An Expert System for Raising Pigs

Tim Menzies
Artificial Intelligence Lab, University of New South Wales, Australia
School of Computer Science and Engineering,
University of New South Wales, P.O. Box 1, Kensington, NSW, Australia, 2033.
timm@spectrum.cs.unsw.oz.au.

John Black, Joel Fleming
DSL Systems Centre, CSIRO Division of Animal Production, Australia
Commonwealth Scientific Industrial Research Organisation.

Murray Dean
Zwiggan Consulting, Australia

1. INTRODUCTION

An intelligent back-end to a DOS-based mathematical modelling package written in ARITY PROLOG is described. The model:
- simulates the growth and reproduction of pigs;
- identifies factors that limit optimal performance of the pig;
- identifies management strategies that maximise enterprise profit.

The expert system:
- presents an abstracted description of the output of the model in a form that a non-mathematician can understand;
- suggests dietary, housing, genotype, or resource input changes that can improve the profitability of the herd.

Verification studies have demonstrated that the expert system can significantly out-perform human experts interpreting the output of the model (performance measured in dollars per square metre per day). In a usual case, the improvement is of the order of 10%. The system is in routine use in America, Holland, Belgium, France, Spain and Australia.

2. THE DOMAIN

Raising pigs is big business. In Australia alone, 300 kilotonnes of pig meat worth some $500 million dollars is produced annually. The Australian pig herd represents one-third of one percent of the international herd.

In response to commercial interest in pig production, CSIRO's Division of Animal Production designed *AUSPIG*, an MS-DOS tool for studying the performance of pigs growing in pens in a piggery. The pig farmer enters information concerning the pigs' diet, genotype, and housing, as well as certain data concerning the local buyers of pigs. *AUSPIG*'s grower pig and breeder pig models then simulate the growth of the pig and report its economic and biological characteristics (e.g. live weight, backfat thickness, amino acid utilisation, profit at sale) at various times during its life [Black 87a].

As well as being useful for studying the pig's performance, *AUSPIG* can be used for experimenting with different ways of improving the profit of a piggery. Once a growing regime has been developed to optimise the growth of one pig, *AUSPIG*'s *PIGMAX* optimisation model can be used to study the effects of this regime on the profitability of the entire herd. Typically, when analysising a piggery, an initial simulation run is made to ensure that the performance of the pigs under their current environment is predicted accurately. If inefficiencies in the pig's diet or environment are detected, a set of potentially beneficial alterations are identified. These alterations are tested by a second run of the *AUSPIG* and/or the *PIGMAX* model. The results of the first and second runs are then compared to see if the tested alterations were successful or if they could be further improved. This process of:

```
analysis => modification => test
```

could repeat any number of times. Experience has shown, however, that there is little utility in repeating the process any more than about five times.

Using *AUSPIG*, it is possible to identify simple changes to the pig's diet, feeding regimes, or environment that can significantly increase the performance and profitability of the herd. For example:
- In one spectacularly successful study of a piggery, CSIRO scientists used *AUSPIG* to develop a growing regime that increased a piggery's net return from $40,992 to

$202,321 (i.e. 496 percent) [Black 88].

- *AUSPIG* demonstrated that increasing the flow of air over a pig in a hot shed (e.g. 32 degrees Celcius) from 0.2 metres per second to 0.6 metres per second can double the animal's growth rate [Black 87b]. Such an increase in air flow would be barely perceptible to a human being.

The potential of the *AUSPIG* system is enormous. The current system has an international market. Some modern piggeries have installed computer-controlled liquid feeding systems for their pigs. Such piggeries can alter a pig's diet every day of the pig's life. With such systems, the challenge is to decide how to best capitalise on this control and systems like *AUSPIG* are essential. Further, the techniques developed for *AUSPIG* could be generalised to produce an *AUS-COW*, *AUS-FARM*, and a *AUS-SHEEP* system.

