# Künstliche Intelligenz 1 – WS 2019/20

Michael Kohlhase
Informatik, FAU Erlangen-Nürnberg
For Course Purposes Only

December 6, 2019

## Contents

# 1 Assignment 1 (Prolog) – Given Oct. 25., Due Nov. 03.

**Hint:** Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, consult the lecture materials, ask a fellow student, your tutor, or on the Forum.

Comment your code or make it otherwise self-explanatory. You do not need to write a lot, but it should be enough to convince your tutor that you understand what the code does. We may deduct up to 30% for uncommented and unclear code, but would prefer not to.

---

This assignment is supposed to get you started with `ProLog`.

You will need a `ProLog` interpreter to solve the problems below. The SWI ProLog interpreter can be downloaded from `http://www.swi-prolog.org/`. An online interpreter is also avilable at `https://swish.swi-prolog.org`. The SWI manual is available at `http://www.swi-prolog.org/pldoc/`.

The lecture notes explain the basics of `ProLog`, in particular handling lists. If there are any commands missing that are strictly necessary to solve these exercises, use the course forum to let us know.

Please hand in your solutions as `ProLog` files – i.e. text files that use the .pl extension, not PDFs or Word files – so that we can run and check them.

Here and in general: *Write comments*! They make it a lot easier for your tutors to give points.

**Problem 1.1 (Basic PrologFunctions)**

Your task is to implement the functions listed below in `ProLog`. Note that many of them are built-in, but we ask you create your own functions. 40pt

1. a function removing multiple occurrances of elements in a list

```
?- removeDuplicates([1,1,1,1,2,2,3,4,1,2,7],A).
A = [1, 2, 3, 4, 7].
```

---

**Hint:** You may want to implement a helper method delete(X,LS,RS), that removes all instances of X in LS and returns the result in RS.

---

2. a function reversing a list

```
?- myReverse([1,2,3,4,2,5],R).
R = [5, 2, 4, 3, 2, 1].
```

3. a function outputting all the permutations of the elements in a list

```
?- myPermutations([1,2,3],Z).
Z = [1, 2, 3] ;
Z = [2, 1, 3] ;
Z = [2, 3, 1] ;
```

```
Z = [1, 3, 2] ;
Z = [3, 1, 2] ;
Z = [3, 2, 1].
```

This one is rather tricky. Try it out for yourself *first*. Ask yourself why this is seemingly difficult.

---

**Hint:** Start with a helper predicate takeout(X,LSA,LSB) that is true iff LSB is the result of removing the first occurence of X from LSA. How does this allow you to define the notion of *permutation* recursively?

---

4. The zip function takes two lists with lengths that differ at most by 1, and outputs a list of lists containing one element from the first list and the element with the same index from the other list, possibly followed by a one-element list with the left-over argument.

   Create a `ProLog` predicate with 3 arguments: the first two would be the two lists you want to zip, and the third one would be the result. For instance:

```
?− zip([1,2,3],[4,5,6],L).
L = [[1, 4], [2, 5], [3, 6]].
```

```
?− zip([1,2],[3,4,5],L).
L = [[1, 3], [2, 4], [5]].
```

---

**Solution**:

1. the remove duplicates function

```
delete(_,[],[]).
delete(X,[X|T],R) :− delete(X,T,R).
delete(X,[H|T],[H|R]) :− not(X=H), delete(X,T,R).
removeDuplicates([],[]).
removeDuplicates([H|T],[H|R]) :− delete(H,T,S), removeDuplicates(S,R).
```

2. the reverse function

```
preReverse([],X,X).
preReverse([X|Y],Z,W) :− preReverse(Y,[X|Z],W).
myReverse(A,R) :− preReverse(A,[],R).
```

3. the permute function

```
takeout(X,[X|T],T).
takeout(X,[H|T1],[H|T2]) :− takeout(X,T1,T2).

myPermutations([],[]).
myPermutations([X|Y],Z) :− myPermutations(Y,W), takeout(X,Z,W).
```

4. the zip function

```
zip([L],[],[[L]]).
zip([],[L],[[L]]).
zip([A],[B],[[A,B]]).
zip([H1|T1],[H2|T2],L) :- zip(T1,T2,T), append([[H1,H2]],T,L).
```

