# The Application of Inductive Logic Programming to Finite Element Mesh Design

Bojan Dolšak and
Stephen Muggleton
The Turing Institute,
36 North Hanover Street,
Glasgow G1 2AD,
UK.

**Abstract**

Finite element methods are used extensively by engineers and modelling scientists to analyse stresses in physical structures. These structures are represented quantitatively as finite collections of elements. The deformation of each element is computed using linear algebraic equations. In order to design a numerical model of a physical structure it is necessary to decide the appropriate resolution for modelling each component part. Considerable expertise is required in choosing these resolution values. Too fine a mesh leads to unnecessary computational overheads when executing the model. Too coarse a mesh produces intolerable approximation errors. In this paper we demonstrate that rules for deciding on appropriate resolution values can be inductively constructed from expert-provided examples. The Inductive Logic Programming algorithm Golem is employed for this purpose. Cross-validation testing of rules produced by Golem in this domain indicate an acuracy of around 79% correct on unseen data. We believe that for this domain the use of a relational learning approach is essential for reflecting the relations between elements of the physical structure being modelled.

## 1 Introduction

Finite element (FE) methods are used extensively by engineers and modelling scientists to analyse stresses in physical structures. Figure 1 is a typical instance of such a structure. The figure illustrates a cross-section of a cylinder from a hydraulic press used in the leather industry. The effects of external and internal pressure on such a structure can be expressed as a set of differential equations. However it is not possible to solve such differential equations in a reasonable amount of computer time for arbitrarily complex structures. Instead engineers partition the structure into a number of finite elements which are interconected at a discrete number of nodal points situated on their boundaries. The displacement of these nodal points are basic unknown parameters of the problem. A set of functions is chosen to define uniquely the state of displacement within each FE in terms of its nodal displacements. With such discretisation instead of differential equations, engineers have to solve the set of algebraic equations. Basic formulation of the finite element method is described in detail in [5].

Figure 1: A typical structure to be analysed

The set of finite elements is called the FE mesh. Figure 2 shows the hand-constructed FE mesh chosen by experts for the structure of Figure 1.

The basic demand for the FE mesh is that the mesh should represent the exact shape of the structure. Fine meshes are used in places where high deformations are expected. Coarser meshes are adequate where the expected deformations are small. In general the coarsest mesh which gives rise to sufficiently low errors is employed. This minimises the required computation time since each additional element adds extra linear algebraic equations to the set which must be solved.

It is very difficult to know in advance where the mesh should be fine and where it should be coarse because a number of parameters have to be considered. These include the shape of the structure and the loadings and boundary conditions. Usually it is necessary to make a few different meshes until we find the right one. The trouble is that each mesh must be analysed since we generate the next mesh on the base of the results derived from the previous mesh. Since each mesh analysis can take several days of computer time, iterative analysis can be very costly. There exists a great need for knowledge based systems which are able to automatically design FE meshes. In this paper we describe an attempt to automatically build part of such a knowledge base using machine learning techniques.

FE methods have been applied extensively for the last thirty years. A large number of published reports (such as [1, 4]) give FE designs in terms of the problem (input data), the final FE mesh (chosen after several trials) and the results of the analysis. To build a knowledge base for an expert system which would help us to find a "reasonable" mesh without previous trials, or at least with less of them, we have to transfer input data and final meshes from these reports into the system. We believe that the best way to do this is to divide each structure into a collection of edges. Figure 3 shows some of the labellings of edges of the structure from Figure 1. If we know how many elements are on the edges of the structure the FE mesh inside the structure is also determined. Thus the number of elements chosen for each edge in published data can be used as a source of examples for a machine learning technique. Certain relationships between edges influence the final mesh. Therefore the target concept cannot be described completely using a propositional learning algorithm. Instead a relational learning algorithm is required.

The expert system should also choose the types of elements which need to be used. There are

Figure 2: Finite element mesh for the structure

Figure 3: Labelled edges of structure

different types of elements for 2D and for 3D structures. These are not all compatible since for some elements the approximation between nodes is linear and for others it is parabolic. So if we have for instance a 3D structure and we want to use parabolic elements there are only a small number of elements that could be used. To represent the shape of the array we must choose some combination of these elements. We believe that this knowledge can be described directly by stating rules without automatic learning.

