Assignment1: Prolog, PEAS
– Given Oct 29, Due Nov 6 –

This exercise sheet gets you started with Prolog. To complete the assignments you can use

- a Prolog interpreter, e.g., `http://www.swi-prolog.org/`
- an online interpreter such as `https://swish.swi-prolog.org`.

However, for submission you will have to upload a single Prolog file for each problem. The files must run using the SWI Prolog interpreter. The manual for SWI Prolog is available at `http://www.swi-prolog.org/pldoc/`.

Now and for all future programming problems, your solutions must

- strictly follow the specification in the problem statement, in particular regarding the names and arities of the predicates — we may do automated testing!
- include comments that explain how your code works including example invocations of your programs — a key grading criterion is how easy it is for the grading tutor to verify that you solved the problem correctly.

Problem 1.1 (Basic Prolog Functions) 35 pt

Implement the functions listed below in Prolog. Note that many of them are built-in, but we ask you create your own functions.

1. a function reversing a list
   Test case:
   ```
   ?- myReverse([1,2,3,4,2,5],R).
   R = [5, 2, 4, 3, 2, 1].
   ```

2. a function removing multiple occurrences of elements in a list
   Test case:
   ```
   ?- 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`.

---

3. a function for zipping two lists
   `zip` takes two lists and outputs a list of pairs (represented as 2-element lists) of elements at the same index in the two lists. If the lists do not have the same length, the zipped list contains only as many pairs as the shorter list.

   Create a Prolog predicate with 3 arguments: the first two are the two lists to zip and the third one 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]].
```

4. a function for computing permutations of a list
   Try it out on paper first and understand why this is difficult.

   Test case:

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

   Note that there are two ways for specifying such a function:

   (a) return a list of all permutations

   (b) return a single permutation each time such that Prolog finds them one by one.

   Here we are using the second way, i.e., `myPermutations(L,P)` must in particular be true if `P` is some permutation of `L`.

Hint: One possible solution is to start with a helper predicate `takeout(X,L,M)` that is true iff `M` is the result of removing the first occurrence of `X` from `L`. Or equivalently: `M` arises by adding `X` somewhere in `L`. How does this allow you to define the notion of permutation recursively?

Problem 1.2 (Binary Tree) 25 pt

A binary tree of (in this case) natural numbers is inductively defined as either

- an expression of the form `tree(n,t1,t2)` where `n` is a natural number (the label of the node) and `t1` and `t2` are themselves binary trees (the children of that node)
- or `nil` for the empty tree. (Normally a tree cannot be empty, but it is more convenient here to allow an empty tree as well.)

In particular, the nodes of the form `tree(n,nil,nil)` are the *leaf* nodes of the tree, the others are the *inner* nodes.

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`.

   Note that there are usually multiple such trees for every list. One example is:

```
?- construct([3,2,4,1,5],T).
T = tree(3, tree(2, tree(1, nil, nil), nil), tree(4, nil, tree(5, nil, nil))).
```

2. Write Prolog functions `count_nodes` and `count_leafs` that take a binary tree and return the number of nodes and leaves, respectively.

3. Write a Prolog function `symmetric` that checks whether a binary tree is symmetric.

Problem 1.3 40 pt

For each of the following agents, develop a PEAS description of the task environment.

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.)

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

Finally, choose suitable designs for the agents.

Assignment1: Prolog, PEAS
– Given Oct 29, Due Nov 6 –

This exercise sheet gets you started with Prolog. To complete the assignments you can use

- a Prolog interpreter, e.g., `http://www.swi-prolog.org/`
- an online interpreter such as `https://swish.swi-prolog.org`.

However, for submission you will have to upload a single Prolog file for each problem. The files must run using the SWI Prolog interpreter. The manual for SWI Prolog is available at `http://www.swi-prolog.org/pldoc/`.

Now and for all future programming problems, your solutions must

- strictly follow the specification in the problem statement, in particular regarding the names and arities of the predicates — we may do automated testing!
- include comments that explain how your code works including example invocations of your programs — a key grading criterion is how easy it is for the grading tutor to verify that you solved the problem correctly.

Problem 1.1 (Basic Prolog Functions) 35 pt

Implement the functions listed below in Prolog. Note that many of them are built-in, but we ask you create your own functions.

1. a function reversing a list
   Test case:
   ```
   ?- myReverse([1,2,3,4,2,5],R).
   R = [5, 2, 4, 3, 2, 1].
   ```

2. a function removing multiple occurrences of elements in a list
   Test case:
   ```
   ?- 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`.

---

3. a function for zipping two lists
   `zip` takes two lists and outputs a list of pairs (represented as 2-element lists) of elements at the same index in the two lists. If the lists do not have the same length, the zipped list contains only as many pairs as the shorter list.

   Create a Prolog predicate with 3 arguments: the first two are the two lists to zip and the third one 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]].
```

4. a function for computing permutations of a list
   Try it out on paper first and understand why this is difficult.

   Test case:

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

   Note that there are two ways for specifying such a function:

   (a) return a list of all permutations

   (b) return a single permutation each time such that Prolog finds them one by one.

   Here we are using the second way, i.e., `myPermutations(L,P)` must in particular be true if `P` is some permutation of `L`.

Hint: One possible solution is to start with a helper predicate `takeout(X,L,M)` that is true iff `M` is the result of removing the first occurrence of `X` from `L`. Or equivalently: `M` arises by adding `X` somewhere in `L`. How does this allow you to define the notion of permutation recursively?

Solution:

1. the reverse function

```
% myReverseAcc uses an additional argument (the second one) as an accumulator
% in which the result is built.
% Its invariant is that myReverserAcc(X,Y,Z) iff reverse(X);Y = Z.
% When the first argument is empty, we return the accumulated result.
myReverserAcc([],X,X).
% When the first argument is non-empty, we take its first element and
% prepend it to the accumulator.
myReverserAcc([X|Y],Z,W) :- myReverserAcc(Y,[X|Z],W).
% To compute the reversal, we initialize the accumulator with the empty list.
myReverse(A,R) :- myReverserAcc(A,[],R).
```

2. 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).
```

3. the zip function

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

4. the permute function

```
takeout(X,[X|T],T).
takeout(X,[H|T1],[H|T2]) :- not(X=H), takeout(X,T1,T2).

% There is exactly one permutation of the empty list.
myPermutations([],[]).
% To find a permutation P of a longer list [H|T], we permute T into Q
% and insert H somewhere into Q.
myPermutations([H|T],P) :- myPermutations(T,Q), takeout(H,P,Q).
```

Note that we defined `takeout` in such a way that the second argument is input and the third one output. But Prolog does not distinguish input and output: when we use it later, we use the third argument as input and the second one as output.

Problem 1.2 (Binary Tree) 25 pt

A binary tree of (in this case) natural numbers is inductively defined as either

- an expression of the form `tree(n,t1,t2)` where `n` is a natural number (the label of the node) and `t1` and `t2` are themselves binary trees (the

children of that node)

- or `nil` for the empty tree. (Normally a tree cannot be empty, but it is more convenient here to allow an empty tree as well.)

In particular, the nodes of the form `tree(n,nil,nil)` are the *leaf* nodes of the tree, the others are the *inner* nodes.

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`.

   Note that there are usually multiple such trees for every list. One example is:

   ```
   ?- construct([3,2,4,1,5],T).
   T = tree(3, tree(2, tree(1, nil, nil), nil), tree(4, nil, tree(5, nil, nil))).
   ```

2. Write Prolog functions `count_nodes` and `count_leafs` that take a binary tree and return the number of nodes and leaves, respectively.

3. Write a Prolog function `symmetric` that checks whether a binary tree is symmetric.

add_node .= base
add left ;
add_right.

Solution:

```
% add(X,S,T) inserts a node with label X into tree S yielding tree T
% Inserting into the empty tree yields a tree with a single node.
add(X,nil,tree(X,nil,nil)).
% To insert an element smaller than the root, insert on the left.
add(X,tree(Root,L,R),tree(Root,L1,R)) :- X @< Root, add(X,L,L1).
% To insert an element bigger than the root, insert on the right.
add(X,tree(Root,L,R),tree(Root,L,R1)) :- X @> Root, add(X,R,R1).

% To construct a binary tree T, from a list L, we insert all elements in order.
% We use an accumulator that we initialize with the empty tree.
construct(L,T) :- constructAcc(L,T,nil).
% At the end of the list, we return the accumulator.
constructAcc([],T,T).
% For each element of the list, we add it to the accumulator A (obtaining A1) and recurse.
constructAcc([N|Ns],T,A) :- add(N,A,A1), constructAcc(Ns,T,A1).

% The empty tree has no nodes.
count_nodes(nil,0).
% An inner node has one more node than its child trees together.
count_nodes(tree(_,L,R),N) :- count_nodes(L,NL), count_nodes(R,NR), N is NL+NR+1.
% Note that we do not need an additional case for leaf nodes here.

% The empty tree has no leaves.
count_leafs(nil,0).
% A leaf node has 1 leaf (itself).
count_leafs(tree(_,nil,nil),1).
% An inner node has as many leaves as its child trees together.
count_leafs(tree(_,L,R),N) :- count_leafs(L,NL), count_leafs(R,NR), N is NL+NR.

% The empty tree is symmetric.
symmetric(nil).
% Any other tree is symmetric if its two child trees are mirror images of each other.
symmetric(tree(_,L,R)) :- mirror(L,R).

% The empty tree is its own mirror image.
mirror(nil,nil).
% Otherwise, the mirror image arises by mirroring and swapping the child trees.
mirror(tree(X,L1,R1),tree(X,L2,R2)) :- mirror(L1,R2), mirror(R1,L2).