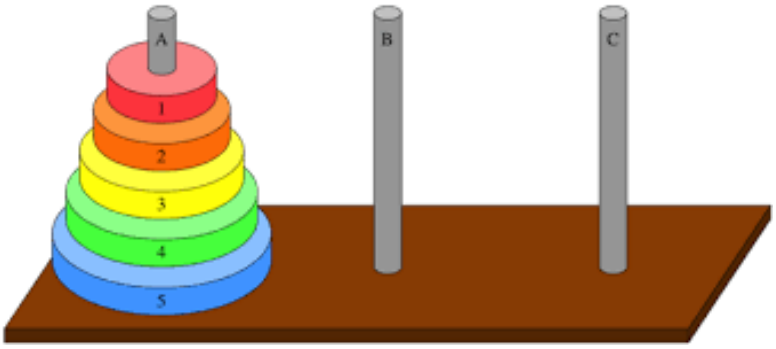
---

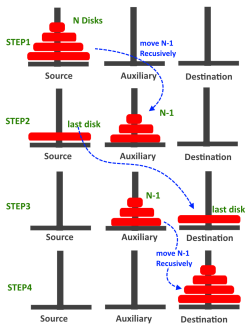
**Problem 1.2 (Towers of Hanoi)** 60pt

The Towers of Hanoi is a mathematical puzzle. It consists of three pegs ($A$, $B$, and $C$) and a number of disks of different sizes, which can slide onto any peg. The puzzle starts with the disks in a stack in ascending order of size on one peg, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move all disks from peg $A$ to peg $B$, while obeying the following rules:

1. only one disk can be moved at a time,
2. each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty peg,
3. no larger disk may be placed on top of a smaller disk.

The idea of the algorithm (for $N > 1$) is to move the top $N - 1$ disks onto the auxiliary peg, then move the bottom disk to the destination peg, and finally moving the remaining $N - 1$ disks from the auxiliary peg to the destination peg.




1. Write a `ProLog` predicate that prints out a solution for the Towers of Hanoi puzzle. Use the **write**(X) predicate that prints the value of $X$ ($X$ can be simple text or any type of argument) to the screen and **nl** that prints a new line to write a rule move(N, A, B, C) that prints out the solution for moving $N$ disks from peg $A$ to peg $B$, using $C$ as the auxiliary peg.

   Each step of the solution should be of the form "Move top disk from X to Y".

   Examples:

```
?- write(hello), write(' world!'), nl.
hello world!
true.
```

```
?− move(3, left, center, right).
Move top disk from left to center
Move top disk from left to right
Move top disk from center to right
Move top disk from left to center
Move top disk from right to left
Move top disk from right to center
Move top disk from left to center
true ;
false.
```

2. Determine the complexity class of your algorithm in terms of the number of disks $N$ and explain how you computed it.

---

**Solution**:

1. 
```
move(1, A, B, _) :−
    write('Move top disk from '),
    write(A), write(' to '), write(B), nl.
move(N, A, B, C) :−
    N>1, M is N−1,
    move(M, A, C, B), move(1, A, B, _), move(M, C, B, A).
```

2. Let $T(N)$ be the number of moves needed to move $N$ disks from one peg to another. Clearly, $T(1) = 1$. For $T(N)$, we have the following recursive relation:

$$T(N) = 2T(N-1) + 1$$

The values for $N = 1, 2, 3, 4, 5$ are $1$, $2+1$, $2^2+2+1$, $2^3+2^2+2+1$, and $2^4+2^3+2^2+2+1$. Thus, $O(T(n)) = 2^{n-1}$, which is exponential.

(You could also solve the non-homogenous linear recurrence to obtain a precise closed formula for $T(N)$.)

---

# 2 Assignment 2 (Prolog and Intelligent Agents) – Given Nov. 1., Due Nov. 10.

**Hint:** Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, consult the lecture materials, ask a fellow student, your tutor, or on the Forum.

Comment your code or make it otherwise self-explanatory. You do not need to write a lot, but it should be enough to convince your tutor that you understand what the code does. We may deduct up to 30% for uncommented and unclear code, but would prefer not to.

**Problem 2.1 (Binary Tree)** 40pt

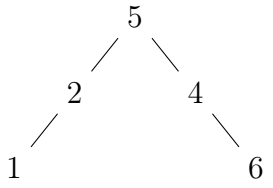
A binary tree of (in this case) natural numbers is inductively defined as a triple `tree(n,t1,t2)`, where `n` is a natural number and `t1` and `t2` are themselves binary trees.

