then repeated states would be automatically detected in CLIPS. But, in this case, we would not know the depth level of the solution nor could we retrieve the solution path.

Following, we show a couple of iterations of the recognize-act cycle of the Rete-based inference engine of CLIPS when applied to the puzzle program by using BREADTH as the control strategy of the Agenda. We also show the equivalence of this process with a tree search process (actually, CLIPS implements the GRAPH-SEARCH version as all nodes are maintained in memory).

The initial situation is shown in Figure 7, which corresponds to the search tree of Figure 8.



```
          CLIPS (Quicksilver Beta 3/26/08)
CLIPS> Loading Selection...
Defining defglobal: nod-gen
Defining defrule: derecha +j+j+j
Defining defrule: izquierda +j+j+j
Defining defrule: abajo +j+j+j
Defining defrule: arriba +j+j+j
Defining defrule: objetivo +j+j+j
Defining defrule: no_solucion +j+j+j+j
Defining deffunction: inicio
CLIPS> (inicio)
Profundidad Maxima:= 10
Tipo de Busqueda
    1.- Anchura
    2.- Profundidad
1
 Ejecuta run para poner en marcha el programa
<Fact-2>
CLIPS>
```

```
f-0     (initial-fact)
f-1     (puzzle 2 8 3 1 6 4 7 0 5 nivel 0 movimiento nulo hecho 0)
f-2     (profundidad-maxima 10)
```

```
0       arriba: f-1,f-2
0       izquierda: f-1,f-2
0       derecha: f-1,f-2
```

Fig. 7 Initial situation of the WM (facts) and the Agenda

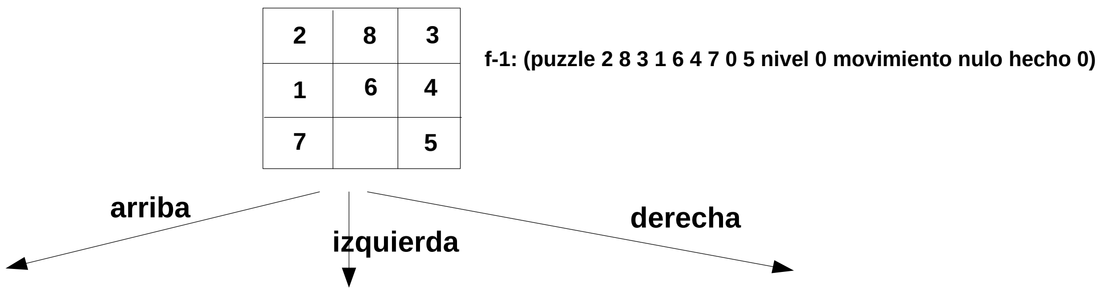
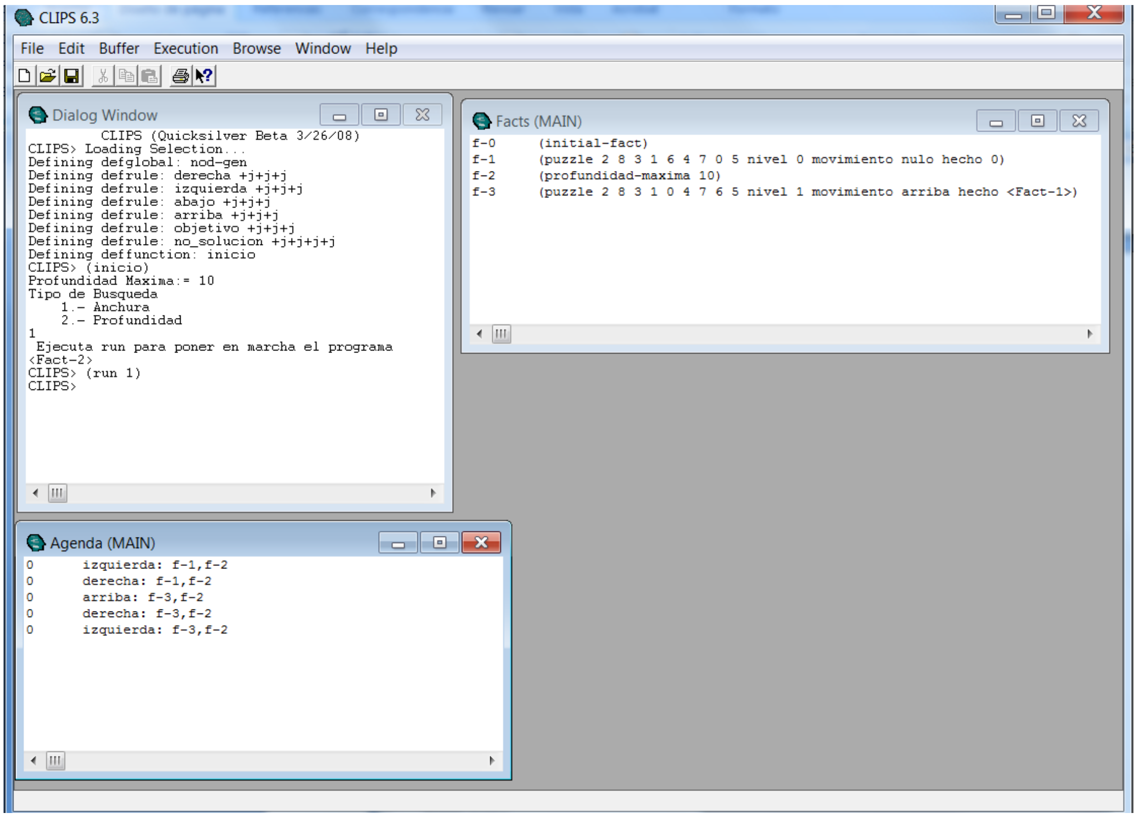

