Università degli Studi di Bari
Dipartimento di Informatica

LACAM
Machine Learning

CLIPS: Knowledge Representation and Problem Solving

Antonio Vergari

May 5, 2016

Knowledge Representation

A simple diagnostic classifier

Getting back to the previous example, the knowledge we wanted to embed could be expressed in such a few rules:

I. **If** the skin and eyes are yellowish, **then** there is *icterus* (partial diagnosis)

II. **If** the eyes are yellowish but not the skin, **then** there is *scleral icterus* (partial diagnosis)

III. **If** scleral icterus is present, there is no fever, and the patient's stressed or without food, **then** *Gilbert's syndrome* can be diagnosed

IV. **If** icterus is present, there is fever, the patient is young and tired, dyspepsia has been diagnosed and the liver is enlarged, **then** *Acute viral hepatitis* can be diagnosed

V. **If** there is icterus, fever and the patient is not young but has recurrent pains and cholecyst pains, **then** it is *cholecystitis*

VI. **If** there is icterus, there is no fever and the patient is not young and abuses alchool, the liver is enlarged and the spleen is enlarged as well, **then** it is *alchoolic cirrhosis*

A very simple conceptualization

The minimal information needed to represent a symptom as a fact is its name and its presence. In this way we are limiting possible extensions (like user profiling), plus each precondition is treated as a symptom.

```
(deftemplate symptom
    (slot name (type SYMBOL))
    (slot observed (default FALSE)))
```

Thus a diagnostic rule could take this linear form:

```
(defrule alchoolic—cirrhosis
    (symptom (name icterus) (observed TRUE))
    (symptom (name fever) (observed FALSE))
    (symptom (name young) (observed FALSE))
    (symptom (name alcohol—abuse) (observed TRUE))
    (symptom (name enlarged—liver) (observed TRUE))
    (symptom (name enlarged—spleen) (observed TRUE))
    =>
    (assert (diagnosis "Alchoolic cirrhosis")))
```

In the case of partial diagnosis we will assert symptom facts and not diagnosis (this is a limitation as well).

A more complex conceptualization

In the previous example we are assuming our expert system user to deal with a patient at a time. At each run, only *his* symptoms are stored as facts in memory.
However, suppose that your expert system shall do inference on *different* patients' symptoms. In this case, it would be necessary to represent each patient as a fact, maybe with a template:

```
(deftemplate patient
    (slot patient-name (type SYMBOL))
    (multislot symptoms (type SYMBOL)))
```

Now, consider the problem of representing one patient symptom history instead. We would need a way to trace the previous diagnosis and their dates:

```
(deftemplate symptom-diagnosis
    (slot name (type SYMBOL))
    (slot date (type STRING)))
```

in the end, the fact representation chosen highly depends on the conceptualization followed and the *requirements* analysed...

Modeling domains, experts, users

Some advices

Designing an expert system means to express an expert knowledge in terms of rules (and facts). Before designing it, be sure to well define all the requirements.

First determine a domain. Stick to some field you know. And for which encoding an expert knowledge in a software can be meaningful.

Then select which expert figure you are going to extract knowledge from. Find . You can be your own expert as well.

Then model the system user. Is he an expert? a novice? When shall he consult the expert system? Keep in mind *interactivity*.

You could use the requirements analysis techniques you studied up to now: interviews, questionnaires,...
Properly write them down (a documentation is required), and *simulate* a user-expert system interaction.

Rule Hierarchy I

To exend the first simple formalism one can conceptualize what some symptoms really mean and how to observe them in more depth. This can be done from the rule point of view as well.

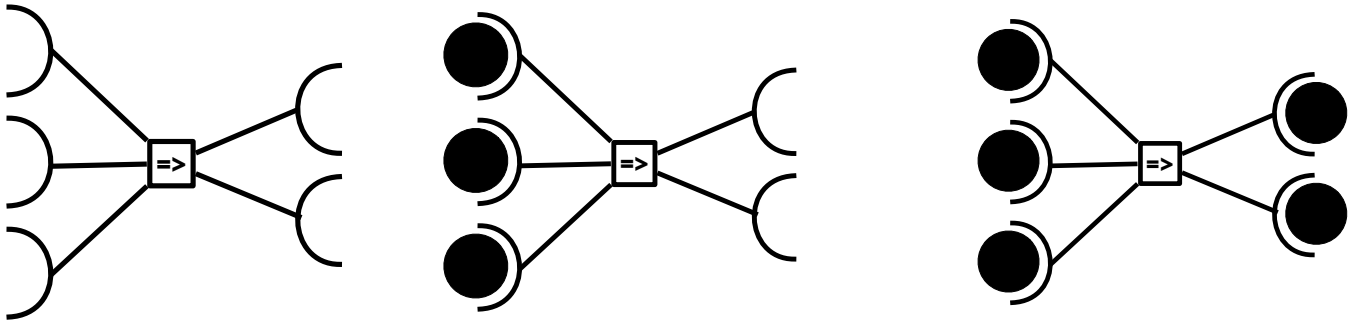
For instance, what does it mean for a patient to be young? Or to have the fever? We can add additional rules acting as *mini-classifiers*.