%A few tests
test1(X):- construct([5,2,4,1,3],Y), count_leafs(Y,X).
% X=2
test2(X):- construct([6,10,5,2,9,4,8,1,3,7],Y), count_leafs(Y,X).
% X=3
symmetric(tree(1,tree(2,nil,nil),tree(2,nil,nil))).
% true.
symmetric(tree(1,tree(3,nil,nil),tree(2,nil,nil))).
% false.
```

To construct a tree use helper

count = 1 + left + right

leaf = 0 ; leaf = 1 left + right.

Symmetric? (Helper method mirror

(mirror(L1,R2), mirror(L2,R1).

Problem 1.2
- Unary add, mul
Unary fibonacci

Problem 1.3 40 pt

For each of the following agents, develop a PEAS description of the task environment.

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.)

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

Finally, choose suitable designs for the agents.

"u-exp"
$x^{(y+1)} = z$
if
$x^y = w$
$w \times x = z$

uadd
$x + (y+1) = (z+1)$
if $x + y = z$
} uadd

$y+1 \in z+1$ represented as $s(y)$ and $s(z)$

umul
$x * (y+1) = z$
if $x * y = w$ and $w + x = z$
} umul.

fibonacci of
$fib(x+2) = y$
if $fib(x+1) = z$, $fib(x) = w$, $z + w = y$
} ufib.

Assignment2: Search
– Given Nov. 3., Due Nov. 13. –

Problem 2.1 20 pt

Explain the difference between agent function and agent program. How many agent programs can there be for a given agent function?

Problem 2.2 20 pt

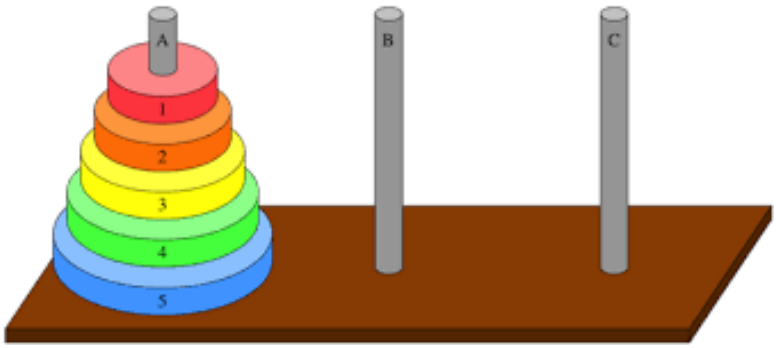
Explain the commonalities of and the differences between the performance measure and the utility function.

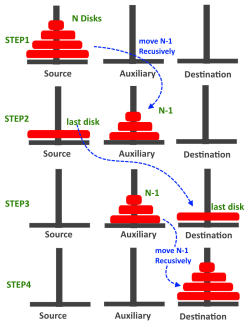
Problem 2.3 (Towers of Hanoi) 50 pt

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.

Problem 2.4 (Mathematical Notation) 10 pt

Let $\mathbb{N}$ be the set of natural numbers. A monoid is a mathematical structure $\langle U, \circ, e \rangle$ where U is a set, $\circ$ is an associative binary function on U, and e is the neutral element of $\circ$.

Express the following concepts in mathematical notation:

1. the set containing all natural numbers
2. the set containing the set of natural numbers
3. the set containing all square numbers

4. the set containing all even natural numbers
5. the set containing all even square numbers
6. the 3-tuple of 0, 1, and 2
7. the n-tuple of all numbers from 0 to $n - 1$
8. the set of pairs of natural numbers and their squares
9. the pair of sets of natural numbers and square numbers
10. the monoid of natural numbers under addition
11. the pair of monoids of the natural numbers under addition and under multiplication
12. the set of the monoids of the natural numbers under addition and under multiplication
13. given a monoid $\langle U, \circ, e \rangle$, the set of elements that are not the neutral element
14. given a monoid $\langle U, \circ, e \rangle$, the monoid in which the operation is flipped

Assignment2: Search
– Given Nov. 3., Due Nov. 13. –

Problem 2.1 20 pt

Explain the difference between agent function and agent program. How many agent programs can there be for a given agent function?

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 2.2 20 pt

Explain the commonalities of and the differences between the performance measure and the utility function.

Solution: Both measure how well an agent is doing.

The performance measure is a meta-level object that defines the quality of any agent used to solve the task. It may be defined informally (but still precisely) because it only needs to be used by an outside observer, such as a human comparing multiple agents.

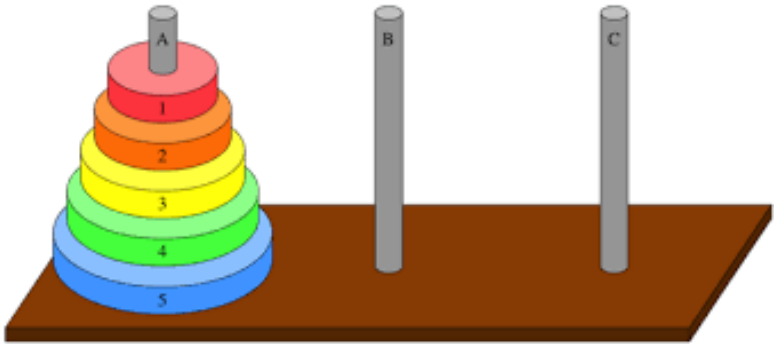
A utility function is a component of a particular utility-based agent. It must be defined formally (e.g., in a specification or programming language) because it must be computed as a part of applying the agent.

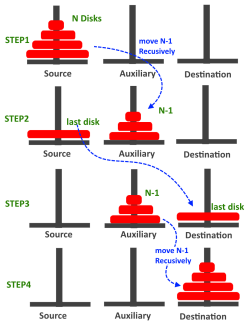
Problem 2.3 (Towers of Hanoi) 50 pt

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.





Move recursively N-1 to Auxiliary.
Move 1 from source to destination.
Move recursively N-1 from Auxiliary to destination.

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 $T(N)$.)

Problem 2.4 (Mathematical Notation) 10 pt

Let $\mathbb{N}$ be the set of natural numbers. A monoid is a mathematical structure $\langle U, \circ, e \rangle$ where U is a set, $\circ$ is an associative binary function on U, and e is the neutral element of $\circ$.

Express the following concepts in mathematical notation:

1. the set containing all natural numbers
2. the set containing the set of natural numbers
3. the set containing all square numbers
4. the set containing all even natural numbers
5. the set containing all even square numbers
6. the 3-tuple of 0, 1, and 2
7. the n-tuple of all numbers from 0 to $n-1$
8. the set of pairs of natural numbers and their squares
9. the pair of sets of natural numbers and square numbers
10. the monoid of natural numbers under addition
11. the pair of monoids of the natural numbers under addition and under multiplication

12. the set of the monoids of the natural numbers under addition and under multiplication
13. given a monoid $\langle U, \circ, e\rangle$, the set of elements that are not the neutral element
14. given a monoid $\langle U, \circ, e\rangle$, the monoid in which the operation is the same but with left and right argument switched.

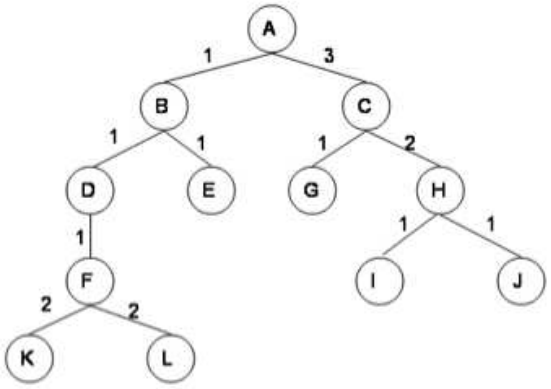
Solution: Note: The notation $\langle a_1, \ldots, a_n\rangle$ for a tuple is specific to the AI lecture. You can also write the tuple as $(a_1, \ldots, a_n)$.

1. $\mathbb{N}$
2. $\{\mathbb{N}\}$
3. $Squares := \{n \in \mathbb{N} | \exists m \in \mathbb{N}. n = m^2\}$ (selecting a subset by a property) or $\{n^2 : n \in \mathbb{N}\}$ (generating a set by applying a function to all elements of a set)
4. $Evens := \{n \in \mathbb{N} | \exists m \in \mathbb{N}. n = 2m\}$ or $\{2n : n \in \mathbb{N}\}$
5. $Squares \cap Evens$
6. $(0, 1, 2)$
7. $(0, \ldots, n-1)$
8. $\{(n, n^2) : n \in \mathbb{N}\}$
9. $(\mathbb{N}, Squares)$
10. $NatAdd := (\mathbb{N}, +, 0)$
11. Let $NatMult := (\mathbb{N}, \cdot, 1)$. Then $(NatAdd, NatMult)$.
12. $\{NatAdd, NatMult\}$
13. $\{u \in U | u \neq e\}$
14. (U, o, e) where o is the function $(x, y) \mapsto y \circ x$

Artificial Intelligence 1
Assignment3: Search
– Given Nov. 12., Due Nov. 20. –

Problem 3.1 (Search Strategy Comparison on Tree Search) 20 pt

Consider the tree shown below. The numbers on the arcs are the arc lengths (i.e., the costs of that arc). In what order are the nodes examined by each type of search if they stop after examining the goal node I?



To enable automatic grading, when submitting your solution, upload a text file containing exactly the following:

- one line for each search algorithm in the order BFS, DFS, Iterative Deepening (with step size increasing by 1), Uniform Cost,
- in each line, the letters identifying the nodes in the correct order (upper case, no whitespace),
- if a search algorithm visits a node multiple times, every occurrence should be included,
- if there is a tie between nodes, the algorithms should examine nodes in alphabetical order.

Problem 3.2 (Tree Search in ProLog) 50 pt

Implement the following tree search algorithms in Prolog:

1. BFS
2. DFS
3. Iterative Deepening (with step size 1)

Remarks:

- In the lecture, we talked about *expanding* nodes. That is relevant in many AI applications where the tree is not built yet (and maybe even too big to hold in memory), such as game trees in move-based games or decision trees of agents interacting with an environment. In those cases, when visiting a node, we have to expand it, i.e., compute what its children are.

  In this problem, we work with smaller trees where the search algorithm receives the fully expanded tree as input. The algorithm must still visit every node and perform some operation on it — the search algorithm determines in which order the nodes are visited.

  In our case, the operation will be to *write out the label* of the node.

- In the lecture, we worked with goal nodes, where the search stops when a goal node is found. Here we do something simpler: we *visit all the nodes and operate on each one* without using a goal state. (Having a goal state is then just the special case where the operation is to test the node and possibly stop.)

Concretely, your submission **must** be a single Prolog file that extends the following implementation:

```
% tree(V,TS) represents a tree.
% V must be a string - the label/value/data V of the root node
% TS must be a list of trees - the children/subtrees of the root node
% In particular, a leaf node is a tree with the empty list of children
istree(tree(V,TS)) :- string(V), istreelist(TS).

% istreelist(TS) holds if TS is a list of trees.
% Note that the children are a list not a set, i.e., they are ordered.
istreelist([]).
istreelist([T|TS]) :- istree(T), istreelist(TS).

% The following predicates define search algorithms that take a tree T
% and visit every node each time writing out the line D:L where
% * D is the depth (starting at 0 for the root)
% * L is the label

% dfs(T) visits every node in depth-first order
dfs(T) :- ???
% bfs(T) visits every node in beadth-first order
```

```
bfs(T) :- ???
%itd(T):- visits every node in iterative deepening order
itd(T) :- ???
```

Here "must" means you can define any number of additional predicates. But the predicates specified above must exist and must have that arity and must work correctly on any input `T` that satisfies `istree(T)`. "working correctly" means the predicates must write out exactly what is specified, e.g.,

```
0:A
1:B
```

for the depth-first search of the tree `tree("A",[tree("B",[])])`.

Problem 3.3 (Formally Modeling a Search Problem) 30 pt

Consider the Towers of Hanoi for 7 disks initially stacked on peg A.

Is this problem deterministic? Is it fully observable?

Formally model it as a Search Problem in the sense of the mathematical definition from the slides. Explain how your mathematical definition models the problem.

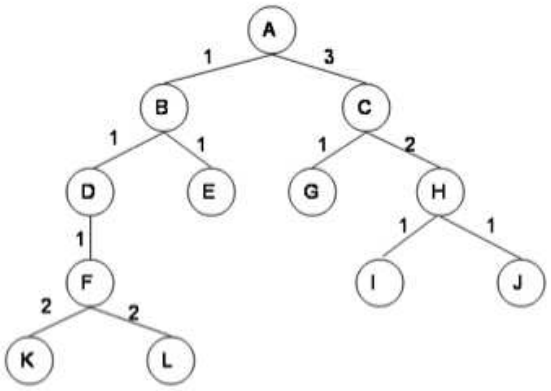
Note that the formal model only defines the problem — we are not looking for solutions here.

Note that modeling the problem corresponds to defining it in a programming language, except that we use mathematics instead of a programming language. Then explaining the model corresponds to documenting your implementation.

Artificial Intelligence 1
Assignment3: Search
– Given Nov. 12., Due Nov. 20. –

Problem 3.1 (Search Strategy Comparison on Tree Search) 20 pt

Consider the tree shown below. The numbers on the arcs are the arc lengths (i.e., the costs of that arc). In what order are the nodes examined by each type of search if they stop after examining the goal node I?



To enable automatic grading, when submitting your solution, upload a text file containing exactly the following:

- one line for each search algorithm in the order BFS, DFS, Iterative Deepening (with step size increasing by 1), Uniform Cost,
- in each line, the letters identifying the nodes in the correct order (upper case, no whitespace),
- if a search algorithm visits a node multiple times, every occurrence should be included,
- if there is a tie between nodes, the algorithms should examine nodes in alphabetical order.

Solution:

Search Type	List of States
Breadth First	ABCDEGHFI
Depth First	ABDFKLECGHI
Iterative Deepening	AABCABDECGHABDFECGHI
Uniform Cost	ABDECFGHKLI

Problem 3.2 (Tree Search in ProLog) 50 pt

Implement the following tree search algorithms in Prolog:

1. BFS
2. DFS
3. Iterative Deepening (with step size 1)

Remarks:

- In the lecture, we talked about *expanding* nodes. That is relevant in many AI applications where the tree is not built yet (and maybe even too big to hold in memory), such as game trees in move-based games or decision trees of agents interacting with an environment. In those cases, when visiting a node, we have to expand it, i.e., compute what its children are.

  In this problem, we work with smaller trees where the search algorithm receives the fully expanded tree as input. The algorithm must still visit every node and perform some operation on it — the search algorithm determines in which order the nodes are visited.

  In our case, the operation will be to *write out the label* of the node.

- In the lecture, we worked with goal nodes, where the search stops when a goal node is found. Here we do something simpler: we *visit all the nodes and operate on each one* without using a goal state. (Having a goal state is then just the special case where the operation is to test the node and possibly stop.)

Concretely, your submission **must** be a single Prolog file that extends the following implementation:

