Chapter 11

MOTORS Implementation

The MOTORS[1] toolset consists of search algorithms, constraint propagation algorithms, and support for the integration of search and propagation. The search algorithms provide for the construction of one, all or possibly some solutions to constraint satisfaction problems. The implemented search and propagation algorithms are described in general terms in Chapter 3. The search algorithms each apply varying degrees of "look-ahead" or remaining problem reduction between selection of variable domain values for instantiation, and are configured so as to allow extension of their propagation before or during search. The constraint propagation algorithms are designed as stand-alone components which take as input a constraint specification or graph, and reduce it accordingly. Essentially, a domain is constructed on top of the search and propagation facilities.

Specific domain implemented are for the N-Queens problem[2], and for the Model Pattern Recognition (MPR) Problem[3]. The Model Pattern Recognition domain implementation will be discussed in the later part of this appendix.

The purpose of this appendix is to conduct a general overview of the tools provided to date in the toolset, and to outline the code for future application, modification, and enhancement.

All source code is commented, and provision has been made for debugging flags for tracing search and propagation output. For further details, please refer to the code and make note of the control flags.

11.1 Tour of the code

The code is separated into the following modules:

[1]Unless otherwise noted, all implementation was done by Steven Woods.
[2]Original code provided by Qiang Yang and Phillip Fong, University of Waterloo.
[3]Code designed and written by Steven Woods, DREV.

- *bt.lisp* The generic backtracking algorithm. This algorithm accepts as input a search-state, the consistency function, and control flags indicating which combinations of search and propagation should be used. Options supported are currently Forward-Checking, Dynamic-Rearrangemnt, BackJumping, application of a pre-defined arc-consistency (or partial arc-consistency) algorithm during search, or application of a pre-defined spatial limiting heuristic during search.

  Currently the arc consistency algorithms available for use must be explicity hardcoded into the algorithm, however, in the future we will wish to make this either an option based on the passed arc-consis parameter, or pass the function to apply itself. The current version has the AC-3 algorithm used as an example.

- *bm.lisp* The BackMarking algorithm is implemented recursively as opposed to the iterative approach of *bm.lisp*. This algorithm accepts as input the variables list, and the consistency function. Currently application of in-search constraint propagation or spatial heuristics are not supported.

  In the future it would be desirable to investigate further how to incorporate in-search propagation without disrupting the maintenance of the various constraint results tables maintained in BackMarking.

- *ct.lisp* The constraint propagation algorithm AC-3[4], supported by the revise algorithm as described by Mackworth. This algorithm is stand-alone, and accepts as input a constraint graph defined in terms of a variable and domain list, a function indicating the presence of arcs in the graph, and a function for determining the consistency of two assignments.

  Some experimentation was done with non-binary constraint graphs for consideration with MPR description, however, this proved problematic with respect to integrating with intelligent backtracking algorithms such as BackJumping. Future research on this integration can be undertaken, and reference made to other work in n-ary graph propagation, such as the algorithm GAC-4 [62, 61, 63]. It should be noted that in [59] it is shown that any CSP with n-ary constraints can be converted into another equivalent binary CSP.

- *Devel/ac4.lisp* The constraint propagation algorithm described in [46][5]. This algorithm may replace AC-3. It also accepts as input the constraint graph and outputs the simplified graph. Please note that the published algorithm is in error, and correction *has* been noted in this file according to personal communication with Hany.Tolba@loria.fr. Please note that this function is not currently operational and needs to be repaired. Notes are provided with the source code as to specific difficulties and testing.

[4]Please note that this source code for AC-3 was provided by Qiang Yang, and Phillip Fong of the University of Waterloo

[5]Implemented by Denis McGonigal under the supervision of Steven Woods.

- *Devel/acq.lisp* The partial arc-consistency algorithms[6]. These algorithms may also replace AC-3 and function in the same way, simplifying an input constraint graph. In conjunction with standard backtracking these may be utilized to form hybrid search algorithms which perform only some arc-consistency simplification of constraint graphs between instantiations. In conjunction with some heuristics determining how much constraint propagation to apply during search it is hoped that improved performance can be obtained.