If the patient's age is less than 25 years, **then** he's young, **otherwise if** it is still less than 50, **then** it is an adult, **else** an elder.
If the patient's temperature is higher than 37.1 Celsius degrees, **then** he has a fever.

We are clearly defining different stages in the diagnosis process, refining available information and collecting missing information. At the same time the interaction (via questions) shall be more focused and complex.
One has to carefully design the set of rules forming his KB.

Rule Hierarchy II

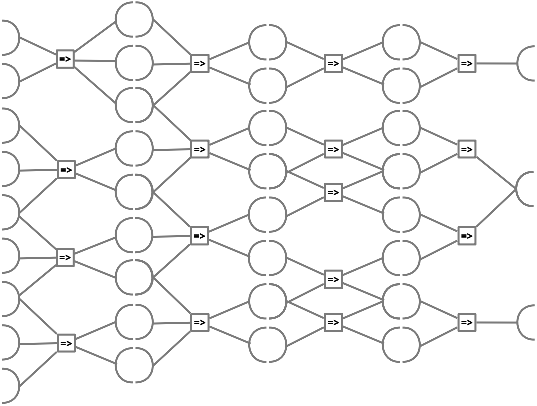
To help you design a complex and consistent rule KB, writing them down is a good starting point.
A rule can be thought of a joint, linking some facts (matching its LHS) to some newly asserted facts (from its RHS).
Multiple rules can share some LHS patterns as well as some RHS slots. Connecting all the rules on paper in a *rule graph* gives a hint of the system inner workings.



A *non-linear* expert system shall activate different rules to different assertions. That is, different paths in the graphs are activated (maybe starting from the same point).

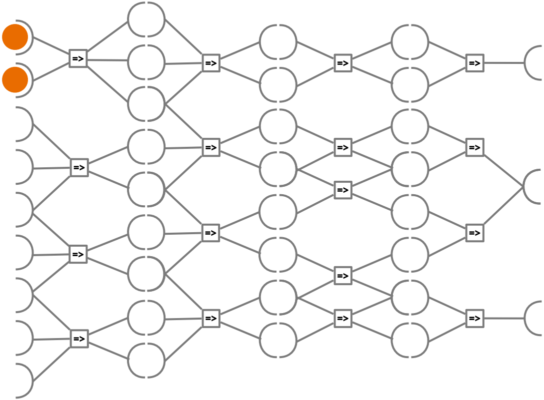
Rule Hierarchy III

Imagine a rule hierarchy written as a graph with three distinct possible outputs (classification results).