```
% tree(V,TS) represents a tree.
% V must be a string - the label/value/data V of the root node
% TS must be a list of trees - the children/subtrees of the root node
% In particular, a leaf node is a tree with the empty list of children
istree(tree(V,TS)) :- string(V), istreelist(TS).

% istreelist(TS) holds if TS is a list of trees.
% Note that the children are a list not a set, i.e., they are ordered.
istreelist([]).
istreelist([T|TS]) :- istree(T), istreelist(TS).

% The following predicates define search algorithms that take a tree T
% and visit every node each time writing out the line D:L where
% * D is the depth (starting at 0 for the root)
% * L is the label

% dfs(T) visits every node in depth-first order
dfs(T) :- ???
% bfs(T) visits every node in beadth-first order
```

```
bfs(T) :- ???
%itd(T):- visits every node in iterative deepening order
itd(T) :- ???
```

Here "must" means you can define any number of additional predicates. But the predicates specified above must exist and must have that arity and must work correctly on any input `T` that satisfies `istree(T)`. "working correctly" means the predicates must write out exactly what is specified, e.g.,

```
0:A
1:B
```

for the depth-first search of the tree `tree("A",[tree("B",[])])`.

Solution:

```
% initialize with depth 0
dfs(T) :- dfsD(T,0).

% write out depth and value V of the current node, then search all children with depth D+1
dfsD(tree(V,TS), D) :- write(D), write(":"), writeln(V), Di is D+1, dfsAll(TS,Di).

% calls dfsD on all trees in a list
dfsAll([],_).
dfsAll([T|TS],D) :- dfsD(T,D), dfsAll(TS,D).

% initialize with the fringe containing T at depth 0
bfs(T) :- bfsFringe([(0,T)]).

% empty fringe - done
bfsFringe([]).
% take the first pair (D,T) in the fringe, write out D and the value V of T
% append children TS of T paired with depth D+1 to the *end* of F, and recurse
bfsFringe([(D,tree(V,TS))|F]) :- write(D), write(":"), writeln(V),
   Di is D+1, pair(Di,TS, DTS), append(F,DTS,F2), bfsFringe(F2).

% pair(D,L,DL) takes value D and list L and pairs every element in L with D, returning DL
pair(_,[],[]).
pair(D,[H|T],[(D,H)|DT]) :- pair(D,T,DT).

% initialize with cutoff 0
itd(T) :- itdUntilDone(T,0),!.

% calls dfsUpTp with cutoff C and initial depth
itdUntilDone(T,C) :- dfsUpTo(T,0,C,Done), increaseCutOffIfNotDone(T,C,Done).
% depending on the value of Done, terminate or increase the cutoff.
increaseCutOffIfNotDone(_,_,Done) :- Done=1.
increaseCutOffIfNotDone(T,C,Done) :- Done=0, Ci is C+1, itdUntilDone(T,Ci).

% dfsUpTo(T,D,U,Done) is like dfs(T,D) except that
% * we stop at cutoff depth U
% * we return Done (0 or 1) if there were no more nodes to explore

% cutoff depth reach, more nodes left
dfsUpTo(_, D, U, Done) :- D > U, Done is 0.
% write data, recurse into all children with depth D+1
dfsUpTo(tree(V,TS), D, U, Done) :- write(D), write(":"), writeln(V),
   Di is D+1, dfsUpToAll(TS,Di,U, Done).