A problem identified with the prototype versions of *AUSPIG* was that it required an expert to fully utilise the output of the mathematical models. Early versions of *AUSPIG* produced dozens of screens reports which the end-user browsed to find ways to optimise the pig growth. Each report was a 20 by 40 array of numbers filling many screens. Trials showed that most users examined few of these reports. In order to make the system more acceptable to the average pig farmer, and to increase its market potential, it required an intelligent post-processor to analyse the output. Ideally, the non-expert user should be able to by-pass the screens of reports and access a single screen that identifies any biological inefficiencies and recommends changed management strategies. The implemented post-processor was called *PigE*[1].

3. WHY PROLOG?

The choice of PROLOG as the expert system implementation language was made after a consideration of the required delivery environment.

- CSIRO's primary concern was to keep the cost of the *AUSPIG* package to a minimum. *AUSPIG* already contained third-party software which could only be distributed under licence. CSIRO did not want to use a commercial expert system shell that would attract more licensing fees and so inflate the cost of the package.
- When the expert system was first considered, *AUSPIG* was targetted for a IBM-AT or AT-clone running MS-DOS[2]. An AT configured for *AUSPIG* leaves 420K of RAM free for other processes. The expert system would need to be able to execute within this memory constraint.
- CSIRO wanted to allow for the extension of any software written during the feasibility study. CSIRO's *AUSPIG* development team includes programmers and it was hoped that this programming team could continue the development of the expert system after the feasibility study.
- The early version of the system was developed using a prototyping methodology. Initially, we were not sure how to reason about the system. The developers wanted a high-level symbolic-processing language that could be easily modified to handle a variety of inferencing techniques.

After a survey of the available tools, PROLOG was selected as the high-level symbolic processing language. At the time of the initial prototype, it was known that certain DOS-based PROLOGs supported the generation of stand-alone executables (i.e. no runtime license fees) as well as memory management tools for DOS. A major factor in the decision was the available programming skills. Menzies and Dean both had extensive PROLOG programming experience.

4. DEVELOPMENT

At first, a rapid prototyping methodology was adopted. The two PROLOG developers (Menzies & Dean) would work on the examples prepared by the experts (Black & Fleming). Rules were written, initially on paper, and a notation was developed that reflected the native idiom of the domain experts (a standard application langauge development approach [Bustany 88]). In between the sessions, the experts would work on further examples or attempt to write rules. The PROLOG developers would work on an interpreter for the notation. Whenever the PROLOG developers recognized that the domain-experts where having to contort the expression of the logic because of limitations with the interpreter, the interpreter was altered to allow for a more

---

1. The running gag of the project was the name *PigE*. *AUSPIG* could be called a **M**anagement **I**nformation **S**ystem so the expert system could have been called *MISs PigE*. The file transfer utility *KERMIT* could have been used to move files from developer to developer. And so on.

---

2. This has now changed. *AUSPIG* runs on a 386 DOS box with 2MB of RAM.

natural expression of the logic.

At the end of the prototype period, the domain experts could write rules and the rules could execute. Verification studies (see section 7) were most encouraging. Testing revealed that the system consistently out-performed the human experts[3]. This "over-performance" of the system is remarkable. We conjecture that a human expert must look at many numbers from the model before making a recommendation. The amount of information that has to be processed is large and it appears to exceed the capacity of human beings. An expert system has no such limitations to its short-term memory. The expert system can therefore successfully study more factors than a human being. Hence the superior performance of the expert system.

Funding was obtained for a continuation of the project. The utility of continuing with PROLOG was reviewed. It was decided:

1. To stay with PROLOG while the specification was under development.
2. To move away from PROLOG if the runtimes became unacceptably slow. This has not happened. The PROLOG reasoning executes at least as fast as the mathematical models so the user doesn't perceive the expert system as being the slowest part of the system[4]. Also, during the reasoning, interim conclusions are displayed to the user. Hence, the user is not left with a blank screen during the reasoning.
3. To move away from the in-house PROLOG used in the prototype to ARITY PROLOG. With ARITY, we gained:
   - Memory management under DOS. Our PROLOG programs were no longer bound to the 640K DOS limit.
   - Screen management tools. These screen tools proved to be less-than-ideal (see section).

