Chapter Overview

Case Studies Knowledge-Based Systems

CLIPS sample programs

- decision tree (animal guessing)
- backward chaining (wine selection)
- monitoring system

R1/XCON

commercial expert system for the
configuration of minicomputer systems

Animal Guessing Decision Tree

problem description

the system asks the user questions about animals, and tries to "guess" the animal the user has in mind

approach

a decision tree with information about animals is built, and possibly expanded with new information from the user

implementation

the information about individual animals is stored in nodes
the nodes are linked into a tree

limitations

limited number of properties for animals
systematic categorization of animals difficult
only yes/no answers

Example Animal Guessing

```
Is the animal warm-blooded? (yes or no) yes
Does the animal purr? (yes or no) no
Does the animal bleat? (yes or no) no
Does the animal moo? (yes or no) no
Is the animal used for riding? (yes or no) yes
Does it have humps (yes or no) yes
I guess it is a camel
Am I correct? (yes or no) yes
Try again? (yes or no) yes
Is the animal warm-blooded? (yes or no) no
Does the animal have legs? (yes or no) no
I guess it is a snake
Am I correct? (yes or no) no
What is the animal? snail
What question when answered yes will distinguish
   a snail from a snake? Does the animal have a slimy bottc
Now I can guess snail
Try again? (yes or no) yes
Is the animal warm-blooded? (yes or no) no
Does the animal have legs? (yes or no) no
Does the animal have a slimy bottom? (yes or no) yes
I guess it is a snail
Am I correct? (yes or no) yes
Try again? (yes or no) no
```


Decision Tree Strategy

ask question for the current node

if yes, go to the node for the yes branch
if no, go to the node for the no branch

check answer for the current node

if correct, display it
otherwise, expand the tree

determine new animal

ask the user for a question which when answered yes will distinguish the animal at the current node from the correct answer for the new animal

tree expansion

replace the answer node with a decision node
- question as provided by the user
- set no branch to the current answer
- set yes branch to the new correct answer

Node Structure

nodes are stored as facts

```
(deftemplate node
  (slot name) ; unique name for the node
  (slot type) ; answer or decision node
  (slot question) ; question to be asked
  (slot yes-node) ; pointer to the next pos
  (slot no-node) ; pointer to the next nega
  (slot answer)) ; only for answer type nod
```

facts are stored in a file

```
(load-facts "animal.dat")
(save-facts "animal.dat" local node))
```


Rules for the Decision Tree

initialization
load facts; start with the root node

ask decision node question
print out question, read answer
check for bad answers (only "yes" or "no")

proceed to next node
either yes or no branch
retract current node, replace with new node
retract current answer

guess animal
print the guess according to the current node
ask for and read feedback

guess was correct
done; ask if user wants to try again

guess was incorrect
proceed to tree expansion phase

tree expansion
replace current answer node with a decision node
get the correct answer from the user
get the distinguishing question from the user
create the new node

initialization

load facts; start with the root node

```
  (defrule initialize
(not (node (name root)))
=>
(load-facts "animal.dat")
(assert (current-node root)))
```

Example Fact File

```
(node (name root) (type decision) (question "Is the animal
 (yes-node node1) (no-node node2) (answer nil))
(node (name node1) (type decision) (question "Does the anim
 (yes-node node3) (no-node node4) (answer nil))
(node (name node3) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer cat))
(node (name node2) (type decision) (question "Does the anim
 (yes-node gen9) (no-node gen10) (answer snake))
(node (name node4) (type decision) (question "Does the anim
 (yes-node gen1) (no-node gen2) (answer dog))
(node (name gen1) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer sheep))
(node (name gen2) (type decision) (question "Does the anima
 (yes-node gen3) (no-node gen4) (answer dog))
(node (name gen3) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer cow))
(node (name gen4) (type decision) (question "Is the animal
 (yes-node gen5) (no-node gen6) (answer dog))
(node (name gen6) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer dog))
(node (name gen9) (type decision) (question "When full-grow
 (yes-node gen7) (no-node gen8) (answer crocodile))
(node (name gen8) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer crocodile))
```



```
(node (name gen7) (type decision) (question "Does it have p
 (yes-node gen11) (no-node gen12) (answer lizard))
(node (name gen11) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer crab))
(node (name gen12) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer lizard))
(node (name gen5) (type decision) (question "Does it have h
 (yes-node gen13) (no-node gen14) (answer horse))
(node (name gen13) (type answer)
 (question nil) (yes-node nil) (no-node nil) (answer camel)
(node (name gen14) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer horse))
(node (name gen10) (type decision) (question "Does the anim
 (yes-node gen15) (no-node gen16) (answer snake))
(node (name gen15) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer snail))
(node (name gen16) (type answer) (question nil)
 (yes-node nil) (no-node nil) (answer snake))
```


