MSc Artificial Intelligence Programming in Prolog, 2004
First assessed exercise: A simple puzzle

Tim Smith (tim.smith@ed.ac.uk)

October 4, 2004

Introduction

This is the first assessed exercise (of two) for the 2004 MSc Artificial Intelligence Programming in Prolog module. It is worth 10% of the total mark for the module. It should take about 8–12 hours to complete. The submission deadline is Monday, 1st November at 4pm (the start of week 7).

A simple puzzle

This exercise is based on a simple puzzle which exists in many incarnations, most of them electronic. The version of the puzzle that we'll be using for this exercise is as follows.

The board for the puzzle is a 3×3 grid, similar to that of tic-tac-toe or noughts and crosses. Each square on the board can have one of two values. We will use the values `x` and `o`. So, a board might look like:

```
 o | x | o
-----------
 x | x | o
-----------
 o | o | x
```

To begin the puzzle, each of the squares is randomly assigned one of the two possible values. The task is then to change the board so that all nine of the squares contain `o`, and there are no squares containing `x`.

In order to do that, the player can *push* the squares on the board, one at a time. Pushing a square flips the value of that square from `x` to `o`, or from `o` to `x`, but *also* flips a number of other squares. The behaviour is:

- If the player pushes one of the corner squares, that square, and the three surrounding squares, are all flipped to the other value.

- If the player pushes a square which is in the middle of an edge, all three squares on that edge are flipped to the other value.
- If the player pushes the central square, then that square, the squares to the left and right of that square, *and* the squares above and below that square, are all flipped to the other value.

So, if we imagine the squares are numbered as follows:

```
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9
```

then:

- Pushing square 1 flips squares 1, 2, 4 and 5.
- Pushing square 2 flips squares 1, 2, and 3.
- Pushing square 3 flips squares 2, 3, 5 and 6.
- Pushing square 4 flips squares 1, 4 and 7.
- Pushing square 5 flips squares 2, 4, 5, 6 and 8.
- Pushing square 6 flips squares 3, 6 and 9.
- Pushing square 7 flips squares 4, 5, 7 and 8.
- Pushing square 8 flips squares 7, 8 and 9.
- Pushing square 9 flips squares 5, 6, 8 and 9.

For example, from this board state:

```
 o | x | o
-----------
 x | o | o
-----------
 x | o | x
```

pushing square 5 produces the following board state:

```
 o | o | o
-----------
 o | x | x
-----------
 x | x | x
```

From any starting state, it is always possible, using a sequence of such pushes, to reach the *goal state*, in which each square has the value `o`.

The exercise

The exercise is in two parts. The first part involves completing an implementation of the puzzle which allows a human player to play with the puzzle. The second part involves completing an implementation of an automatic *solver* which will randomly generate a puzzle board, and then automatically find a sequence of pushes that leads to the goal state.

For each part you are given a skeleton solution, which you must complete, by implementing a number of predicates, according to a specification. If you write these predicates according to the specification, the entire program will work. Make sure you follow the specification for each predicate carefully.

Attempt the following sections in the order they're listed, since later sections may depend on your solutions to earlier sections.

Download the skeleton solution `puzzle_template.pl` from the course website. Save it in your home directory and rename it something memorable. In the file you will find partial code and commented sections indicating where you should write your new code. You should write all your answers to this file, using comments to answers questions that do not require code.

1 Implementing a human-playable puzzle

You are given a skeleton solution for an implementation of the puzzle that a human player can play. The skeleton looks like this:

```
play :-
   initialise_randomness,
   initialise_game(Game),
   display_game(Game),
   play(Game,0).

play(Game, Count) :-
   game_over(Game),
   write('Congratulations you have '),
   write('finished the game in '),
   write(Count)
   write(' moves!'), nl, !.

play(Game, Count) :-
   choose_move(Move),
   update_game(Game, Move, NewGame),
   display_game(NewGame),
   Count1 is Count +1,
   play(NewGame, Count1).
```

In fact, this is a very general skeleton for a one-player game, and might be used for many different games. You will find this skeleton in the template that is provided for this exercise.

The skeleton works as follows:

- `play/0` initialises SICStus's random number generator (this is not something you need to worry about), then calls `initialise_game/1` to generate a random board for the start of the puzzle. It calls `display_game/1` to display the board on the screen, then calls `play/2` to allow a player to play with the puzzle.

- The first clause of `play/2` calls `game_over/1`, which succeeds if the current board state is the *goal state*. If it is, then the program writes the number of moves taken to reach the goal state (Count) and then stops—the puzzle has been solved.