# 2 Golem

In [3] Muggleton and Feng describe a relational learning algorithm called Golem. Since Golem constructs Logic Programs from examples and background knowledge it is an Inductive Logic Programming system [2]. Golem examples and background knowledge are represented as ground atomic facts.

The hypotheses constructed by Golem are restricted by a particular condition. A variable found in the body of a Prolog clause and not in the head of the clause is normally assumed to be existentially quantified. In Golem it is assumed that existential quantification is restricted to being "exists exactly one". This is otherwise known as Hilbert $\epsilon*$ quantification. Target clauses using non $\epsilon*$ quantification cannot be learned using Golem. This caused some difficulty for the learning task described in this paper (see Appendixes C and D).

# 3 Presentation of examples and background knowledge for mesh learning

Golem needs three types of input files to build the rules.

- foreground examples
- negative examples
- background facts

In the file with foreground examples (Appendix A) we have stated the examples of classified edges of FE meshes. For each edge there is a sentence such as

mesh(a13,1).

which means that on the edge "a13" there is "1" finite element. For different structures we have used different letters for the first letter in the name of the edge.

In the file with negative examples (Appendix B) the sentences have the same format as those in the foreground example file. The negative examples contain all possible combinations of numbers of elements other than those found in the foreground examples for each edge name. Golem needs negative examples for reducing the rules.

The file with background facts (Appendix C) is in some ways the most important. This file contains definitions of the vocabulary that can be used to describe hypotheses about meshes. The contents of the background file is divided into five parts.

- declarations
- types of edges

- boundary conditions
- loads
- geometric representation

The declarations part defines the *modes* and *determinations* of each predicate. The mode of a predicate specifies its input and output arguments. The determinations are the set of background predicates which are expected in the bodies of rules for this predicate. Thus a sentence such as

determ(mesh(_,_),mesh(_,_)).

means that recursive rules for the predicate 'mesh' are allowed. We have classified edges into twelve different types.

- important long
- important
- important short
- not important
- circuit
- half circuit
- quarter circuit
- short for a hole
- long for a hole
- circuit hole
- half circuit hole
- quarter circuit hole

We believe that these are all the relevant types of FE edges. As an example, the following is a simple fact which states the type of an edge.

not_important(a2).

There are four possibilities boundary conditions. An edge can be

- free
- fixed on one side
- fixed on two sides
- fixed completely

There are similar types for loads. Edges can have

- no loading

- one side loaded
- two sides loaded
- continuous loading

There is a possibility that an edge is continued and one or two sides loaded at the same time, which means that some edges may be described with two "load" predicates.

The geometric representation is needed because of the relationships between edges. We think that the most important relations are "neighbour" and "opposite". In other words we believe that the edges which are neighbours or which are opposite influence each other in the FE mesh. The third relation which is interesting is that some edges are not only opposite but also have the same length or form. For instance concentric circles have the same form. Such pairs of edges we have described using the predicate "same". All three predicates have two arguments. Each argument (edge) can be input or output, which means for example that if A is a neighbour to B then B is also a neighbour to A. Using these predicates we have described the FE meshes for three different training structures. The sizes of input data for Golem were

- 75 foreground examples
- 618 negative examples
- 588 background facts

# 4 Results

## 4.1 Learning using the complete set of examples

We ran Golem with the described data several times. However some parameters varied with each Golem run.

First of all we knew that even an FE mesh chosen after several trials is not perfect so we expected some of the data to be "noisy". Therefore we allowed the rules induced by Golem to cover in one case a maximum of three negative examples and in another case only one negative example. We also tried an assumption of no noise.

We ran Golem with and without the ability to produce recursive rules. When recursive rules were not allowed more rules were derived for exact numbers of elements than in the case in which recursive rules were allowed.

