UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Sistemas basados en reglas[1]

Alfons Juan
Albert Sanchis
Jorge Civera

DSIC

Departamento de Sistemas Informáticos y Computación

[1] Para una correcta visualización, se requiere Acrobat Reader v. 7.0 o superior

Objetivos

- Describir las bases de los sistemas basados en reglas (Rule-Based Systems, RBS).
- Usar CLIPS para el diseño, construcción y ejecución de RBS.

Índice

1. Introducción

- 1984: el grupo de IA del *NASA's Johnson Space Center* decide desarrollar una herramienta C de construcción de sistemas expertos
- 1985: se desarrolla la versión prototipo de ***C Language Integrated Production System (CLIPS)***, idónea para formación
- 1986: CLIPS se comparte con grupos externos
- 1987–2002: mejoras de rendimiento y nuevas funcionalidades; por ejemplo, programación procedural, OO e interfaces gráficas
- Desde 2008: Gary Riley mantiene CLIPS fuera de la NASA [1, 2] (URL: `https://www.clipsrules.net/`)
- Documentación:
  - ***Manual de referència I: Guia de programación bàsica*** [3]
  - Manual de referencia II: Guía de programación avanzada [4]
  - Manual de referencia III: Guía de interfaces [5].
  - Guía del usuario [6]

2. Sistemas basados en reglas con CLIPS

CLIPS permite construir SBRs con 3 componentes:

1. ***Base de hechos (BH):***
   - Cada estado del problema suele representarse con un único hecho de acuerdo con un cierto patrón de ***hecho-estado***
   - A cada paso de ejecución, los hecho-estado representan estados del problema ya explorados o pendientes de exploración
   - El resto de hechos son información ***estática*** del problema
2. ***Base de reglas (BR):***
   - Cada posible acción aplicable a uno o más estados del problema suele representarse con una única regla `izq=>der`
   - La parte izquierda escoge el conjunto de estados al cual es aplicable
   - La parte derecha suele resultar con nuevos hecho-estado añadidos (o eliminados) a la BH
3. ***Motor de inferencia:*** instanciación, selección y ejecución de reglas

- ***Motor de inferencia:***
  - ***Entrada:*** base de hechos y base de reglas iniciales, BH y BR
  - ***Salida:*** base de hechos final, BH
  - ***Método:***

$CC = \emptyset$ // conjunto conflicto de instancias de reglas

repetir

// añadimos nuevas instancias al CC a partir de nuevos hechos:
$CC =$ Instancia(BH, BR, CC)

si $CC = \emptyset$: **salir** // objetivo no conseguido

// seleccionamos una instancia con algún criterio:
$InstRule =$ Selecciona(CC)

// ejecutamos $InstRule$ y actualizamos BH y CC:
$(BH, CC) =$ Ejecuta$(BH, CC, InstRule)$

hasta objectivo conseguido

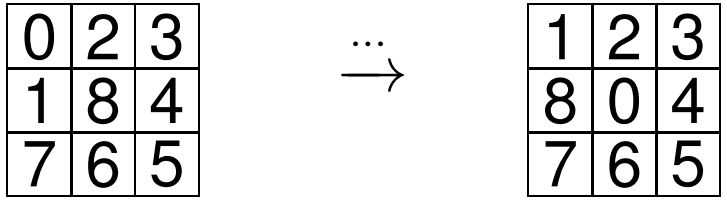
- ***Tres pasos básicos en inferencia:***

1. ***Instancia:*** añade nuevas instancias al CC a partir de nuevos hechos, sin repetir instancias añadidas anteriormente ***(refracción)***
2. ***Selecciona:*** aplica un criterio de selección como ahora:
   - ***Profundidad:*** primero la instancia más reciente
   - ***Anchura:*** primero la instancia más antigua
   - ***Prioridad:*** primero la instancia de la regla más prioritaria
3. ***Ejecuta:*** aplica las órdenes de la instancia seleccionada:
   - ***Eliminación de hechos*** en la BH
   - ***Eliminación de instancias*** en el CC con hechos eliminados
   - ***Inserción de hechos*** nuevos en la BH ***sin repeticiones***

- ***No duplicidad de hechos y refracción***:
  - Por defecto, los hechos en CLIPS no se duplican en la BH
  - ***Refracción:*** Una regla sólo se puede instanciar una vez con el mismo hecho y con la misma instanciación de valores a variables
  - No duplicación y refracción previenen de la activación infinita de reglas
  - La inserción de un nuevo hecho en la BH puede provocar la instanciación de nuevas reglas