   A interface was developed between ARITY and PASCAL to simplify data transfer and allow for the seamless embedding of the expert system into the *AUSPIG* package.

Using the experience gained during the prototype, it was possible to estimate the times required to build the remaining system. A project plan was developed for the next year's work. The rule set was extended from 72 rules to 486 rules and is currently stable.

The system was developed and fielded using PROLOG. Occasionally, some thought was given to re-coding the rules in PASCAL. However, other enhancements were considered to be of higher priority than a rewrite of a sub-system that was performing adequately. Currently, *AUSPIG/PigE* is in use in Australia, Holland, Belgium, France, Spain, and America.

The domain experts report that maintaining the system is not a major problem. Experts can access reports generated by the inference engine on rules that never fire or rules that fire too much. Hence, they have some automated support for detecting inappropriate logic. The 486 rules divide up into 9 knowledge bases (KB). Each KB consists of several rule groups of between 5 to 15 rules. Within a group, assertions can be made that other rules require. Between rule groups, there exists a coarse-grain level of communication (e.g. some areas of management are not altered until others are resolved). However, in general, rules remained modular chunks of knowledge and can be edited freely without unwanted side-effects to the rest of the system[5].

5. SHELL DESIGN

The domain is data-driven so a forward chaining rule interpreter was written to process the expert's rules. As a technique for allowing the succinct expression of generic queries and meta-level logic, *PigE's* rules support variables. Rule conditions can be instantiated in numerous ways and the associated rule actions are applied for each different instantiation of the condition. All the source code for the project was written especially for CSIRO and CSIRO's programmers were trained in the maintenance of this code. Where ever possible, procedural attachments were used to implement commonly performed tasks. Such procedural attachments could be called from the RHS or LHS of a rule.

---

3. The developers were somewhat alarmed at this finding. Would the experts take offence at the system beating them at their own game? Fortunately, the experts' pride in their rule-writing ability out-weighed any other considerations.

4. To the user, data entry is the most tedious and slowest part of using the system.

---

5. This is in marked contrast with the rest of the *AUSPIG* system. The maintainers of the mathematics models are somewhat envious of the rule system. No portion of the models can be changed without extensively effecting the rest of the model.

At start up, *PigE* converts the *AUSPIG* report files into Prolog facts. For example, if three lines in the first grower report where:

```
32 11.0 462 425 4.2 1.75 1.71 18 28 38
39 14.5 535 465 7.3 1.89 1.78 17 27 37
45 18.0 606 498 9.4 1.98 1.84 16 27 37
```

then *PigE* would load in the Prolog facts:

```
g1(32,11.0,462,425,4.2,1.75,1.71,18,28,38).
g1(39,14.5,535,465,7.3,1.89,1.78,17,27,37).
g1(45,18.0,606,498,9.4,1.98,1.84,16,27,37).
```

PigE refers to these files via a set of database selectors defined using a data dictionary. The data dictionary for *g1* gives each column in this report file a succinct label. For example, Figure One shows the data dictionary for report *g1* defines "live weight (kg)" as *lwt*.

```
dd(g1, [
want    age : "age (days)",
want    lwt : "live weight (kg)",
        lwt_gain :
            "live weight gain (g/day)",
want    lwt_av_gain :
            "average daily gain (g/day)",
want    p2 : "backfat thickness (mm)",
want    fcr : "fat conversion ratio: fcr",
want    fcr_av : "average fcr",
want    lct : "lct (degrees C)",
want    ect : "ect (degrees C)",
        uct : "uct (degrees C)"
]).
```

Selectors are automatically generated by the shell for each *want*-ed column. The use of selectors makes calls to the database easier and insulates programs from database changes. If programs talk to the database via the selectors, then queries to the database can be made without having to remember how many parameters each clause has and without a lot of tedious typing. Also, if the database structure is changed, or more selectors are required, then the data dictionary is edited and the selectors generation program is called again. Any programs that access this database via the selectors would not need changing.