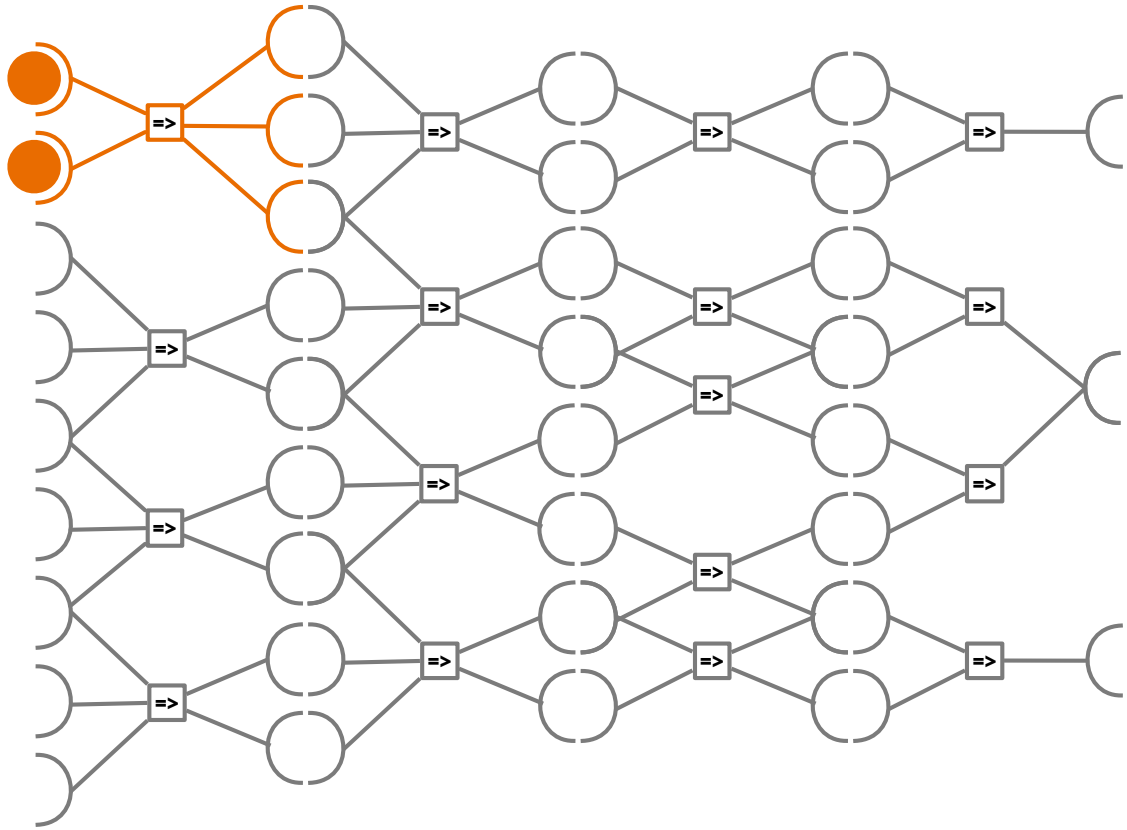
Rule Hierarchy IV

The facts asserted at the beginning determine the *root* of the activation path.



Rule Hierarchy V

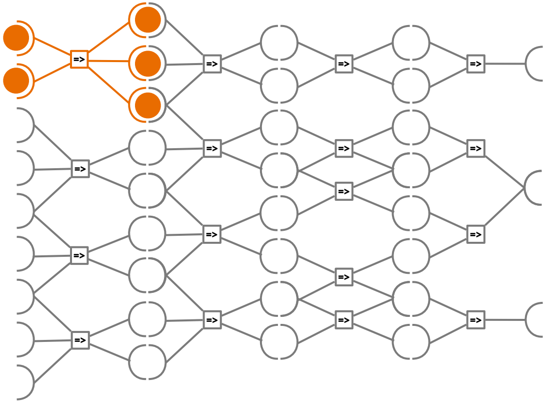
When a LHS side of a rule gets satisfied we traverse it (we color it).



It means that the rule gets instantiated, goes to the agenda, then...

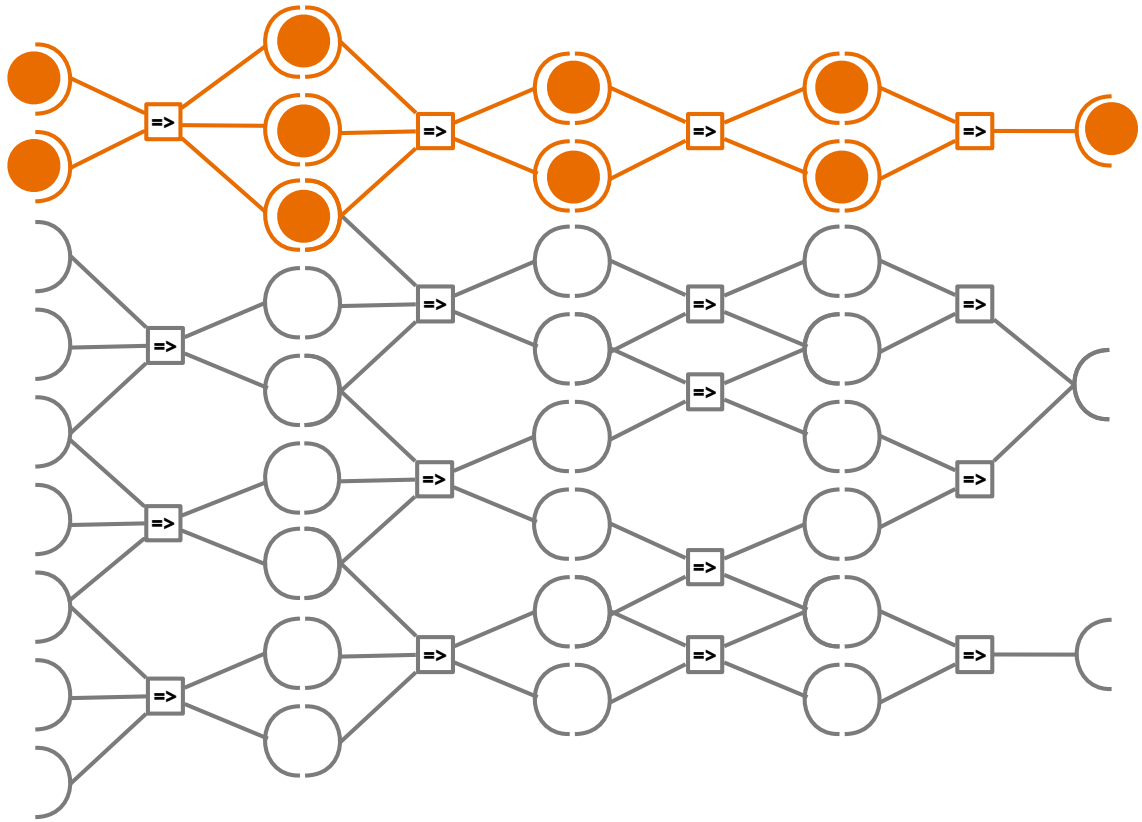
Rule Hierarchy VI

...when fired it will probably alter the WM, allowing other branches in the path...