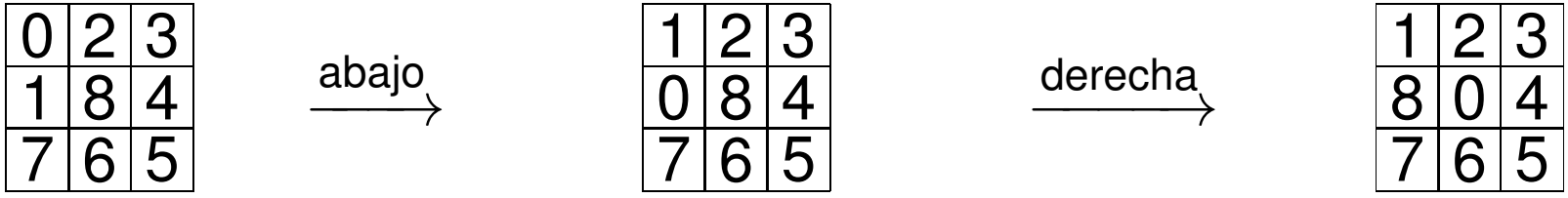
3. El problema del 8-puzle en CLIPS

Dado un estado inicial, se busca llegar a un estado objetivo mediante movimientos de la ficha en blanco (ficha 0): derecha, izquierda, arriba y abajo



Estado inicial Objetivo

Solución para el estado inicial y objetivo anterior:

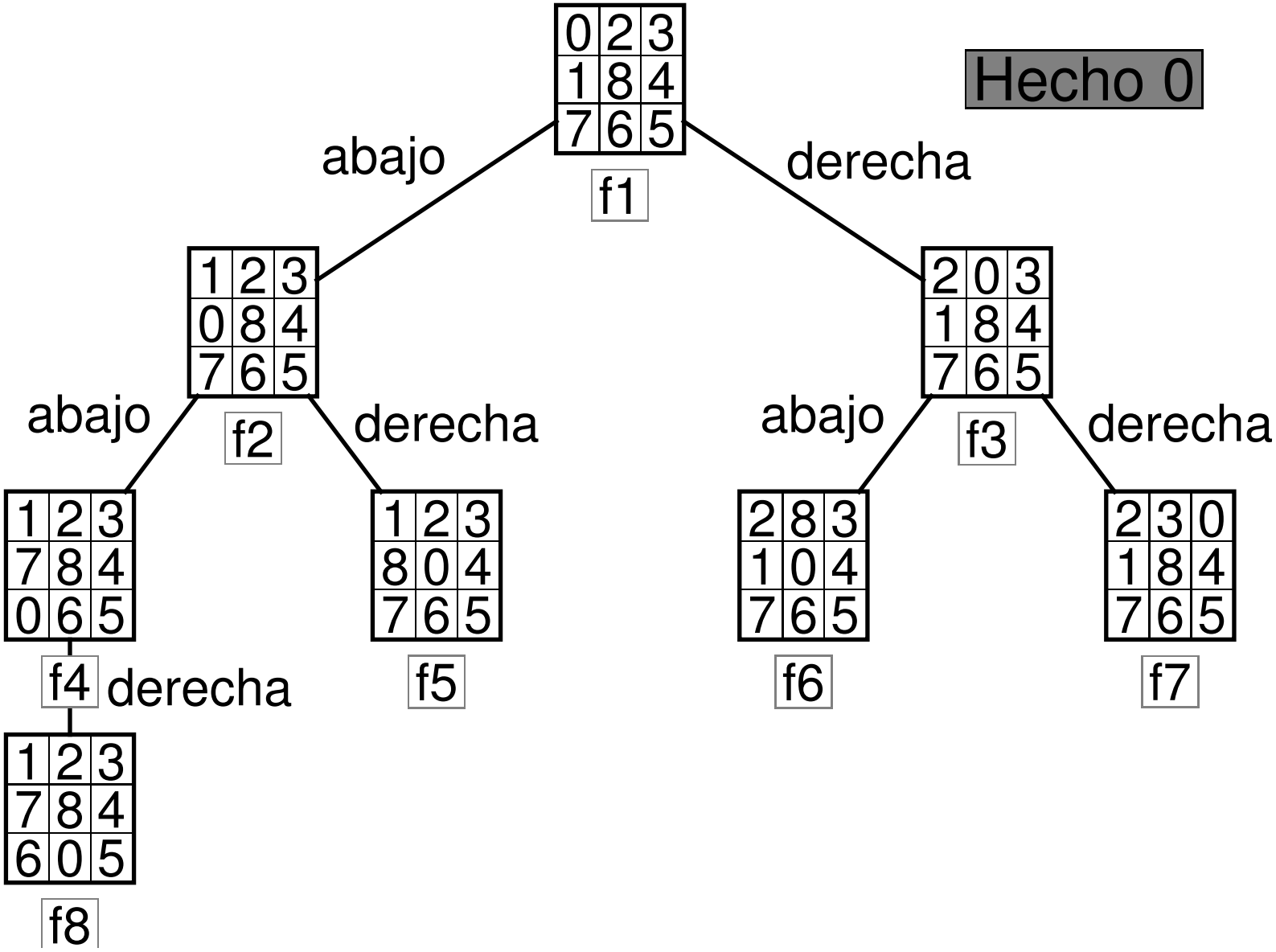


Estado inicial Estado intermedio Estado objetivo

Un RBS sencillo para el problema del 8-puzle

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

(defrule izquierda
  (puzzle $?x ?y 0 $?z)
  (test (<> (length$ $?x) 2))
  (test (<> (length$ $?x) 5)) =>
  (assert (puzzle $?x 0 ?y $?z)))

(defrule derecha
  (puzzle $?x 0 ?y $?z)
  (test (<> (length$ $?x) 2))
  (test (<> (length$ $?x) 5)) =>
  (assert (puzzle $?x ?y 0 $?z)))

(defrule arriba
  (puzzle $?x ?a ?b ?c 0 $?y) =>
  (assert (puzzle $?x 0 ?b ?c ?a $?y)))

(defrule abajo
  (puzzle $?x 0 ?a ?b ?c $?z) =>
  (assert (puzzle $?x ?c ?a ?b 0 $?z)))

(defrule objetivo
  (puzzle 1 2 3 8 0 4 7 6 5) =>
  (printout t "Solución encontrada!" crlf)
  (halt))
```

Traza de BFS con CLIPS en el problema del 8-puzzle



4. Hechos

- Lista de símbolos entre paréntesis donde el primero indica la "relación" ***(puzzle 0 2 3 1 8 4 7 6 5)***
- Hechos iniciales: con ***deffacts***

  ```
  (deffacts <name> [<comment>] <fact>*)
  ```

  donde la sintaxis de cada ***hecho*** es:

  ```
  <fact> ::= (<symbol> <constant>*)
  ```

- Ejemplo:

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

- Órdenes sobre hechos:
  - ***assert*** (para insertar un nuevo hecho en la BH)
  - ***retract*** (para eliminar un hecho de la BH)

5. Reglas

- Constan de dos partes, la LHS y la RHS:
  - Antecedente o parte izquierda (LHS):
    - ↦ Condiciones a cumplir para que se ejecute la RHS
  - Consecuente o parte derecha (RHS):
    - ↦ Acciones a ejecutar si se cumple la LHS

- Sintaxis:

  `(defrule <name> <LHS> => <RHS>)`

  donde la parte izquierda ***(LHS)*** es:

  `LHS ::== <conditional-element>*`

  y la parte derecha ***(RHS)***:

  `RHS ::= <action>*`

- ***Activación y ejecución de reglas***:
  - Se ejecuta o dispara (*fire*) en función de la existencia o no de hechos con los cuales se cumplen las condiciones
  - El motor de inferencia es el encargado de encajar (hacer *matching* de) hechos con reglas
    - ↦ Llamamos ***instancias*** de una regla a los diferentes matchings de hechos con la regla que se puedan hacer (cero, uno o más)
    - ↦ ***Agenda o conjunto conflicto (CC):*** conjunto de instancias de todas las reglas pendientes de ejecución

- ***Ciclo básico de ejecución de reglas***:
  - ***Motor de inferencia:*** bucle con los siguientes pasos básicos

a) ***Selección de una instancia (de regla) de la agenda***

↦ Si no hay ninguna instancia en la agenda, se acaba

b) ***Ejecución de la parte derecha de la regla seleccionada***

c) ***Activación y desactivación de instancias de reglas*** como consecuencia de la ejecución de la regla seleccionada

↦ Las activadas se añaden a la agenda

↦ Las desactivadas se eliminan de la agenda

d) ***Re-evaluación de prioridades dinámicas de instancias en la agenda:*** si utilizamos prioridades dinámicas con `salience`

- ***Estrategias de resolución de conflictos***
  - ***Ordenación de instancias de reglas en la agenda:***
    - ***Por prioridad:*** las nuevas instancias se situan encima (delante) de las de menor prioridad y bajo (detrás) de las de mayor. `(salience <entero>)` define la prioridad de la regla
      - Prioridad mínima: -10000; máxima: 10000; por defecto: 0

