CLIPS
A Rule-Based Programming Language

CLIPS stands for "C Language Integrated Production System" and has been developed at NASA, Software Technology Branch, Lyndon B. Johnson Space Center. The CLIPS 6.1 version that it used here is from 1998. CLIPS is a rule-based language used for symbolic computation, in particular as an expert system shell. Although CLIPS can be seen as a logical reasoning system based on rules that encode logical cause-effect implications, here it is seen from the perspective of associative processing and declarative programming. From this point of view, the processing in CLIPS is similar to that of a Markov Algorithmic Machine, except from a series of pragmatic extensions thought for programming convenience.

The simplest way to observe the main points of programming in CLIPS and to discuss the main concepts that underline the associative programming paradigm is to solve a problem.

The cheapest paths between two given vertices of a directed graph

Consider a directed graph **G=(V,E)** with vertices **V** and edges **E** and **cost:E→R+** a weighting function that associates a real positive cost to each edge from **E**. Let **start** and **stop** be two vertices from **V**. Find the shortest paths from **start** to **stop** in the graph.

One conventional solution is to use the Dijkstra's algorithm. However, there is a possible declarative solution that is based on the following statements:

1. The cheapest paths between **start** and **stop** are the *useful* paths between **start** and **stop** when there are no other useful paths that extend the existing useful paths.

2. Initially, when starting to solve the problem, there is a single path that contains the **start** vertex only and has the cost **0**.

3. A path **x...y,z** extends a path **x...y** if **(y,z)∈E** and **z** is not in the path **x...y**. The cost of **x...y,z** is the cost of **x...y** augmented by the cost of the edge **(y,z)**.

4. A path between two vertices **x** and **y** is *useful* if there are no other cheaper paths between **x** and **y**. There is no point in preserving non-useful paths for further processing. A non-useful path **x...y** is discarded immediately after a better path **x...y** is discovered.

The statements above are merely declarative. They state what is meant by the cheapest paths between two vertices of a directed graph and what should be done to solve the problem: start with an initial path, extend the current paths as long as it is possible do so, and to get rid of the non-useful paths. In the end, all paths between **start** and **stop** are the cheapest such paths.

Let us translate in CLIPS the above story. First, there is the problem of representation: how the graph and the paths in the graph are represented.

Data representation

In CLIPS the problem universe is represented symbolically by *facts*. A fact is similar to a symbol from the DR of MAM. It is a statement about the significant attributes of an explicit or implied problem object. For example, a graph can be fully described by its edges. Each edge specifies the connected vertices and the connection cost. The goal of the problem is also a fact. It specifies the **start** and the **stop** vertices of the cheapest paths to be computed.

For example, the cheapest path problem for the directed graph in figure 1 is represented in CLIPS by the following facts:

```
(edge (from a) (to b) (cost 1))
(edge (from a) (to e) (cost 9))
(edge (from b) (to c) (cost 2))
(edge (from b) (to d) (cost 1))
(edge (from c) (to d) (cost 1))
(edge (from c) (to e) (cost 4))
(edge (from c) (to a) (cost 2))
(edge (from d) (to e) (cost 2))
(edge (from a) (to c) (cost 1))

(cheapest_paths (start a) (stop e))
```

The facts above are like values of a data type. Here it is more appropriate to talk of instances of a set of *templates*. The fact **`(edge (from a) (to c) (cost 1))`** is an instance of the template **`(edge (from ...) (to ...) (cost ...))`**.

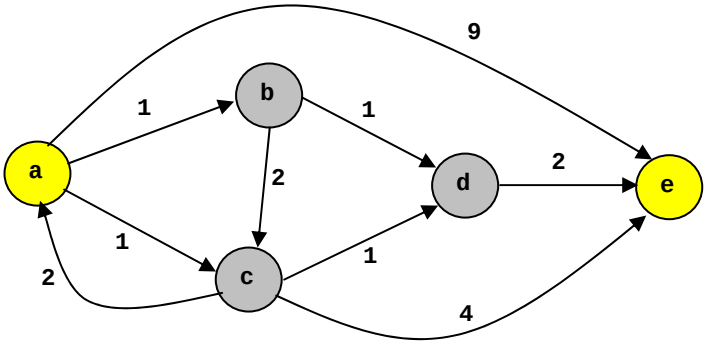

Figure 1. A weighted directed graph

A template defines a structure similar to a **`record`** in Pascal or to a **`struct`** in C, structure that contains field selectors (called here slot selectors). For example, the definition of the template **`edge`** is

`(deftemplate edge (slot from) (slot to) (slot cost))`.