Golem induced 56 different rules. All of them we believe to be correct. Some of them cover only two foreground examples, some of them more than ten. But from a practical point of view some of these rules are not useful. For example rule

```
mesh(A,1) :-
    neighbour(A,B),
    neighbour(A,C),
    cont_loaded(C),
    fixed(B),
    not_loaded(B).
```

is not useful, since it does not say anything about what kind of edge A is. Another example of a useless rule is

```
mesh(A,3) :- opposite(A,a3).
```

The rule is true for two training examples but it can't be used in practice for other edges since only two edges are opposite edge a3.

There were 26 useful rules. 20 of them were unrecursive and six rules were recursive. Some of the rules were more general than others. So we had to eliminate some of them. We chose to retain the rules which covered the most foreground examples. The following is a rule for one element

```
mesh(A,1) :-
    not_important(A),
    not_loaded(A).
```

which covers 18 foreground examples and no negative examples. But when we allowed some negative cover the induced rule was

```
mesh(A,1) :- not_important(A).
```

which covers 27 foreground examples and only one negative example. It is obvious that this one example is debatable so we accepted the second rule, treating this example as noise.

A similar situation occurred with recursive rules. The following rule covered 10 foreground examples and one negative

```
mesh(A,B) :-
    same(A,C),
    con_loaded(C),
    mesh(C,B).
```

The rule does not say anything about edge A. However it does say that since A and C are both opposite and have the same form they should have the same number of FEs. This is because the pressure on edge C influences edge A. We chose this rule in preference to the rule

```
mesh(A,B) :-
    cont_loaded(A),
    same(A,C),
    cont_loaded(C),
    mesh(C,B).
```

which is more restrictive and covers only 6 foreground examples. We also know from experience that there is no need for both edges to be loaded for them to have the same number of FEs.

After eliminating rules there were 13 unrecursive and two recursive rules left. In the foreground examples we had examples for 1,2,3,4,5,7,8,9 and 12 elements on an edge. There were too few examples for Golem to induce rules for 4 and 5 elements.

## 4.2 Cross-validation

In order to estimate the accuracy of the rules obtained and described in the previous subsection we carried out a cross-validation test. In this test we removed a random subset containing 10% (7 examples) from the training examples and used these for testing the resulting rules. Training and testing was carried out twice with different random 10% test samples removed from the training sample each time. In the first case 6 out of the 7 test instances were still correctly predicted, i.e.

the induced rules were 86% correct. In the second experiment 5 out of the 7 test instances were correctly predicted, i.e. the resultant rules were 71% correct. The average result is thus 79%, though as shown by the discrepancy between the first and second test, this may have a rather large standard deviation. To accurately assess the standard deviation would require many more runs.

# 5 Conclusion

Rules derived by Golem can be used as a knowledge base for an expert system which would help in the design of FE meshes. We can use the rules simply as a PROLOG program and use PROLOG as an expert system shell. We use PROLOG to "consult" the background facts and the rules. We can then ask PROLOG about the number of FEs on edges.

We carried out this experiment using training structures. Clearly, when using the complete set of data and ignoring noise, the rules agreed with the training set. In addition, by use of cross-validation techniques we found that the rules have an expected accuracy of around 79% on unseen data. In order to apply these rules in real world domains a higher accuracy in excess of 90% should be achieved. This will require larger amounts of data and further testing.

Generally, we believe that derivation of the knowledge base using automatic learning is the correct approach for this domain. For practical use there have to be more rules for predicting the number of elements, which means that more training examples should be given to the system.

Finite element methods are used by thousands of engineers all over the world. Mesh design is a major bottleneck in this process. We believe that extensions of our approach should have very wide and far-ranging applications.

**Acknowledgements.** This work was supported partly by the Esprit Ecoles project 3059 and an SERC Post-graduate fellowship held by Stephen Muggleton.

# References

[1] A. Jezernik, S. Gorosnik, and S. Bader. Deformation and stress analysis of cylinder F30 using fem, Final report. Technical report, Faculty of Technical Sciences, Maribor, Yugoslavia, 1989.

[2] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4), (To Appear) 1991.

[3] S.H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.