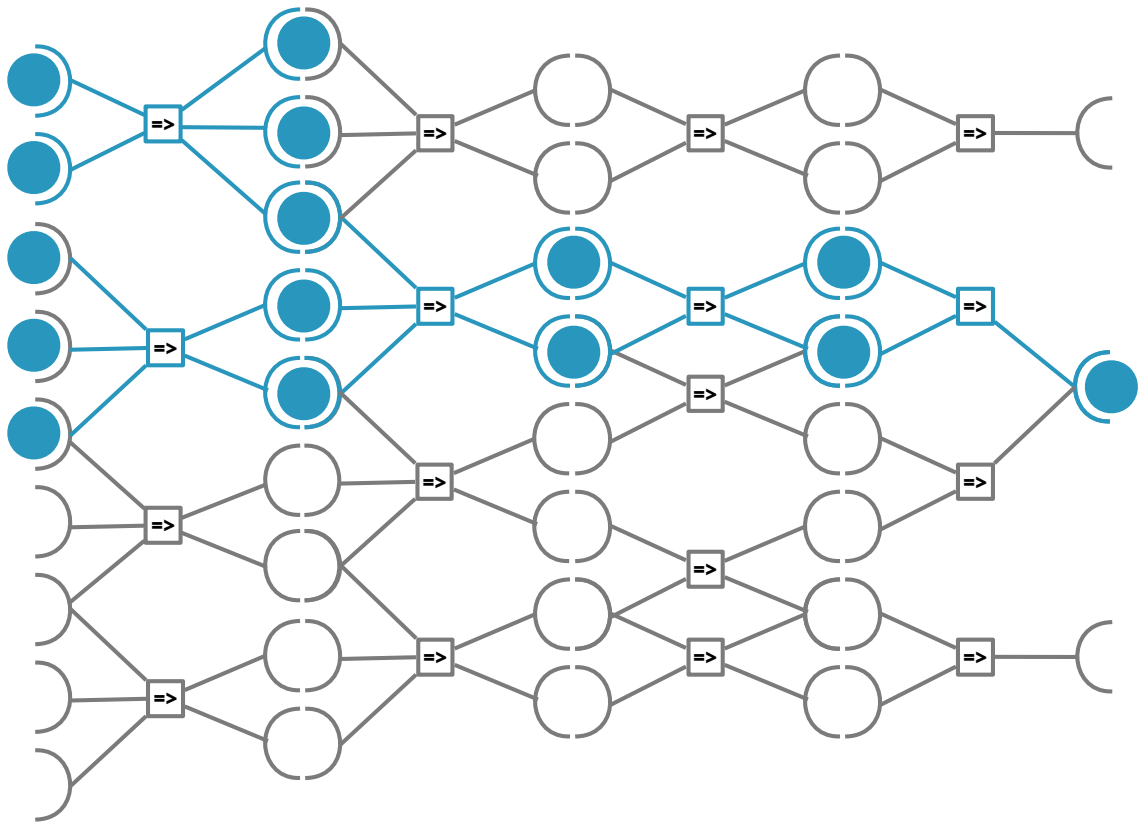
Rule Hierarchy VII

a full path is traversed when no more rules can get activated.



The complexity and non linearity can be measured by the number of alternative paths...

Rule Hierarchy VIII



Now consider this as a single module. Repeat the process at a higher abstraction level.

Randomizing order

Our naive conversation behaviour could be more banal if we could start the conversation from any possible rule.

We can achieve this by imposing a random conflict resolution strategy (all rules to ask questions shall have the same priority):

```
(clear)
(set—strategy random)
(load icterus.clp)
(reset)
(run)
```

However, please note that a real diagnostic interview can start from every point, *but then it is then guided by the partial diagnosed diseases and symptoms*. In order to do this we have to introduce meta-rules…

If there is a symptom that can be asked that is characterizing a disease we could diagnose, **then** ask about that symptom

Revising Asserted Knowledge

If we assume each asserted symptom to be erroneously observed, we could provide a routine that lets you change the truth value for them all, calling inference again on the new WM.

```
Would you like to revise the diagnosis? (yes/y/no/n): yes
Would you like to change some observed symptom?
(1) recurrent—pain: TRUE
(2) tired: TRUE
(3) enlarged—liver: TRUE
(4) yellowish—skin: TRUE
...
(11) enlarged—spleen: TRUE
(12) alcohol—abuse: TRUE
(13) cholecyst—pain: TRUE
(14) fever: TRUE
Enter the symptom number or 'e' to return or 'h' to stop:
```

Suppose you are setting yellowish-skin to FALSE, will the scleral icterus rule fire? *what if scleral icterus has already been asserted before*?