  In the future, additional research will be done on determining how to best apply these propagation algorithms, and how to select which algorithm to apply. Please note that these functions have been defined separately from MOTORS and are not integrated at this time, however the functions have been tested independently.

- *queens.lisp* The N-queens implementation. Useful for testing and construction of simple, explainable examples, the queens domain has been widely used as an example of constraint propagation. In this source is also provided the alternate Confused-N-queens problem. This file defined the calls available to solve the queens or confused problems, including the calls to initial propagation, and to search. Control flags cover domain selection, search mode selection, application of node consistency, arc consistency, and optiononal flags to control search such as Forward Checking, etc. Further options include in-advance sorting of variable domains (not useful in symettric queens problems, except after constraint propagation may have affected domain sizes), and as well output options such as output to file, debugging, or printing search information at checkpoints.

  This and any other domain files must define the consistency function, and a function which indicates the presence or absence of domain constraint arcs.

  Future extension should unify the calls from the domains under a single initiation procedure, and relegate this file to domain definition only.

- *mpr-simple.lisp* The Model Pattern Recognition domain is implemented in two files. This is the first, which contains the domain function as does the queens.lisp file. Consistency functions are defined in this file as described later in this appendix. Initial calls to search and propagation modules are provided here as described for queens, as well as all constrol flags over selection of instance options such as distribution control over domain element type in random problem instances, and selection of random-key for reproduction of earlier random experiments runs.

- *mpr-setup.lisp* The second of the MPR domain definition files, this file contains all possible situation instances, and template definitions. These instances and templates are identified by name for use in the mpr-simple.lisp function calls as desired. Essentially

[6]Implemented by Denis McGonigal under the supervision of Steven Woods.

we maintain a complete list of all created test cases to facillitate later reproduction of results. In addition, this file maintains the distribution control files for the random creation of attributes in random situations. The random situation creation code resides in this file also.

- *utility.lisp* The general utility file contains functions not specific to either domain. Generally support functions such as global variable instantiation, and output. I/O functions reside here such as maintenance of random state files, creation of random situation output for later display via gnuplot, and output of test results. Note that the gnuplot interface functions as follows: when a new random situation sample is created, that sample configuration is stored in a set of files with unique file names including the random key used, and can be displayed via gnuplot creation of a postscript file and displayed via ghostview or laser printing.

- *load.lisp* The initial load file simply loads all required source code. If you load this file after starting lisp, the system loads.

- *compile.lisp* Loading this file will recompile the entire system.

- *prt-it* Shell script to dump all lisp source code to a printer.

- *COPYRIGHT-NOTICE* File containing version, ownership and authorship information.

- *EXAMPLES* File containing examples of running MOTORS from start to finish of a lisp session. This example file does not demonstrate all MOTORS functions, only a sample.

- *Batch* This directory contains scripts and output from Batch runs of MOTORS. The shell scripts to initialize the lisp process and the lisp input control files are included, and may be used as example of batch runnning the system.

- *Interest* This directory contains extend.notes, a working document detailing the work of Yang in implementing abstraction on top of constraint propagation. This work is not currently part of MOTORS.

- *Random* This directory contains random state descriptors from Lucid Common Lisp which are written out and read in by MOTORS when a unique state is generated. These allow reproduction of particular runs for further investigation. Each filename includes the key identifier needed by MOTORS to run that case.

- *Situation* This directory containts situation descriptions as they are created, including the size and random key used. From these descriptions, the situation can be drawn or shown using the shell scripts draw and show. Future integration of saved situation objects could speed up testing by eliminating the need to repeat construction of identical situations. This is currently not supported.

11.2 Model Pattern Recognition Implementation

Refer to mpr-simple.lisp and mpr-setup.lisp.