A *leaf* is a terminal node, i.e. a tree of the form `tree(n,nil,nil)`. An example tree in prolog would be:

```
tree(1,tree(2,nil,nil),tree(2,nil,nil))
```

1. Write a `ProLog` function `construct` that constructs a binary tree out of a list of (distinct) numbers, such that for every subtree `tree(n,t1,t2)`, all values in `t1` are smaller than `n` and all values in `t2` are larger than `n`. E.g. the list $(5, 2, 4, 1, 3)$ would yield `tree(5, tree(2, tree(1, nil, nil), tree(4, tree(3, nil, nil), nil)), nil)`.
2. Write a `ProLog` function `tree_sum` that given a binary tree returns the sum of all integers in the tree.
3. Write a `ProLog` function `tree_wsum` that given a binary tree returns a weighted sum of all integers in the tree, using the depth of the node as a weight.
4. A tree is symmetrical if its right subtree is a mirror image of its left subtree (see examples below). Write a `ProLog` function `symmetric` that checks whether a binary tree is symmetrical.

Examples:




The sum of all the integers in the tree above is 18. The weighted sum using depth as a weight is

$$w_5 \cdot 5 + w_2 \cdot 2 + w_4 \cdot 4 + w_1 \cdot 1 + w_6 \cdot 6 = 1 \cdot 5 + 2 \cdot 2 + 2 \cdot 4 + 3 \cdot 1 + 3 \cdot 6 = 39.$$

The right subtree of this tree is a mirror image of the left subtree, so the tree is symmetrical (note that the values do not matter).

**Solution**:

```
add(X,nil,tree(X,nil,nil)).
add(X,tree(Root,L,R),tree(Root,L1,R)) :- X @< Root, add(X,L,L1).
add(X,tree(Root,L,R),tree(Root,L,R1)) :- X @> Root, add(X,R,R1).

construct(L,T) :- construct(L,T,nil).

construct([],T,T).
construct([N|Ns],T,T0) :- add(N,T0,T1), construct(Ns,T,T1).

tree_sum(nil,0).
tree_sum(tree(X,L,R),S) :- tree_sum(L,SL), tree_sum(R,SR), S is X + SL + SR.

tree_wsum(nil,_,0).
tree_wsum(tree(X,L,R),D1,S) :-
    D2 is D1+1,
    tree_wsum(L,D2,SL),
    tree_wsum(R,D2,SR),
    S is D1*X + SL + SR.

mirror(nil,nil).
mirror(t(_,L1,R1),t(_,L2,R2)) :- mirror(L1,R2), mirror(R1,L2).

symmetric(nil).
symmetric(t(_,L,R)) :- mirror(L,R).

symmetric(tree(1,tree(2,nil,nil),tree(2,nil,nil))).
% true.
symmetric(tree(1,tree(3,nil,nil),tree(2,nil,nil))).
% true.
```

---

**Problem 2.2** Explain the difference between agent function and agent program. How many agent programs can there be for a given agent function? 20pt

---

**Solution**: The function specifies the input-output relation (outside view). The program implements the function (inside view).

The function takes the full sequence of percepts as arguments. The program uses the internal state to avoid that.

There are either none or infinitely many programs for a function.

---

**Problem 2.3** For each of the following agents, develop a PEAS description of the task environment. 40pt

1. Robot soccer player
2. Internet book-shop agent (that is: an agent for book shops that stocks up on books depending on demand)
3. Autonomous Mars rover
4. Mathematical theorem prover

5. First-person shooter (Counterstrike, Unreal Tournament etc.)

Characterize the environments of these agents according to the properties discussed in the lecture. Where not "obvious", justify your choice with a short sentence.

Choose suitable designs for the agents.

# 3 Assignment 3 (Search) – Given Nov. 8., Due Nov. 17.

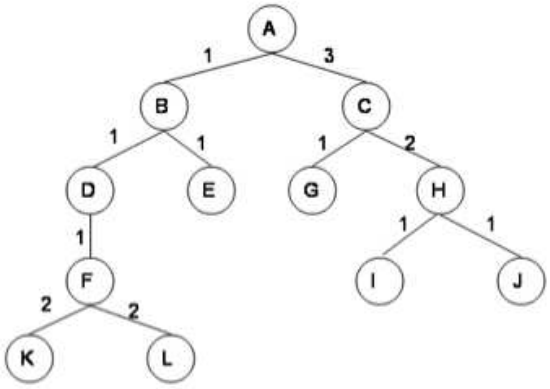
**Hint:** Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, consult the lecture materials, ask a fellow student, your tutor, or on the Forum.