Revising Asserted Knowledge

A simple, manual approach:

```
1   (deffunction get-all-facts-by-names ($?template-names)
2       (bind ?facts (create$))
3       (progn$ (?f (get-fact-list)) ;; this is a foreach
4           (if (member$ (fact-relation ?f) $?template-names)
5               then (bind ?facts (create$ ?facts ?f)))) ?facts)
6
7   (deffunction change-symptom-by-index (?index)
8       (bind ?f (nth ?index (get-all-facts-by-names symptom)))
9       (modify ?f (observed (not (fact-slot-value ?f observed)))))
10
11  (deffunction ask-to-change-symptom (?question)
12      (printout t "Would you like to change some observed symptom?" crlf)
13      (print-all-symptoms-status)
14      (bind ?n (length$ (get-all-facts-by-names symptom)))
15      (bind ?response (ask-question ?question (create$ (gen-int-list ?n) e h)))
16      (switch ?response  (case h then (printout t "Halt" crlf) (halt))
17                         (case e then (return))
18                         (default (change-symptom-by-index ?response)
19      (ask-to-change-symptom ?question))))
```

Searching

Search State Representation

We can represent the inference problem as the search in the space of possible diagnosed states.
To implement a problem solver in CLIPS we have to model, for each problem[1]:

states	that form our search space (e.g. problem configurations)
transitions	that allow to jump from one state to its neighbors, the next states (e.g. rules in a game)
constraints	eventually imposed on states (e.g. no repeated or illegal states) and to check for a state to be a solution
search strategy	that controls which transition to apply and which state to move on at each time
explored tree	representing the states explored so far

[1] How could we model a *general problem solver*?

A tentative implementation

The simplest implementation we can try relies on the direct exploitation of the constructs that CLIPS offers, using in a non-transparent way its inference cycle

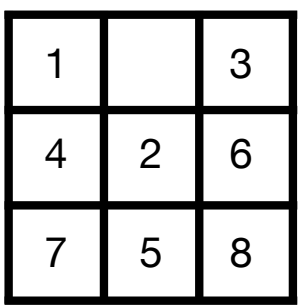
To be more precise we would have:

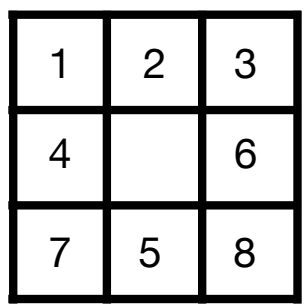
states	as facts in the current configuration of the WM.
transitions	encoded directly as CLIPS rules with the same priority
constraints	as constrained pattern in the LHS of transitions (e.g. the **not** logical operator). Some rules can be explicitly used for solution states
search strategy	directly relying on CLIPS conflict resolution strategy
explored tree	building it by some other facts as we proceed

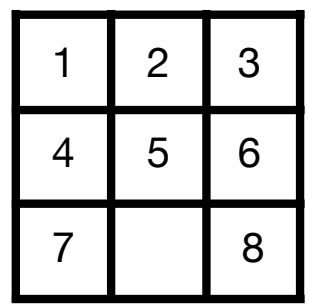
Even though it lacks flexibility and constraints us in many ways, we will explore a use case by implementing a very simple game before moving forward.

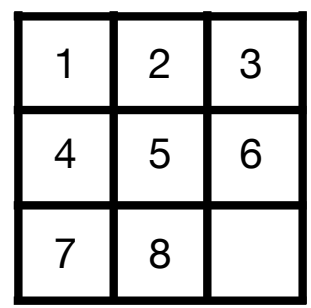
8-Puzzle

The 8-puzzle game is one of the earliest games exploited in AI. Numbered tiles, from 1 to 8, are placed on a 3×3 grid.
Starting from an initial grid configuration the objective is to reach a final state in which all tiles are ordered numerically with the last one left empty.
The allowed moves consist in exchange the position of the the empty tile with one of its (at most) four neighbors.









At this point of the course you should be pretty familiar with this game.

8-Puzzle: Facts and Rules

Concerning states we can represent a grid configuration with a single fact. The information to be stored concerns *the numbered positions* and *the actual value* in them. For instance, we can use a special number like -1 to represent the empty tile value.