Ask Decision Node Question

print out question, read answer
check for bad answers (only “yes” or “no”)

```
(defrule ask-decision-node-question
  ?node <- (current-node ?name)
  (node (name ?name)
        (type decision)
        (question ?question))
  (not (answer ?))
  =>
  (printout t ?question " (yes or no) ")
  (assert (answer (read))))
```


Proceed to Next Node

either yes or no branch
retract current node, replace with new node
retract current answer

```
(defrule proceed-to-yes-branch
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (yes-node ?yes-branch))
   ?answer <- (answer yes)
   =>
   (retract ?node ?answer)
   (assert (current-node ?yes-branch)))
```

similar rule for no branch

Guess Animal

print the guess according to the current node
ask for and read feedback

```
(defrule ask-if-answer-node-is-correct
  ?node <- (current-node ?name)
  (node (name ?name) (type answer) (answer ?val
  (not (answer ?))
  =>
  (printout t "I guess it is a " ?value crlf)
  (printout t "Am I correct? (yes or no) ")
  (assert (answer (read))))
```


Guess Correct

done; ask if user wants to try again

```
(defrule answer-node-guess-is-correct
  ?node <- (current-node ?name)
  (node (name ?name) (type answer))
  ?answer <- (answer yes)
  =>
  (assert (ask-try-again))
  (retract ?node ?answer))
```


Guess Incorrect

proceed to tree expansion phase

```
(defrule answer-node-guess-is-incorrect
  ?node <- (current-node ?name)
  (node (name ?name) (type answer))
  ?answer <- (answer no)
  =>
  (assert (replace-answer-node ?name))
  (retract ?answer ?node))
```


Tree Expansion

replace current answer node with a decision node
get the correct answer from the user
get the distinguishing question from the user
create the new node

```
(defrule replace-answer-node
   ?phase <- (replace-answer-node ?name)
   ?data <- (node (name ?name)
                  (type answer)
                  (answer ?value))
   =>
   (retract ?phase)
   ; Determine what the guess should have been
   (printout t "What is the animal? ")
   (bind ?new-animal (read))
   ; Get the question for the guess
   (printout t "What question when answered yes
   (printout t "will distinguish " crlf "   a ")
   (printout t ?new-animal " from a " ?value "?
   (bind ?question (readline))
```

```
(printout t "Now I can guess " ?new-animal cr
; Create the new learned nodes
(bind ?newnode1 (gensym*))
(bind ?newnode2 (gensym*))
(modify ?data (type decision)
              (question ?question)
              (yes-node ?newnode1)
              (no-node ?newnode2))
(assert (node (name ?newnode1)
              (type answer)
              (answer ?new-animal)))
(assert (node (name ?newnode2)
              (type answer)
              (answer ?value)))
; Determine if the player wants to try again
(assert (ask-try-again)))
```


Evaluation of the Program

Initialization
decision tree is loaded
agenda contains
`ask-decision-node-question`

Questions
user answers questions
system reads answers, navigates the tree

Incorrect Guess
if the system makes an incorrect guess, the user is asked for the new animal, and a question that distinguishes it