Note that the type of the slot values is not specified. A slot can have any value (symbol, number, string, etc.). The validity of the operations performed with these values is checked at run time. CLIPS is a *dynamically typed* language. The template declarations from the cheapest paths program are:

```
(deftemplate edge (slot from) (slot to) (slot cost))
(deftemplate cheapest_paths (slot start) (slot stop))
(deftemplate path (multislot nodes) (slot cost))
```

Note that the template **`path`** has a **`multislot`** called **`nodes`**. The value of a **`multislot`** can be a sequence, eventually empty, of values (possibly of different types). The value of a **`slot`** is a single value. For example, two valid instances of the template **`path`** are:

```
(path (nodes a b d e) (cost  4))
(path (nodes) (cost 0))
```

There are special templates that need not be declared. Templates that reduce themselves to a single **`multislot`** are called implied templates and need not be declared. For example, the fact:

`(PC_manufacturers Fujitsu IBM Compaq Dell HP)`

corresponds to the implied template **`(PC_manufacturers ...)`**. It does not need be explicitly declared.

The facts that exist during the execution of a CLIPS program form the *factual knowledge base* of the program. The knowledge base varies as new facts are added or retracted while the program is running. Notice that, apart from the DR of MAM, there is no explicit[1] ordering of facts in the knowledge base.

Rules

The statements of the algorithm above are coded in CLIPS as symbolic structures called rules. A rule is somewhat similar to the rule of a Markov algorithm and has the format:

```
(defrule rule_name
   pattern1 ... patternn   similar to the MA identification_pattern
   =>
   action1 ... actionm)    similar to the MA substitution_pattern
```

The patterns are parameterized facts (facts that contain variables that designate generic slot values), eventually incomplete, that correspond to the templates of the program. The patterns of a rule control the applicability of the rule as the *identification_pattern* of a rule does in the Markov Algorithmic Machine. There must exist facts that match all the patterns of a rule for the rule to be applicable. In the process, the rule variables are bound to values taken from the matching facts. For example, the rule

```
(defrule initial_path
   (cheapest_paths (start ?x))
   =>
   (assert (path (nodes ?x) (cost 0))))
```

encodes the statement " Initially, when starting to solve the problem, there is a single path that contains the **start** vertex only and has the cost **0**". The rule is applicable only if there is at least an instance of the template **cheapest_paths** in the factual knowledge base of the program. Assume, the fact is

```
(cheapest_paths (start a) (stop e))
```

Then, the rule **initial_path** becomes applicable, and the variable **?x** is bound to the value **a**, the symbol of the **start** vertex. Note that the pattern **(cheapest_paths (start ?x))** is incomplete and, moreover, it is parameterized. Indeed, for building the root path there is no need to know the **stop** vertex. In addition, the name of the **start** node need not be known and is represented by the variable **?x**.

If there are several patterns, then the pattern matching is sequential. In this way the variables that occur in $\mathbf{pattern_k}$ can be used in $\mathbf{pattern_j}$, **j>k**. However, the scope of the variables is restricted to the rule within which they occur.
An applicable rule can be applied. If it is applied, the actions $\mathbf{action_1 \ ... action_m}$ are executed in sequence. For the rule **initial_path** a new fact is constructed and added to the factual knowledge base of the program. The other statements of the cheapest path algorithm can be encoded in a similar way. We have to state what are the patterns that make each rule applicable and what actions are executed if the rule is applied.

- The statement “A path **x...y,z** extends a path **x...y** if **(y,z)∈E** and **z** is not in the path **x...y**. The cost of **x...y,z** is the cost of **x...y** augmented by the cost of the edge **(y,z)**." is encoded as the rule **extended_path**.