To alter the rule set, CSIRO's experts edited an ASCII file storing special Prolog facts. These facts are entered using pseudo-English Prolog operators. For example, if browsing this ASCII file, the domain expert could read the rule shown in Figure One.

Line 1 of Figure One is the rule label. The rule label has a unique identification number and a brief synopsis of the function of the rule. Lines 2 to 7 are the rule condition. The keyword *if* indicates that this rule condition can only be satisfied once. If this keyword was replaced by *forall*, then *PigE* would attempt to find all the different instantiations of the variables that

```
rule # 6000 - 'use temperature option ?'

if      climate(off) and           % 2
        lower_piggery_temp := L and  % 3
        upper_piggery_temp := U and  % 4
        lct is (LCT,ROW)and         % 5
        ect is (ECT,ROW)and         % 6
        (L < LCT or U > ECT)        % 7
then
        observe
        "When next you run the model,
        turn on the temperature option." %8
        and zap pig is ok.           % 9
```

Figure 1. A sample rule.

satisfy this rule condition. The rule action would be applied for each different instantiation. Variables set in the condition can be accessed by the rule action. Rule conditions and actions consist of executable Prolog code separated by the keywords *and, or* and *not*.

Rule number 6000 tests to see if the user should have run the *AUSPIG* model with the climate response option turned on. If this option is on, the model studies the pig's growth hour by hour instead of day by day. This increases execution time of the system considerably, but is useful for investigating pigs under hot or cold conditions. After testing that the climate option is off, lines 3 to 6 gather the information required to test if the pig was under stress. Lines 3 and 4 ask the user for the extremes of temperature in the piggery. The procedure *X := Y* uses a *question* fact to generate a question to the user; e.g.:

```
question(
  lower_piggery_temp,
  "How cold does your piggery get ? ",
  0 to 50
).
```

The user is prompted with the text shown here and is pestered until they provide a numeric value in the range 0 to 50.

The actual logic of the rule is in line 7. The temperature option should be turned on if the lower piggery temperature is less than the pig's lower critical temperature (the point at which the pig is forced to increase its energy expenditure by shivering) or if the upper piggery temperature is greater than the pig's evaporate critical temperature (the point at which the pig starts panting).

The procedure *zap X* (line 9 of *rule # 6000*) removes an assertion made at start up. Assertions are made by a *++ X* rule action. These assertions can be tested for by a call to *{ X }*. At start up, *++ pig is ok* is asserted. Certain rules test for *not { pig is ok}*. If this is true, i.e. a rule

like *rule # 6000* has fired, then certain remedies are explored. *PigE* tests its rules in a top-down manner. This allows for a prioritisation of possible remedies. The rules are ordered in such a way that important problems are resolved first, regardless of their consequences on less important problems.

6. SYSTEM VALIDATION

The primary motivation for attempting an expert system was to make the system more accessible to the average user. Hence, an assessment criteria was chosen that reflected the perspective of such a user.

PigE is assessed according to the speed at which it reaches a conclusion and how much its recommendations can improve the profit per pig. *Speed* is defined in terms of the number of runs of the model required to eliminate biological inefficiencies. *Profit* is defined as dollars per pig per day (if the model's housing option is off) or as dollars per square metre per day (if the housing option is on). By graphing profit vs runs for both *PigE* and a CSIRO expert, a simple visual impression can be gained of the level of expertise reached by the program (see Figure Two). The expert system out performed the human expert by 6.5 percent and increased the profit per pig by 227 percent (using the example in [Black 88]).
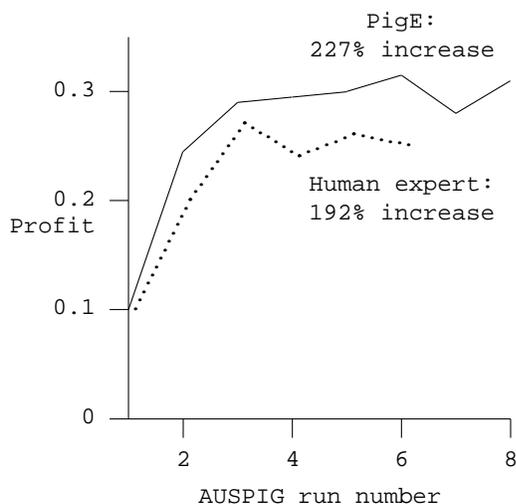