[4] M. Oblak, A. Jezernik, and J. Ducep. Deformation and stress analysis of paper mill for the Sladogorska Company, Final report. Technical report, Faculty of Technical Scineces, Maribor, Yuoslavia, 1986.

[5] O.C. Zienkiewicz and R.L. Taylor. Basic formulation and linear problems. In *The Finite Element Method*, volume 1. McGraw-Hill, London, 1988.

# A Foreground examples

```
% a – cylinder

mesh(a1,17).
mesh(a2,1).
mesh(a3,8).
mesh(a4,1).
mesh(a5,1).
mesh(a6,2).
mesh(a7,1).
mesh(a8,1).
mesh(a9,3).
mesh(a10,1).
mesh(a11,3).
mesh(a12,1).
mesh(a13,1).
mesh(a14,1).
mesh(a15,4).
mesh(a16,1).
mesh(a17,2).
mesh(a18,1).
mesh(a19,4).
mesh(a20,1).
mesh(a21,1).
mesh(a22,2).
mesh(a23,2).
mesh(a24,1).
mesh(a25,2).
mesh(a26,1).
mesh(a27,1).
mesh(a28,2).
mesh(a29,1).
mesh(a30,1).
mesh(a31,3).
mesh(a32,2).
mesh(a33,2).
mesh(a34,11).
mesh(a35,1).
mesh(a36,12).
mesh(a37,12).
mesh(a38,12).
mesh(a39,5).
mesh(a40,2).
mesh(a41,1).
mesh(a42,5).

% b – hook

mesh(b1,9).
mesh(b2,1).
mesh(b3,2).
mesh(b4,7).
mesh(b5,1).
mesh(b6,1).
mesh(b7,1).
mesh(b8,9).
mesh(b9,1).
mesh(b10,2).
mesh(b11,7).
mesh(b12,1).
mesh(b13,1).
mesh(b14,1).

% c – paper mill

mesh(c1,1).
mesh(c2,2).
mesh(c3,1).
mesh(c4,1).
mesh(c5,3).
mesh(c6,2).
mesh(c7,2).
mesh(c8,3).
mesh(c9,1).
mesh(c10,2).
mesh(c11,1).
mesh(c12,2).
mesh(c13,1).
mesh(c14,2).
mesh(c15,8).
mesh(c16,8).
mesh(c17,8).
mesh(c18,8).
mesh(c19,8).
mesh(c20,8).
mesh(c21,8).
```

# B Negative examples

If we consider that we have two sets:

- set of the names of the edges
  $\mathrm{E} = \{\mathrm{a1},\ldots,\mathrm{a42}, \mathrm{b1},\ldots,\mathrm{b14}, \mathrm{c1},\ldots,\mathrm{c21}\}$
- set of possible numbers of finite elements
  $\mathrm{N} = \{1, 2, 3, 4, 5, 7, 8, 9, 12\}$

and that we have also set of foreground examples:

$$\mathrm{F} = \{\ \ldots,\ \mathrm{mesh}\,(\mathrm{a2}, 1),\ \mathrm{mesh}\,(\mathrm{a3}, 8),\ \mathrm{mesh}\,(\mathrm{a4}, 1),\ \ldots\,\}$$

the set of negative examples can be constructed as:

$$\{\,\mathrm{mesh}(\ \mathrm{Edge},\ \mathrm{Number}\ )\mid \mathrm{Edge} \in \mathrm{E},\ \mathrm{Number} \in \mathrm{N}\,\} - \mathrm{F}$$

For instance:

```
mesh(a1,1).
mesh(a3,1).
mesh(a6,1).
...
...
...
mesh(c19,12).
mesh(c20,12).
mesh(c21,12).
```

