CPS721: Assignment 3

Due: October 22, 2024, 9pm
Total Marks: 100 (worth 4% of course mark)
You MUST work in groups of 2 or 3

Late Policy: The penalty for submitting even one minute late is 10%. Assignments are not accepted more than 24 hours late.

Clarifications and Questions: Please use the discussion forum on the D2L site to ask questions as they come up. These will be monitored regularly. Clarifications will be made there as needed. A Frequently Asked Questions Page will also be created. You may also email your questions to your instructor. Check the D2L forum and frequently asked questions first.

Collaboration Policy: You can only discuss this assignment with your group partners or with your CPS721 instructor. By submitting this assignment, you acknowledge that you have read and understood the course policy on collaboration as stated in the CPS721 course management form.

PROLOG Instructions: When you write your rules in PROLOG, you are not allowed to use ";" (disjunction), "!" (cut), and "->" (if-then). You are only allowed to use ";" to get additional responses when interacting with PROLOG from the command line. Note that this is equivalent to using the "More" button in the ECLiPSe GUI.

We will be using ECLiPSE Prolog release 6 to mark the assignments. If you run any other version of PROLOG, it is your responsibility to check that it also runs on this version.

SUBMISSION INSTRUCTIONS: You should submit ONE zip file called `assignment3.zip` containing 6 files:

`q1a_generate_and_test.pl`	`q1b_interleaving.pl`	`q1_puzzle_session.txt`
`q2_game.pl`	`q2_game_session.txt`	
`q3_fertilizers.pl`	`q3_fertilizers_session.txt`	

These files have been given to you and you should follow the format given. Your submission should not include any other files. If you submit a `.rar`, `.tar`, `.7zip`, or other compression format aside from `.zip`, you will lose marks. All submissions should be made on D2L. Submissions by email will not be accepted. As long as you submit your assignment with the file name `assignment3.zip` your group will be able to submit multiple times as it will overwrite an earlier submission. You do not have to inform anyone if you do. The time stamp of the last submission will be used to determine the submission time. Do not submit multiple `zip` files with different names. If you do, we will use the last submitted one, but you may lose marks.

Make sure the files are saved in plain text and readable on Linux machines. Ensure your PROLOG code does not contain any extra binary symbols and that they can be compiled by ECLiPSE Prolog release 6.

1 Crypt-Arithmetic [30 marks]

Use Prolog to solve the following crypt-arithmetic puzzle involving **multiplication**:

```
          J  E  T
      *      A  X
-----------------
  +    A  X  L  E
       J  E  T  .
----------------
       L  O  V  E
```

This notation means that JET * AX = LOVE, and the words between the bars correspond to performing the usual multiplication algorithm. That is AXLE + 10*JET = LOVE. For example, the calculation of 123 * 12 would appear as

```
          1  2  3
      *      1  2
-----------------
  +       2  4  6
       1  2  3  .
----------------
       1  4  7  6
```

Each letter stands for a distinct digit and leading digits are not zeroes. You cannot make any other assumptions. You will solve this problem in two parts.

a. [10 marks]

First, try to solve this problem using the *pure generate and test* technique, without any interleaving, and notice how much time it takes. In particular, you should write the rules for *solve*, which takes in your list of variables and finds an assignment that solves the puzzle. You should also write the rules for *solve_and_print* which calls your *solve* rule and prints out your solution in a human-readable form. These should be put in the given file `q1a_generate_and_test.pl` following the directions given in the file. Ensure that both the *solve* and *solve_and_print* can be run.

You can then determine how much computer time your computation takes using the following query:

```
?- Start is cputime, solve_and_print, End is cputime, Time is End - Start.
```

The value of `Time` will be the time taken. Note, that the timing code should not be included in `q1a_generate_and_test.pl`.

Add the results of this query, including the time and solution, to the file `q1_puzzle_session.txt` in the correct section.

b. [20 marks] Next, solve this problem using a smart interleaving of generate and test. Again implement *solve* and *solve_and_print*, and add them to the file `q1b_interleaving.pl`. In this file, add comments in your file which explain briefly the order of constraints you have chosen and why this has an effect on computation time. You can draw a dependency graph by hand and include a PDF image if you decide to use one (but it is not a requirement).

Find also how much time your program takes to compute an answer in this case. Again, include the results of your session in `q1_puzzle_session.txt`

2 Game Puzzle [30 marks]

There was a tournament of 5 teams, where each team played with every other team exactly once. In total, the tournament had 10 matches. In each match, two teams played against each other. The teams are from Oakville, Pickering, Richmond Hill, Scarborough, and Toronto (downtown). The games are scheduled over 5 rounds. In each round, one of the teams did not participate, while the other four teams played two matches once. The outcome for each match is either a draw **d** for both teams, or a win **w** for one team and a loss **l** for another team.

Your task is to write a Prolog program that finds the outcome of each match for each team, based on the clues provided below. You have to write a Prolog program that implements precisely not only the given constraints, but also constraints that remain implicit. Never try to guess part of solution by yourself: all reasoning should be done by your program. You have to demonstrate whether you learned well a program design technique. Your implementation should follow exactly one of the design patterns explained in class. You will lose marks, if the TA sees that you embedded your own reasoning into your program. Points per match must satisfy all of the following constraints.

1. Pickering lost to Scarborough in the first round, but won over Oakville in the second round.
2. Toronto did not play in the third round; they had one win and one loss in the previous two rounds.
3. Oakville did not participate in the fourth round, but they already won twice in the preceding three matches.
4. All matches in the fourth and in the fifth round finished with a draw.
5. Before the fourth round, Richmond Hill won only once and lost once.
6. None of the matches in the first, in the second and in the third rounds finished with a draw.

a. [**25 marks**] Write a Prolog program `q2_game.pl` that solves this problem using interleaving of generate and test technique considered in class. In your programs, the variables should range over the following finite domain of characters: **n** (a team did not participate in a round), **l** (loss), **d** (draw), **w** (win).