- Otherwise, the second clause of `play/2` calls `choose_move/1` to get the player's next move, calls `update_game/3` to apply this move to the board, creating a new board, calls `display_game/1` to display the new board, then calls `play/2` recursively to repeat the whole cycle, which terminates if and when the player manages to reach the goal state.

Your task for the first part of the exercise is to implement the following five predicates, which are not yet implemented in the skeleton that you've been given (the behaviour of each of these predicates is outlined in the next section):

```
initialise_game/1
display_game/1
game_over/1
choose_move/1
update_game/3
```

Don't modify any of the existing code in the template, otherwise the program might not work. As you implement each predicate, add your code underneath the skeleton, in the appropriate marked area.

If you feel that it would be useful to write other predicates to help with the implementation of the predicates you are asked to implement, then go ahead. So long as the five predicates listed above are implemented according to the specifications shown below, it doesn't matter which other predicates you implement.

Don't load any Prolog library packages, other than those that the template loads already. If you need standard list-processing predicates, such as `member/2` and `append/3`, take these from a standard Prolog reference, such as Clocksin & Mellish, or the lecture notes—and say in your code where they came from.

1.1 Design a board representation

The first step is to decide how you're going to represent the game board. This should be a Prolog *term*, but exactly what this looks like is up to you. At the minimum, it needs to represent:

- Each of the nine squares on the board.

- The current value, `x` or `o`, of each square on the board.

This is not a programming task. You don't need to write any code. You do need to decide how you'll be representing the board as a Prolog data-structure. The predicates that you go on to implement will then need to make use of the representation that you've chosen. So think about what might be a *good* way to represent the game board.

1.2 Implement `initialise_game/1`

`initialise_game/1` is called with a variable as its single argument, and succeeds by instantiating this variable to a board state, in the representation you have chosen, in which each square is randomly assigned to either `x` or `o`.

Your attention is drawn to this section of the SICStus manual:

http://www.dai.ed.ac.uk/dai/computing/software_manuals/sicstus/sicstus_22.html#SEC173

which describes a 'package' that makes available a number of predicates for dealing with random numbers.

This package is already loaded by the template you are given, so the predicates in the package are available for you to use, and you don't need to load it yourself. And note that the template also initialises the random number generator for you.

The predicate `random/1` is likely to be particularly useful in implementing this predicate.

Write your code in the section marked `initialise_game/1`.

1.3 Implement `display_game/1`

`display_game/1` has a single argument, which is a board state, in the representation you have chosen. It succeeds by displaying the board state on the screen, in an appropriate, clear format.

Your attention is drawn to the built-in predicate `write/1`, which writes a term to the screen, and the built in predicate `nl/0`, which writes a newline character to the screen.

A simple, clear picture of the board state is all that's required here. You won't get any extra marks for something elaborate, so don't waste time on that.

1.4 Implement `game_over/1`

`game_over/1` has a single argument, which is a board state, in the representation you have chosen. It succeeds if that board state represents the *goal state* for the puzzle; that is, when the board contains only `o` values, and no `x` values are left. Otherwise, it fails.

1.5 Implement `choose_move/1`

`choose_move/1` is called with an uninstantiated variable as its single argument. It succeeds by instantiating this variable to the player's choice of the next square to push.

It should:

- Display a suitable prompt, asking the player to enter the square they chooses to push.

- Read from the keyboard the player's choice of square.
- Check that the input represents a valid square, and repeat the input process if it's invalid.
- Instantiate the argument to the chosen square.

Your attention is drawn to the built-in predicate `read/1`, which reads a *term* from the keyboard and instantiates it to the variable when followed by a full-stop and *Return*.

How you choose to refer to the squares on the board is up to you. You might use the numbers 1 to 9, as I did earlier, or you might use something entirely different. Whatever you choose, you should make it clear to the player what input is expected, and this predicate should succeed only when a valid square has been chosen.

1.6 Implement `update_game/3`

`update_game/3` is called with its first two arguments instantiated, and its third argument as a variable. The first argument is a board state, in the representation you have chosen. The second argument is a square, using whatever naming you have chosen to refer to squares (whatever `choose_move/1` accepts as valid input).

This predicate succeeds by instantiating the third argument to the board state which results from pushing the square which is the second argument, in the board state which is the first argument. In other words, it generates the new board state after making the move the player has chosen.

To implement this you will need to encode the game rules as outlined before in some form that they can be used to modify the board state.

1.7 That's it!

Once you've implemented each of these five predicates, you should be able to consult the program, then type:

```
?- play.
```

at the SICStus prompt, and play with the puzzle.