Comment your code or make it otherwise self-explanatory. You do not need to write a lot, but it should be enough to convince your tutor that you understand what the code does. We may deduct up to 30% for uncommented and unclear code, but would prefer not to.

**Problem 3.1 (Search Strategy Comparison on Tree Search)**

Consider the tree shown below. The numbers on the arcs are the arc lengths. 20pt




Assume that the nodes are expanded in alphabetical order when no other order is specified by the search, and that the goal is state $G$. No visited or expanded lists are used. What order would the states be expanded by each type of search? Stop when you expand $G$. Write only the sequence of states expanded by each search.

| Search Type | Sequence of States |
|---|---|
| Breadth First | |
| Depth First | |
| Iterative Deepening (step size 1) | |
| Uniform Cost | |

**Solution**:

| Search Type | List of States |
|---|---|
| Breadth First | ABCDEG |
| Depth First | ABDFKLECG |
| Iterative Deepening | AABCABDECG |
| Uniform Cost | ABDECFG |

**Problem 3.2 (Tree Search in ProLog)**

70pt

Implement the tree search algorithms presented in the lectures in ProLog, meaning:

1. BFS
2. DFS
3. Iterative Deepening with variable step size
4. Uniform cost search

Use the following implementation of trees:

```
subtrees([]). % The empty list is a valid list of subtrees
istree(tree(Value,Children)) :- string(Value),subtrees(Children).
% A (valid) tree is a pair of some value/label
% (represented as a string) and a valid list of subtrees
subtrees([(Cost,T)|Rest]) :- number(Cost),istree(T), subtrees(Rest).
% A non-empty list is a valid list of subtrees, if it consists of pairs (Cost,T), where T
% is a valid tree and Cost is a number representing the step cost. i.e. the simple binary
% tree with root A, two children B, C and step costs A->B=2 and A->C=3 would be
%represented as: tree("A",[(2,tree("B",[])),(3,tree("C",[]))])
```

Here are the definitions and some test trees you can use: `https://swish.swi-prolog.org/p/Tree%20search%20-%20definitions.swinb`

Please print out the trace of expansion, that is, in which order the nodes were expanded in your solutions. This makes it much easier for us to grade. For example, for the tree

```
tree("A",[(2,tree("B",[])),(3,tree("C",[]))])
```
(1)

and goal `"C"`, print something like `"ABC"`.

**Solution**: `https://swish.swi-prolog.org/p/Tree%20search.swinb`

**Problem 3.3 ($A^*$ vs. BFS)**

10pt

Does $A^*$ search always expand fewer nodes than BFS? Justify you answer.

**Solution**: No. With a bad heuristic, $A^*$ can be forced to explore the whole space, just like BFS.

# 4 Assignment 4 (Game Play) – Given Nov. 14., Due Nov. 24.

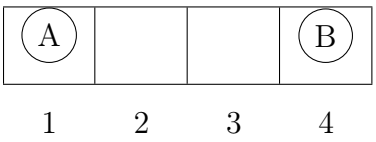
**Hint:** Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, consult the lecture materials, ask a fellow student, your tutor, or on the Forum.

Comment your code or make it otherwise self-explanatory. You do not need to write a lot, but it should be enough to convince your tutor that you understand what the code does. We may deduct up to 30% for uncommented and unclear code, but would prefer not to.

**Problem 4.1 (Simple Game)**

The figure below shows a starting position of a simple game. Player $A$ moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if $A$ is on 3 and $B$ is on 2, then $A$ may move back to 1.) The game ends when one player reaches the opposite end of the board. If player $A$ reaches space 4 first, then the value of the game to $A$ is $+1$; if player $B$ reaches space 1 first, then the value of the game to $A$ is $-1$. 60pt




(1) Draw the complete game tree, using the following conventions:

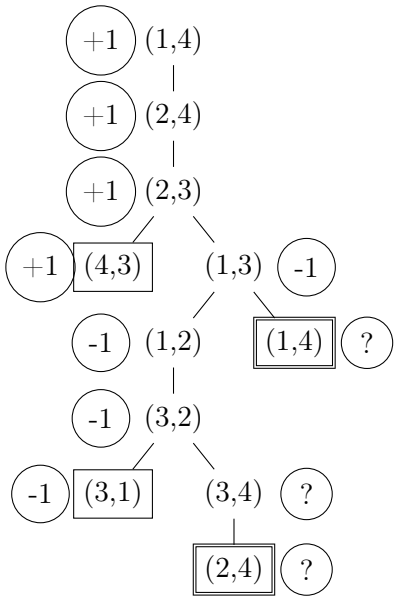
- Write each state as $(s_A, s_B)$, where $s_A$ and $s_B$ denote the token locations.
- Put each terminal state in a square box and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.

(2) The value of which nodes do the *loop states* affect? Mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

(3) Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (2). Does your modified algorithm give optimal decisions for all games with loops?

**Solution**:

(1) Game tree with game values and backed up minimax values.




(2) One way to deal with loop states is to make the assumption that the agent with a choice between winning the game and entering a loop state will always choose to win. In this case, the loop states affect all states that do not have a terminal state as a descendant; equivalently, these are all the states that only have states with value "?" as descendants.

(3) Standard minimax is depth-first and would go into an infinite loop. We can fix this by comparing the current state agains the stack to see if the state is repeated. This works in this case because the choice is always between a loop state and a win (either for Max of for Min). It would not work, for example, in games where the agent would need to make the decision between a draw and a loops state, as it is not clear how the comparison could be made.

---

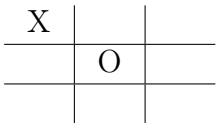
**Problem 4.2 (Tic Tac Toe - Gameplay)**

This problem exercises the basic concepts of game playing, using tic-tac-toe. We define $X_n$ as the number of rows, columns, or diagonals with *exactly* $n$ $X$'s and *no* $O$'s. Similarly, $O_n$ is the number of rows, columns, or diagonals with *exactly* $n$ $O$'s and *no* $X$'s. The utility function assigns $+1$ to any position with $X_3 = 1$ and $-1$ to any position with $O_3 = 1$. All other terminal positions have utility 0. For nonterminal positions, we use a linear evaluation function defined as 40pt

$$\text{Eval}(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$$

(1) Approximately how many possible games of tic-tac-toe are there?

(2) Draw (for yourself) the game tree starting from an empty board down to depth 2 (i.e., one $X$ and one $O$ on the board), taking symmetry into account. That is, you do not need to show "duplicate" states that could be obtained from ones already in the tree by rotation or reflection. (*Hint.* You should end up with 12 leaves.) List the states at depth 2 as lists of triples, so that each triple represents a row. For example, the state




can be represented as

$$[[X, \_, \_], [\_, O, \_], [\_, \_, \_]]$$

(3) Compute the evaluation function for the leaf nodes from (2).

(4) Which of the leaf nodes from (2) with values from (3) would *not* be evaluated in the next step if alpha–beta pruning were applied, assuming the nodes are generated in the optimal order for alpha-beta pruning.

---

**Solution**:

(1) There are at most 9! games. This is the number of move sequences that fill up the board, but many wins and losses end before the board is full.

(2) Leaf nodes with values for the evaluation function:

| leaf node | value | evaluated in next step? |
|---|---|---|
| $[[X, O, \_], [\_, \_, \_], [\_, \_, \_]]$ | 1 | no |
| $[[X, \_, \_], [\_, O, \_], [\_, \_, \_]]$ | $-1$ | yes |
| $[[X, \_, O], [\_, \_, \_], [\_, \_, \_]]$ | 0 | no |
| $[[X, \_, \_], [\_, \_, \_], [\_, \_, O]]$ | 0 | no |
| $[[X, \_, \_], [\_, \_, O], [\_, \_, \_]]$ | 1 | no |
| $[[O, \_, \_], [\_, X, \_], [\_, \_, \_]]$ | 1 | yes |
| $[[\_, O, \_], [\_, X, \_], [\_, \_, \_]]$ | 2 | yes |
| $[[\_, X, O], [\_, \_, \_], [\_, \_, \_]]$ | $-1$ | no |
| $[[\_, X, \_], [\_, O, \_], [\_, \_, \_]]$ | $-2$ | yes |
| $[[\_, X, \_], [\_, \_, \_], [\_, O, \_]]$ | 0 | no |
| $[[\_, X, \_], [\_, \_, \_], [O, \_, \_]]$ | $-1$ | no |
| $[[\_, X, \_], [O, \_, \_], [\_, \_, \_]]$ | 0 | no |