You will need to be careful in your program regarding the order of constraints. Also, provide comments where in your program which constraint is implemented. Explain briefly the ordering you have chosen (write comments in you program file). You should implement both a *solve* and *solve_and_print* as described in question 1. Make sure that *solve_and_print* has no arguments and prints the answers returned by your program in an easy to read format: you will lose marks if output is obscure. Arrange your output similar to the predicate *print_solution(List)* discussed in class. The TA should be able to call both your predicate *solve(List)* and your predicate *solve_and_print* to see the solution.

Before, you write a program that solves this CSP, implement the 2-argument predicate *fourExactly(X,List)* and use it in your program. Specifically, this predicate can be useful when you implement the 4th constraint: the list of points awarded to the teams has exactly 4 occurrences of "d". This predicate is true, if X occurs exactly four times in the given *List*. In your implementation, do not make assumptions about the number or kind of elements in *List*. This is because we will test your predicate independently of your solution. For example,

`fourExactly(q,[q,a,b,q,q,b,q,a,c,b,c]` should succeed. You might wish to introduce additional helping predicates to implement constraints. In any case, you have to implement all your helping predicates yourself. You cannot use any of the library predicates, unless they were discussed in class.

b. [**5 marks**] We should be able to query your program using both *solve(List)* and *solve_and_print* (no arguments). Make sure it is obvious from your output what the solution is. Find also how much time your program takes to compute an answer using the process described in the previous Part 1 of this assignment. Your Prolog session should be included in the file `q2_game_session.txt`

Thus, your submission should include both `q2_game.pl` and `q2_game_session.txt`

3 Science puzzle [40 marks]

As an exhibit for the annual school science fair, a school student planted five tomato plants in a row and labeled them from one through five. The student wanted to test the effectiveness of various fertilizers on tomato plant growth. The student decided to use a different fertilizer for each plant and record the final plant height, the number of tomatoes each plant yielded, and the total weight of the yield to determine fertilizer effectiveness. The fertilizers used were, in no particular order, bone-meal, compost, egg-shells, manure, and seaweed. The final height for the plants were 1 foot, 2 feet, 4 feet, 5 feet and 7 feet, in no specific order. Each plant yielded, in no particular order, 4, 6, 9, 12, and 21 tomatoes. Each plant produced, in no specific order, a weight of 3, 9, 10, 14, and 19 pounds. Write a Prolog program that follows a CSP technique explained in class. Based on the provided clues, your program must find how to match each plant number with the fertilizer, the plant height, the number of tomatoes produced, and the total weight of each plant's yield.

1. The 3rd plant was not fertilized with manure, and it produced a heavier weight than plant two, but that was a lighter weight than the manure-fertilized plant.
2. The 4th plant grew taller than plant three, but plant two grew taller than plant four.
3. Plants three, four and five were not fertilized with either seaweed or bone-meal.
4. The tallest and shortest plants did not produce either the most tomatoes or the heaviest weight.
5. The plant fertilized with egg-shells produced half as many tomatoes as plant one, but it did produce the heaviest weight.
6. Seaweed fertilized the tallest plant, but that plant produced the fewest tomatoes, and bone-meal fertilized the plant that produced the lightest weight.
7. Plant one was taller than plant three by one foot, but it only grew half as tall as plant five.
8. Plant five did not produced the heaviest weight.

a. [35 marks] Your task is to write a PROLOG program to solve this problem using the smart interleaving of generate-and-test technique explained in class: your program has to compute the match between the plants and their heights, yields and weights of produced tomatoes. *Do* **not** *attempt to solve any part of this puzzle yourself*, i.e., do not make any conclusions from the statements given to you. *You lose marks if your program implements your own reasoning.* To get full marks, you have to follow a design technique from class. Start with implementing two helping predicates *maxList(L,M)* and *minList(L,M)*. Use them when you formulate some of the constraints given above. These predicates should not make any assumptions about the length or contents of the input lists aside from the fact that you can assume the list contains unique numbers. This is because we will be testing your predicates independently of your CSP solution. For example, the query `maxList([5,-2,7,-89,100,-25],X)` should succeed and return `X=100`. *Hint:* You might wish to introduce a predicate for plants that associates a plant number with its yield, height, and weight. This can simplify the set of variables in your program, so it is easier to access the variables of a given plant when you formulate the constraints.

You will have to be careful in your program regarding the order of constraints. Explain briefly the ordering you have chosen (write comments in your program). Remember to implement all implicitly stated and hidden constraints. They must be formulated and included in the program to find a correct solution satisfying all constraints. You can use predicates from

class in your program, but rename them. If you like, you can introduce any additional helping predicates.

Determine how much computer time your computation takes using `cputime` construct. Never try to guess part of solution by yourself: all reasoning should be done by your program. You have to demonstrate whether you learned well a program design technique. You will lose marks, if the TA finds that you embedded some of your own reasoning into your program. Your *solve(List)* program should **not** use any of the numbers mentioned above as heights, weights or yields.

You should implement both a *solve(List)* and *solve_and_print* as described in question 1. These should be put in the file `q3_fertilizers.pl`. Explain briefly the ordering you have chosen (write comments in your program file). Make sure that *solve_and_print* prints the answers returned by your program in an easy to read format: you will lose marks if output is obscure.

b. [5 marks] You should also include your session testing your program in `q3_fertilizers_session.txt`. This test should include information about the runtime. We should be able to query your program using both *solve(List)* and *solve_and_print* (no arguments).

Thus, your submission should include both `q3_fertilizers.pl` and `q3_fertilizers_session.txt`