```
(defrule <name>
   (declare (salience <integer>))
   <LHS> => <RHS>)
```

      - Ejemplo:

```
(defrule objetivo
   (declare (salience 1))
   (puzzle 1 2 3 8 0 4 7 6 5) =>
   (printout t "Solución encontrada!" crlf)
   (halt))
```

    - ***Profundidad (*depth*)***: Las nuevas instancias se sitúan encima de todas las de igual prioridad; es la estrategia per defecto
    - ***Anchura (*breadth*)***: Las nuevas reglas se sitúan bajo de todas las de igual prioridad

Sintaxis de la LHS

- ***Elementos condicionales (CEs):*** serie de cero, uno o más elementos de que consta la LHS de una regla y que se han de satisfacer para que se añada una instancia de la regla al conjunto conflicto (o agenda)
- Hay ocho tipos de CEs, pero sólo hacemos uso de cinco:
  - ***CEs patrón:*** restricciones sobre los hechos que lo satisfacen
  - ***CEs test:*** evalúan expresiones durante el encaje de patrones
  - ***CEs or:*** dado un grupo de CEs, al menos uno se ha de satisfacer
  - ***CEs and:*** dado un grupo de CEs, todos se han de satisfacer
  - ***CEs not:*** dado un CE, *no* se ha de satisfacer

```
<conditional-element> ::=
  <pattern-CE> | <assigned-pattern-CE> |
  <test-CE> | <or-CE> | <and-CE> | <not-CE>
```

CE patrón

- ***<pattern-CE>:*** lista ordenada con un símbolo inicial, seguido de constantes, comodines y variables

```
<pattern-CE> ::= (<symbol> <constraint>*)
<constraint> ::= <constant> | <variable> | ? | $?
<constant> ::= <symbol> | <string> | <integer> | <float>
<variable> ::=  <single-vble> | <multi-vble>
<single-vble> ::= ?<symbol>
<multi-vble> ::= $?<symbol>
```

- Ejemplo:

```
(puzzle $?x ?y 0 $?z)
```

  - ***$?x*** es una variable multi-valuada instanciándose a cero o más elementos
  - ***?y*** es una variable mono-valuada intanciándose a exactamente un elemento
  - ***0*** es una constante

- ***comodines ?*** y ***$?*** se comportan igual que las variables mono y multi-valuadas, respectivamente, pero los valores instanciados no se almacenan

Pattern matching

- Posibles “encajes” (***matchings***) entre patrones y hechos
  - un patrón “encaja” una vez con un hecho

Hechos	Patrón
f-1: ***(puzzle 1 2 3 0 7 4 8 6 5)***	***(puzzle $?x 0 $?y 7 $?z)***
f-2: ***(puzzle 4 7 1 5 8 0 6 3 2)***	

Match#	***$?x***	***$?y***	***$?z***
f-1	(1 2 3)	()	(4 8 6 5)

  - un patron “encaja” una vez con diferentes hechos

Hechos	Patrón
f-1: ***(puzzle 1 2 3 0 7 4 8 6 5)***	***(puzzle $?x 0 $?y)***
f-2: ***(puzzle 4 7 1 5 8 0 6 3 2)***	

Match#	***$?x***	***$?y***
f-1	(1 2 3)	(7 4 8 6 5)
f-2	(4 7 1 5 8)	(6 3 2)

▷ diferentes patrones “encajan” con el mismo hecho

Hechos	Patrón
f-1: ***(puzzle 1 2 3 0 7 4 8 6 5)***	***(puzzle $?x 0 ?y $?z)***
	(puzzle $?x ?y 0 $?z)

Match#	***$?x***	***?y***	***$?z***
f-1	(1 2 3)	(7)	(4 8 6 5)
f-1	(1 2)	(3)	(7 4 8 6 5)

- diferentes “encajes” entre un patrón y un hecho debido a diferentes instanciaciones de variables

Hechos	Patrón
f-1: ***(puzzle 1 2 3 0 7 4 8 6 5)***	***(puzzle $?x ?y $?z)***

Match#	***$?x***	***?y***	***$?z***
f-1	()	(1)	(2 3 0 7 4 8 6 5)
f-1	(1)	(2)	(3 0 7 4 8 6 5)
f-1	(1 2)	(3)	(0 7 4 8 6 5)
f-1	(1 2 3)	(0)	(7 4 8 6 5)
f-1	(1 2 3 0)	(7)	(4 8 6 5)
f-1	(1 2 3 0 7)	(4)	(8 6 5)
f-1	(1 2 3 0 7 4)	(8)	(6 5)
f-1	(1 2 3 0 7 4 8)	(6)	(5)
f-1	(1 2 3 0 7 4 8 6)	(5)	()

Asignación de patrones a variables

- ***<assigned-pattern-CE>:*** Asignación del índice de un hecho con el que hace ***matching*** un patrón a una variable con el fin de borrar el hecho de la base de hechos mediante una acción ***retract*** en la RHS