(3) See above for which nodes get evaluated. In the optimal order for alpha-beta pruning, the children of the node $[[\_, \_, \_], [\_, X, \_], [\_, \_, \_]]$ get evaluated first, giving us $\alpha = 1$. In the optimal order for the other two nodes the algorithm finds the values $-1$ for node $[[X, \_, \_], [\_, \_, \_], [\_, \_, \_]]$ and $-2$ for node $[[\_, X, \_], [\_, \_, \_], [\_, \_, \_]]$. Both of these are less than $\alpha$, so the rest of the children do not get evaluated.

---

# 5 Assignment 5 (Constraint Satisfaction 1) – Given Nov. 21., Due Dec. 1.

**Hint:** Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, consult the lecture materials, ask a fellow student, your tutor, or on the Forum.

Comment your code or make it otherwise self-explanatory. You do not need to write a lot, but it should be enough to convince your tutor that you understand what the code does. We may deduct up to 30% for uncommented and unclear code, but would prefer not to.

**Problem 5.1 (Map Coloring in Prolog)** 50pt

We know that it is possible to color a (sufficiently nice) map with at most four colors, so that no two adjacent countries are the same color. Online (linked below), you will find an incomplete Prolog program. The program already has a predicate `paint(Map,ColoredMap)` that produces a colored map.

You will notice that the variable `TempList` appears in several places. It always holds the current color assignments in the format $[country1 : color1, country2 : color2, ...]$.

This problem is based on the one here:
`https://www.tjhsst.edu/~rlatimer/assignments2004/mapcolorsProlog.html`.
**Please only submit the "Edit section".**

(1) Edit `arrangecolors` so that `paint` will produce a coloring of the map so that no two adjacent countries are the same color. You can use the predicate `different`.

(2) Which line causes Prolog to try new colors for some country?

(3) Implement the Minimum Remaining Values heuristic. Write a predicate `restrictedness` (a template is already in the notebook) such that `Neighbors` is a list of countries that are neighbors of country $C$ and `R` is the number of colors that we can use color country $C$.

(4) The predicate `restrictedness` is used in `sortedCountries` to produce a list of countries ordered by restrictedness (smallest restrictedness first). Edit `arrangecolors` so that at each step, the algorithm first assigns a color to the most restricted country.

(5) What would be a good heuristic to use a tiebreaker whenever there is more than one country with minimal remaining values?

The notebook:
`https://swish.swi-prolog.org/p/Map%20Coloring%20-%20setup.swinb`.
In case Swish is offline, here is a file with the same code:
`https://gl.kwarc.info/teaching/AI/blob/master/WS1920/hw5-setup.txt`

**Solution**:

**Problem 5.2 (Kalah Framework)**
Familiarize yourself with the Kalah framework at 0pt

`https://github.com/KWARC/Kalah-Framework`.

Over the course of the next weeks, implement your own agent as teams of maximally 3 students per team. The precise deadline will be announced next week. You can already start implementing your agent – e.g. by implementing methods that simulate a move, or compute the full game tree...

We follow the rules described at the introduction section of `https://en.wikipedia.org/wiki/Kalah` with variable numbers of houses and seeds.

The team with the best agent receives an additional(!) 100 points, the 2nd team 90 points, the 3rd 80 etc.

**Please read the instructions and rules specified in the readme of the git repository carefully!**

**Solution**:

# 6 Assignment 6 (Constraint Satisfaction 2 and Kalah Tournament) – Given Nov. 27., Due Dec. 10.

**Hint:** Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, consult the lecture materials, ask a fellow student, your tutor, or on the Forum.

Comment your code or make it otherwise self-explanatory. You do not need to write a lot, but it should be enough to convince your tutor that you understand what the code does. We may deduct up to 30% for uncommented and unclear code, but would prefer not to.