[1] The facts can be ordered implicitly by force of the time stamp of each fact (the time when the fact is constructed). Using the time stamps, the control engine of CLIPS may decide what applicable rules are effectively applied.

```
(defrule extended_path
   (path (nodes $?n ?y) (cost ?w))
   (edge (from ?y) (to ?z & ~?y & :(not (member ?z $?n))) (cost ?we))
 =>
(assert (path (nodes $?n ?y ?z) (cost (+ ?w ?we)))))
```

The first pattern of the rule uses the **multifield** variable **$?n** that can be bound to a sequence – possibly empty – of values. The pattern stands for a path which has some vertices, designated collectively as **$?n**, and a terminal vertex designated by **?y**. The cost of the path is **?w**.

The second pattern of the rule states clearly that the edge selected to extend the path must link the vertex **?y** to a vertex **?z** not member of the path. The restriction "not member of the path" is encoded as a series of predicates that follow the variable **?z** and are separated by the reserved symbol **&**. The reading of **?z & ~?y & :(not (member ?z $?n))** is: the value of **?z** must be different from the value of **?y** and must not be member in the list of values bound to **$?n**.

The action **(assert (path (nodes $?n ?x ?y) (cost (+ ?w ?we))))** builds a new path and adds it to the facts of the problem factual knowledge base.

The syntax of the above rule may seem intricate. However, the rule could be written more explicitly, although not so compact, in the format:

```
(defrule extended_path
      (path (nodes $?n ?y) (cost ?w))
      (edge (from ?y) (to ?z) (cost ?we))
      (test (and (neq ?z ?y)
                 (not (member ?z $?n))))
      =>
      (assert (path (nodes $?n ?y ?z) (cost (+ ?w ?we)))))
```

Here the pattern **(test *predicate*)** succeeds if the ***predicate*** evaluates to true. There are many built-in CLIPS functions that can be used to restrict the values of the variables from the patterns of a rule and, therefore, to control the applicability of the rule.

- The statements "A path between vertices **x** and **y** is *useful* if there are no other cheaper paths between **x** and **y**. There is no point in preserving non-useful paths for further processing. A non-useful path **x...y** is discarded immediately after a better path **x...y** is discovered." correspond to the rule below.

```
(defrule prune_non-useful_path
      (declare (salience 10))
      (path (nodes $? ?x) (cost ?w1))
      ?f <- (path (nodes $? ?x) (cost ?w2&:(> ?w2 ?w1)))
      =>
      (retract ?f))
```

The rule starts with the declaration **(declare (salience 10))** that makes the rule more important than all the other rules in the program. Indeed, we wish that a non-useful path be destroyed immediately after a better one is found. The salience of rules can be used to control the rule application process. If there are several applicable rules, then the most salient will be applied. If there are several applicable rules with the same salience, usually the rule applied is randomly selected (see a later lecture on control features). If a rule does not contain a salience declaration its salience is **0** by default.

Another important element of the rule above is the pattern

```
?f <- (path (nodes $? ?x) (cost ?w2&:(> ?w2 ?w1)))
```

The facts present in the factual knowledge base of a program are characterized by a special reference number (a reference for short). If the pattern matching process of a ***pattern***

succeeds, the CLIPS reference of the fact that matched the ***pattern*** can be saved for later use by assigning it to a variable. The way of specifying the assignment is ***variable <- pattern*** as above.

The action **(retract ?f)** deletes a fact form the factual knowledge base of the problem. The deleted fact is specified using its reference, assigned to the variable **?f**. Here the retracted fact is precisely the fact that matched the pattern **(path (nodes $? ?x) (cost ?w2&:(> ?w2 ?w1)))**, i.e. a non-useful path.

- The statement "The cheapest paths between **start** and **stop** are the *useful* paths between **start** and **stop** when the process of extending any existing path cannot be performed any longer" corresponds to the rule **result**.

```
(defrule result
       (declare (salience -10))
       (cheapest_paths (stop ?x))
       (path (nodes $?n ?x) (cost ?w))
       =>
       (printout t "path " (create$ $?n ?x) crlf "cost " ?w crlf))
```

The rule is the least important from the program. It will be applied when no other rule of the program can be applied. Therefore, the rule is applied when there are no more paths that can be extended and when there are no more non-useful paths to be retracted.

