COMP3411/9814 Artificial Intelligence
Term 1, 2021

Assignment 1 – Prolog and Search

Due: Friday 19 March, 10:00 pm

Marks: 20% of final assessment for COMP3411/9814 Artificial Intelligence

Part 1 - Prolog

In this part, you are to write some Prolog programs.

At the top of your file, place a comment containing **your full name, student number and assignment name**. You may add additional information like the date the program was completed, etc. if you wish.

At the start of each Prolog predicate, write a comment describing the operation of the predicate.

Testing Your Code

A significant part of completing this assignment will be testing the code you write to make sure that it works correctly. To do this, you will need to design test cases that exercise every part of the code.

You should pay particular attention to "boundary cases", that is, what happens when the data you are testing with is very small, or in some way special. For example:

- What happens when the list you input has no members, or only one member?
- Does you code work for lists with both even and odd numbers of members?
- Does your code work for negative numbers?

Note: not all of these matter in all cases, so for example with `sqrt_table`, negative numbers don't have square roots, so it doesn't make sense to ask whether your code works with negative numbers.

With each question, some example test data are provided to clarify what the code is intended to do. You need to design *further* test data. Testing, and designing test cases, is part of the total programming task.

It is important to use *exactly* the names given below for your predicates, otherwise the automated testing procedure will not be able to find your predicates and you will lose marks. Even the capitalisation of your predicate names must be as given below.

Question 1.1: List Processing

Write a predicate `sumsq_even(Numbers, Sum)` that sums the squares of only the even numbers in a list of integers. For example,

```
?- sumsq_even([1,3,5,2,-4,6,8,-7], Sum).
Sum = 120
```

Note that it is the element of the list, not its position, that should be tested for oddness. (The example computes 2*2 + (-4)*(-4) + 6*6 + 8*8). Think carefully about how the predicate should behave on the empty list — should it fail or is there a reasonable value that Sum can be bound to?

To decide whether a number is even or odd, you can use the built-in Prolog operator **N mod M**, which computes the remainder after dividing the whole number ***N*** by the whole number ***M***. Thus a number ***N*** is even if the goal 0 is **N mod 2** succeeds. Remember that arithmetic expressions like X + 1 and N mod M are only evaluated if they appear after the ***is*** operator. So **0 is N mod 2** works, but **N mod 2 is 0** doesn't work.

Question 1.2: List Processing

Eliza was the name of the first "chatbot" written by Joseph Weizenbaum at MIT in the mid-1960s. It pretended to be a psychiatrist, so that it only had to do simple transformations on the input and turn a statement into a sentence. If a sentence is represented by a list of words, an example of a simple transformation is:

```
?- eliza1([you,do,not,like,me], X).
X = [what,makes,you,say,i,do,not,like,you]
```

Here, the transformation is to put "What makes you say" in the front of the sentence and replace "you" with "i" and "me" with "you".

Write a Prolog program that takes a sentence in the form of a list with replacements:

you → i
me → you
my → your

and prepends the list [what, makes, you, say] to the transformed list.

- Your can write helper predicates, but the top-level predicate MUST be called "eliza1".
- Assume that all sentence begins with "you", but if you had " … me and you …" would that make a difference to your program?
- An input list that is empty i.e. "[]", should return an empty list.
- Assume sentences are grammatical correct, so we won't test something like [you, me].
- By default, SWI Prolog limits the number of elements in a list that it prints. You might see the answer to your query ending with [a, b c | …]. You can force printing longe lists with the directive:

```
:- set_prolog_flag(answer_write_options,[max_depth(0)]).
```

which you can put at the top of your file. max_depth(0) means no limit.

You MUST include ":–" at the start of the line. This is a Prolog directive.

Question 1.3: List Processing

The rules in Question 1.2 work if "you" starts a sentence but won't make much sense for an example like this:

```
?- eliza1([i,wonder,if,you,would,help,me,learn,prolog], X).
X = [what,makes,you,say,i,if,wonder,i,would,help,you,learn,prolog]
```

What would be better is:

```
?- eliza2([i,wonder,if,you,would,help,me,learn,prolog], X).
X = [what,makes,you,think,i,would,help,you]
```

Write a new predicate ***eliza2*** (don't forget the "2") that takes a list of words:

```
[ …, you, <some words>, me, …]
```

and creates a new list of the form:

```
[what, makes, you, think, i, <some words>, you]
```

i.e. skip the words before "you" and after "me", and insert the words in between "you" and "me" into the new sentence between "i" and "you".

Hint: You can use the built-in predicate "append(X, Y, Z)" to do a lot of the work for you. Remember, "append" can be used to split a list, as well as concatenating lists.

Question 1.4: Prolog Terms

Arithmetic expressions can be written in *prefix* format, e.g 1+2*3 can be written as `add(1, mul(2, 3))`. If the operators available are `add`, `sub`, `mul`, `div`, write a Prolog program, `eval(Expr, Val)`, that will evaluate an expression, e.g.

```
?- eval(add(1, mul(2, 3)), V).
V = 7
?- eval(div(add(1, mul(2, 3)), 2), V).
V = 3.5
```

- If you wish, you may use the builtin predicate **number(N)**, which is true when **N** is a number and false otherwise (e.g. when N is a compound term like **mul(2, 3)**.
- You will need is use the **is** built-in predicate to do the actual arithmetic.
- We will not test for division by 0, but if you include a check for division by 0, your coding style will be much better.