The model pattern recognition system is implemented by considering template models as constraint graphs. The following is an example of a template.

```
'( "test1-m"
  (
   (ts1 ARI  * * * Co)
   (ts2 ARM  * * * Co)
   (ts3 ARI  * * * Co)
   (ts5 (ARI AIHQ) * * * Co)
   )
  (
   (sep (ts1 ts2) (4 8))
   (left-of   (ts1 ts2))
   (sep (ts2 ts3) (4 8))
   (right-of  (ts3 ts2))
   (sep (ts2 ts5) (6 10))
   (behind-of  (ts5 ts2))
   (echelon   (ts2 ts5) (4))
   (same-attr-orient (ts1 ts3))
   )
  ( (ts1 20) (ts2 20) (ts3 20) (ts5 20) )
  )
```

This description corresponds to the csp described earlier in Chapter 7, on page 104. The first element "test1-m" is the template name. The second element is a list of node definitions prefixed by node name, including constraints on the *Type*, *Size*, *Activity*, *Orientation*, and *Absolute Location* of the template elements.

- *Type* Indicates which type of unit may be identifed as this template slot. This may be either a list of types or a single type. Possible types in the current implementation are: ARM (Armoured), AIHQ (Armoured Infantry HeadQuarters), ARI (Armoured Infantry), ART (Armoured Tank), or * (unknown).
- *Size* Indicates the possible size of a unit. This may be either a list of sizes, or a single size. Possible sizes in the current implementation are: Grp (Group), Co (Company), Bn (Battallion), PL (Platoon), or * (unknown).

- *Activity* Indicates the possible activity of a unit. This may be either a list of activities, or a single activity. Possible activities in the current implementation are: Adv (Advance), Rec (Reconnaissance), Dep (Deployed), Ret (Retreat), oir * (unknown).

- *Orientation* Indicates the possible orientation of a unit. This may be either a list of orientations, or a single orientation. Possible orientations in the current implementation are: N, NE, E, SE, S, SW, W, NW, or * (unknown).

- *Absolute Position* Indicates the known range of a template slot in absolute terms (if known). Typically this value would be used to either limit a search-in-progress for a template, or to radically restrict a search based upon incontrovertible evidence received. This takes the form of a list consisting of minimum x coordinate, minimum y coordinate, maximum x coordinate, and maximum y coordinate. Generally this field will use a value of *, indicating the known range of this template slot is unknown.

The third element of our template definition is a list of inter-slot constraints. Currently supported constraints are Sep (Separation), Left-of, Right-of, Behind-of, Ahead-of, Echelon, same-attr-orient, same-attr-type, same-attr-size, and same-attr-activity. All these constraints are binary between two and only two slots. In addition, a 3-ary constraint has been used for experimentation purposes: Medial. Please note that medial is not robust, and current search and propagation does not support constraints with airity greater than 2. Each of these arc constraints are described below.

- *Separation* Two slots are constrained by the minimum and maxium distance between them. The form of this slot is a list consisting of the term "sep", a list of the two slots concerned, and a list of the minimum distance they may be apart and the maximum. The distance used is the absolute distance between the given coordinates of two situation elements. Later in this appendix the form of situation elements will be described, however, in the current implementation, each situation element has an absolute coordinate position, not a range or area. In future extensions it will be necessary to give a situation element a ranging area as opposed to a coordinate position.

- *Left-of, Right-of* These constraints are directional. A "destination" slot is constrained by a "source" slot to be either left of, or right of, the source. Left-of is defined by first defining a direction vector by taking the orientation of the source as a vector commencing at infinity and running through the situation object in the direction of the orientation. Any object to the right of the direction vector is "Right-of" the source object, and any object to the left of the direction vector is "Left-of" the source object. This method is only an approximation, and while effective, might not fully capture the intended meaning of left or right. If the source slot instance has no known orientation, these constraints should not apply, and may be thought of as being universally satisfied. Note that Left-of and Right-of are *not* necessarily complimentary since each are defined in terms of the orientation vector of the source, if one exists.