2
8
3
1
6
4
7
5
f-1: (puzzle 2 8 3 1 6 4 7 0 5 nivel 0 movimiento nulo hecho 0)
arriba
izquierda
derecha

Fig. 8 Search tree corresponding to the situation of Figure 7

The three activations in the Agenda represent the three movements that can be applied in the initial state; `up` (arriba), `left` (izquierda) and `right` (derecha). Unlike a classical search process, the child nodes have not been generated yet (the only fact in the WM is `f-1`, besides the static fact `f-2`).

If we run one step of the inferential process (one recognize-act cycle) we get:



```
         CLIPS (Quicksilver Beta 3/26/08)
CLIPS> Loading Selection...
Defining defglobal: nod-gen
Defining defrule: derecha +j+j+j
Defining defrule: izquierda +j+j+j
Defining defrule: abajo +j+j+j
Defining defrule: arriba +j+j+j
Defining defrule: objetivo +j+j+j
Defining defrule: no_solucion +j+j+j+j
Defining deffunction: inicio
CLIPS> (inicio)
Profundidad Maxima:= 10
Tipo de Busqueda
    1.- Anchura
    2.- Profundidad
1
 Ejecuta run para poner en marcha el programa
<Fact-2>
CLIPS> (run 1)
CLIPS>
```

```
f-0     (initial-fact)
f-1     (puzzle 2 8 3 1 6 4 7 0 5 nivel 0 movimiento nulo hecho 0)
f-2     (profundidad-maxima 10)
f-3     (puzzle 2 8 3 1 0 4 7 6 5 nivel 1 movimiento arriba hecho <Fact-1>)
```

```
0       izquierda: f-1,f-2
0       derecha: f-1,f-2
0       arriba: f-3,f-2
0       derecha: f-3,f-2
0       izquierda: f-3,f-2
```

Fig. 9 Status of the Agenda and WM after one recognize-act cycle

which corresponds to the search tree of Figure 10.