% dfsUpToAll(TS,D,U,Done) calls dfsUpTo(T,D,U,_) on all elements of TS; it returns 1 if all
dfsUpToAll([],_,_,Done) :- Done is 1.
dfsUpToAll([T|TS],D,U,Done) :- dfsUpTo(T,D,U,DoneT),
   dfsUpToAll(TS,D,U,DoneTS), Done is DoneT*DoneTS.
```

Problem 3.3 (Formally Modeling a Search Problem) 30 pt

Consider the Towers of Hanoi for 7 disks initially stacked on peg A.

Is this problem deterministic? Is it fully observable?

Formally model it as a Search Problem in the sense of the mathematical definition from the slides. Explain how your mathematical definition models the problem.

Note that the formal model only defines the problem — we are not looking for solutions here.

Note that modeling the problem corresponds to defining it in a programming language, except that we use mathematics instead of a programming language. Then explaining the model corresponds to documenting your implementation.

Solution: We need to give $(S, \mathcal{A}, T, I, G)$.

Because the problem is deterministic, we know $|T(a, s)| \leq 1$. Because the problem is fully observable, we know $|I| = 1$.

Let $D = \{1, 2, 3, 4, 5, 6, 7\}$ (the set of disks) and $P = \{A, B, C\}$ (the set of pegs). We put:

- $S = D \to P$, i.e., a state s is a function from disks to pegs.
  Explanation: In state s, the value $s(d)$ is the peg that disk d is on. Because disks must always be ordered by size, we do not have to explicitly store the order in which the disks sit on the pegs.

- $\mathcal{A} = \{(A, B), (B, A), (A, C), (C, A), (B, C), (C, B)\}$, i.e., an action a is a pair of different pegs.
  Explanation: (p, q) represents the action of moving the top disk of peg p to peg q.

- For $s \in S$ and $p \in P$, we abbreviate as $top(s, p)$ the smallest $d \in D$ such that $s(d) = p$.
  Explanation: $top(s, p)$ is the top (smallest) disk on peg p in state s.

  Then $T : \mathcal{A} \times S \to \mathcal{P}(S)$ is defined as follows:
  - If $top(s, q) > top(s, p)$, we put $T((p, q), s) = \{s'\}$ where $s' : D \to P$ is given by
    - $s'(d) = q$ if $d = top(s, p)$
    - $s'(d) = s(d)$ for all other values of d
  - otherwise, $T((p, q)), s) = \emptyset$

  Explanation: In state s, if the top disk of q is bigger than the top disk of p, the action (p, q) is applicable, and the successor state s' of s after applying (p, q) is the same as s except that the top (smallest) disk on peg p is now on peg q.

- $I = \{i\}$ where the state i is given by $i(d) = A$ for all $d \in D$.
  Explanation: Initially, all disks are on peg A.

- $G = \{g\}$ where the state g is given by $g(d) = B$ for all $d \in D$.
  Explanation: There is only one goal state, described by all disks being on peg B.

Note that there are many different correct solutions to this problem. In particular, you can use different definitions for S (i.e., model the state space differently), in which case everything else in the model will be different, too. Often a good model for the state space can be recognized by how straightforward it is to define the rest of the model formally.

Even if you used a different state space, a good self-study exercise is to check that the above (a) is indeed a search problem and (b) correctly models the Towers of Hanoi. Continuing the above analogy to programming languages, (a) corresponds to compiling/type-checking your implementation and (b) to checking that your implementation is correct.

Artificial Intelligence 1 2022
Assignment4: Search
– Given Nov. 21, Due Nov. 27 –

Problem 4.1 (Heuristic Searches) 0 pt

Consider the graph of Romanian cities with edges labeled with costs $c(m, n)$ of going from m to n. $c(m, n)$ is always bigger than the straight-line distance from m to n. $c(m, n)$ is infinite if there is no edge.

Our search algorithm keeps:

- a list E of expanded nodes n together with the cost $g(n)$ of the cheapest path to n found so far,
- a fringe F containing the unexpanded neighbors of expanded nodes.

We want to find a cheap path from Lugoj to Bucharest. Initially, E is empty, and F contains only Lugoj. We terminate if E contains Bucharest.

Expansion of a node n in F moves it from F to E and adds to F every neighbor of n that is not already in E or F. We obtain $g(n)$ by minimizing $g(e) + c(e, n)$ over expanded nodes e.

As a heuristic $h(n)$, we use the straight-line distance from n to Bucharest as given by the table in the lecture.

1. Explain how the following algorithms choose which node to expand next:
   - greedy search with heuristic h
   - A* search with path cost g and heuristic h
2. Explain what h^* is here and why h is admissible.
3. For each search, give the order in which nodes are expanded.
   (You only have to give the nodes to get the full score. But to get partial credit in case you're wrong, you may want to include for each step all nodes in the fringe and their cost.)

Problem 4.2 (Heuristics) 20 pt

Consider heuristic search with heuristic function h.

1. Briefly explain what is the same and what is different between A^* search and greedy search regarding the decision which node to expand next.
2. Is the constant function $h(n) = 0$ an admissible heuristic for A^* search?

Problem 4.3 (Games for Adversarial Search) 30 pt

For each of the following games and properties, state whether the game has the property.

Properties:

A 2 players alternating moves

I discrete state space

C players have complete information about state

F finite number of move options per state

E deterministic successor states

T games guaranteed to terminate

U terminal state has zero-sum utility.

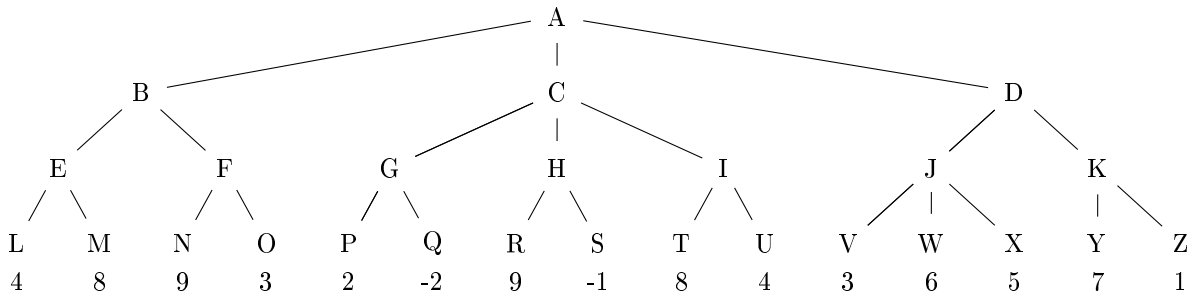
Games:

1. 2-player poker (until one player is bankrupted)
2. Backgammon
3. Wrestling (one 5 minute round)
4. Connect Four
5. Rock-Paper-Scissors (with a repeat to break ties)
6. Meta-Game (player 1's first move is to choose a game that satisfies all properties, subsequent moves play that game)

Submit a text file containing one line per game (in the order given above) such that each line contains the upper-case letters (without space) of all conditions that are violated.

Problem 4.4 (Game Tree) 20 pt

Consider the following game tree. Assume it is the maximizing player's turn to move. The values at the leaves are the static evaluation function values of the states at each of those nodes.



1. Compute the minimax game value of nodes A, B, C, and D

2. Which move would be selected by Max?

3. List the nodes that the alpha-beta algorithm would prune (i.e., not visit). Assume children of a node are visited left-to-right.

Submit your solution as a text file containing the following:

1. Line 1: 4 numbers, separated by space, corresponding to the nodes in alphabetical order, e.g., "1 3 2 5" means A=1, B=3, C=2, D=5.

2. Line 2: The upper-case letter for the selected move.

3. Line 3: The upper-case letters of the pruned moves.

Problem 4.5 (Minimax Search in ProLog) 30 pt

Consider the following game:

1. There is a pile of n matches in the middle.

2. Two players alternate taking away 1, 2, or 3 matches.

3. The winner is whoever takes the last match.

Solve this game (for all values of n) by implementing the minimax algorithm in Prolog. Specifically, implement exactly the following

- a Prolog predicate `value(S,P)` that holds if player `P` wins from initial state `S`,
- where the Prolog constructor `state(N,P)` represents the game state with `N` remaining matches and player `P` going next,
- where we represent players `P` using 1 for the starting player and -1 for the opponent.

Note: A partial solution will be explained in the tutorials, especially the use of `\+` for negation-as-failure and `!` for cut.

Artificial Intelligence 1 2022
Assignment4: Search
– Given Nov. 21, Due Nov. 27 –

Problem 4.1 (Heuristic Searches) 0 pt

Consider the graph of Romanian cities with edges labeled with costs $c(m, n)$ of going from m to n. $c(m, n)$ is always bigger than the straight-line distance from m to n. $c(m, n)$ is infinite if there is no edge.

Our search algorithm keeps:

- a list E of expanded nodes n together with the cost $g(n)$ of the cheapest path to n found so far,
- a fringe F containing the unexpanded neighbors of expanded nodes.

We want to find a cheap path from Lugoj to Bucharest. Initially, E is empty, and F contains only Lugoj. We terminate if E contains Bucharest.

Expansion of a node n in F moves it from F to E and adds to F every neighbor of n that is not already in E or F. We obtain $g(n)$ by minimizing $g(e) + c(e, n)$ over expanded nodes e.

As a heuristic $h(n)$, we use the straight-line distance from n to Bucharest as given by the table in the lecture.

1. Explain how the following algorithms choose which node to expand next:
   - greedy search with heuristic h
   - A* search with path cost g and heuristic h
2. Explain what h^* is here and why h is admissible.
3. For each search, give the order in which nodes are expanded.
   (You only have to give the nodes to get the full score. But to get partial credit in case you're wrong, you may want to include for each step all nodes in the fringe and their cost.)

Solution:

1. Both searches expand the node n that minimizes a function. The functions are
   (a) greedy search: $h(n)$
   (b) A*: $g(n) + h(n)$
2. $h^*(n)$ is the cost of the shortest path from n to Bucharest (which we do not know unless we expand all nodes). Because $c(m, n)$ is always bigger than the straight-line distance, every path is longer than the straight-line distance between its end points. Thus $h(n) \leq h^*(n)$.
3. Greedy search: Lugoj (244), Mehadia (241), Drobeta (242), Craiova (160), Pitesti (100), Bucharest (0)
   A* search: Lugoj (0+244), Mehadia (70+241), Drobeta ((70+75)+242), Craiova (70+75+120)+160), Timisoara (111+329), Pitesti ((70+75+120+138)+100), Bucharest ((70+75+120+138+101)+0)

Problem 4.2 (Heuristics) 20 pt

Consider heuristic search with heuristic function h.

1. Briefly explain what is the same and what is different between A^* search and greedy search regarding the decision which node to expand next.
2. Is the constant function $h(n) = 0$ an admissible heuristic for A^* search?

Solution:

1. Both choose the node that minimizes a certain function. As that function, A^* uses the sum of path cost and heuristic whereas greedy only uses the heuristic.
2. Yes. (But it's a useless one.)

Problem 4.3 (Games for Adversarial Search) 30 pt

For each of the following games and properties, state whether the game has the property.

Properties:

A 2 players alternating moves

I discrete state space

C players have complete information about state

F finite number of move options per state

E deterministic successor states

T games guaranteed to terminate

U terminal state has zero-sum utility.

Games:

1. 2-player poker (until one player is bankrupted)
2. Backgammon
3. Wrestling (one 5 minute round)
4. Connect Four
5. Rock-Paper-Scissors (with a repeat to break ties)
6. Meta-Game (player 1's first move is to choose a game that satisfies all properties, subsequent moves play that game)

Submit a text file containing one line per game (in the order given above) such that each line contains the upper-case letters (without space) of all conditions that are violated.

Solution: The games violate the following restrictions:

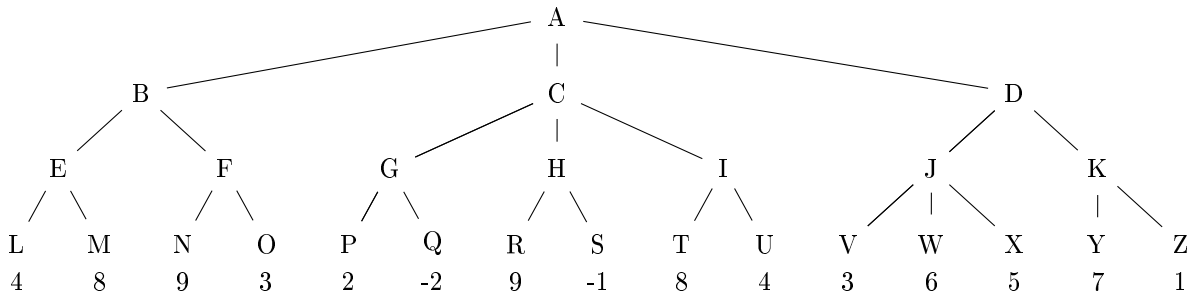
- 2-player poker: C, E, T
- Backgammon: E, T
- Wrestling: A, I, F
- Connect Four: none, solvable
- Rock-Paper-Scissors: A, T
- Meta-game: F

Games are solvable if they satisfy all properties.

Note: Because we have never made (and could not easily make) a formal definition of *game* that includes all possible games, it can be difficult to define the criteria in a way that allows checking them for any game. And whenever we do not have formal definitions, some answers may be unclear.

Problem 4.4 (Game Tree) 20 pt

Consider the following game tree. Assume it is the maximizing player's turn to move. The values at the leaves are the static evaluation function values of the states at each of those nodes.



1. Compute the minimax game value of nodes A, B, C, and D
2. Which move would be selected by Max?
3. List the nodes that the alpha-beta algorithm would prune (i.e., not visit). Assume children of a node are visited left-to-right.

Submit your solution as a text file containing the following:

1. Line 1: 4 numbers, separated by space, corresponding to the nodes in alphabetical order, e.g., "1 3 2 5" means A=1, B=3, C=2, D=5.
2. Line 2: The upper-case letter for the selected move.
3. Line 3: The upper-case letters of the pruned moves.

Solution:

1. B = 8, C = 2, D = 6, A = 8
2. B
3. O, H (and R and S), I (and T and U), K (and Y and Z)

Problem 4.5 (Minimax Search in ProLog) 30 pt

Consider the following game:

1. There is a pile of n matches in the middle.
2. Two players alternate taking away 1, 2, or 3 matches.
3. The winner is whoever takes the last match.

Solve this game (for all values of n) by implementing the minimax algorithm in Prolog. Specifically, implement exactly the following

- a Prolog predicate `value(S,P)` that holds if player `P` wins from initial state `S`,

- where the Prolog constructor `state(N,P)` represents the game state with `N` remaining matches and player `P` going next,
- where we represent players `P` using 1 for the starting player and -1 for the opponent.

Note: A partial solution will be explained in the tutorials, especially the use of `\+` for negation-as-failure and `!` for cut.

Solution:

```
% Game state: number N of remaining matches and current player P=1 or P=-1

% possible moves in state(N,P) yielding successor state T
successor(state(N,P),T) :- N>0, N2 is N-1, P2 is -P, T=state(N2,P2).
successor(state(N,P),T) :- N>1, N2 is N-2, P2 is -P, T=state(N2,P2).
successor(state(N,P),T) :- N>2, N2 is N-3, P2 is -P, T=state(N2,P2).

% membership in a list
contains([H|T],A) :- not(H=A), contains(T,A).
contains([A|_],A).

% find list Ts of succesor states of S using accumulator Acc
successors(S, Acc, Ts) :- successor(S,T), \+ contains(Acc,T), !, successors(S, [T|Acc], Ts).
successors(_, Acc, Acc).

% shown until here in the tutorials

% minvalue(Ss,Sofar,V) holds if V is the minimum value of list of states Ss
% Sofar is accumulator for minimum value seen so far

% end of list - return accumulator
minvalue([],Sofar,Sofar).
% next state has smaller value, replace accumulator and continue with rest
minvalue([S|Ss],Sofar,V) :- value(S,V1), V1<Sofar, minvalue(Ss,V1,V).
% next state has non-smaller value, keep accumulator and continue with rest
minvalue([S|Ss],Sofar,V) :- value(S,V1), V1>=Sofar, minvalue(Ss,Sofar,V).

% like minvalue
maxvalue([],Sofar,Sofar).
maxvalue([S|Ss],Sofar,V) :- value(S,V1), V1>=Sofar, maxvalue(Ss,V1,V).
maxvalue([S|Ss],Sofar,V) :- value(S,V1), V1<Sofar, maxvalue(Ss,Sofar,V).

% value(S,V) holds if state S has winner V (1 or -1)

% our turn (P=1): choose successor with maximal value
% we lose if no possible move (Ts=[], accumulator initialized to -1)
value(S,V) :- state(_,P)=S, P = 1, successors(S,[],Ts), maxvalue(Ts,-1,V).