2 Implementing an automatic puzzle solver

You are given a skeleton solution for an implementation of a *solver* for this puzzle. The skeleton looks like:

```
solve :-
   initialise_randomness,
   initialise_game(Game),
   display_game(Game),
   agenda_item(Game, [], FirstAgendaItem),
   solve([FirstAgendaItem], [], Moves),
   write(Moves).

solve([FirstAgendaItem | _], _, Moves) :-
   agenda_item(Game, Moves, FirstAgendaItem),
   game_over(Game),
   display_solution(Game,Moves),
   !.

solve([FirstAgendaItem|RestOfAgenda], AlreadyTried, Moves) :-
   get_possible_moves(FirstAgendaItem, PossibleMoves),
   get_worthwhile_moves(PossibleMoves, RestOfAgenda,
                        AlreadyTried, WorthwhileMoves),
   update_agenda(WorthwhileMoves, RestOfAgenda, NewAgenda),
   solve(NewAgenda, [FirstAgendaItem|AlreadyTried], Moves).

display_solution(_,[]).
display_solution(Board,[H|T]):-
   update_game(Board, H, NewBoard),
   display_solution(NewBoard, T),
   write('Push '), write(H), nl,
   display_game(Board).
```

Note that this skeleton makes use of the predicates `initialise_game/1`, `display_game/1`, `update_game/1` and `game_over/1`, which you implemented for the first part of this exercise.

The skeleton works as follows:

- `solve/0` initialises SICStus's random number generator (this is not something you need to worry about), then calls `initialise_game/1` to generate a random board for the start of the puzzle. It calls `display_game/1` to display the board on the screen, then calls `solve/3` to generate a solution to the puzzle.
- `solve/3` solves the puzzle by making use of an *agenda*. Using an agenda is a common technique in AI search algorithms.

  Here, the agenda is a list which keeps track of the board states and moves we've generated so far, but which we haven't checked yet. Each item in the agenda represents two things:

  1. A board state; and
  2. A list of the moves that have been made to get to this board state from the initial board state.

  The first argument of `solve/3` is the current agenda, the second argument is a list of agenda items that have been checked already[1], and the third argument is the result: a list of moves which solves the puzzle.

[1]This is often called the 'closed list'. The current agenda, similarly, is often called the 'open list'.

solve/3 is called with its first two arguments instantiated, and its third argument *uninstantiated*. It succeeds by instantiating its third argument to the list of moves which solves the puzzle.

- The first clause of solve/3 takes the first item from the current agenda, then calls agenda_item/3 to split the agenda item up into its two parts: the board state, and the moves so far. It calls game_over/1, which succeeds if the board state from the first agenda item is the *goal state*. If it is, then solve/3 writes out the solution using display_solution/2 and then succeeds, returning the moves so far as its result.
- Otherwise, the second clause of solve/3 takes the first item from the current agenda, then calls get_possible_moves/2 to generate all of the board states and moves so far which may be reached by making one move from the first agenda item. It then calls get_worthwhile_moves/4, which picks out those agenda items which are worth following up, and then calls update_agenda/3 to add these new agenda items to the agenda. Finally, it calls solve/3 recursively with the new agenda, adding the agenda item it's just checked to its second argument.

Your task is to implement the following predicates, which are not yet implemented in the skeleton that you've been given:

```
agenda_item/3
get_possible_moves/2
get_worthwhile_moves/4
update_agenda/3
```

You *don't* need to have a complete understanding of how this search algorithm works in order to implement these predicates. So long as you follow the specifications described below, everything will work fine. Nevertheless, do try to follow how solve/3 uses the agenda to search through the space of possible board states in order to find a solution, and a path *to* that solution.

Don't modify any of the existing code in the template, otherwise the program might not work. As you implement each predicate, add your code underneath the skeleton, in the appropriate marked area.

If you feel that it would be useful to write other predicates to help with the implementation of the predicates you are asked to implement, then go ahead. So long as the four predicates listed above are implemented according to the specifications shown below, it doesn't matter which other predicates you implement.

Don't load any Prolog library packages, other than those that the template loads already. If you need standard list-processing predicates, such as member/2 and append/3, take these from a standard Prolog reference, such as Clocksin & Mellish or the lecture notes—and say in your code where they came from.

2.1 Design a representation for an agenda item

The first step is to decide how you're going to represent each item in the agenda. This should be a Prolog *term*, but exactly what this looks like is up to you. At the minimum, it needs to represent:

- A board state, using the representation you designed for the first part of this exercise.
- A list of moves, showing the moves taken to reach this board state from the initial state. It would make sense for each move to be represented using the naming scheme for the squares that you chose for the first part of this exercise.