# C Background facts

% DECLARATIONS

```
mode(important_long(-1)).
mode(important(-1)).
mode(important_short(-1)).
mode(circuit(-1)).
mode(half_circuit(-1)).
mode(short_for_hole(-1)).
mode(long_for_hole(-1)).
mode(circuit_hole(-1)).
mode(half_circuit_hole(-1)).
mode(not_important(-1)).
mode(free(-1)).
mode(one_side_fixed(-1)).
mode(two_side_fixed(-1)).
mode(fixed(-1)).
mode(not_loaded(-1)).
mode(one_side_loaded(-1)).
mode(cont_loaded(-1)).
mode(neighbour_xy_r(-1,1)).
mode(neighbour_yz_r(-1,1)).
mode(neighbour_zx_r(-1,1)).
mode(neighbour_xy_l(1,-1)).
mode(neighbour_yz_l(1,-1)).
mode(neighbour_zx_l(1,-1)).
mode(opposite_r(-1,1)).
mode(opposite_l(1,-1)).
mode(same_r(-1,1)).
mode(same_l(1,-1)).
mode(mesh(-1,1)).

determ(mesh(_,_),important_long(_)).
determ(mesh(_,_),important(_)).
determ(mesh(_,_),important_short(_)).
determ(mesh(_,_),circuit(_)).
determ(mesh(_,_),half_circuit(_)).
determ(mesh(_,_),short_for_hole(_)).
determ(mesh(_,_),long_for_hole(_)).
determ(mesh(_,_),circuit_hole(_)).
determ(mesh(_,_),half_circuit_hole(_)).
determ(mesh(_,_),not_important(_)).
determ(mesh(_,_),free(_)).
determ(mesh(_,_),one_side_fixed(_)).
determ(mesh(_,_),two_side_fixed(_)).
determ(mesh(_,_),fixed(_)).
determ(mesh(_,_),not_loaded(_)).
determ(mesh(_,_),one_side_loaded(_)).
determ(mesh(_,_),cont_loaded(_)).
determ(mesh(_,_),neighbour_xy_r(_,_)).
determ(mesh(_,_),neighbour_yz_r(_,_)).
determ(mesh(_,_),neighbour_zx_r(_,_)).
determ(mesh(_,_),neighbour_xy_l(_,_)).
determ(mesh(_,_),neighbour_yz_l(_,_)).
determ(mesh(_,_),neighbour_zx_l(_,_)).
determ(mesh(_,_),opposite_r(_,_)).
determ(mesh(_,_),opposite_l(_,_)).
determ(mesh(_,_),same_r(_,_)).
determ(mesh(_,_),same_l(_,_)).
determ(mesh(_,_),mesh(_,_)).
```

% TYPES OF THE EDGES

```
important_long(a1).
important_long(a34).
important_long(b1).
important_long(b8).

important(a3).
important_short(a9).
important_short(a11).
important_short(a11).

important(a39).
important(b4).
important(b11).
important(c2).
important(c5).
important(c6).
circuit(c18).
circuit(c19).
circuit(c19).

important(c8).
important(c10).
important(c12).
important(c14).

important_short(a6).
not_important(a12).
not_important(a14).
not_important(a14).
```

important_short(a13).
important_short(a15).
important_short(a19).
important_short(a22).
important_short(a23).
important_short(a25).
important_short(a26).
important_short(a28).
important_short(a31).
important_short(a33).
important_short(a35).
important_short(a40).
important_short(b3).
important_short(b10).
important_short(c4).
important_short(c7).
important_short(c13).

circuit(c15).
circuit(c16).
circuit(c17).
circuit(c20).

half_circuit(a36).
half_circuit(a37).

short_for_hole(a16).
short_for_hole(a18).

long_for_hole(a17).

circuit_hole(c21).

half_circuit_hole(a38).
half_circuit_hole(a42).

not_important(a2).
not_important(a4).
not_important(a5).
not_important(a7).
not_important(a8).
not_important(a10).
not_important(a20).
not_important(a21).
not_important(a24).
not_important(a27).
not_important(a29).
not_important(a30).
not_important(a32).
not_important(a41).
not_important(b2).
not_important(b5).
not_important(b6).
not_important(b7).
not_important(b9).
not_important(b12).
not_important(b13).
not_important(b14).
not_important(c1).
not_important(c3).
not_important(c9).
not_important(c11).