% opponent's turn (P=-1): choose successor with minimal value
% we win if no possible move (Ts=[], accumulator initialized to 1)
value(S,V) :- state(_,P)=S, P = -1, successors(S,[],Ts), minvalue(Ts,1,V).
```

Artificial Intelligence 1
Assignment5: Constraint Satisfaction
– Given Nov. 28, Due Dec. 4 –

Problem 5.1 (3 Rooks on a Small Board) 40 pt

Consider the following problem: We want to place 3 rooks (german: Turm) on a 4×7 chess-board such that no two rooks threaten each other. (Rooks can move horizontally and vertically as far as they like.)

Model the problem above as a *constraint satisfaction problem* $\langle V, D, C \rangle$.
Explain your model briefly by saying how rook placements correspond to the *variable assignments* for the problem.

Problem 5.2 (CSP as a Search Problem) 30 pt

We consider a binary CSP P with

- a set V of variables
- a family D of domains D_v for $v \in V$
- a family C of constraints $C_{uv} \subseteq D_u \times D_v$ for $u, v \in V$, $u \neq v$ where C_{uv} is the dual of C_{vu}

Note: We assume here that a constraint C_{uv} is given for all pairs of unequal variables — if we want to omit a constraint, we can simply assume $C_{uv} = D_u \times D_v$ or $C_{uv} = \texttt{true}$, i.e., all pairs are allowed and thus there is no constraint. That could be problematic in implementations, but is practical on paper.

Define the search problem (S, A, T, I, G) corresponding to P.

Note: This problem formalizes the informal statement that CSPs are a special case of search problems.

Problem 5.3 (Basic Definitions) 30 pt

Consider the following binary CSP:

- $V = \{a, b, c, d\}$
- $D_a = \texttt{bool}$, $D_b = D_c = \{0, 1, 2, 3\}$, $D_d = \{0, 1, 2, 3, 4, 5, 6\}$
- Constraints:
  - if a, then $b \leq 2$
  - if $c < 2$, then a
  - $b + c < 4$
  - $b > d$

- $d = 2c$

1. Give all solutions.
2. Give an inconsistent total assignment.
3. Give all consistent partial assignments α such that $\mathrm{dom}(\alpha) \subseteq \{a, b\}$.

Artificial Intelligence 1
Assignment5: Constraint Satisfaction
– Given Nov. 28, Due Dec. 4 –

Problem 5.1 (3 Rooks on a Small Board) 40 pt

Consider the following problem: We want to place 3 rooks (german: Turm) on a 4×7 chess-board such that no two rooks threaten each other. (Rooks can move horizontally and vertically as far as they like.)

Model the problem above as a *constraint satisfaction problem* $\langle V, D, C \rangle$. Explain your model briefly by saying how rook placements correspond to the *variable assignments* for the problem.

Make sure you give a formally exact definition, i.e., explicitly define the sets V and all sets D_v. You can describe each *constraint* as a set of tuples or as a formula.

Solution: $V = a_1, a_2, b_1, b_2, c_1, c_2$
$D_{a_1} = D_{b_1} = D_{c_1} = \{1, 2, 3, 4\}$
$D_{a_2} = D_{b_2} = D_{c_2} = \{1, 2, 3, 4, 5, 6, 7\}$
Constraints in C:

- $v_1 \neq w_1$ for all $(v, w) \in \{(a, b), (a, c), (b, c)\}$
- $v_2 \neq w_2$ for all $(v, w) \in \{(a, b), (a, c), (b, c)\}$

The assignments to (a_1, a_2), (b_1, b_2), and (c_1, c_2) correspond to the coordinates of the squares where the rooks are placed.

Problem 5.2 (CSP as a Search Problem) 30 pt

We consider a binary CSP P with

- a set V of variables
- a family D of domains D_v for $v \in V$
- a family C of constraints $C_{uv} \subseteq D_u \times D_v$ for $u, v \in V$, $u \neq v$ where C_{uv} is the dual of C_{vu}

Note: We assume here that a constraint C_{uv} is given for all pairs of unequal variables — if we want to omit a constraint, we can simply assume $C_{uv} = D_u \times D_v$ or $C_{uv} = \texttt{true}$, i.e., all pairs are allowed and thus there is no constraint. That could be problematic in implementations, but is practical on paper.

Define the search problem (S, A, T, I, G) corresponding to P.

Note: This problem formalizes the informal statement that CSPs are a special case of search problems.

Solution: The search problem is defined as follows:

- states are the assignments, i.e., S is the set of the partial mappings with domain V that map each $v \in V$ to an element of D_v
  More formally, we can write this as

$$S = \{a : V \rightharpoonup \bigcup_{v \in V} D_v \mid \forall v \in \mathrm{dom}(a).a(v) \in D_v\}$$

  Typically, V is finite. In that case, we can use the simpler definition

$$S = \prod_{v \in V} (D_v \cup \{\bot\})$$

  where $a(v) = \bot$ represents that a is partial at v.

- An action assigns to a variable a concrete value of its domain. So

$$A = \{(v, x) \mid v \in V, x \in D_v\}$$

- The transition model simply updates the assignment: $T((v, x), a) = \{a'\}$ where $a'(v) = x$ and $a'(w) = a(w)$ if $w \neq v$.
  Note: alternatively, we could put $T((v, x), a) = \emptyset$ if a already assigns a value to v.

- In the initial state, no variable is assigned: $I = \{i\}$ where i is undefined everywhere.

- The terminal states are the solutions, i.e., T is the set of all $a \in S$ such that
  - a is total
  - for all $u, v \in V$ with $u \neq v$, we have $(a(u), a(v)) \in C_{uv}$

Problem 5.3 (Basic Definitions) 30 pt

Consider the following binary CSP:

- $V = \{a, b, c, d\}$
- $D_a = \texttt{bool}$, $D_b = D_c = \{0, 1, 2, 3\}$, $D_d = \{0, 1, 2, 3, 4, 5, 6\}$
- Constraints:
  - if a, then $b \leq 2$
  - if $c < 2$, then a
  - $b + c < 4$
  - $b > d$
  - $d = 2c$

1. Give all solutions.
2. Give an inconsistent total assignment.
3. Give all consistent partial assignments α such that $\mathrm{dom}(\alpha) \subseteq \{a, b\}$.

Solution:

- There are 2 solutions: a true, $b \in \{1, 2\}$, $c = 0$, $d = 0$
- Any total assignment that is not a solution (see previous question).
- We classify the possibly assignments a by their domain:
  - $\mathrm{dom}(\alpha) = \emptyset$: 1 assignment, namely α undefined everywhere
  - $\mathrm{dom}(\alpha) = \{a\}$: 2 assignments, namely $\alpha(a) \in D_a$, undefined elsewhere
  - $\mathrm{dom}(\alpha) = \{b\}$: 4 assignments, namely $\alpha(b) \in D_b$, undefined elsewhere
  - $\mathrm{dom}(\alpha) = \{a, b\}$: 7 assignments, namely
    - 4 assignments with $\alpha(a) = false$, $\alpha(b) \in D_b$, undefined elsewhere
    - 3 assignments with $\alpha(b) = true$, $\alpha(b) \in \{0, 1, 2\}$, undefined elsewhere

---

Artificial Intelligence 1
Assignment6: Constraint Propagation
– Given Dec. 1, Due Dec. 11 –

Problem 6.1 (Scheduling CS Classes as a CSP) 40 pt

You are in charge of scheduling for computer science classes. There are 5 classes and 3 professors to teach them. You are constrained by the fact that each professor can only teach one class at a time. The classes are:

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

The professors are:

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

1. Formulate this problem as a binary CSP problem in which there is one variable per class, stating the domains, and constraints. Constraints should be specified formally and precisely.
2. Give the constraint graph associated with your CSP.
3. Give examples of
   - a total inconsistent assignment
   - a solution

Problem 6.2 (Scheduling CS Classes with Constraint Propagation) 30 pt

Consider the CSP problem for scheduling CS classes from the previous assignment.

1. Show the CSP obtained by running arc-consistency. As usual, you can visualize this as a graph whose
   - nodes are labeled with the variable names and domains
   - edges are labeled with the constraints.
2. Give all optimal cutsets for the CSP.

Problem 6.3 (CSP Formalization) 30 pt

Consider the following binary CSP $\Pi := (V, D, C)$:

- Variables $V = \{x, y, z\}$
- Domains D: $D_x = \{0, 1, 2\}$, $D_y = \{1, 2\}$, $D_z = \{0, 1\}$
- Constraints C: $x \neq y$, $y > z$

1. Give all pairs (v, w) of variables such that v is arc-consistent relative to w.
2. Give all solutions that would remain if we added the constraint $x \neq z$ to Π.
3. Assume we assign $y = 1$ in Π and apply forward-checking. Give the resulting domains D_x, D_y, D_z.

Problem 6.4 (Kalah Tournament) 200 pt

This is an extraordinary problem, in which we implement adversarial search as a tournament. You can implement all search methods, e.g., to simulate a move or compute the full game tree etc.

Submission parameters:

- Team size: 3 people per team
- Deadline: 2023-01-08
- Site: The submission site will be opened later.
- Format: The details will be published later on the studon forum. But you can use any programming language, and your program might be subject to resource constraints (overall space for the binary, time per move, etc.).
- Points: Submissions that are better than a relatively low baseline (e.g., win against a player that makes random moves) will receive 100 points. The team with the best agent receives an additional 100 points, the 2nd team 90 points, the 3rd 80 etc.

Further details will be announced in the forum as they come up.

Artificial Intelligence 1
Assignment6: Constraint Propagation
– Given Dec. 1, Due Dec. 11 –

Problem 6.1 (Scheduling CS Classes as a CSP) 40 pt

You are in charge of scheduling for computer science classes. There are 5 classes and 3 professors to teach them. You are constrained by the fact that each professor can only teach one class at a time. The classes are:

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

The professors are:

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

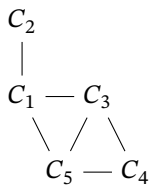
1. Formulate this problem as a binary CSP problem in which there is one variable per class, stating the domains, and constraints. Constraints should be specified formally and precisely.
2. Give the constraint graph associated with your CSP.
3. Give examples of
   - a total inconsistent assignment
   - a solution

Solution:

1.

Variables	Domains
C_1	A
C_2	A,C
C_3	A,B,C
C_4	A,B,C
C_5	A,C

Constraints: $C_1 \neq C_2, C_1 \neq C_3, C_1 \neq C_5, C_3 \neq C_4, C_3 \neq C_5, C_4 \neq C_5$

2. 

3. Various options.

Problem 6.2 (Scheduling CS Classes with Constraint Propagation) 30 pt

Consider the CSP problem for scheduling CS classes from the previous assignment.

1. Show the CSP obtained by running arc-consistency. As usual, you can visualize this as a graph whose
   - nodes are labeled with the variable names and domains
   - edges are labeled with the constraints.
2. Give all optimal cutsets for the CSP.

Solution:

1.

Variable	Domain
C_1	A
C_2	C
C_3	B
C_4	A
C_5	C

2. The two optimal cutsets are $\{C_3\}$ and $\{C_5\}$.

Problem 6.3 (CSP Formalization) 30 pt

Consider the following binary CSP $\Pi := (V, D, C)$:

- Variables $V = \{x, y, z\}$
- Domains D: $D_x = \{0, 1, 2\}$, $D_y = \{1, 2\}$, $D_z = \{0, 1\}$
- Constraints C: $x \neq y$, $y > z$

1. Give all pairs (v, w) of variables such that v is arc-consistent relative to w.
2. Give all solutions that would remain if we added the constraint $x \neq z$ to Π.
3. Assume we assign $y = 1$ in Π and apply forward-checking. Give the resulting domains D_x, D_y, D_z.