The program contains the additional rule **load_data** for loading the factual knowledge base of the problem with the initial content: the facts that describe the problem instance.

```
(defrule load_data
       =>
       (printout t "File: ")
       (load-facts (read)))
```

Apparently this rule has no patterns to control its application. However, it can be applied once only, by force of the *refraction principle*, discussed later. In order to apply this principle the rule is automatically transformed by CLIPS into

```
(defrule load_data
       (initial-fact)
       =>
       (printout t "File: ")
       (load-facts (read)))
```

The factual knowledge base of a CLIPS program is not empty initially. Before any program is loaded and starts execution there is always an **(initial-fact)** present in the factual knowledge base. It acts as the empty string in the Data Register of MAM for making applicable rules of the kind: **:-> *substitution_pattern*;**

The complete CLIPS program follows. Note that the textual order of the rules is not important. Parting from the MAM control, CLIPS selects the rule to be applied using the salience property and other implicit – data dependent – control mechanisms. In addition, note that the program has a certain degree of non-determinacy. Starting from the initial path, the paths can be extended in any order still preserving the correctness of the result.

```
; The minimum cost paths between two given vertices of a directed graph

(deftemplate edge (slot from) (slot to) (slot cost))
(deftemplate cheapest_paths (slot start) (slot stop))
(deftemplate path (multislot nodes) (slot cost))

(defrule initial_path
   (cheapest_paths (start ?x))
   =>
   (assert (path (nodes ?x) (cost 0))))
```

```
(defrule extended_path
   (path (nodes $?n ?y) (cost ?w))
   (edge (from ?y) (to ?z & ~?y & :(not (member ?z $?n))) (cost ?we))
   =>
   (assert (path (nodes $?n ?y ?z) (cost (+ ?w ?we)))))

(defrule prune_non-useful_path
   (declare (salience 10))
   (path (nodes $? ?x) (cost ?w1))
   ?f <- (path (nodes $? ?x) (cost ?w2&:(> ?w2 ?w1)))
   =>
   (retract ?f))

(defrule cheapest_paths
   (declare (salience -10))
   (cheapest_paths (stop ?x))
   (path (nodes $?n ?x) (cost ?w))
   =>
   (printout t crlf "path " (create$ $?n ?x) crlf "cost " ?w crlf))

(defrule load_data
  =>
  (printout t "File: ")
  (load-facts (read)))
```

A CLIPS session to run the program is illustrated below. The contents of the data file corresponds to the graph in figure 1. The goal is **`(cheapest_paths (start a) (stop e))`**. For details, including ways to trace the execution (a useful experience), see the CLIPS reference manual.

```
CLIPS> (load “H:\\CPSC-432\\CLIPS\\Minpath.clp”)
CLIPS> (reset)
CLIPS> (run)

File: H:\CPSC-432\CLIPS\Minpath.dat
path (a b d e)
cost 4
path (a c d e)
cost 4
CLIPS> (exit)
```

Correctness proof of the CLIPS program

Although the correctness of the CLIPS program is intuitive it can be formally proved by induction. The general induction schema is shown in figure 2, where **$Data_0$** are the initial data of the program, say **Prog**, **Data** are the data processed by the program at a given moment in time, and **Prop** is a property that helps proof the correctness. The aim is to show that if **$Prop(Data_0)$** is true and that whenever the program modifies some data **Data** such that **Prop(Data)** is true then the resulting data **Data'=Prog(Data)** satisfy the property **Prop**, i.e. **Prop(Data')** is true. Therefore, **Prop** is an invariant throughout the execution of the program and, consequently, it will hold for the results **$Data_f$** of the program.

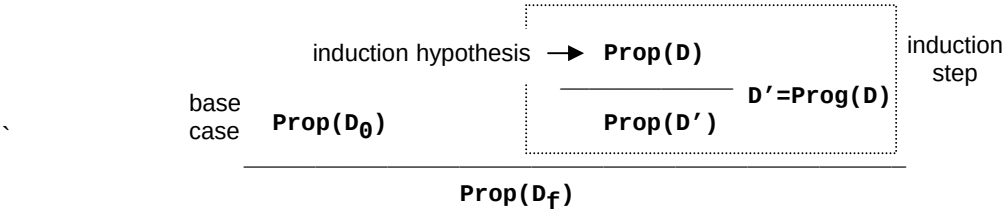

Figure 2. Proving the correctness of a program by induction

The following notations are used to characterize the data processed by the cheapest path CLIPS program at any given moment **t** of time during execution for a given directed graph **G=(V,E)**:

- **All(x)** is the set of all paths **start...x** that exist in the given graph. **All(z)** stays the same throughout the execution of the program.
- **P(x)** is the set of paths **start...x** stored in the factual knowledge base of the program at the moment **t**.
- **D(x)** is the set of all paths **start...x** discarded as being non-useful from the beginning of the program execution until the moment **t**.