New Node
a new node is inserted into the decision tree

see also spreadsheet

Summary Animal Guessing

decision tree
based on an implementation of decision trees in CLIPS

queries
questions are answered by searching for the right answer in a decision tree

expansion of the tree
if necessary, new animals with distinguishing questions can be integrated

storage
factual knowledge is stored via `deftemplates` in a file

Example Wine Selection

domain
selection of a wine type for a meal

approach
ask the user about important properties of the meal, and offer a recommendation from a selection of wine types

goal-based reasoning
based on backward chaining
more appropriate for the problem

backward chaining
emulated by the CLIPS forward chaining inference method

Knowledge Representation

as backward chaining rules (`if-then`)

representation of `if-then` rules
backward chaining rules are represented as facts
CLIPS rules are used to specify the evaluation mechanism (inference engine) for the evaluation of the backward chaining rules

representation of goal and subgoals
also as facts, with an attribute as slot

attributes
facts with name and value slots

Emulation of Backward Chaining

in CLIPS

generate goals and attributes
either from available `if-then` rules, or from the user

evaluation of backward chaining rules
remove goals with determined attribute values
for satisfied rules, add the consequent to the list of facts
remove rules that are currently not applicable
modify partially satisfied rules

systems that implement backward chaining directly (such as PROLOG are far more efficient for larger problems

Representation of Rules and Goals

```
(deftemplate BC::rule
   (multislot if)
   (multislot then))

(deftemplate BC::goal
   (slot attribute))

(deftemplate BC::attribute
   (slot name)
   (slot value))
```

`BC::rule` indicates that a module `BC` is used

Knowledge Base

```
(deffacts MAIN::wine-rules
   (rule (if main-course is red-meat and
                meal-is-pork is yes and white-sa
         (then best-color is white))
   (rule (if main-course is red-meat and
                meal-is-pork is yes and white-sa
         (then best-color is rose))
   (rule (if main-course is red-meat and
                meal-is-pork is no)
         (then best-color is red))
   (rule (if main-course is fish)
         (then best-color is white))
   (rule (if main-course is poultry and
             meal-is-turkey is yes)
         (then best-color is red))
   (rule (if main-course is poultry and
             meal-is-turkey is no)
         (then best-color is white)))
```

Initialization

```
(defmodule MAIN (import BC deftemplate rule goal

(deffacts MAIN::initial-goal
   (goal (attribute best-color)))

(defrule MAIN::start-BC
   =>
   (focus BC))
```


Goals and Attributes

```
(defrule BC::attempt-rule
   (goal (attribute ?g-name))
   (rule (if ?a-name $?)
         (then ?g-name $?))
   (not (attribute (name ?a-name)))
   (not (goal (attribute ?a-name)))
   =>
   (assert (goal (attribute ?a-name))))

(defrule BC::ask-attribute-value
   ?goal <- (goal (attribute ?g-name))
   (not (attribute (name ?g-name)))
   (not (rule (then ?g-name $?)))
   =>
   (retract ?goal)
   (printout t "What is the value of " ?g-name '
   (assert (attribute (name ?g-name)
                      (value (read)))))
```


Goal Satisfied?

```
(defrule BC::goal-satisfied
   (declare (salience 100))
   ?goal <- (goal (attribute ?g-name))
   (attribute (name ?g-name))
   =>
   (retract ?goal))
```


Rule Satisfied?

```
(defrule BC::rule-satisfied
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name)
              (value ?a-value))
   ?rule <- (rule (if ?a-name is ?a-value)
                  (then ?g-name is ?g-value))
   =>
   (retract ?rule)
   (assert (attribute (name ?g-name)
                      (value ?g-value))))
```

Rule Not Applicable

```
(defrule BC::remove-rule-no-match
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name) (value ?a-value))
   ?rule <- (rule (if ?a-name is ~?a-value)
                  (then ?g-name is ?g-value))
   =>
   (retract ?rule))
```