Testing

This assignment will be marked on functionality in the first instance. However, you should always adhere to good programming practices in regard to structure, style and comments, as described in the Prolog Dictionary. Submissions that score very low in the automarking will be examined by a human marker, and may be awarded some marks, provided the code is readable.

Your code must work under the version of SWI Prolog used on the Linux machines in the UNSW School of Computer Science and Engineering. If you develop your code on

any other platform, it is your responsibility to re-test and, if necessary, correct your code when you transfer it to a CSE Linux machine prior to submission.

Your code will be run on a few simple tests when you submit. So, it is a good idea to submit early and often so that potential problems with your code can be detected early. You will be notified at submission time if your code produces any compiler warnings. Please ensure that your final submission does not produce any such warnings (otherwise, marks will be deducted).

Part 2 - Search

Question 1: Search Algorithms for the 15-Puzzle

In this question you will construct a table showing the number of states expanded when the 15-puzzle is solved, from various starting positions, using four different searches:

(i) Uniform Cost Search (with Dijkstra's Algorithm)

(ii) Iterative Deepening Search

(iii) A* Search (using the Manhattan Distance heuristic)

(iv) Iterative Deepening A* Search

Go to theWebCMS. Under "Assignments" you will find Prolog Search Code "`prolog_search.zip`". Unzip the file and change directory to prolog search, e.g.

```
unzip prolog_search.zip
cd prolog_search
```

Start prolog and load puzzle15.pl and ucsdijkstra.pl by typing

```
[puzzle15].
[ucsdijkstra].
```

Then invoke the search for the specified start10 position by typing

```
start10(Pos),solve(Pos,Sol,G,N),showsol(Sol).
```

When the answer comes back, just hit Enter/Return. This version of Uniform Cost Search (UCS) uses Dijkstra's algorithm which is memory efficient, but is designed to return only one answer. Note that the length of the path is returned as G, and the total number of states expanded during the search is returned as N.

a) Draw up a table with four rows and five columns. Label the rows as UCS, IDS, A* and IDA*, and the columns as `start10, start12, start20, start30` and `start40`. Run each of the following algorithms on each of the 5 start states:

`(i)[ucsdijkstra]`

```
(ii)[ideepsearch]
(iii)[astar]
(iv)[idastar]
```

In each case, record in your table the number of nodes generated during the search. ***If the algorithm runs out of memory, just write "Mem" in your table. If the code runs for five minutes without producing out- put, terminate the process by typing Control-C and then "a", and write "Time" in your table. Note that you will need to re-start prolog each time you switch to a different search.***

b) Briefly discuss the efficiency of these four algorithms (including both time and memory usage).

Question 2: Heuristic Path Search for 15-Puzzle

In this question you will be exploring an Iterative Deepening version of the Heuristic Path Search algorithm discussed in the Week 3 Tutorial. Draw up a table in the following format:

	start50		start60		start64	
IDA*	50	14642512	60	321252368	64	1209086782
1.2 1.4 1.6						
Greedy						

The top row of the table has been filled in for you (to save you from running some rather long computations).

(a) Run [greedy] for start50, start60 and start64, and record the values returned for G and N in the last row of your table (using the Manhattan Distance heuristic defined in puzzle15.pl).

(b) Now copy idastar.pl to a new file heuristic.pl and modify the code of this new file so that it uses an Iterative Deepening version of the Heuristic Path Search algorithm discussed in the Weak 3 Tutorial Exercise, with w = 1.2 .
In your submitted document, briefly show the section of code that was changed, and the replacement code.

(c) Run [heuristic] on start50, start60 and start64 and record the values of G and N in your table. Now modify your code so that the value of w is 1.4, 1.6 ; in each case, run the algorithm on the same three start states and record the values of G and N in your table.

(d) Briefly discuss the tradeoff between speed and quality of solution for these five algorithms.

Submitting your assignment

Your submission will consist of two files: *assign1_part1.pl* should contain all of your Prolog programs; and *assign1_part2.pdf* should contain the results of your search experiments in part 2.

To hand in, log in to a School of CSE Linux workstation or server, make sure that your files are in the current working directory, and use the Unix command:

```
% give cs3411 assign1 assign1_part1.pl assign1_part2.pdf
```

Please make sure your code works on CSE's Linux machines and generates no warnings. Remove all test code from your submission. Make sure you have named your predicates correctly.

You can submit as many times as you like - later submissions will overwrite earlier ones. Once `give` has been enabled, you can check that your submission has been received by using one of these commands:

The submission deadline is Friday 19 March, 10:00 pm.

10% penalty will be applied to the (maximum) mark for every 24 hours late after the deadline.

Questions relating to the project can be posted to the forums on the course Web site.

If you have a question that has not already been answered on the forum, you can email it to cs3411@unsw.edu.au

Plagiarism Policy

Group submissions are not allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for any similar projects from previous years) and serious penalties will be applied, particularly in the case of repeat offences.

DO NOT COPY FROM OTHERS. DO NOT ALLOW ANYONE TO SEE YOUR CODE

Please refer to the UNSW Policy on Academic Honesty and Plagiarism if you require further clarification on this matter.