Even if inefficiently we can take advantage of the WM by simply asserting and never retracting (remind assertion and refraction properties).

We can have each rule modeling a single possible empty cell movement (24 rules total), something like:

If the grid has X_1 in the 1st position, -1 in the 2nd, X_3 in the third, X_4 in the fourth, X_5 in the fifth, ..., **then** the new grid will be: X_1 in the 1st position, X_5 in the 2nd, X_3 in the third, X_4 in the fourth, -1 in the fifth,...

We could reduce the number of rules by exploiting functional abstraction.

8-Puzzle: Constraints and Strategy

Avoiding loops is done by the WM use we are doing and by enforcing to match only rules leading to unseen configurations.
Refraction helps, but we need to avoid even to trigger rules that would assert and already asserted state.
One possible solution is to put a constraint as a pattern in the LHSs:

If the grid has X_1 in the 1st position, -1 in the 2nd, X_3 in the third, X_4 in the fourth, X_5 in the fifth, ..., **AND** there is *not* X_1 in the 1st position, X_5 in the 2nd, X_3 in the third, X_4 in the fourth, -1 in the fifth..., **then** the new grid will be: X_1 in the 1st position, X_5 in the 2nd, X_3 in the third, X_4 in the fourth, -1 in the fifth,...

The search strategy can be demanded to CLIPS by enforcing all rules to have the same salience and by employing the breadth or depth conflict resolution strategy.

```
(set-strategy depth)
```

What is, for this particular problem, the best strategy to employ? why?

8-Puzzle: Constraints and Search Tree

As a last constraint we need rules to check for a state to be the final one (solution). This is simply done like in the previous fashion.

To build a search tree we can store its edge information as facts containing information about the two configurations of the world, the parent and child grids.

After a solution has been found we have to crawl back from that state to the root. To do this we need a simple rule that searches for a state matching for the parent attribute in our edge facts

Exercise

Implement such a problem solver for the 8-puzzle.

Hint 1:
While programming apply a debugging strategy, inspect the WM after each activated rule execution. Moreover, keep track of strategy resolution by printing to stout the state that activates a rule and the one that the rule asserts.

Hint 2:
To set the strategy and maybe constants use another file to by loaded with the batch function.

Hint 3:
For the tree representation you can use something like this for edges, containing simply the fact addresses (pointers):

```
(deftemplate move
    (slot parent (type FACT-ADDRESS) (default ?NONE))
    (slot next (type FACT-ADDRESS) (default ?NONE)))
```

8-Puzzle: a possible solution I

A very simple (and verbose) representation for states as facts:

```
(deftemplate 8-puzzle
    (slot one (type INTEGER) (default ?NONE))
    (slot two (type INTEGER) (default ?NONE))
    (slot three (type INTEGER) (default ?NONE))
    (slot four (type INTEGER) (default ?NONE))
    (slot five (type INTEGER) (default ?NONE))
    (slot six (type INTEGER) (default ?NONE))
    (slot seven (type INTEGER) (default ?NONE))
    (slot eight (type INTEGER) (default ?NONE))
    (slot nine (type INTEGER) (default ?NONE)))

(deftemplate move
    (slot parent (type FACT-ADDRESS) (default ?NONE))
    (slot next (type FACT-ADDRESS) (default ?NONE)))

(deffacts initial-board
   (8-puzzle (one 1) (two 2) (three 3) (four 4) (five -1) (six 6)
             (seven 7) (eight 5) (nine 8)))
```

8-Puzzle: a possible solution II

An equally simple (and verbose) formulation for the rules:

```
(defrule from-one-to-two
    ?s <-(8-puzzle (one -1) (two ?T) (three ?H)
                   (four ?F) (five ?I) (six ?S)
                   (seven ?E) (eight ?G) (nine ?N))
    (not (8-puzzle (one ?T) (two -1) (three ?H)
                   (four ?F) (five ?I)  (six ?S)
                   (seven ?E) (eight ?G) (nine ?N)))
    =>
    (bind ?n (assert (8-puzzle (one ?T) (two -1) (three ?H)
                               (four ?F) (five ?I) (six ?S)
                               (seven ?E) (eight ?G) (nine ?N))))
    (print-boards ?s ?n)
    (printout t "Moved from one to two" crlf)
    (assert (move (parent ?s) (next ?n)))
    (yes-or-no-halt))
```

The check to not activate a rule leading to an already seen configuration is embedded as a **not** CE in the LHS.