During execution, the data processed by the program are **$Data(x) =_{def} P(x) \cup D(x)$** for all **$x \in V$**. Initially, the data of the program are:

$P_0(start) = \{start\}$, and **$P_0(x) = \varnothing$** for all **$x \in V$**, **$x \neq start$**
$D_0(x) = \varnothing$ for all **$x \in V$**

Let us define the following property of **Data(x)**, relevant for proving the correctness of the program:

$Prop(P(x),D(x)) =_{def}$
$\forall p \in P(x), \forall q \in P(x), \forall r \in D(x).\ cost(p)=cost(q) \wedge cost(p)<cost(r)$

where the symbol $\wedge$ is the logical *and* connector.

The only ways to change the data of the program is by extending a path and, eventually, by discarding non-useful paths. These actions are encoded by the rules **extend_path** and **prune_non-useful_path**.

1. Extending a path **start...y** to **start...y,x** means to select at random a new path from the set **All(x)**. The effect of applying the rule **extend_path** can be described conventionally as

P'(x),D'(x)= extend(P(x),D(x))
$=_{def}$ { p = select(All(x)); P'(x)= P(x)∪{p}; D'(x)= D(x);}

Due to the refraction control mechanism (discussed in a later lecture) each particular path **start...x** is generated once only.

2. Discarding a non-useful path **p = start...x** from **P(x)** can be seen as deleting **p** from the set **P(x)** and adding **p** to the set **D(x)**. Therefore, the effect of a single application of the rule **prune_non-useful_path** is **D'(x)= D(x)∪{p}, P'(x)=P(x)\{p}**, where **p** is a non-useful path. Since the rule has the highest salience we can consider that it is applied instantly for all non-useful paths existing in the current knowledge base of the program at a given moment in time. Obviously, if there are non-useful paths in **P(x)** the rule is not applied and the paths **D(x)**, **P(x)** are not modified. We shall note:

non-useful(x) = { p∈P(x) | ∀p'∈P(x). cost(p) > cost(p') }

P'(x),D'(x)= discard(P(x),D(x))
$=_{def}$ { D'(x)= D(x)∪ non-useful(x); P'(x)= P(x)\non-useful(x);}

the effect of *eventually* applying the rule **prune_non-useful_path** for all non-useful paths from **P(x)** at a given moment in time during the execution of the program. Actually, such a moment occurs just after an operation **p = select(All(x))** is performed.

The main compound solving action performed at a given moment in time during the execution of the program is described conventionally by:

P"(x),D"(x)= Prog(P(x),D(x))$=_{def}$ { P'(x),D'(x)= extend(P(x),D(x));
P"(x),D"(x)= discard(P'(x),D'(x));}.

We have now all the ingredients necessary to particularize the generic induction schema for the CLIPS program as shown in figure 3, for any **x∈V**, and to prove the following theorem.

Theorem. The CLIPS program for computing the minimum cost paths in a weighted directed graph is correct.

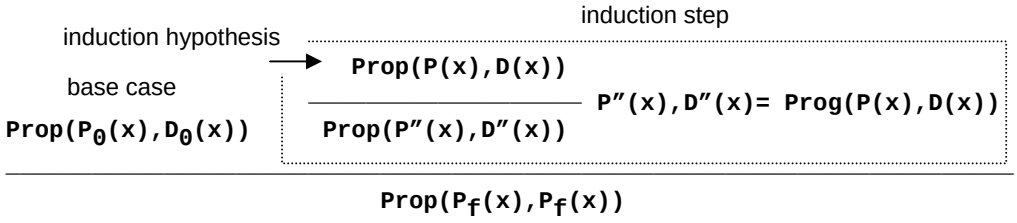

Figure 3. The induction schema for the CLIPS program

To prove the correctness of the program we have to show that the base case and the induction step of the induction schema in figure 3 hold.

Base case. For the **start** node **Prop(P(x),D(x))** holds trivially. For any other node **x ≠ start** the property holds also since the sets **P(x)**and **D(x)** are empty.