Partially Satisfied Rule

```
(defrule BC::modify-rule-match
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name) (value ?a-value))
   ?rule <- (rule (if ?a-name is ?a-value and
                      $?rest-if)
                  (then ?g-name is ?g-value))
   =>
   (retract ?rule)
   (modify ?rule (if $?rest-if)))
```


Evaluation of the Program

Initialization
: facts are put on the fact list
: module `BC` is activated
: agenda contains `start-BC`

Meal Selection
: user answers questions about the meal
: system reads answers, identifies appropriate BC rules

More Information
: if the system doesn't have enough information, the user is asked for further details on the meal

Rule Satisfied
: a rule for which all antecedents are satisfied is found

see also spreadsheet

Summary Wine Selection

knowledge representation
: `if-then` rules are represented as facts in CLIPS

backward chaining
: CLIPS rules specify the evaluation of `if-then` rules

goal
: goals and subgoals are generated
: `if-then` rules are applied or discarded towards the (sub-)goal

storage
: factual knowledge is stored via `deftemplates` in the source code

Example Monitoring

domain

process control: sensor values are read, and appropriate actions taken
a system consisting of several devices needs to be monitored
each device has one or more sensors that provide information about the device's status
a device may depend on another device for proper functioning

approach

for each sensors, important values are defined

- low guard line (LGL)
- low red line (LRL)
- high guard line (HGL)
- high red line (HRL)

readings between the guard lines are normal
readings between guard and red line are acceptable, but may be cause for concern

readings below or above the low/high red line indicates problems with the device, and must lead to a shut-down of the device

data-driven reasoning

forward chaining
certain values of the sensor data may trigger actions

Knowledge Representation

sensors

sensor name
affiliated device
sensor reading
state
guard- and red-line values

devices

name
status
device dependencies

Sensors

```
(deftemplate MAIN::sensor
   (slot name (type SYMBOL))
   (slot device (type SYMBOL))
   (slot raw-value (type SYMBOL NUMBER)
                 (allowed-symbols none)
                 (default none))
   (slot state (allowed-values low-red-line
                               low-guard-line
                               normal
                               high-red-line
                               high-guard-line)
               (default normal))
   (slot low-red-line (type NUMBER))
   (slot low-guard-line (type NUMBER))
   (slot high-guard-line (type NUMBER))
   (slot high-red-line (type NUMBER)))
```

Facts about Sensors

```
(deffacts MAIN::sensor-information
  (sensor (name S1) (device D1)
          (low-red-line 60) (low-guard-line 70)
          (high-guard-line 120) (high-red-line 1
  (sensor (name S2) (device D1)
          (low-red-line 20) (low-guard-line 40)
          (high-guard-line 160) (high-red-line 1
  (sensor (name S3) (device D2)
          (low-red-line 60) (low-guard-line 70)
          (high-guard-line 120) (high-red-line 1
  (sensor (name S4) (device D3)
          (low-red-line 60) (low-guard-line 70)
          (high-guard-line 120) (high-red-line 1
  (sensor (name S5) (device D4)
          (low-red-line 65) (low-guard-line 70)
          (high-guard-line 120) (high-red-line 1
  (sensor (name S6) (device D4)
          (low-red-line 110) (low-guard-line 115
          (high-guard-line 125) (high-red-line 1
```


Devices

```
(deftemplate MAIN::device
   (slot name (type SYMBOL))
   (slot status (allowed-values on off)))

(deffacts MAIN::device-information
  (device (name D1) (status on))
  (device (name D2) (status on))
  (device (name D3) (status on))
  (device (name D4) (status on)))
```


Execution Control

sensors are read in cycles
status information is printed out

```
(deffacts MAIN::cycle-start
  (data-source user)
  (cycle 0))

(defrule MAIN::Begin-Next-Cycle
   ?f <- (cycle ?current-cycle)
   =>
   (retract ?f)
   (assert (cycle (+ ?current-cycle 1)))
   (focus INPUT TRENDS WARNINGS))
```