8-Puzzle: a possible solution III

To detect if we have reached a final configuration, we can employ a fixed LHS rule:

```
(defrule found-solution
    (declare (salience ?*highest-priority*))
    ?s <- (8-puzzle (one 1) (two 2) (three 3)
                    (four 4) (five 5) (six 6)
                    (seven 7) (eight 8) (nine -1))
    (move (parent ?p) (next ?s))
    =>
    (print-boards ?s ?s)
    (assert (moves ?p ?s))
    (printout t "FOUND solution" crlf))
```

While, to govern the search strategy we have to set the conflict resolution strategy in CLIPS, which can be done in a batch file:

```
(clear)
(set-strategy breadth)
(load 8_puzzle.clp)
(reset)
(run)
```

8-Puzzle: a possible solution IV

To retrieve the path leading to the final correct configuration, we can use also other rules, collecting back the **move** facts a create a multifield **moves** fact which (we can print at the end).

```
(defrule get-solution-path
    (declare (salience ?*highest-priority*))
    ?m <- (moves ?last-move $?other-moves)
    ?k <- (move (parent ?previous) (next ?last-move))
    =>
    (retract ?m ?k)
    (assert (moves ?previous ?last-move $?other-moves)))

(defrule print-solution-path
    (declare (salience ?*highest-priority*))
    ?m <- (moves ?last-move $?other-moves)
    (not (move (parent ?previous) (next ?last-move))) ;; this is the root
    =>
    (printout t "Found a solution" crlf)
    (print-board-moves ?m)
    (halt))
```

Pros&Cons

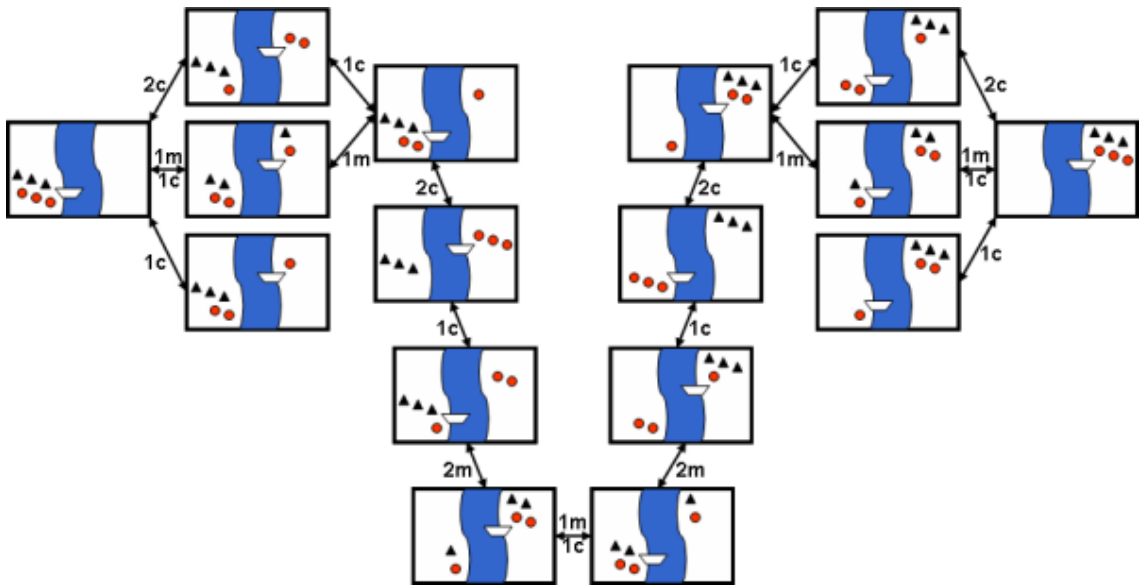
What ***we gain for free*** is CLIPS inference routines and constructs, i.e. we do not have to implement a breadth first search or the neighbor space generation by ourselves (we just describe how they are generated via transition rules).

However, *we cannot explore the search space in a more sophisticated way*.
If we use CLIPS conflict resolution strategy for free, we cannot assign different priorities to rules. Neither it is possible to implement other search techniques like best-first search, A^*, and so on. We could implement the Manatthan distance heuristic score as a function, but we would have no way to embed its evaluation on the pairs of states.

A higher level representation for transition rules and states is needed.

Cannibals and Missionaries (CaM)

We have an initial equal number N of cannibals and missionaries on one shore of a river. There is one boat that can transport at most k people at a time from one side to the other. We start with $2N$ people on one shore and we have to move all to the other. considering that, at each time, on both shore and on the boat, the number of missionaries shall be equal or greater than that of cannibals.
Supposing $N = 3$ and $k = 2$ a possible state search space is:



CaM: State formulation I

To be able to govern more the search, we need to embed in the state conceptualization some references to the current search advancement (e.g. depth, references to the parent state,...)[2]:

```
(deftemplate MAIN::status
    (slot search-depth (type INTEGER) (range 1 ?VARIABLE))
    (slot parent (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
    (slot shore-1-missionaries (type INTEGER) (range 0 ?VARIABLE))
    (slot shore-1-cannibals (type INTEGER) (range 0 ?VARIABLE))
    (slot shore-2-missionaries (type INTEGER) (range 0 ?VARIABLE))
    (slot shore-2-cannibals (type INTEGER) (range 0 ?VARIABLE))
    (slot boat-location (type SYMBOL) (allowed-values shore-1 shore-2))
    (slot last-move (type STRING)))
```

as well as taking into account the boat position and number of missionaries and cannibals on both shores.