Solution:

- (x, y), (x, z), (y, x), (y, z), (z, x), (z, y)
- Solutions (x, y, z) are $(0, 2, 1)$, $(1, 2, 0)$, $(2, 1, 0)$
- $D_x = \{0, 2\}$, $D_y = \{1\}$, $D_z = \{0\}$

Artificial Intelligence 1
Assignment7: Propositional Logic
– Given Dec 8, Due Dec 18 –

Problem 7.1 (PL Concepts) 30 pt

Which of the following statements are true? In each case, give an informal argument why it is true or a counter-example.

1. Every satisfiable formula is valid.
2. Every valid formula is satisfiable.
3. If A is satisfiable, then $\neg A$ is unsatisfiable.
4. If $A \models B$, then $A \wedge C \models B \wedge C$.
5. Every admissible inference rule is derivable.
6. If $\vdash$ is sound for $\models$ and $\{A, B\} \vdash C$, then C is satisfiable if A and B are.

Problem 7.2 (Equivalence of CSP and SAT) 30 pt

We consider

- CSPs (V, D, C) with finite domains as before
- SAT problems (V, A) where V is a set of propositional variables and A is a propositional formula over V.

We will show that these problem classes are equivalent by reducing their instances to each other.

1. Given a SAT instance $P = (V, A)$, define a CSP instance $P' = (V', D', C')$ and two bijections
   - f mapping satisfying assignments of P to solutions of P'
   - f' the inverse of f

   We already know that binary CSPs are equivalent to higher-order CSPs. Therefore, it is sufficient to give a higher-order CSP.
2. Given a CSP instance (V, D, C), define a SAT instance (V', A') and bijections as above

Problem 7.3 (Calculi Comparison) 60 pt

Prove (or disprove) the validity of the following formulae in i) Natural Deduction ii) Tableau and iii) Resolution.

1. $(P \wedge Q) \Rightarrow (P \vee Q)$ (to be done in the tutorial, not part of grading)
2. $((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C$
3. $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$

Artificial Intelligence 1
Assignment7: Propositional Logic
– Given Dec 8, Due Dec 18 –

Problem 7.1 (PL Concepts) 30 pt

Which of the following statements are true? In each case, give an informal argument why it is true or a counter-example.

1. Every satisfiable formula is valid.
2. Every valid formula is satisfiable.
3. If A is satisfiable, then $\neg A$ is unsatisfiable.
4. If $A \models B$, then $A \wedge C \models B \wedge C$.
5. Every admissible inference rule is derivable.
6. If $\vdash$ is sound for $\models$ and $\{A, B\} \vdash C$, then C is satisfiable if A and B are.

Solution:

1. Not true. Counter-example: p is satisfiable (put $\varphi(p) = \mathtt{T}$) but not valid (falsified by $\varphi(p) = \mathtt{F}$).

2. True. Assume F is valid. Then F is satisfied by all assignments. We know (This is a subtle step that can easily be overlooked.) that there is at least one assignment a. (Even if there are no propositional variables, we would still have the empty assignment.) So a must satisfy F and therefore F is satisfiable.

3. Not true. Counter-example: p is satisfiable (put $\varphi(p) = \mathtt{T}$), but $\neg p$ is also satisfiable (put $\varphi(p) = \mathtt{F}$).

4. True. Assume $A \models B$ (H) and an assignment φ such that $I_\varphi(A \wedge C) = \mathtt{T}$ (A).
   We need to show that also $I_\varphi(A \wedge C) = \mathtt{T}$ (G).
   By definition, (A) yields $I_\varphi(A) = \mathtt{T}$ (A1) and $I_\varphi(C) = \mathtt{T}$ (A2).
   By definition of (H), we obtain from (A1) that $I_\varphi(B) = \mathtt{T}$ (B).
   Then we obtain (G) from its definition and (B) and (A2).

5. Not true. Counter-example: The empty derivation relation has no inference rules and thus no derivable formulas. Then any rule with non-empty set of assumptions is admissible. But no rule is derivable.

6. Not true. The assumptions do show that $A, B \models C$. So if we have an assignment that satisfies both A and B, then that assignment also satisfies C and thus C is satisfiable. But we only know that A and B are satisfiable by some assignments, not necessarily the same one. A counter-example, is $A = p$, $B = \neg p$, C any unsatisfiable formula. Then $A, B \models C$ holds (because there are no assignments that satisfy both A and B), and A and B but not C are satisfiable.

Problem 7.2 (Equivalence of CSP and SAT) 30 pt

We consider

- CSPs (V, D, C) with finite domains as before
- SAT problems (V, A) where V is a set of propositional variables and A is a propositional formula over V.

We will show that these problem classes are equivalent by reducing their instances to each other.

1. Given a SAT instance $P = (V, A)$, define a CSP instance $P' = (V', D', C')$ and two bijections
   - f mapping satisfying assignments of P to solutions of P'
   - f' the inverse of f

We already know that binary CSPs are equivalent to higher-order CSPs. Therefore, it is sufficient to give a higher-order CSP.

2. Given a CSP instance (V, D, C), define a SAT instance (V', A') and bijections as above

---

Solution:

1. We define P' by $V' = V$, $D_v = \{\mathtt{T}, \mathtt{F}\}$ for every $v \in V$, and $C = \{A\}$, i.e., C contains the single higher-order constraint that holds if an assignment to V' (seen as an propositional assignment to V) satisfies A.
   f and f' are the identity.

2. We define P' as follows. V' contains variables p'_{va} for every $v \in V$ and $a \in D_v$. The intuition behind p'_{va} is that v has value a.
   A' is the conjunction of the following formulas:

   - for all $v \in V$ with $D_v = \{a_1, \dots, a_n\}$, the formula $p'_{va_1} \vee \dots \vee p'_{va_n}$ (i.e., v must have at least one value)
   - for all $v \in V$, and $a, b \in D_v$ with $a \neq b$, the formula $p'_{va} \Rightarrow \neg p'_{vb}$ (i.e., v can have at most one value)
   - for all C_{vw} and $(a, b) \notin C_{vw}$, the formula $\neg(p'_{va} \wedge p'_{wb})$ (i.e., every constraint must be satisfied)

   The bijection f maps a solution α of P to a A'-satisfying propositional assignment φ for V' as follows: for all v, a, we put $\varphi(p'_{va}) = \mathtt{T}$ if $\alpha(v) = a$ and $\varphi(p'_{va}) = \mathtt{F}$ otherwise.
   The inverse bijection f' maps an A'-satisfying assignment φ to a solution α of P as follows: for all v we put $\alpha(v) = a$ where a is the unique value for which $\varphi(p'_{va}) = \mathtt{T}$.

Problem 7.3 (Calculi Comparison) 60 pt

Prove (or disprove) the validity of the following formulae in i) Natural Deduction ii) Tableau and iii) Resolution.

1. $(P \wedge Q) \Rightarrow (P \vee Q)$ (to be done in the tutorial, not part of grading)
2. $((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C$
3. $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$

Solution: ND

1.

(1)	1	$(P \wedge Q)$	Assumption
(2)	1	P	$\wedge E_\ell$ (on 1)
(3)	1	$(P \vee Q)$	$\vee I_\ell$ (on 2)
(4)		$(P \wedge Q) \Rightarrow (P \vee Q)$	$\Rightarrow I$ (on 1 and 3)

2.

(1)	1	$(A \vee B) \wedge ((A \Rightarrow C) \wedge (B \Rightarrow C))$	Assumption
(2)	1	$(A \vee B)$	$\wedge E_\ell$ (on 1)
(3)	1	$(A \Rightarrow C) \wedge (B \Rightarrow C)$	$\wedge E_r$ (on 1)
(4)	1	$(A \Rightarrow C)$	$\wedge E_\ell$ (on 3)
(5)	1	$(B \Rightarrow C)$	$\wedge E_r$ (on 3)
(6)	1,6	A	Assumption
(7)	1,6	C	$\Rightarrow E$ (on 4 and 6)
(8)	1,8	B	Assumption
(9)	1,8	C	$\Rightarrow E$ (on 5 and 8)
(10)	1	C	$\vee E$ (on 2, 7 and 9)
(11)		$((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C$	$\Rightarrow I$ (on 1 and 10)

3.

(1)		$(P \vee \neg P)$	TND
(2)	2	P	Assumption
(3)	2,3	$(P \Rightarrow Q) \Rightarrow P)$	Assumption
(4)	2	$((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$	$\Rightarrow I$ (on 3 and 2)
(5)	5	$\neg P$	Assumption
(6)	5,6	$(P \Rightarrow Q) \Rightarrow P)$	Assumption
(7)	5,6,7	P	Assumption
(8)	5,6,7	F	FI (on 5 and 7)
(9)	5,6,7	Q	FE (on 8)
(10)	5,6	$P \Rightarrow Q$	$\Rightarrow I$ (on 7 and 9)
(11)	5,6	P	$\Rightarrow E$ (on 6 and 10)
(12)	5	$((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$	$\Rightarrow$ I (on 6 and 11)
(13)		$((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$	$\vee E$ (on 1, 4 and 12)

Solution: Tableau

1.

	(1)	$(P \wedge Q) \Rightarrow (P \vee Q)^F$	
	(2)	$(P \wedge Q)^T$	(from 1)
	(3)	$(P \vee Q)^F$	(from 1)
	(4)	P^T	(from 2)
	(5)	Q^T	(from 2)
	(6)	P^F	(from 3)
		close on P	

2.

(1)	$((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C^F$					
(2)	$(A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)^T$					(from 1)
(3)	C^F					(from 1)
(4)	$(A \vee B)^T$					(from 2)
(5)	$(A \Rightarrow C)^T$					(from 2)
(6)	$(B \Rightarrow C)^T$					(from 2)
(7)	A^T			B^T		(split on 6)
(8)	A^F close on A	C^T close on C	(split on 5)	B^F close on B	C^T close on C	(split on 4)

3.

	(1)	$((P \Rightarrow Q) \Rightarrow P) \Rightarrow P^F$		
	(2)	$(P \Rightarrow Q) \Rightarrow P)^T$		(from 1)
	(3)	P^F		(from 1)
	(4)	$P \Rightarrow Q^F$	P^T	(split on 2)
	(5)	P^T (from 4)	close on P	
	(6)	Q^F (from 4)		
		close on P		

Solution:

Resolution

1. $(P \wedge Q) \Rightarrow (P \vee Q)$: We negate and build a CNF:

$$\begin{aligned}&(P \wedge Q) \wedge \neg(P \vee Q)\\ \equiv &P \wedge Q \wedge \neg P \wedge \neg Q\end{aligned}$$

   yielding clauses $\{P^T\}, \{Q^T\}, \{P^F\}, \{Q^F\}$
   Resolving the two red clauses yields $\{\}$.

2. $((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \Rightarrow C$: We negate and build a CNF:

$$\begin{aligned}&((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C)) \wedge \neg C\\ \equiv &(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee C) \wedge \neg C\end{aligned}$$

   yielding clauses $\{A^T, B^T\}, \{A^F, C^T\}, \{B^F, C^T\}, \{C^F\}$.
   Resolving yields:

$$\begin{aligned}\{A^F, C^T\} + \{C^F\} &\Longrightarrow \{A^F\}\\ \{B^F, C^T\} + \{C^F\} &\Longrightarrow \{B^F\}\\ \{A^T, B^T\} + \{A^F\} &\Longrightarrow \{B^T\}\\ \{B^T\} + \{B^F\} &\Longrightarrow \{\}\end{aligned}$$

3. $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$: We negate and build a CNF:

$$\begin{aligned}&((P \Rightarrow Q) \Rightarrow P) \wedge \neg P\\ \equiv &(\neg(P \Rightarrow Q) \vee P) \wedge \neg P\\ \equiv &((P \wedge \neg Q) \vee P) \wedge \neg P\\ \equiv &((P \vee P) \wedge (\neg Q \vee P) \wedge \neg P\end{aligned}$$

   yielding clauses $\{P^T\}, \{Q^F, P^T\}, \{P^F\}$.
   Resolving the two red clauses yields $\{\}$.

Artificial Intelligence 1
Assignment8: First-Order Logic
– Given Dec 20, Due Jan 08 –

Problem 8.1 (Propositional Logic in Prolog) 50 pt

We implement propositional logic in Prolog.

We use the following Prolog terms to represent Prolog formulas

- lists of strings for signatures (each element being the name of a propositional variables)
- `var(s)` for a propositional variable named `s`, which is a string,
- `neg(F)` for negation,
- `disj(F,G)` for disjunction,
- `conj(F,G)` for conjunction,
- `impl(F,G)` for implication.

1. Implement a Prolog predicate `isForm(S,F)` that checks if `F` is well-formed formula relative to signature `isForm(S)`.

   Examples:

   ```
   ?- isForm(["a","b"],neg(var("a"))).
   True