Reading Sensor Values

from actual sensors in a real system
from the user, facts, or a file in a simulation

```
(defrule INPUT::Read-Sensor-Values-From-Sensors
  (data-source sensors)
  ?s <- (sensor (name ?name)
                (raw-value none)
                (device ?device))
  (device (name ?device) (status on))
  =>
  (modify ?s (raw-value (get-sensor-value ?name)
  )
```

Status Information

```
(defrule TRENDS::Normal-State
  ?s <- (sensor (raw-value ?raw-value&~none)
                (low-guard-line ?lgl)
                (high-guard-line ?hgl))
  (test (and (> ?raw-value ?lgl) (< ?raw-value ?
  =>
  (modify ?s (state normal) (raw-value none)))

(defrule TRENDS::High-Guard-Line-State
  ?s <- (sensor (raw-value ?raw-value&~none)
                (high-guard-line ?hgl)
                (high-red-line ?hrl))
  (test (and (>= ?raw-value ?hgl) (< ?raw-value
  =>
  (modify ?s (state high-guard-line) (raw-value
```

similar for other states

Identify Trends in Sensor Data

```
(deftemplate MAIN::sensor-trend
   (slot name)
   (slot state (default normal))
   (slot start (default 0))  ; first cycle in cu
   (slot end (default 0)) ; current state
   (slot shutdown-duration (default 3))) ; maxim

(deffacts MAIN::start-trends
  (sensor-trend (name S1) (shutdown-duration 3))
  (sensor-trend (name S2) (shutdown-duration 5))
  (sensor-trend (name S3) (shutdown-duration 4))
  (sensor-trend (name S4) (shutdown-duration 4))
  (sensor-trend (name S5) (shutdown-duration 4))
  (sensor-trend (name S6) (shutdown-duration 2))
```


Monitoring Trends

```
(defrule TRENDS::State-Has-Not-Changed
  (cycle ?time)
  ?trend <- (sensor-trend (name ?sensor)
                          (state ?state)
                          (end ?end-cycle&~?time
  (sensor (name ?sensor) (state ?state)
          (raw-value none))
  =>
  (modify ?trend (end ?time)))
```



```
(defrule TRENDS::State-Has-Changed
  (cycle ?time)
  ?trend <- (sensor-trend (name ?sensor)
                          (state ?state)
                          (end ?end-cycle&~?time
  (sensor (name ?sensor)
          (state ?new-state&~?state)
          (raw-value none))
  =>
  (modify ?trend (start ?time)
                 (end ?time)
                 (state ?new-state)))
```

Warnings

```
(defrule WARNINGS::Sensor-In-Guard-Region
  (cycle ?time)
  (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line | low-guard-
      (shutdown-duration ?length)
      (start ?start) (end ?end))
  (test (< (+ (- ?end ?start) 1) ?length))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in " ?state cr
```


Shutdown Red Region

```
(defrule WARNINGS::Shutdown-In-Red-Region
  (cycle ?time)
  (sensor-trend
     (name ?sensor)
     (state ?state&high-red-line | low-red-line)
  (sensor (name ?sensor) (device ?device))
  ?on <- (device (name ?device) (status on))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in " ?state cr
  (printout t "   Shutting down device " ?device
             crlf)
  (modify ?on (status off)))
```


Shutdown Guard Region

```
(defrule WARNINGS::Shutdown-In-Guard-Region
  (cycle ?time)
  (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line | low-guard-
      (shutdown-duration ?length)
      (start ?start) (end ?end))
  (test (>= (+ (- ?end ?start) 1)  ?length))
  (sensor (name ?sensor) (device ?device))
  ?on <- (device (name ?device) (status on))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in " ?state "
  (printout t "for " ?length " cycles "
             crlf)
  (printout t "   Shutting down device " ?device
             crlf)
  (modify ?on (status off)))
```


Summary Sensor Monitoring

knowledge representation
read sensor values, store them in facts
sensors are affiliated with devices

forward chaining
CLIPS rules are triggered by combinations of sensor values and state information