- *Ahead-of, Behind-of* These constraints are directional also. We define a balance line through the "source" slot by rotating the direction vector 90 degrees. Any object to the side of the balance line in the direction of the orientation is "Ahead-of" the source, and any object to the side of the balance line in the opposite direction of the source orientation is "Behind-of" the source object. If the source slot instance has no known orientation, these constraints should not apply, and may be thought of as being universally satisfied. Note that Ahead-of and Behind-of are *not* necessarily complimentary since each are defined in terms of the orientation vector of the source, if one exists.

- *Echelon* Once again, the echelon constraint is a directed one. The source slot and its orientation are used to limit the location of the destination slot. We once again create the direction vector from infinity through the source slot in the direction of the source slot's orientation. We now imagine a corridor created by making this direction vector a plane of some width. For the echelon constraint to hold, the destination object must be within the plane created from the source object location and orientation. If a source slot instance has no orientation, the echelon constraint should not apply, and may be thought of as being satisfied universally. Note that echelon is not necessarily in both directions from some instance A to B and then from B to A, since each direction is determined by the orientation of the source, if it has one.

- *same-attr-orient* Indicates that each of the two affected slot instances must possess an orientation value determined to be "equal". This equality is currently defined as identity. Note that this constraint needs to be extended to accomodate lists of possible values as accomodates by the similar node constraint.

- *same-attr-type* Indicates that each of the two affected slot instances must possess a type value determined to be "equal". This equality is currently defined as identity. Note that this constraint needs to be extended to accomodate lists of possible values as accomodates by the similar node constraint.

- *same-attr-size* Indicates that each of the two affected slot instances must possess a size value determined to be "equal". This equality is currently defined as identity. Note that this constraint needs to be extended to accomodate lists of possible values as accomodates by the similar node constraint.

- *same-attr-activity* Indicates that each of the two affected slot instances must possess an activity value determined to be "equal". This equality is currently defined as identity. Note that this constraint needs to be extended to accomodate lists of possible values as accomodates by the similar node constraint.

The final element of the template definition is an experimental heuristic currently usually nil, known as the SCH (spatial cohesion heuristic) list. This element defines our attempt to capture the conception of "spatial locality" or cohesion in the construction of possible template instances matching this template. Currently, if the SCH flag is set to true during backtracking search, then when any slot with a value in the template SCH list is instantiated, the remaining variable domains are each reduced according to the range limit accompanying the slot identifier. Essentially we are forcing propagation of a spatial constraint similar to separation after each listed instantiation. This heuristic is under experimentation and should generally not be used. It may be very useful in cases where there are many situation elements, and a template is under constrained. It would appear that only certain slots, perhaps locationally critical ones, should be used as SCH limiters.

We have now explained how to define a particular template. The constraints provided must be viewed as examples of the types of constraints typical to this problem, as well as other spatial problems. Other constraints may be added by examining the way these samples are implemented in mpr-simple.lisp, function test-constraint-2.

Situation elements are our candidates from which we must identify a template instance. For the purposes of our experiments we define a particular instance (or instances) in a predefined set of situation elements, and then clutter the situation with additional randomly created situation elements. The following is an example of an instance of the template described above.

```
'( "test-1b"
     ( sit3-1  (70 30)  ARI   Adv ne co)
     ( sit3-2  (74 26)  ARM   Adv ne co)
     ( sit3-3  (79 21)  ARI   Adv ne co)
     ( sit3-4  (69 21)  ART   Adv ne co)
     ( sit3-5  (67 19)  AIHQ  Adv ne co) )
```

This is a partial situation composed of only the elements of the template described. In Chapter 7, Figure 7.5 on page 107 this list of situation elements is graphed as "Middle Abstract Solution", and in Figure 7.4, these elements are embedded in an additional 300 situation elements. The format of each situation element is a list composes of the sighting element identifier, the absolute sighting position consisting of an x and y coordinate, the type of the sighting if known, the activity of the sighted unit if known, the orientation of the sighted unit if known, and the size of the sighted unit if known.