   ?- isForm(["a","b"],neg(var("c"))).
   False

   ?- isForm(["a","b"],conj(var("a"),impl(var("b")))).
   False
   ```

2. Implement a Prolog predicate `simplify(F,G)` that replaces all disjunctions and implications with conjunction and negation.

   Examples:

   ```
   ?- simplify(disj(var("a"),var("b")), X).
   X = not(and(not(var("a")),not(var("b")))).
   ```

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

3. Implement a predicate `eval(P,F,V)` that evaluates a formula under assignment `P`. Here `P` is a list of terms `assign(s,v)` where `s` is the name of a propositional variable and `v` is a truth value (either 1 or 0). You can assume that `P` provides exactly one assignment for every propositional variable in `F`.

   Example:

```
?- eval([assign("a",1),assign("b",0)], conj(var("a"), var("b")), V).
V = 0.

?- eval([assign("a",1),assign("b",1)], conj(var("a"), var("b")), V).
V = 1.
```

Problem 8.2 (PL Semantics) 30 pt

We work with a propositional logic signature declaring variables A and B and consider the following two formulas:

1. $A \Rightarrow (B \Rightarrow A)$
2. $(A \wedge B) \Rightarrow (A \wedge C)$

We use a fixed but arbitrary assignment φ for the propositional variables.

For each of the two formulas F, apply the definition of the interpretation $\mathcal{I}_\varphi(F)$ step-by-step to obtain the semantic condition that F holds under φ. Afterwards determine if F is valid or not by one of the following:

- argue why $\mathcal{I}_\varphi(F)$ is true, which means F is valid because it holds for an arbitrary φ,
- give an assignment φ that makes $\mathcal{I}_\varphi(F)$ false

Problem 8.3 (FOL-Signatures) 20 pt

1. Model the following situation as a FOL signature. (FOL and PLNQ signatures are the same.)
   - We have constants (= nullary functions) called `zero` and `one`.
   - We have a binary function called `plus`.
   - We have a unary function called `minus`.
   - We have a binary predicate called `less`.
2. Now consider the signature given by
   - $\Sigma^f_0 = \{a, b\}$
   - $\Sigma^f_1 = \{f, g\}$
   - $\Sigma^f_2 = \{h\}$
   - $\Sigma^p_0 = \{p\}$
   - $\Sigma^p_1 = \{q\}$
   - $\Sigma^p_2 = \{r\}$
   - all other sets empty

Give

- a term over this signature that uses all function symbols
- a formula over this signature that uses all function and predicate symbols

Artificial Intelligence 1
Assignment8: First-Order Logic
– Given Dec 20, Due Jan 08 –

Problem 8.1 (Propositional Logic in Prolog) 50 pt

We implement propositional logic in Prolog.

We use the following Prolog terms to represent Prolog formulas

- lists of strings for signatures (each element being the name of a propositional variables)
- `var(s)` for a propositional variable named `s`, which is a string,
- `neg(F)` for negation,
- `disj(F,G)` for disjunction,
- `conj(F,G)` for conjunction,
- `impl(F,G)` for implication.

1. Implement a Prolog predicate `isForm(S,F)` that checks if `F` is well-formed formula relative to signature `S`.

   Examples:

   ```
   ?- isForm(["a","b"],neg(var("a"))).
   True

   ?- isForm(["a","b"],neg(var("c"))).
   False

   ?- isForm(["a","b"],conj(var("a"),impl(var("b")))).
   False
   ```

2. Implement a Prolog predicate `simplify(F,G)` that replaces all disjunctions and implications with conjunction and negation.

   Examples:

   ```
   ?- simplify(disj(var("a"),var("b")), X).
   X = neg(conj(neg(var("a")),neg(var("b")))).
   ```

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

3. Implement a predicate `eval(P,F,V)` that evaluates a formula under assignment `P`. Here `P` is a list of terms `assign(s,v)` where `s` is the name of a propositional variable and `v` is a truth value (either 1 or 0). You can assume that `P` provides exactly one assignment for every propositional variable in `F`.

   Example:

```
?- eval([assign("a",1),assign("b",0)], conj(var("a"), var("b")), V).
V = 0.

?- eval([assign("a",1),assign("b",1)], conj(var("a"), var("b")), V).
V = 1.
```

Solution:

```
contains([H|_],H).
contains([_|L],X) :- contains(L,X).

% isForm(S,F) holds if F is a PL-formula over signature S
% the signature is given as a list of names of propositional variables
isForm(S,var(N)) :- string(N), contains(S,N).
isForm(S,neg(F)) :- isForm(S,F).
isForm(S,conj(F,G)) :- isForm(S,F), isForm(S,G).
isForm(S,disj(F,G)) :- isForm(S,F), isForm(S,G).
isForm(S,impl(F,G)) :- isForm(S,F), isForm(S,G).

% simplify(F,G) holds if G is the result of replacing in F
% disjunction and implication with conjunction and negation
simplify(var(S),var(S)).
simplify(neg(F), neg(FS)) :- simplify(F,FS).
simplify(conj(F,G), conj(FS,GS)) :- simplify(F,FS), simplify(G,GS).
simplify(disj(F,G), neg(conj(neg(FS),neg(GS)))) :- simplify(F,FS), simplify(G,GS).
simplify(impl(F,G), neg(conj(FS,neg(GS)))) :- simplify(F,FS), simplify(G,GS).