R1/XCON

one of the first commercially successful expert systems

application domain
configuration of minicomputer systems

- selection of components
- arrangement of components into modules and cases

approach
data-driven, forward chaining
consists of about 10,000 rules written in OPS5

results
quality of solutions similar to or better than human experts
roughly ten times faster (2 vs. 25 minutes)
estimated savings $25 million/year

System Configuration

complexity
tens or hundreds of components that can be arranged in a multitude of ways
in theory, an exponential problem
in practice many solutions "don't make sense", but there is still a substantial number of possibilities

components
important properties of individual components
stored in a data base

constraints

- *functional* constraints derived from the functions a component performs
  e.g. CPU, memory, I/O controller, disks, tapes
- *non-functional* constraints such as spatial arrangement, power consumption, ...

Knowledge Representation

configuration space
constructed incrementally by adding more and more components
the correctness of a solution often can only be assessed after it is fully configured
subtasks are identified to make the overall configuration space more manageable

component knowledge
retrieved from the external data base as needed

control knowledge
rules that govern the sequence in which subtasks are performed

constraint knowledge
rules that describe properties of partial configurations

Example Component

partial description of RK611* disk controller

```
RK611*
  Class: UniBus module
  Type: disk drive
  Supported: yes
  Priority Level: buffered NPR
  Transfer Rate: 212
  . . .
```

facts are retrieved from the data base and then stored in templates

Example Rule

Distribute-MB-Devices-3

If the most current active context is distributing Massbus devices
& there is a single port disk drive that has not been assigned to a Massbus
& there are no unassigned dual port disk drives
& the number of devices that each Massbus should support is known
& there is a Massbus that has been assigned at least one disk drive and that should support additional disk drives
& the type of cable needed to connect the disk drive to the previous device is known

Then assign the disk drive to the Massbus

rules incorporate expertise from configuration experts, assembly technicians, hardware designers, customer service, ...

Configuration task

and its subtasks

1. **check order**; identify and correct omissions, errors
2. **configure CPU**; arrange components in the CPU cabinet
3. **configure UniBus modules**; put modules into boxes, and boxes into expansion cabinets
4. **configure panels**; assign panels to cabinets and associate panels with modules
5. **generate floor plan**; group components and devices
6. **determine cabling**; select cable types and calculate distances between components

this set of subtasks and its ordering is based on expert experience with manual configurations

Reasoning

data-driven (forward chaining)
components are specified by the customer/sales person
identify a configuration that combines the selected components into a functioning system

pattern matching
activates appropriate rules for particular situations

execution control
a substantial portion of the rules are used to determine what to do next
groups of rules are arranged into subtasks

Performance Evaluation

notoriously difficult for expert systems

evaluation criteria
usually very difficult to define
sometimes comparison with human experts is used

empirical evaluation
Does the system perform the task satisfactorily?
Are the users/customers reasonably happy with it?

benefits
faster, fewer errors, better availability, preservation of knowledge, distribution of knowledge, ...
often based on estimates

Development of R1/XCON

R1 prototype
the initial prototype was developed by Carnegie Mellon University for DEC

XCON commercial system
used for the configuration of various minicomputer system families
first VAX 11/780, then VAX 11/750, then other systems

reimplementation
more systematic approach to the description of control knowledge
clean-up of the knowledge base
performance improvements

Extension of R1/XCON

addition of new knowledge

wider class of data
additional computer system families
new components

refined subtasks
more detailed descriptions of subtasks
revised descriptions for performance or systematicity reasons

extended task definition
configuration of “clusters” (tightly interconnected multiple CPUs)

related system XSEL
tool for sales support

Summary R1/XCON

commercial success
after initial reservations within the company, the system was fully accepted and integrated into the company's operation
widely cited as one of the first commercial expert systems

domain-specific control knowledge
the availability of enough knowledge about what to do next was critical for the performance and eventual success of the system

suitability of rule-based systems
appropriate vehicle for the encoding of expert knowledge
subject to a good selection of application domain and task