Figure 2. Comparison of human expertise vs PigE

When these figures are extended to cover the profit of the entire piggery, then human expert's analysis increased the piggery's net profit from $40,992 to $202,321. The expert system increased the piggery's net profit to $273,300; i.e. it outperformed the human expert by 34 percent and made $70,979 extra dollars.

Note that in all the above cases, the expert system out-performed the human expert.

7. LESSONS FROM THE SYSTEM

7.1 Heuristic optimisation

Traditionally, with many biological problems, linear programs are used to optimise inputs to a system, such as occurs in least-cost diet formulation packages. However, with complex simulation models like *AUSPIG*, the values of variables that must be satisfied during the optimisation procedure are widely depending on the conditions of the simulation. This makes a linear programming approach unsatisfactory. Other approaches, such as selected variations in a few main variables, with and without "hill climbing" techniques have been used, but these are computationally extremely time-consuming and inefficient. The heuristic approach outlined in this paper provides a far superior method of optimising these complex systems than those previously adopted because it uses information indicating which factors are limiting biological efficiency and the rules of "experts" to select and remedy in order of importance of these limitations.

7.2 Arity Prolog

This project gave us some insights into the strengths and weaknesses of ARITY PROLOG. In order to retain a balanced perspective, the reader should view ARITY's weaknesses (listed below) the broader context. Using ARITY, we rapidly built an extensive meta-interpreter and fielded a DOS-based PROLOG product internationally.

7.2.1 Strengths

PigE made extensive use of meta-level programming. ARITY preserved the semantics of our code in both interpretative and compiled mode.

We had some difficulties implementing the interface from ARITY to PASCAL. This problems proved to be with the PASCAL, not with ARITY. Once we changed PASCALs to one that had a rational memory structure, the ARITY/PASCAL interface went very smoothly.

ARITY's memory management was quite transparent to our PROLOG developers. After some initial parameter tuning, we could ignore memory management altogether.

The ARITY compiler speed up our execution times by a factor of five. This was an unexpected surprise since the rule-interpreter used non-compiled rules (see below).

7.2.2 Weaknesses

The screen-management tools are overly-complicated. The specification of each screen required the assertion of numerous facts. One of our design goals was that the CSIRO programmers could maintain the code. We realised this goal, except with the ARITY screen drivers. In the end, the CSIRO programmers replaced the ARITY system with their own home-grown interface toolkit. We were surprised that this home-grown system was easier to use and more sophisticated than the screen package distributed with ARITY.

The ARITY compiler had a few limitations:

- Debugging compiled code is not fast. The ARITY compiler produces one *internal database file (called a .idb* file) for the whole application. If a module is re-compiled, its old *.obj* file is replaced but the *.idb* file is merely appended to. In practice, this means that *edit-compile-link-test* requires a re-compilation of all modules in order to completely reset the *.idb* file; i.e. no incremental compilation.
- The conclusions of our rules could be very long and they exceeded the maximum fact size that the ARITY compiler could handle. Hence, we had to distribute our knowledge base in an interpreted form.
- The declarations required at the top of each compiled files were involved and awkward to maintain.

7.3 Explanation

The standard view of expert systems is that explicit knowledge representations (such as rule-based systems) make an explanation of the reasoning simple to implement. According to this view, expert systems should have an explanation facility since this makes the reasoning more like a human (humans can explain their reasoning) and it makes the system more acceptable to the user. Humans will not accept, it is argued, the recommendations of a black box and need to have a justification for the conclusions.