Randomly created situation elements providing noise for our test cases are distributed according to space, and as well according to attribute frequency assignment. The following is a characterization of the current ability to adjust the frequency of each attribute assignment.

```
( "ddist"
  ( (ARM 1) (AIHQ 1) (ARI 1) (ART 1) (* 1) )
  ( (Grp 1) (Co 1) (Bn 1) (Pl 1) (* 1) )
  ( (Adv 1) (Rec 1) (Dep 1) (Ret 1) (* 1) )
  ( (n 1) (ne 1) (e 1) (se 1) (s 1) (sw 1) (w 1) (nw 1) (* 0) ))
```

Here, each of the element attributes Type, Size, Activity, Orientation, or * (unknown) can have configurable frequency when generated as noise. For instance, in this above example distribution "ddist", each element has equal probability of selection in Type, Size and Activity, while all possible Orientations are equally probable except * (unknown) which is never selected. Our method essentially creates a list for each attribute composes of N elements of each attribute value where N is the value corresponding to the value. An element is selected at random from this created list and assigned to the situation element. For instance, consider a distribution for Type that looked like the following.

```
( (ARM 2) (AIHQ 1) (ARI 3) (ART 3) (* 0) )
```

We can then imagine a list of 8 elements composed from this distribution as

```
( ARM ARM AIHQ ARI ARI ART ART ART )
```

where there is a 2/8 chance of selecting ARM, a 1/8 chance of selecting AIHQ, a 3/8 chance of selecting ARI, a 3/8 chance of selecting ART, and no chance at all of selection * (unknown).

In the current implementation, selection of "noise" situation element positions is purely random, however, further investigation of different distributions should be investigated and the corresponding affect on search identified. The domain needs to be characterized for arriving message distribution both spatially and by type in order to better simulate reality in the identification process.

11.3 Running Motors

11.3.1 Input

We have discussed briefly the source code portions responsible for running the MOTORS domain problems, the process of template creation, and the process of establishing instances of the templates amidst noise. We show below the source code used in the calling procedure for MPR.

```
(defun mpr (&key
    (situation-id       "test-1")
    (sit-noise                 0)
    (template-id      "test1-w")
    (dom (list 'mpr situation-id template-id))
    (search-mode            "bt")
    (node-consis               t)
    (node-force-all          nil)
    (arc-consis              nil)
    (sch-call                nil)
```

```
(rand-dist            "ddist")
(forward-checking        nil)
(dynamic-rearrangement   nil)
(advance-sort            nil)
(one-solution-only       nil)
(cpu-sec-limit           600)
(ck-pt-interval          150)
(debug                   nil)
(debug-csp               nil)
(random-ident       'default)
(output-file             nil)
)
```

Situation-id identifies the situation instance we have created and wish to use, sit-noise the number of additional elements to generate as noise, template-id identifies the template to search for, dom is an optional domain name, search-mode one of bt (backtracking), bj (backjumping), or bm (backmarking), node-consis indicates whether or not to perform node-consistency checking prior to search (generally true), node-force-all is a indicator that allows for all node constraints to be checked if you desire (never true), arc-consis identifes when to perform the arc-consis algorithm implemented - either before search, during search, or both (this will be extended greatly in the future), sch-call indicates whether or not to apply the spatial cohesion heuristic, rand-dist determines which random situation element distribution to use, forward-checking and dynamic rearrangement are backtrack options as described earlier in the paper, advance-sort indicates whether or not to sort the domain variables according to some heuristic such as domain size, one-solution-only indicates whether or not to find all solutions, cpu-sec-limit stops the search process when it is met, ck-pt-interval prints statistics on at every expiration of the interval, debug indicates that we wish to debug the search process and see the search space, debug-csp indicates that we wish to debug the constraint application process, random-ident indicates which random seed we wish to use (default is the standard case, nil creates a new one, or some identifier indicates a predefined seed in the Random directory), and output-file indicates a file name to write output to if desired.