This is not a programming task. You don't need to write any code. You do need to decide how you'll be representing the agenda item as a Prolog data-structure. The predicates that you go on to implement will then need to make use of the representation that you've chosen. So think about what might be a *good* way to represent the agenda item. It doesn't need to be complex.

2.2 Implement `agenda_item/3`

`agenda_item/3` is used to construct an agenda item from a board state and a list of moves, or to pull apart an agenda item *into* the board state and list of moves that it represents.

Its first argument is a board state. Its second argument is a list of moves. And its third argument is an agenda item. It succeeds where the third argument is the agenda item obtained by combining the board state and the list of moves.

It may be called in two ways. If the first two arguments are instantiated, and the third is a variable, then it succeeds, instantiating the third argument to the agenda item which represents the board state from the first argument and the list of moves from the second argument. This way of using `agenda_item/3` is that of *constructing* an agenda item from its parts.

If the third argument is instantiated to an agenda item, and the first two arguments are *uninstantiated*, then it succeeds, instantiating the first argument to the board state contained by the agenda item, and instantiating the second argument to the list of moves contained by the agenda item. This way of using `agenda_item/3` is that of *destructing* an agenda item into its parts.

Correctly implemented, this will be a remarkably short predicate. If you find yourself writing much code, you've misunderstood what's needed here.

2.3 Implement `get_possible_moves/2`

`get_possible_moves/2` generates new agenda items from an existing agenda item. It should be called with its first argument instantiated to an agenda item, and its second argument as an uninstantiated variable.

It succeeds by instantiating the second argument to a list of all of the agenda items which may be reached from the agenda item in the first argument, by making a single new move. (Since there are nine squares in the puzzle, and therefore nine possible moves, this list of new agenda items should always include nine items.)

Each new agenda item in this list represents, using the representation you have chosen for agenda items:

- The new board state reached by making a new move;
- The updated list of moves, constructed by adding the new move to the previous list of moves.

You will almost certainly find the implementation of `update_game/3` that you wrote for the first part of this exercise useful here. You might also find your implementation of `agenda_item/3` useful.

2.4 Implement `get_worthwhile_moves/4`

`get_worthwhile_moves/4` is used to select from a list of agenda items those which are 'worthwhile'. Here, we'll define 'worthwhile' as: agenda items which don't take us back to a board state that we've already generated.[2]

It is called with its first three arguments instantiated, and its fourth argument *uninstantiated*.

The first argument is a list of agenda items: these are the items we want to decide are worthwhile or not. The second argument is the current agenda. The third argument is the list of agenda items that we've already checked (the 'closed list').

It succeeds by instantiating the fourth argument to a list of those agenda items in the first argument which contain board states that *do not* appear in the current agenda (second argument), or in the 'closed list' of already-checked agenda items (third argument). That is, it keeps those agenda items from the first list which move the puzzle to a board state that we've not generated yet.

You will almost certainly find the usual definition of list membership, `member/2`, useful here. You might also find your implementation of `agenda_item/3` useful.

2.5 Implement `update_agenda/3`

`update_agenda/3` creates a new agenda by adding new agenda items to an existing agenda.

It is called with its first two arguments instantiated, and its third argument *uninstantiated*. The first argument is a list of agenda items we want to add to the existing agenda. The second argument is the existing agenda.

It succeeds by instantiating the third argument to the new agenda which is created by adding the agenda items in the first argument to the existing agenda in the second argument.

This should not be a very complex predicate. You might remember a predicate from the lectures on list processing that was able to append two lists together to create a third.

2.6 That's it!

Once you've implemented each of these four predicates, you should be able to consult the program, then type:

```
?- solve.
```

at the SICStus prompt, and watch the solver do its stuff. It will randomly instantiate the puzzle, then work out a solution, and display the moves in the solution. Check to see if it really *is* a solution.

[2]There's a very simple practical reason why we don't want to consider board states that we've already generated. If we do, we'll end up going round in circles and never reach a solution. Keeping only the agenda items that contain new board states means we must reach a solution eventually.

2.7 Looking at different search techniques

You should find that the effectiveness of the search algorithm used here is very much dependent on the way in which new agenda items are added to the agenda. The time taken to find a solution to each puzzle, and the length of the solution found, depend on exactly how you have implemented `update_agenda/3`.