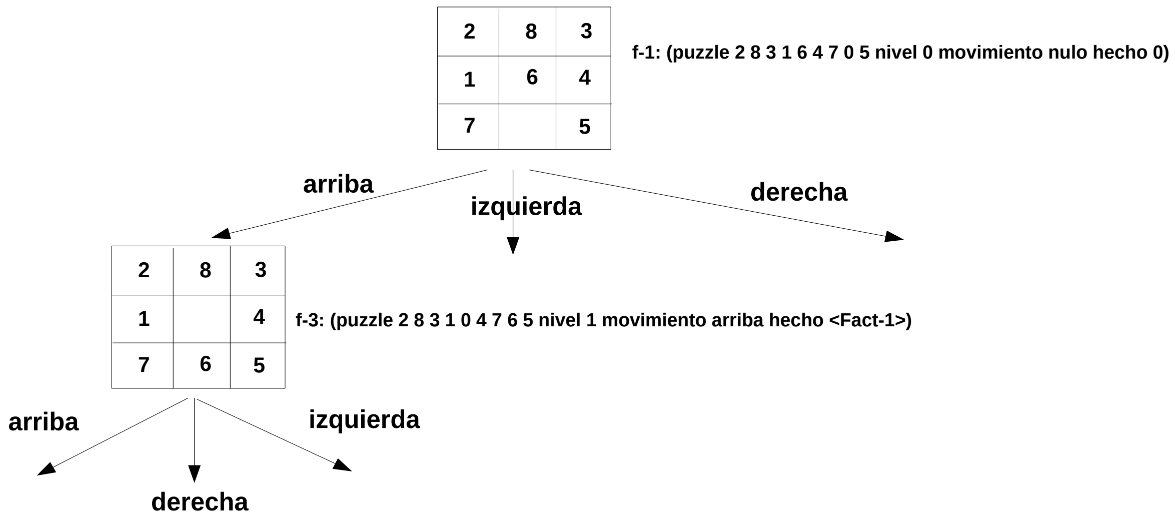
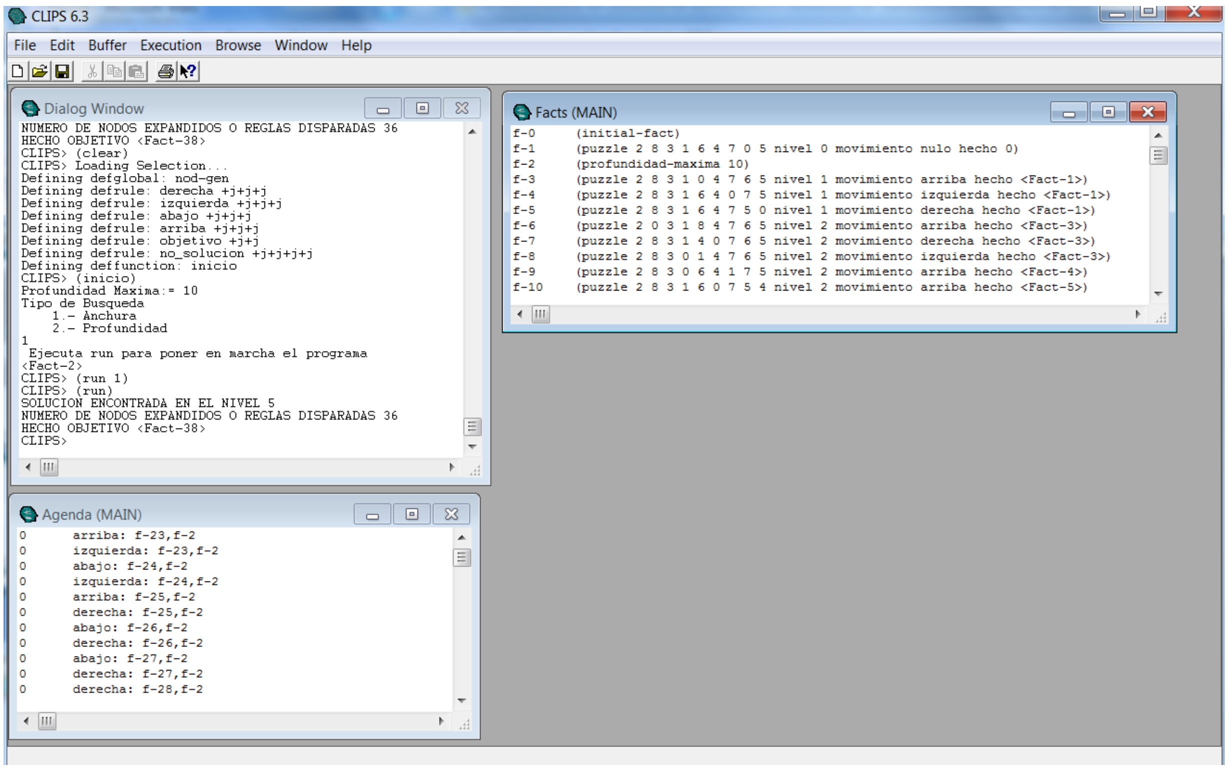