11.3.2 Output

Sample output from MOTORS looks something like the following example of a default test run.

```
***** OPTIONS for CSP ( (MPR test-1 test1-w) ) *****

 Situation id         :   test-1
 Situation noise add  :   0
```

```
 Situation Size       :   12
 Template id          :   test1-w
 Template Size        :   5
 Distribution id      :   ddist
 Distributions    :
((ARM 1) (AIHQ 1) (ARI 1) (ART 1) (* 1))
((GRP 1) (CO 1) (BN 1) (PL 1) (* 1))
((ADV 1) (REC 1) (DEP 1) (RET 1) (* 1))
((N 1) (NE 1) (E 1) (SE 1) (S 1) (SW 1) (W 1) (NW 1) (* 0))
 Num of constraints   :   11

 Search Mode          : bt
 Node Consis Mode     : T
 Node Force all       : NIL
 Arc  Consis Mode     : NIL
 Forward Checking     : NIL
 Dynamic Rearrangement: NIL
 Advance sort option  : NIL
 Backjump option      : NIL
 One Solution Only    : NIL
 CPU Seconds limit    : 600
 CK Point interval    : 150
 Debug                : NIL
 Debug CSP            : NIL

 Random Ident         : default
 Spatial Cohes Prune  : NIL

Domain size ( INITIAL ) statistics ...
Max[(TS1 12)]  Min[(TS1 12)]  Avg[12.0]

Domain size ( NODE ) statistics ...
Max[(TS1 6)]  Min[(TS2 2)]  Avg[3.6]

***** BACKTRACK Information *****

 Exit Location    : NULL-PARENT-STATE
 No. Solutions    : 1
 Node Consis. cks   : 132
 Ttl constr chks  : 36
 BT nodes created : 10
```

```
 Nodes visited    : 34

Advance Processing CPU Seconds: 0.02
Search             CPU Seconds: 0.06
Total              CPU Seconds: 0.08

COMPLETE
```

After a description of the options selected, the output is composed of several parts:

- *Problem Characterization* Domain sizes are evaluated in the initial problem formulation and then again after node and pre-search arc constraint propagation. The improvement can be seen in the average (Avg) measure which indicates the average size of the variable domains before search commenced. Maximum and minimum reflect similar bounds for the same information amongts all variables.
- *Exit Location* This indicates where search exited. It can be in error, from a checkpoint, or from a null-parent-state indicating the search space was completely searched.
- *No. Solutions* As one would expect, the number of template instances found in the situation.
- *Node Consis cks* The number of constraints applied to reduced the variable domains based on node constraints defined. In an implemented system these constraints would be applied in an ongoing fashion, and would not need to be applied at the beginning of a search.
- *Ttl constr chks* The number of inter-slot constraints applied during recognition, including propagation in advance and during search.
- *BT nodes created* The number of backtracking positions during search.
- *Nodes visisted* The size of the search space explored.
- *Advance Processing* The time in CPU seconds spent processing Node constraints or Arc constraints before search commenced.
- *Search* The time in CPU seconds spent in search, including in-search constraint propagation.
- *Total* The sum of Advance Processing time and Search time.

Additionally, further details are available if desired. These are obtained through the function "(show-solution)" which basically repeats the display procedure, but with more detail. The additional detail includes:

- *Matches-attribute* A breakdown for each attribute of the number of node propagation constraints.

- *Consistent-nulls* A counter of the number of times an assignment was made which required no constraint checks at all as a result of the variable lacking applicable constraints.

- *Unique restricts* A counter of the number of times an attempt was made to check a constraint between identical situation elements. This situation arises when template slots have identical node characteristics.

- *Spatical Cohesion* Identification of the cost in terms of additional constraint checks and savings in terms of removed domain values incurred by applying the spatial cohesion heuristic.

- *Constraint Breakdown* A breakdown for each inter-slot constraint of how many types it was checked during the entire propagation and search process.

Two additional output functions are available demonstrating some of the system flexibility and future use for explanation purposes. First, during node consistency checking, a record is kept of failing nodes and why they failed to qualify for assignment to a template slot. This structure can be queried by "(explain)" which shows all reasoning behind removal, or "(explain-node-fail slot instance)" which will tell why a particular instance was not a candidate for a particular slot. Second, the function "(show-cons-fail)" details each application of a failed arc constraint indicating which situation elements failed the constraint.