% BOUNDARY CONDITIONS

free(a39).
free(a40).
free(b2).
free(b3).
free(b7).
free(b9).
free(b10).
free(b14).
free(c6).
free(c7).
free(c11).
free(c12).
free(c13).
free(c17).
free(c18).

one_side_fixed(a34).
fixed(a13).
fixed(a14).
fixed(a15).
fixed(a16).
fixed(a17).
one_side_fixed(a35).
one_side_fixed(a41).
one_side_fixed(b1).
one_side_fixed(b4).
one_side_fixed(b8).
one_side_fixed(b11).
one_side_fixed(c2).
one_side_fixed(c3).
one_side_fixed(c5).
one_side_fixed(c8).
one_side_fixed(c10).
one_side_fixed(c14).

two_side_fixed(a36).
two_side_fixed(a37).
two_side_fixed(a38).
two_side_fixed(a42).
fixed(a23).
fixed(a24).
fixed(a25).
fixed(a26).
fixed(a27).
two_side_fixed(b5).
two_side_fixed(b6).
two_side_fixed(b12).
two_side_fixed(b13).

fixed(a1).
fixed(a2).
fixed(a3).
fixed(a4).
fixed(a5).
fixed(a6).
fixed(a7).
fixed(a8).
fixed(a9).
fixed(a10).
fixed(a11).
fixed(a12).
fixed(a33).
fixed(c1).
fixed(c4).
fixed(c9).
fixed(c15).

fixed(a18).
fixed(a19).
fixed(a20).
fixed(a21).
fixed(a22).
fixed(a28).
fixed(a29).
fixed(a30).
fixed(a31).
fixed(a32).
fixed(c16).
fixed(c19).
fixed(c20).
fixed(c21).

% LOADS

not_loaded(a1).
not_loaded(a2).
not_loaded(a3).
not_loaded(a4).
not_loaded(a5).
not_loaded(a6).
not_loaded(a7).
not_loaded(a23).
not_loaded(a24).
not_loaded(a25).
not_loaded(a26).
not_loaded(a27).
not_loaded(a28).
not_loaded(a29).
not_loaded(a33).
not_loaded(a36).
not_loaded(a42).
not_loaded(b1).
not_loaded(b2).
not_loaded(b5).
not_loaded(b6).
not_loaded(b7).
not_loaded(b8).
not_loaded(b9).
not_loaded(b12).
not_loaded(b13).
not_loaded(b14).
not_loaded(c1).
not_loaded(c2).
not_loaded(c3).
not_loaded(c4).
not_loaded(c5).
not_loaded(c6).
not_loaded(c7).
not_loaded(c8).
not_loaded(c9).
not_loaded(c15).
not_loaded(c20).
not_loaded(c21).

one_side_loaded(a34).
one_side_loaded(a35).
one_side_loaded(a40).
one_side_loaded(a41).
one_side_loaded(b3).
one_side_loaded(b4).
one_side_loaded(b10).
one_side_loaded(b11).

cont_loaded(a8).
cont_loaded(a9).
cont_loaded(a10).
cont_loaded(a11).
cont_loaded(a12).
cont_loaded(a13).
cont_loaded(a14).
cont_loaded(a15).
cont_loaded(a16).
cont_loaded(a17).
cont_loaded(a18).
cont_loaded(a19).
cont_loaded(a20).
cont_loaded(a21).
cont_loaded(a22).
cont_loaded(a30).
cont_loaded(a31).
cont_loaded(a32).
cont_loaded(a37).
cont_loaded(a38).
cont_loaded(a39).
cont_loaded(c10).
cont_loaded(c11).
cont_loaded(c12).
cont_loaded(c13).
cont_loaded(c14).
cont_loaded(c16).
cont_loaded(c17).
cont_loaded(c18).
cont_loaded(c19).