2 8 3
1 6 4
7 5
f-1: (puzzle 2 8 3 1 6 4 7 0 5 nivel 0 movimiento nulo hecho 0)
arriba
izquierda
derecha
2 8 3
1 4
7 6 5
f-3: (puzzle 2 8 3 1 0 4 7 6 5 nivel 1 movimiento arriba hecho <Fact-1>)
arriba
izquierda
derecha

Fig. 10 Search tree corresponding to the situation of Figure 9

Following, the rule `left` over the initial state (fact `f-1`) would be fired, then the rule `right`, and then the rule `up` over the state represented by fact `f-3`; and so on, thus implementing a breadth search process. When the search process is over, we would end up in a situation like the one shown in Figure 11.



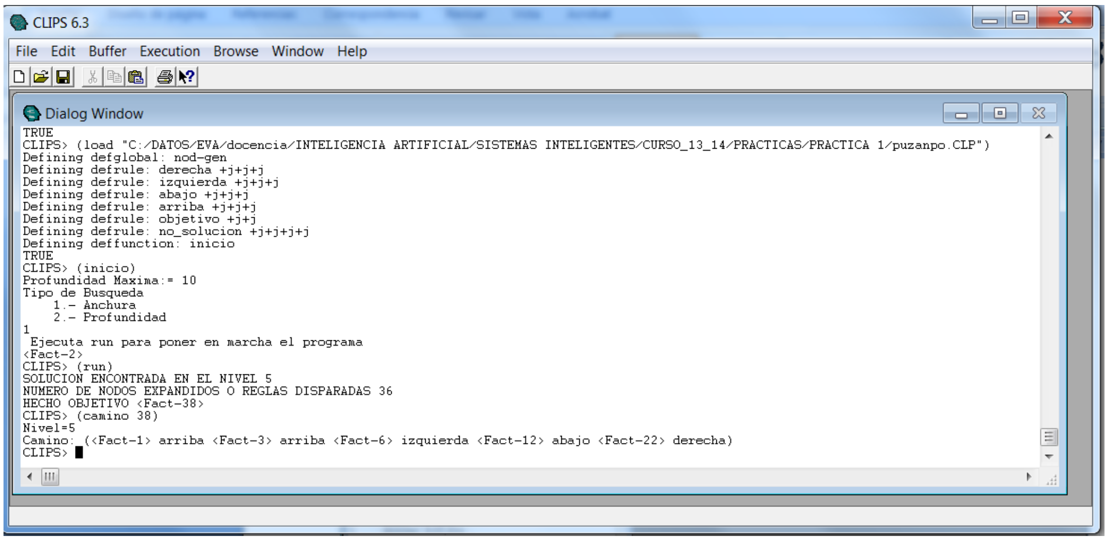
```
NUMERO DE NODOS EXPANDIDOS O REGLAS DISPARADAS 36
HECHO OBJETIVO <Fact-38>
CLIPS> (clear)
CLIPS> Loading Selection...
Defining defglobal: nod-gen
Defining defrule: derecha +j+j+j
Defining defrule: izquierda +j+j+j
Defining defrule: abajo +j+j+j
Defining defrule: arriba +j+j+j
Defining defrule: objetivo +j+j
Defining defrule: no_solucion +j+j+j+j
Defining deffunction: inicio
CLIPS> (inicio)
Profundidad Maxima:= 10
Tipo de Busqueda
    1.- Anchura
    2.- Profundidad
1
 Ejecuta run para poner en marcha el programa
<Fact-2>
CLIPS> (run 1)
CLIPS> (run)
SOLUCION ENCONTRADA EN EL NIVEL 5
NUMERO DE NODOS EXPANDIDOS O REGLAS DISPARADAS 36
HECHO OBJETIVO <Fact-38>
CLIPS>
```

```
f-0     (initial-fact)
f-1     (puzzle 2 8 3 1 6 4 7 0 5 nivel 0 movimiento nulo hecho 0)
f-2     (profundidad-maxima 10)
f-3     (puzzle 2 8 3 1 0 4 7 6 5 nivel 1 movimiento arriba hecho <Fact-1>)
f-4     (puzzle 2 8 3 1 6 4 0 7 5 nivel 1 movimiento izquierda hecho <Fact-1>)
f-5     (puzzle 2 8 3 1 6 4 7 5 0 nivel 1 movimiento derecha hecho <Fact-1>)
f-6     (puzzle 2 0 3 1 8 4 7 6 5 nivel 2 movimiento arriba hecho <Fact-3>)
f-7     (puzzle 2 8 3 1 4 0 7 6 5 nivel 2 movimiento derecha hecho <Fact-3>)
f-8     (puzzle 2 8 3 0 1 4 7 6 5 nivel 2 movimiento izquierda hecho <Fact-3>)
f-9     (puzzle 2 8 3 0 6 4 1 7 5 nivel 2 movimiento arriba hecho <Fact-4>)
f-10    (puzzle 2 8 3 1 6 0 7 5 4 nivel 2 movimiento arriba hecho <Fact-5>)
```

```
0       arriba: f-23,f-2
0       izquierda: f-23,f-2
0       abajo: f-24,f-2
0       izquierda: f-24,f-2
0       arriba: f-25,f-2
0       derecha: f-25,f-2
0       abajo: f-26,f-2
0       derecha: f-26,f-2
0       abajo: f-27,f-2
0       derecha: f-27,f-2
0       derecha: f-28,f-2
```