[2] This code is taken from Giarratano's and Riley's book

Modular Programming

In CLIPS it is possible to define different modules, each one with its deftemplated facts, rules, function, global variables (everything that can be build with a def construct).
To define a module, use the construct defmodule.
To specify the name of a module as a ***namespace***, one shall use the scope operator :: (like in C++)

One module can import from others the constructs it needs to operate. Image we use a MAIN module to represent the state and the rules to operate on them, we could use a CONSTRAINT module for rules checking illegal states and a module SOLUTION for getting the path from the root of the search tree to the final configuration state. To import the status template in the CONSTRAINT module we could write:

```
(defmodule CONSTRAINTS
   (import MAIN deftemplate status))
```

The switch among modules can be governed by the focus property (see rules later).

Focus-Stack

In order to manage the modules of a program CLIPS maintains a stack of module references, the focus-stack. The module at the top of the focus-stack is active; all the others are dormant.
The rules in a module can contain actions that modify the contents of the focus-stack and thus force the activation of other modules.

If the rule of a module is declared auto-focus, by using the special pattern

```
(declare (auto—focus TRUE))
```

then the module will be automatically activated and inserted at the top of the focus-stack when the rule becomes instantiated.
Note that this creates an effect similar to that of putting a very high priority on that rule.

This way not only we can *divide* rules into modules (doing different tasks), but we can also *govern* their activation (by determining their modules activation).

CaM: State formulation II

We can represent the initial status as shown in the previous image with a deffacts construct.

```
(deffacts MAIN::initial-positions
     (status (search-depth 1)
             (parent no-parent)
             (shore-1-missionaries 3)
             (shore-2-missionaries 0)
             (shore-1-cannibals 3)
             (shore-2-cannibals 0)
             (boat-location shore-1)
             (last-move "No move.")))

(deffacts boat-information
    (boat-can-hold 2))
```

CaM: Generate & Test I

To search the state space we can adopt a ***generate & test*** approach: that is, we generate (assert) all the possible *syntactical* states by moving one or more cannibals and/or missionaries from one shore to the other. After the states have been generated, we can use other rules to prune (retract) those that are illegal, for instance statuses where the number of cannibals exceeds that of missionaries:

```
(defrule CONSTRAINTS::cannibals-eat-missionaries
      (declare (auto-focus TRUE))
      ?node <- (status (shore-1-missionaries ?s1m)
      (shore-1-cannibals ?s1c)
      (shore-2-missionaries ?s2m)
      (shore-2-cannibals ?s2c))
      (test (or (and (> ?s2c ?s2m) (<> ?s2m 0))
                (and (> ?s1c ?s1m) (<> ?s1m 0))))
      =>
      (retract ?node))
```

The property auto-focus tells CLIPS to apply inference in the referenced module as soon as it is possible to do so. In this case if this rule gets activated by a status, than the focus of execution will switch to the CONSTRAINTS module.

CaM: Generate & Test II

To generate all the states we can use just two rules that, for each shore, will calculate all the possible movements. (See the attached script for the full code)

```
(defrule shore-1-move
    ?node <- (status (search-depth ?num)
                     (boat-location shore-1)
                     (shore-1-missionaries ?s1m)
                     (shore-1-cannibals ?s1c)
                     (shore-2-missionaries ?s2m)
                     (shore-2-cannibals ?s2c))
                     (boat-can-hold ?limit)
    =>
    (bind ?max-missionaries (min ?s1m ?limit))
    (loop-for-count (?missionaries 0 ?max-missionaries)
                    ...
                    (loop-for-count (?cannibals ?min-cannibals ?max-cannibals)
                                    (duplicate ?node (search-depth =(+ 1 ?num))
                                                     (parent ?node)
                                                     ...))))
```

CaM: exercise

Given the incomplete script for this problem, complete the CONSTRAINTS module by implementing the following rules:

- A rule to consider illegal all the states that have been generated over a certain depth limit
- A rule to check whether a state has already been visited before (hint, given this formulation, such a state would have all attributes equal to an already asserted state, for the exception of...)

Implement a SOLUTIONS module that, like in the icterus case, is able to retrieve the paths from the root to a solution (final state). Hint: let the last slot of a status be its printed proxy.
Keep in mind to import all the constructs that you need from other modules.

Extend the module to find all the possible solutions.