Induction step. Consider a node **x∈V** selected at random such that the operation **extend(P(x),D(x))** can be performed, i.e. there are still non-selected paths in **All(x)**. The program computes:

P"(x),D"(x)= Prog(P(x),D(x))$=_{def}$ { P'(x),D'(x)= extend(P(x),D(x));
P"(x),D"(x)= discard(P'(x),D'(x));}.

The operation **extend(P(x),D(x))** is executed first. Assume **p** is the path selected from **All(x)**. The result of **extend(P(x),D(x))** is:

P'(x) = P(x)∪{p}
D'(z) = D(x)

The result **D"(x),P"(x)**of **discard(P'(x),D'(x))**, and hence the result of **Prog(P(x),D(x))**, depends of the cost of the path **p**.

Case 1. The path **p** is non-useful, i.e. **∀p'∈P(x). cost(p) > cost(p')**, hence **non-useful(x) = {p}**. In this case the result of the operation **discard(P'(x),D'(x))** is:

P"(x) = P'(x)\non-useful(x) = P(x)∪{p}\{p} = P(x)
D"(x) = D'(x)∪non-useful(x) = D(x)∪{p}

According to the induction hypothesis **Prop(P(x),D(x))** holds. Then it is easy to see that it holds as well for **P(x), D(x)∪{p}**, i.e. for the pair **P"(x),D"(x)**.

Case 2. The path **p** has the property **∀p'∈P(x). cost(p)=cost(p')**, hence **non-useful(x)=∅**. In this case the result of **discard(P'(x),D'(x))** is:

P"(x) = P'(x)\non-useful(x) = P(x)∪{p}
D"(x) = D'(x)∪non-useful(x) = D(x)

Since the induction hypothesis **Prop(P(x),D(x))** holds, it is easy to verify that the property **Prop(P(x)∪{p},D(x))** holds. Therefore, **Prop(P"(x),D"(x))** holds.

Case 3. The path **p** is the best so far, i.e. **∀p'∈P(x). cost(p) < cost(p')**, hence **non-useful(x) = P(x)**. The result of **discard(P'(x),D'(x))** is:

P"(x) = P'(x)\non-useful(x) = P(x)∪{p}\P(x) = {p}
D"(x) = D'(x)∪non-useful(x) = D(x)∪P(x)

Since the induction hypothesis **Prop(P(x),D(x))** holds, it is easy to see that the property **Prop({p},D(x)∪P(x))** holds. Hence **Prop(P"(x),D"(x))** is true.

Since the property **Prop(P(x),D(x))** holds during the program execution, it holds just before the program stops. The program stops when all the paths from **All(x)** have been selected, for all **x∈V**. At this moment we have

$$\forall x \in V \;.\; P_f(x) \cup D_f(x) = All(x)$$

where $P_f(x)$ is the final set of paths in the factual knowledge base and $D_f(x)$ is the overall set of paths discarded by the program for the node **x**. Each path **start...x** from **All(x)** is either in $P_f(x)$ or in $D_f(x)$ and, according to the property **Prop**$(P_f(x),D_f(x))$:

$$\forall p \in P_f(x), \forall q \in P_f(x), \forall r \in D_f(x).\; cost(p)=cost(q) \wedge cost(p)<cost(r)$$

It follows that the set $P_f(x)$ contains all possible optimal paths **start...x** ■