% GEOMETRIC REPRESENTATION

neighbour_xy_r(a34,a35).
neighbour_xy_r(a35,a26).
neighbour_xy_r(a26,a36).
neighbour_yz_r(c20,c2).
neighbour_yz_r(c21,c4).
neighbour_zx_r(b5,b1).
neighbour_zx_r(b8,b9).
neighbour_zx_r(b9,b10).

neighbour_xy_r(a36,a4).
neighbour_xy_r(a4,a34).
neighbour_xy_r(b1,b13).
neighbour_xy_r(b13,b8).
neighbour_xy_r(b8,b7).
neighbour_xy_r(b7,b1).
neighbour_xy_r(b4,b6).
neighbour_xy_r(b6,b11).
neighbour_xy_r(b10,b14).
neighbour_xy_r(b14,b3).
neighbour_xy_r(c15,c9).
neighbour_xy_r(c16,c9).
neighbour_xy_r(c17,c11).
neighbour_xy_r(c12,c17).
neighbour_xy_r(c18,c12).
neighbour_xy_r(c13,c18).
neighbour_xy_r(c19,c1).
neighbour_xy_r(c20,c1).
neighbour_xy_r(c21,c3).

neighbour_yz_r(a39,a41).
neighbour_yz_r(a40,a39).
neighbour_yz_r(a35,a40).
neighbour_yz_r(a25,a35).
neighbour_yz_r(a42,a25).
neighbour_yz_r(a24,a42).
neighbour_yz_r(b5,b6).
neighbour_yz_r(b6,b12).
neighbour_yz_r(b12,b13).
neighbour_yz_r(b13,b5).
neighbour_yz_r(b2,b7).
neighbour_yz_r(b7,b9).
neighbour_yz_r(b9,b14).
neighbour_yz_r(b14,b2).
neighbour_yz_r(c15,c8).
neighbour_yz_r(c16,c10).
neighbour_yz_r(c19,c14).
opposite_r(b11,b8).
opposite_r(c6,c12).
opposite_r(c2,c14).
opposite_r(c10,c14).
opposite_r(c15,c16).
opposite_r(c16,c17).
opposite_r(c17,c18).
opposite_r(c18,c19).
opposite_r(c19,c20).
opposite_r(c20,c21).

neighbour_zx_r(a1,a2).
neighbour_zx_r(a2,a3).
neighbour_zx_r(a3,a4).
neighbour_zx_r(a4,a5).
neighbour_zx_r(a5,a6).
neighbour_zx_r(a6,a7).
neighbour_zx_r(a7,a8).
neighbour_zx_r(a8,a9).
neighbour_zx_r(a9,a10).
neighbour_zx_r(a10,a11).
neighbour_zx_r(a11,a12).
neighbour_zx_r(a12,a13).
neighbour_zx_r(a13,a14).
neighbour_zx_r(a14,a15).
neighbour_zx_r(a15,a16).
neighbour_zx_r(a16,a17).
neighbour_zx_r(a17,a18).
neighbour_zx_r(a18,a19).
neighbour_zx_r(a19,a20).
neighbour_zx_r(a20,a21).
neighbour_zx_r(a21,a22).
neighbour_zx_r(a22,a23).
neighbour_zx_r(a23,a24).
neighbour_zx_r(a24,a1).
neighbour_zx_r(a25,a26).
neighbour_zx_r(a26,a27).
neighbour_zx_r(a27,a28).
neighbour_zx_r(a28,a29).
neighbour_zx_r(a29,a30).
neighbour_zx_r(a30,a31).
neighbour_zx_r(a31,a32).
neighbour_zx_r(a32,a33).
neighbour_zx_r(a33,a25).
neighbour_zx_r(b1,b2).
neighbour_zx_r(b2,b3).
neighbour_zx_r(b3,b4).
neighbour_zx_r(b4,b5).
same_r(a33,a23).
same_r(a36,a37).
same_r(a38,a37).
same_r(a39,a42).
same_r(b1,b8).
same_r(c6,c12).
same_r(c2,c14).
same_r(c10,c14).
same_r(c15,c16).
same_r(c16,c17).