Fig. 11 Agenda and WM after the execution of the search process

If we call the function `(path 38)`, where 38 is the index of the goal fact, we get this output:



```
TRUE
CLIPS> (load "C:/DATOS/EVA/docencia/INTELIGENCIA ARTIFICIAL/SISTEMAS INTELIGENTES/CURSO_13_14/PRACTICAS/PRACTICA 1/puzanpo.CLP")
Defining defglobal: nod-gen
Defining defrule: derecha +j+j+j
Defining defrule: izquierda +j+j+j
Defining defrule: abajo +j+j+j
Defining defrule: arriba +j+j+j
Defining defrule: objetivo +j+j
Defining defrule: no_solucion +j+j+j+j
Defining deffunction: inicio
TRUE
CLIPS> (inicio)
Profundidad Maxima:= 10
Tipo de Busqueda
    1.- Anchura
    2.- Profundidad
1
 Ejecuta run para poner en marcha el programa
<Fact-2>
CLIPS> (run)
SOLUCION ENCONTRADA EN EL NIVEL 5
NUMERO DE NODOS EXPANDIDOS O REGLAS DISPARADAS 36
HECHO OBJETIVO <Fact-38>
CLIPS> (camino 38)
Nivel=5
Camino: (<Fact-1> arriba <Fact-3> arriba <Fact-6> izquierda <Fact-12> abajo <Fact-22> derecha)
CLIPS>
```

Fig. 12 Solution path for the puzzle configuration of Figure 8

3. Execution of the RBS

We provide the following files:

- The file `puzanpo.clp` implements the 8-puzzle problem in breadth/depth and heuristic, respectively.

- The file `auxiliar.clp` contains the function `check_conf` that checks the solvability of a given puzzle configuration.

In order to run a particular example, we have to change the initial puzzle configuration in the function `start` of the file `puzanpo.clp` before loading the file. Before running the example, one can execute the function `check_conf` in the file `auxiliar.clp` to check that the puzzle configuration we want to run is solvable.

4. Evaluation

The execution of an individual problem requires setting the initial puzzle configuration in the function `start`. For example:

8	**1**	**3**
7	**2**	**5**
4		**6**

The fact that represents this puzzle configuration is as follows:

```
(puzzle 8 1 3 7 2 5 4 0 6 level 0 movement null fact 0))
```

The following table shows the results obtained when running the configuration with alternative search strategies and different maximum depth levels:

Search Strategy	*Max. depth level*	*Solution length (level)*	*#Expanded nodes*
Breadth	10	9	442
Breadth	20	9	442
Depth	10	9	479
Depth	20	15	68318