```
<assigned-pattern-CE> ::= <single-vble> <- <pattern-CE>
```

- Ejemplo:

```
(defrule izquierda
  ?f <- (puzzle $?x ?y 0 $?z)
  (test (<> (length$ $?x) 2))
  (test (<> (length$ $?x) 5))
  =>
  (retract ?f)
  (assert (puzzle $?x 0 ?y $?z)))
```

CE Test

- ***<test-CE>*** se satisface si ***<function-call>*** no devuelve ***False***

```
<test> ::= (test <function-call>)
<function-call> ::=  (<function-name> <expression>*)
<function-name> ::=  > | < | = | <> | eq | neq | <member>
<expression> ::= <constant> | <variable> | <function-call>
```

- Ejemplo:

```
(test (<> (length$ $?x) 2))
```

  - **<>** es el operador “desigualdad” para ***<integer>*** o ***<float>***
  - ***length$*** es una función pre-definida que calcula la longitud de una lista

- Hay que tener en cuenta que se usa notación prefija (el operador va delante de los operandos)

CEs or, and, not

- `(or <CE>+)` se satisface si cualquier de los `<CE>+` lo hace

```
(or (test (= ?x 1)) (test (= ?y 2)))
```

- `(and <CE>+)` se satisface si todos los `<CE>+` lo hacen

```
(and (test (= ?x 1)) (test (= ?y 2)))
```

- `(not <CE>+)` se satisface si `<CE>` no lo hace

```
(not (test (= ?x 0)))
```

Sintaxis de la RHS

- Acciones en la RHS permiten insertar y eliminar hechos, mostrar texto, detener el motor de inferencia, etc.:

```
<action> ::=
(assert <fact>+) |
(retract <fact-index>+)  |
(printout t <string> crlf) |
(halt)

<fact-index> ::= <integer> | <single-vble>
```

- Ejemplos:

```
(defrule abajo
  ?f <- (puzzle $?x 0 ?a ?b ?c $?z) =>
  (retract ?f)
  (assert (puzzle $?x ?c ?a ?b 0 $?z)))
(defrule objetivo
  (puzzle 1 2 3 8 0 4 7 6 5) =>
  (printout t "¡Solution found!" crlf)
  (halt))
```

6. Funciones multicampo

- crea valor multicampo

```
(create$ a b)
(a b)
```

- n-ésimo campo del multicampo

```
(nth$ 2 (create$ a b))
 b
```

- longitud de multicampo

```
(length$ (create$ a b c))
3
```

- posicion(es) de valor en multicampo

```
(member$ b (create$ a b b))
 2
(member$ (create$ b b) (create$ a b b))
(2 3)
(member$ c (create$ a b b))
 FALSE
```

Referencias

[1] G. Riley. CLIPS: A Tool for Building Expert Systems.

[2] G. Riley. CLIPS: SourceForge Project Page.

[3] C. Culbert et al. CLIPS Reference Manual I: Basic Programming Guide (v6.4.1).

[4] C. Culbert et al. CLIPS Reference Manual II: Advanced Programming Guide (v6.4.1).

[5] C. Culbert et al. CLIPS Reference Manual III: Interfaces Guide (v6.4.1).

[6] J. Giarratano. CLIPS User's Guide (v6.4.1).