**Problem 6.1 (Scheduling CS Classes)** 50pt

You are in charge of scheduling for computer science classes that meet Mondays, Wednesdays and Fridays. There are 5 classes that meet on these days and 3 professors who will be teaching these classes. You are constrained by the fact that each professor can only teach one class at a time. The classes are:

- Class 1 - Intro to Programming: meets from 8:00-9:00am
- Class 2 - Intro to Artificial Intelligence: meets from 8:30-9:30am
- Class 3 - Natural Language Processing: meets from 9:00-10:00am
- Class 4 - Computer Vision: meets from 9:00-10:00am
- Class 5 - Machine Learning: meets from 9:30-10:30am

The professors are:

- Professor A, who is available to teach Classes 3 and 4.
- Professor B, who is available to teach Classes 2, 3, 4, and 5.
- Professor C, who is available to teach Classes 1, 2, 3, 4, 5.

1. Formulate this problem as a CSP problem in which there is one variable per class, stating the domains, and constraints. Constraints should be specified formally and precisely, but may be implicit rather than explicit.

2. Give the constraint graph associated with your CSP

3. Show the domains of the variables after running arc-consistency on this initial graph (after having already enforced any unary constraints).

4. List all optimal cutsets for the constraint graph associated with the CSP.

**Problem 6.2 (Kalah Tournament)** 100pt

Download the Kalah framework at `https://github.com/KWARC/Kalah-Framework`. Implement your own agent as teams of maximally 3 students per team.

We follow the rules described at the introduction section of `https://en.wikipedia.org/wiki/Kalah` with variable numbers of houses and seeds.

The team with the best agent receives an additional 100 points, the 2nd team 90 points, the 3rd 80 etc.

**Please read the instructions and rules specified in the readme of the git repository carefully!**

**Use the forum to ask questions!**

If you have a question, it is almost certain that somebody else has the same question.

# 7 Assignment 7 (Propositional Logic) – Given Dec. 6., Due Dec. 15.

**Hint:** Exercises need to be handed in via StudOn at 23:59 on the day they are due or earlier. Please use only the exercise group of your tutor to hand in your work.

If any concepts here seem unfamiliar to you or you have no idea how to proceed, consult the lecture materials, ask a fellow student, your tutor, or on the Forum.

Comment your code or make it otherwise self-explanatory. You do not need to write a lot, but it should be enough to convince your tutor that you understand what the code does. We may deduct up to 30% for uncommented and unclear code, but would prefer not to.

**Problem 7.1 (PL Semantics)** 30pt

Prove whether the following formulae are valid. If not, give an assignment that serves as counterexample.

Use the definition of *interpretation* and *assignments* for your proofs (i.e. specifically no truth tables!)

- $A \Rightarrow (B \Rightarrow A)$
- $(A \wedge B) \Rightarrow (A \wedge C)$

**Problem 7.2 (Natural Deduction)**
Prove (or disprove) the validity of the following formulae in Natural Deduction: 30pt

1. $(P \wedge Q) \Rightarrow (P \vee Q)$
2. $((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C$
3. $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$

**Problem 7.3** We want to work with propositional formulae in Prolog. We introduce new Prolog operators to handle the logical connectives (this is already provided in the template). We can use the new operators to represent e.g. the expression $(a \wedge b) \Leftrightarrow (\neg a \vee b)$ as (a and b) equi (**not** a **or** b). The template can be found at `https://kwarc.info/teaching/AI/resources/`. 40pt

- Implement a predicate simplify replaces all logical connectives by conjunction and negation. Examples:

```
?− simplify(a or b, X).
X = (not (not a and not b)).

?− simplify(not (a or b), X).
X = (not (not (not a and not b))).

?− simplify(a equi b, X).
X = (not (not (not a)and not b)and not (not (not b)and not a)).
```

Note that there is more than one possible simplification of a term, so your results may be different (but should be correct nevertheless!).

- Implement a predicate eval that evaluates a formula given a list of variable assignments. Examples:

```
?− eval(a impl b, [a:true, b:false], R).
R = false.

?− eval(a impl b, [a:true, b:true], R).
R = true.
```

If not all variables are assigned, the predicate should fail:

```
?− eval(a impl b, [a:true], R).
false.
```

- Bonus: Implement a predicate model that generates all variable assignments that make a formula true. Examples:

```
?− model(a impl b, M).
M = [a:true, b:true] ;
M = [a:false, b:true] ;
M = [a:false, b:false].

?− model(a and not a, M).
false.

?− model((not a) and (a or b), M).
M = [a:false, b:true] ;
false.
```

---

**Hint:** You may want to create a predicate that generates all possible variable assignments and then check which ones work.

---

**Hint:** You may want to use the simplify predicate for the second and third problem.

---