This has not been the *PigE* experience. A full explanation of *PigE*'s conclusions requires an inspection of many screens. Users will happily accept the expert system's recommendations in preference to viewing these screens and will never elect to view the screens. So, expert systems may not need an explanation facility so much as a facility to give the users the feeling that they might be able to check the conclusions should they wish to. The paradox we found is that once the users know such a facility exists, they may never use it.

7.4 Knowledge Acqusition

Knowledge acqusition is described as the bottleneck in building expert systems. In the case of *PigE*, the use of the application language approach simplified knowledge acqusition. Our domain experts (Black and Fleming) wrote the 486 rules in the knowledge base with extensive initial support, some support in the post-prototype period, and minimal support for the on-going maintenance.

It should be noted that our experts were both experienced programmers and had written tens of thousands of lines of FORTRAN each for the rest of the package. Perhaps this prior programming experience made it easier for them to construct rules. Also, our experts were not "computer-scared". They were using the rules to augment a package they had built entirely themselves and, hence, were very motivated. Still, the ease of knowledge acquisition was surprising.

7.5 Knowledge Maintenance

Knowledge maintenance is a subject discussed extensively in the expert systems literature. It has not proved to be a problem with *PigE*. The acid test for maintenance is when the development crew changes and this has not happened in the history of the current project. However, the division of the rules into knowledge bases and rule groups simplifies the maintenance problem as does the application language approach. The use of domain-specific procedural attachments that mimic the natural language of the domain experts significantly simplifies maintenance.

8. FUTURE DIRECTIONS

Following on from the success of *AUSPIG*, CSIRO is now planning a version of the system for beef cattle. The expert systems development will continue. As the pig system is refined, the expert system assumes a higher and higher prominence. Originally developed by mathematical modelling experts, *AUSPIG* is an expert's tool more than a tool for the computer novice. The expert system component, originally developed as an intelligent back-end to the mathematical model, is being assessed as a tool for assisting the computer novice for *all* their interaction with the system:

- Human users could be assessed on a continuum of expertise and the environment could adapt appropriately.
- Much of the tedium of data entry could be removed if an expert system could guess at many of the parameters and offer these guesses to the user for them to confirm or override. A frame representation of the screens is being considered. Generic screens could be modelled as classes and the actual screens could be instances, each entry field being one slot. Default values for the slots could be inherited from the class structure and the user could override the slot values as appropriate.

9. CONCLUSION

The logic programming language PROLOG was chosen for business reasons as the implementation tool of choice for a DOS-based farm management expert system. The use of a compiled PROLOG allowed for the rapid development of a royalty-free high-level domain-specific shell. Using PROLOG, we constructed a application language system in which the knowledge representations were customised to reflect the natural idiom of the domain experts. Domain experts have found no trouble in understanding, developing, and maintaining this representation. embedded into an existing application, and fielded around the world. The choice of PROLOG as the expert system tool has been reviewed and no pressing reason has been found to move from this language. The system has been a complete success, partially due to the adaptablity and extendability of PROLOG.

10. REFERENCES

[Black 87a] Black J.L., Fleming J.F. & Davies G.T. *A Computer Modelling Package for the Nutritional Management of Pigs*, in **Australian Poultry and Food Convention**, 1987, pp273-278.

[Black 87b] Black J.L. & Davies G.T. *Predicting the Effects of Climate on the Performance of Growing Pigs*, in **Proceedings of NSW Dept. of Agriculture Pig Advisory Office Conference**, 1987, pp1-13.

[Black 88] Black J.L. & Barron A. *Application of the AUSPIG Computer Package for the Management of Pigs* in **Proceedings of the Australian Farm Management Society**, 15, 1988, 5-1 - 5-10.

[Bustany 88] Bustany A. & Skingle B *Knowledge-based Development via Application Languages* in **Proceedings of the Fourth Australian Conference on Applications of Expert Systems**, May 11-13, 1988, pp277-302.

--   --