As a final task:

- Write two versions of `update_agenda/3`: one which constructs a new agenda by adding new agenda items to the *beginning* of the current agenda, and another which adds new agenda items to the *end*. When testing one version remember to comment out the other version.
- Compare the efficiency of these two versions by examining how many moves it takes each strategy to find a solution and how direct the solution appears to be. You can compare the two search strategies by using `solve/1` which takes a starting board state as an argument and generates a solution. This can be used to directly compare the two strategies.
- Write a short report, of about 250 words:
    1. Describe the solutions the solver finds, and how they are different for the different implementations of `update_agenda/3`.
    2. Explain why you think the different implementations of `update_agenda/3` cause this different behaviour.
    3. Describe any other ways you can think of in which new agenda items might be added to the agenda, other than simply at the end or the beginning. Could they be sorted in some way? How?
- Put the report into the same file as your Prolog solutions, in the marked area. Make sure it's properly commented, so that the file loads correctly.

3 Assessment

This exercise will be marked out of 100%. The mark will break down as follows:

- Part 1:
    - 10% for the design of the board representation.
    - 5% for the implementation of `initialise_game/1`.
    - 5% for the implementation of `display_game/1`.
    - 5% for the implementation of `game_over/1`.
    - 10% for the implementation of `choose_move/1`.
    - 15% for the implementation of `update_game/3`.

- Part 2:
  - 5% for the design of an agenda item and the implementation of `agenda_item/3`.
  - 15% for the implementation of `get_possible_moves/2`.
  - 10% for the implementation of `get_worthwhile_moves/4`.
  - 10% for the different implementations of `update_agenda/3`, and the report on their behaviour.
- The remaining 10% will be given for issues of general programming style, such as:
  - Clear layout of Prolog code.
  - Clear and useful commenting.
  - Suitable and effective variable names.

A maximum of 60% of the available marks will be given to a predicate which does not work (or which does not work according to the specification!), no matter how minor any errors might be.

Nevertheless, marks *are* available for a predicate which is not complete and working, so please present partial solutions too. Make sure incomplete predicates are commented out, so that they don't prevent your file from loading into SICStus.

4 Hints and advice

- Read this specification carefully, several times, before thinking about starting on the exercise.
- Don't be intimidated by the length of this specification. Good Prolog programs are usually quite short, and a good solution to this exercise will almost certainly be shorter than you expected. If you find yourself writing what seems like a lot of Prolog, you might be making things more complicated than they need to be. Take a step back and rethink.
- As you implement each predicate, test it separately, before moving on to the next one. If you wait until they're all implemented before you test them, it'll be much harder to find out where things are going wrong.
- Take lots of care with laying out your code clearly. Use lots of white space, and plenty of clear comments. Part of your mark is for formatting and commenting of code. If you don't take care over this, you will lose easy marks unnecessarily.
- Make sure that what you write conforms to the specification given here. There are no extra marks for additional cleverness, and you'll lose marks if you implement something that doesn't conform to the specification, no matter how neat it might be.
- If you're having problems, remember that you can use `trace/0` and `notrace/0` to enable Prolog's tracing, and `spy/1` to look at individual predicates. You can also use `write/1` to write any term out on the screen, and `nl/0` to write a newline character. See a Prolog reference for more details.

5 Plagiarism

This is an *individual* exercise, so what you submit should be your own work. Please make sure that you understand the Division's Guidelines on Plagiarism, which you can find here:

> http://www.informatics.ed.ac.uk/admin/ITO/DivisionalGuidelinesPlagiarism.html

If you submit work that isn't wholly your own, you must say so clearly in your submission. Discussing general issues concerning an exercise with other students is a good thing, and is expected, but genuine collaboration on all or part of an assessed exercise must be explicitly acknowledged and will be penalised.

6 How to submit

Your submission for this exercise should take the form of a single Prolog file, containing the templates you were given, the Prolog you have added, and the report you are asked to write in part 2.

- *Put your name and matriculation number at the top of the file.* Make sure that they're commented out, so that they don't stop the file loading into SICStus.
- *Make sure that your submission loads into SICStus.* Make sure that comments are properly commented. If your submission fails to load, and there are simple and obvious changes which allow it to load (like commenting out something which should have been commented already), I *might* make those changes, but you will lose marks because of that. If the file needs more significant corrections in order to load, I won't attempt any changes, and you will be marked down accordingly. If you make any changes to your submission at the last moment, *no matter how small they seem*, test that the file loads before you submit.

Submit the file electronically, by 4pm on Monday, 1st November (the start of week 7), on any DICE machine using the command:

```
submit msc aipp 1 <filename> <time spent>
```

where `<filename>` is the name of your file, and `<time spent>` is an integer representing approximately how many hours you spent on the exercise. (This value in no way affects your mark for the exercise. It simply helps to gauge whether the size and scope of the exercise was appropriate, for future reference.)

Late submissions will not be accepted without a extremely good reason. If you have a case for an extension, please let me know as soon as possible.