A declarative model for the min cost paths weighted oriented graphs

The model is biased for generating the paths procedurally, by extending existing paths.
The entities and the constraints in red convey operational information for the above mentioned process.

```
Node = Symbol;

ENTITY Edge;
  from,to: Node;
  cost: NUMBER;
WHERE cost ≥ 0;
END_ENTITY;

ENTITY Graph;
  nodes: SET OF Node;
  edges: SET OF Edge;
WHERE
  ∀e∈edges • e.from∈nodes ∧ e.to∈nodes;
END_ENTITY;

ENTITY Path ABSTRACT SUPERTYPE;
  graph: Graph;        // parent graph
  nodes: SEQ OF Node;  // nodes of the path

DERIVE
  nodes_set: SET OF Node = nodes.seq_to_set();
  // seq_to_set converts a SEQ to a SET

  cost: NUMBER = sumof{∀e∈graph.edges,u∈nodes_set,v∈nodes_set |
                       nodes.is_next(v,u) ∧ e.from = u ∧ e.to = v • e.cost}
  // is_next(v,u) tests whether v is next to u in a SEQ
WHERE
   nodes ⊆ graph.nodes;

   // Stepping through the nodes of the path. The constraint is useless
   // if there is no need to generate "possible" paths.
   ∀u∈nodes_set,v∈nodes_set | nodes.is_next(v,u) •
                 (∃e∈graph.edges • e.from = u ∧ e.to = v)
END_ENTITY;

ENTITY MinCostPath SUBTYPE OF Path;
WHERE
   ∀Path(path) | path.graph = graph ∧
                 path.nodes.first() = nodes.first() ∧
                 path.nodes.last() = nodes.last()
              • cost ≤ path.cost;
END_ENTITY;

ENTITY ProblemGoal;
  graph: Graph;
  start,stop: Node;
  path: MinPath;
WHERE
  start∈graph.nodes ∧ stop∈graph.nodes;

  path.graph = graph;
  path.nodes.first() = bounds.start;
  path.nodes.last() = bounds.stop;

  // Wait until all graph path have been instanced
  ∀Path(p),Edge(e) |
     p.graph=graph ∧ e∈p.graph.edges ∧ p.last()=e.from ∧ e.to∉p.nodes_set
              • (∃ExtendedPath(p') • p'.path = p ∧ p'.edge = edge);
END_ENTITY;
```

```
ENTITY SingletonPath SUBTYPE OF Path;
   node: Node;
WHERE
   node∈graph.nodes;
   nodes = {node};
END_ENTITY;

ENTITY ExtendedPath SUBTYPE OF Path;
   path: Path; // the parent path of the extended path
   edge: Edge; // the edge that extends p
WHERE
   path.graph = graph;
   edge∈graph.edges;
   path.last() = edge.from;
   // last returns the last element of a non-empty SEQ

   edge.to ∉path.nodes_set; // acyclic path
   nodes = path.nodes.append(edge.to);
   // procedural interpretation: binding of the slot nodes to a value
END_ENTITY;
```

A variant of

```
ENTITY ExtendedPath SUBTYPE OF Path;
WHERE
   ∃Path(p),Edge(e) •
               p.graph=graph
             ∧ e∈graph.edges
             ∧ p.last() = e.from
             ∧ e.to ∉p.nodes_set
             ∧ nodes = p.nodes.append(e.to);
END_ENTITY;
```

Possible alternatives for instancing the **ExtendedPath** entity.

1. Starting from the constraint, automatically derive an instancing algorithm, such as

```
for-each (Path(p)∈KB)
  for-each(e∈path.graph.edges)
    if(p.last() = e.from ∧ e.to∉p.nodes_set)
      make-instance-of Path(nodes = p.nodes.append(e.to),graph = p.graph);
```

2. Possible **ExtendedPath** candidates are considered. Some are already in **KB**, other could be generated (how?). Those candidates that satisfy the constraint become instances of the **ExtendedPath.**

```
; The minimum cost paths between two given vertices of a directed graph
; A CLIPS program deriving from the above model
; The program works with a single graph

(deftemplate edge (slot from) (slot to) (slot cost))
(deftemplate path-bounds (slot start) (slot stop))
(deftemplate path (multislot nodes) (slot cost))

(defrule singleton_path
   (path-bounds (start ?x))
   =>
   (assert (path (nodes ?x) (cost 0))))

(defrule extended_path
   (path (nodes $?n ?y) (cost ?w))
   (edge (from ?y) (to ?z & ~?y & :(not (member ?z $?n))) (cost ?we))
   =>
   (assert (path (nodes $?n ?y ?z) (cost (+ ?w ?we)))))

(defrule destroy_non_min_paths
   (declare (salience 10))
   (path (nodes $? ?x) (cost ?w1))
   ?f <- (path (nodes $? ?x) (cost ?w2&:(> ?w2 ?w1)))
   =>
   (retract ?f))

(defrule solution
   (declare (salience -10))
   (path-bounds (stop ?x))
   (path (nodes $?n ?x) (cost ?w))
   =>
   (printout t crlf "path " (create$ $?n ?x) crlf "cost " ?w crlf))

(defrule load_data
  =>
  (printout t "File: ")
  (load-facts (read))
```