neighbour_zx_r(b10,b11).
neighbour_zx_r(b11,b12).
neighbour_zx_r(b12,b8).
neighbour_zx_r(c1,c2).
neighbour_zx_r(c2,c3).
neighbour_zx_r(c3,c4).
neighbour_zx_r(c4,c5).
neighbour_zx_r(c5,c6).
neighbour_zx_r(c6,c7).
neighbour_zx_r(c7,c8).
neighbour_zx_r(c8,c9).
neighbour_zx_r(c9,c10).
neighbour_zx_r(c10,c11).
neighbour_zx_r(c11,c12).
neighbour_zx_r(c12,c13).
neighbour_zx_r(c13,c14).
neighbour_zx_r(c14,c1).

opposite_r(a11,a3).
opposite_r(a9,a3).
opposite_r(a31,a25).
opposite_r(a13,a1).
opposite_r(a15,a1).
opposite_r(a17,a1).
opposite_r(a19,a1).
opposite_r(a22,a1).
opposite_r(a23,a1).
opposite_r(a32,a22).
opposite_r(a33,a23).
opposite_r(a34,a37).
opposite_r(a36,a37).
opposite_r(a38,a37).
opposite_r(a39,a42).
opposite_r(b1,b8).
opposite_r(b3,b1).
opposite_r(b4,b1).
opposite_r(b10,b8).
same_r(c17,c18).
same_r(c18,c19).
same_r(c19,c20).
same_r(c20,c21).

Because the arguments in the relations "neighbour", "opposite" and "same" can be output or input, we also need to have the following predicates defined in the background facts:

neighbour_xy_l
neighbour_yz_l
neighbour_zx_l
opposite_l
same_l

which have exactly the same arguments as the predicates with suffix "_r". Only the order of input and output is changed using "mode" declarations.

# D Rules derived by Golem

Because of the restriction for hypotheses constructed by Golem (Section 2) we introduced three different suffixes for predicate neighbour: "_xy", "_yz" and "_zx" (Appendix C) and divided the problem on three plains in xyz space. When applying the rules, the suffixes "_r" and "_l" are not necessary. Therefore we deleted them from the rules derived by Golem.

```
mesh(A,1) :- not_important(A).

mesh(A,1) :- short_for_hole(A).

mesh(A,2) :-
    important_short(A),
    neighbour(A,B),
    fixed(B),
    not_loaded(B).

mesh(A,2) :-
    important_short(A),
    neighbour(B,A),
    not_important(B),
    not_loaded(B).

mesh(A,2) :-
    important_short(A),
    free(A),
    neighbour(B,A),
    not_loaded(B).

mesh(A,2) :-
    free(A),
    one_side_loaded(A).

mesh(A,2) :-
    important(A),
    opposite(A,B),
    important(B).
```

```
mesh(A,3) :-
    important(A),
    not_loaded(A),
    neighbour(B,A),
    important_short(B).

mesh(A,3) :-
    important_short(A),
    cont_loaded(A),
    neighbour(A,B),
    not_important(B),
    neighbour(C,A),
    C \== B,
    not_important(C).

mesh(A,7) :-
    important(A),
    one_side_loaded(A).

mesh(A,8) :- circuit(A).

mesh(A,9) :-
    important_long(A),
    one_side_fixed(A),
    not_loaded(A).

mesh(A,12) :- half_circuit(A).
```

```
mesh(A,B) :-
    same(A,C),
    cont_loaded(C),
    mesh(C,B).
```

```
mesh(A,B) :-
    not_loaded(A),
    same(A,C),
    mesh(C,B).
```

In the second rule for three elements both neighbour predicates had different suffixes before we deleted them. Insead of suffixes we added declaration "C \== B" to make clear that C and B

are not the same edge.

If we want to use the rules without suffixes with the set of background facts we need the following additional rules for geometric representation.

```
neighbour(A,B) :-
    neighbour_xy_d(A,B);
    neighbour_xy_d(B,A);
    neighbour_yz_d(A,B);
    neighbour_yz_d(B,A);
    neighbour_zx_d(A,B);
    neighbour_zx_d(B,A).

opposite(A,B) :-
    opposite_d(A,B);
    opposite_d(B,A).

same(A,B) :-
    same(A,B);
    same(B,A).
```