% eval(P,F,V) holds if I_P(F) = V
% the assignment P is given as a list [assign(N,V), ...]
% where N is the name of a propositional variable and V is 0 or 1
eval(P,var(N), V) :- contains(P,assign(N,V)).
eval(P,neg(F), V) :- eval(P,F,FV), V is 1-FV.
eval(P,conj(F,G), V) :- eval(P,F,FV), eval(P,G,GV), V is FV*GV.
eval(P,disj(F,G), V) :- eval(P,F,FV), eval(P,G,GV), V is FV+GV-FV*GV.
eval(P,impl(F,G), V) :- eval(P,F,FV), eval(P,G,GV), V is (1-FV)+GV-(1-FV)*GV.
```

Problem 8.2 (PL Semantics) 30 pt

We work with a propositional logic signature declaring variables A and B and consider the following two formulas:

1. $A \Rightarrow (B \Rightarrow A)$
2. $(A \wedge B) \Rightarrow (A \wedge C)$

We use a fixed but arbitrary assignment φ for the propositional variables.

For each of the two formulas F, apply the definition of the interpretation $\mathcal{I}_\varphi(F)$ step-by-step to obtain the semantic condition that F holds under φ. Afterwards determine if F is valid or not by one of the following:

- argue why $\mathcal{I}_\varphi(F)$ is true, which means F is valid because it holds for an arbitrary φ,

- give an assignment φ that makes $\mathcal{I}_\varphi(F)$ false

---

Solution: We use $\top/\bot$ as the two truth values here. They are sometimes also written as $1/0$ or T/F.

- $A \Rightarrow (B \Rightarrow A)$ is valid:
  For any assignment φ:

$$\begin{aligned}\mathcal{I}_\varphi(A \Rightarrow (B \Rightarrow A)) &= \mathcal{I}_\varphi(\neg(A \wedge \neg\neg(B \wedge \neg A))\\ &= \top \text{ iff } \mathcal{I}_\varphi(A \wedge \neg\neg(B \wedge \neg A)) = \bot\\ &\text{iff not both } \varphi(A) = \top \text{ and } \mathcal{I}_\varphi(\neg\neg(B \wedge \neg A)) = \top\\ &\text{The latter is the case iff } \mathcal{I}_\varphi(B \wedge \neg A) = \top\\ &\text{iff } \varphi(B) = \top \text{ and } \varphi(A) = \bot\end{aligned}$$

  So the formula is false iff both $\mathcal{I}_\varphi(A) = \top$ and $\mathcal{I}_\varphi(A) = \bot$, which is impossible. So the formula is true for every assignment.

- $(A \wedge B) \Rightarrow (A \wedge C)$: Not valid. Counterexample: $\varphi(A) = \varphi(B) = \top, \varphi(C) = \bot$.

Problem 8.3 (FOL-Signatures) 20 pt

1. Model the following situation as a FOL signature. (FOL and PLNQ signatures are the same.)
   - We have constants (= nullary functions) called `zero` and `one`.
   - We have a binary function called `plus`.
   - We have a unary function called `minus`.
   - We have a binary predicate called `less`.
2. Now consider the signature given by
   - $\Sigma_0^f = \{a, b\}$
   - $\Sigma_1^f = \{f, g\}$
   - $\Sigma_2^f = \{h\}$
   - $\Sigma_0^p = \{p\}$
   - $\Sigma_1^p = \{q\}$
   - $\Sigma_2^p = \{r\}$
   - all other sets empty

   Give

- a term over this signature that uses all function symbols
- a formula over this signature that uses all function and predicate symbols

---

Solution:

1. $\Sigma_0^f = \{\texttt{zero}, \texttt{one}\}$, $\Sigma_1^f = \{\texttt{minus}\}$, $\Sigma_2^f = \{\texttt{plus}\}$, $\Sigma_2^p = \{\texttt{less}\}$, and all other sets are empty
2. E.g., $t = h(f(a), g(b))$ for the term $r(t, t) \wedge q(t) \wedge p$ for the formula

---

AI 1 2022/23
Assignment9: First-Order Logic
– Given Jan. 12, Due Jan. 22 –

Problem 9.1 (Induction) 20 pt

Use structural induction on terms and formulas to define a function C that maps every term/formula to the number of occurrences of free variables. For example, $C(\forall x.P(x,x,y,y,z)) = 3$ because the argument has 2 free occurrences of y and 1 of z.

Hint: Use an auxiliary function $C'(V,A)$ that takes the set V of bound variables and a term/formula A. Define C' by structural induction on A. Then define $C(A) = C'(\emptyset, A)$.

Problem 9.2 (First-Order Semantics) 30 pt

Let $= \in \Sigma_2^p$, $P \in \Sigma_1^p$ and $+ \in \Sigma_2^f$. We use the semantics of first-order logic without equality.

Prove or refute the following formulas semantically. That means you must show that $I_\varphi(A) = T$ for all models I and assignments φ (without using a proof calculus) or to give some I, φ such that $I_\varphi(A) = F$.

1. $P(X)$
2. $\forall X.\forall Y. = (+(X,Y), +(Y,X))$
3. $\exists X.(P(X) \Rightarrow \forall Y.P(Y))$
4. $P(Y) \Rightarrow \exists X.P(X)$

Problem 9.3 (Natural Deduction) 25 pt

Let $R \in \Sigma_2^p$, $P \in \Sigma_1^p$, $c \in \Sigma_0^f$.

Prove the following formula in Natural Deduction:

$$((\forall X.\forall Y.R(Y,X) \Rightarrow P(Y)) \ \wedge \ (\exists Y.R(c,Y))) \Rightarrow P(c)$$

AI 1 2022/23
Assignment9: First-Order Logic
– Given Jan. 12, Due Jan. 22 –

Problem 9.1 (Induction) 20 pt

Use structural induction on terms and formulas to define a function C that maps every term/formula to the number of occurrences of free variables. For example, $C(\forall x.P(x,x,y,y,z)) = 3$ because the argument has 2 free occurrences of y and 1 of z.

Hint: Use an auxiliary function $C'(V,A)$ that takes the set V of bound variables and a term/formula A. Define C' by structural induction on A. Then define $C(A) = C'(\emptyset, A)$.

Solution: C' is defined as follows for terms

- variables X: $C'(V,X) = 0$ if $X \in V$ and $C'(V,X) = 1$ if $X \notin V$
- applications of n-ary function symbol f: $C'(V, f(t_1,...,t_n)) = \Sigma_i C'(V,t_i)$

and for formulas

- applications of n-ary predicate symbol p: $C'(V, p(t_1,...,t_n)) = \Sigma_i C'(V,t_i)$
- nullary connectives: $C'(V,T) = C'(V,F) = 0$
- unary connectives: $C'(V,\neg A) = C'(V,A)$
- binary connectives: $C'(V, A_1 \wedge A_2) = C'(V, A_1 \vee A_2) = C'(V, A_1 \rightarrow A_2) = C'(V,A_1) + C'(V,A_2)$
- quantifiers: $C'(V, \forall x.A) = C'(V, \exists x.A) = C'(V \cup \{x\}, A)$

This definition exhibits the typical pattern of structural induction:

- An additional argument (V) is used to track the bound variables.
- When recursing into a quantifier that argument is updated by adding the bound variable x. (In general, additional information about could be added, e.g., whether it is bound by $\forall$ or $\exists$.)
- At the leafs of the syntax tree (the base cases of the induction, here the variables), the additional argument is used.
- The main function is defined by initializing the additional argument (here with $\emptyset$).

Problem 9.2 (First-Order Semantics) 30 pt

Let $= \in \Sigma_2^p$, $P \in \Sigma_1^p$ and $+ \in \Sigma_2^f$. We use the semantics of first-order logic without equality.

Prove or refute the following formulas semantically. That means you must show that $I_\varphi(A) = T$ for all models I and assignments φ (without using a proof calculus) or to give some I, φ such that $I_\varphi(A) = F$.

1. $P(X)$
2. $\forall X.\forall Y. = (+(X,Y), +(Y,X))$
3. $\exists X.(P(X) \Rightarrow \forall Y.P(Y))$
4. $P(Y) \Rightarrow \exists X.P(X)$

Solution: Let φ be any value function.

1. Not valid. One out of many counter-examples is given by domain $\mathbb{N}$, $I(P) = \{0\}$, and $\varphi(X) = 1$.
2. Not valid. A counter-model is $\mathcal{I}_\varphi(=) = \emptyset$ with an arbitrary domain.
3. Valid:

   $\mathcal{I}_\varphi(\exists X.(P(X) \Rightarrow \forall Y.P(Y))) = \top$
   $\Leftrightarrow$There is some $a \in \mathcal{D}_\mathcal{I}$ s.t. $\mathcal{I}_\varphi((P(a) \Rightarrow \forall Y.P(Y))) = \top$
   $\Leftrightarrow$There is some $a \in \mathcal{D}_\mathcal{I}$ s.t. $\mathcal{I}_\varphi(\neg(P(a) \wedge \neg\forall Y.P(Y))) = \top$
   $\Leftrightarrow$There is some $a \in \mathcal{D}_\mathcal{I}$ s.t. $\mathcal{I}_\varphi(P(a) \wedge \neg\forall Y.P(Y)) = \bot$
   $\Leftrightarrow$There is some $a \in \mathcal{D}_\mathcal{I}$ s.t. $\mathcal{I}_\varphi(P(a)) = \bot$ or $\mathcal{I}_\varphi(\neg\forall Y.P(Y)) = \bot$
   $\Leftrightarrow$There is some $a \in \mathcal{D}_\mathcal{I}$ s.t. $\mathcal{I}_\varphi(P(a)) = \bot$ or $\mathcal{I}_\varphi(\forall Y.P(Y)) = \top$
   $\Leftrightarrow$There is some $a \in \mathcal{D}_\mathcal{I}$ s.t. $\mathcal{I}_\varphi(P(a)) = \bot$ or for all $b \in \mathcal{D}_\mathcal{I}$: $\mathcal{I}_\varphi(P(b)) = \top$

   Now the last statement holds because if the left side does not hold, then the right side must hold.

Problem 9.3 (Natural Deduction) 25 pt

Let $R \in \Sigma_2^p$, $P \in \Sigma_1^p$, $c \in \Sigma_0^f$.

Prove the following formula in Natural Deduction:

$$((\forall X.\forall Y.R(Y,X) \Rightarrow P(Y)) \ \wedge \ (\exists Y.R(c,Y))) \Rightarrow P(c)$$

Solution:

1(Assumption)[1]	$(\forall X.\forall Y.R(Y,X) \Rightarrow P(Y)) \ \wedge \ (\exists Y.R(c,Y))$
2 $\wedge$ -Elimination on 1	$\forall X.\forall Y.R(Y,X) \Rightarrow P(Y)$
3 $\wedge$ -Elimination on 1	$\exists Y.R(c,Y)$
4$\forall$-Elimination on 2	$\forall Y.R(Y,X) \Rightarrow P(Y)$
5$\forall$-Elimination on 4	$R(c,X) \Rightarrow P(c)$
6$\forall$-Introduction on 5	$\forall X.R(c,X) \Rightarrow P(c)$
7(Assumption)[2]	$R(c,d)$
8$\forall$-Elimination on 6	$R(c,d) \Rightarrow P(c)$
9 $\Rightarrow$ -Elimination on 8, 7	$P(c)$
10$\exists$-Elimination[2] on 3, 9	$P(c)$
11 $\Rightarrow$ -Introduction[1] on 10	$((\forall X.\forall Y.R(Y,X) \Rightarrow P(Y)) \ \wedge \ (\exists Y.R(c,Y))) \Rightarrow P(c)$

AI 1 2022/23
Assignment10: Knowledge Representation
– Given Jan. 21, Due Jan. 29 –

Problem 10.1 (Unification) 30 pt

Decide whether (and how or why not) the following pairs of terms are unifiable.
$S_1 \in \Sigma_2^p, S_2 \in \Sigma_3^p, f \in \Sigma_1^f, g \in \Sigma_2^f, c \in \Sigma_0^f$

1. $S_1(g(f(x), g(x, y)), y)$ and $S_1(g(z, v), f(w))$
2. $S_2(g(f(x), g(x, u)), f(y), z)$ and $S_2(g(g(g(u, v), f(w)), f(c)), f(g(u, v)), f(c))$

Problem 10.2 (First-Order Resolution) 35 pt

Prove the following formula using resolution.
$P \in \Sigma_1^p, R \in \Sigma_2^p, a, b \in \Sigma_0^f$

$$\exists X.\forall Y.\exists Z.\exists W.((\neg P(Z) \wedge \neg R(b, a)) \vee \neg R(a, b) \vee R(W, a) \vee (P(Y) \wedge R(X, b)))$$

Problem 10.3 (First-Order Tableaux) 35 pt

Prove the following formula using the first-order free variable tableaux calculus.
We have $P \in \Sigma_1^p$.

$$\exists X.(P(X) \Rightarrow \forall Y.P(Y))$$

AI 1 2022/23
Assignment10: Knowledge Representation
– Given Jan. 21, Due Jan. 29 –

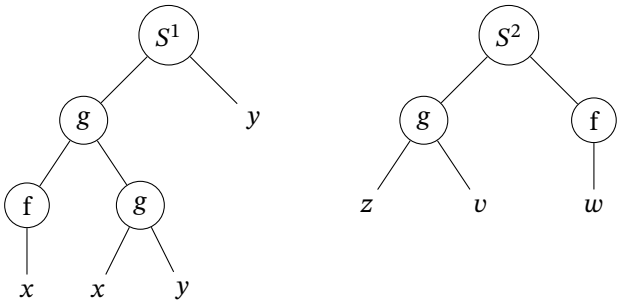
Problem 10.1 (Unification) 30 pt

Decide whether (and how or why not) the following pairs of terms are unifiable.
$S_1 \in \Sigma_2^p, S_2 \in \Sigma_3^p, f \in \Sigma_1^f, g \in \Sigma_2^f, c \in \Sigma_0^f$

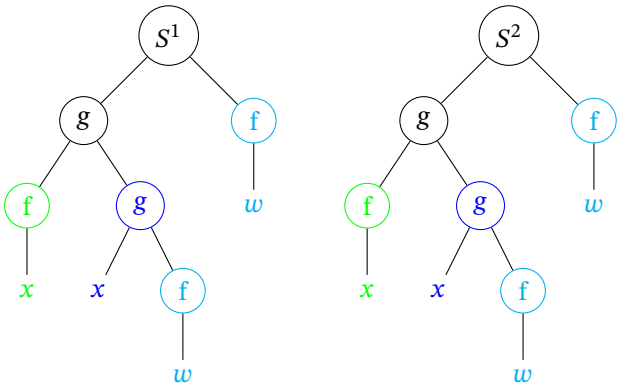
1. $S_1(g(f(x), g(x, y)), y)$ and $S_1(g(z, v), f(w))$
2. $S_2(g(f(x), g(x, u)), f(y), z)$ and $S_2(g(g(g(u, v), f(w)), f(c)), f(g(u, v)), f(c))$

Solution:

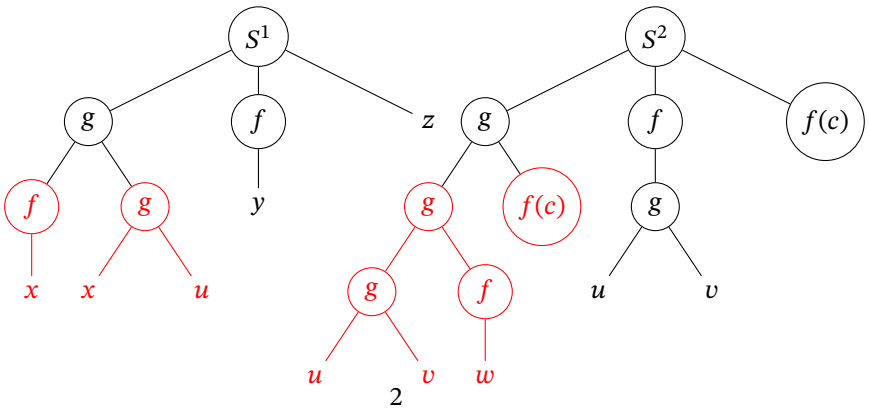
1. The term trees look like this:



Obviously, we need to perform the following substitutions to make the two trees equal:



2. The term trees look like this:



Obviously, the red subtrees can't be unified.

Problem 10.2 (First-Order Resolution) 35 pt

Prove the following formula using resolution.
$P \in \Sigma_1^p, R \in \Sigma_2^p, a, b \in \Sigma_0^f$

$$\exists X.\forall Y.\exists Z.\exists W.((\neg P(Z) \wedge \neg R(b,a)) \vee \neg R(a,b) \vee R(W,a) \vee (P(Y) \wedge R(X,b)))$$

Solution: We negate:

$$\forall X.\exists Y.\forall Z.\forall W.(P(Z) \vee R(b,a)) \wedge R(a,b) \wedge \neg R(W,a) \wedge (\neg P(Y) \vee \neg R(X,b))$$

We skolemize:

$$(P(Z) \vee R(b,a)) \wedge R(a,b) \wedge \neg R(W,a) \wedge (\neg P(f_Y(X)) \vee \neg R(X,b))$$

This yields the clauses $\{P(Z)^T, R(b,a)^T\}, \{R(a,b)^T\}, \{R(W,a)^F\}, \{P(f_Y(X))^F, R(X,b)^F\}$.
We resolve:

$$\{P(Z)^T, R(b,a)^T\} + \{R(W,a)^F\}[b/W] \Longrightarrow \{P(Z)^T\}$$
$$\{R(a,b)^T\} + \{P(f_Y(X))^F, R(X,b)^F\}[a/X] \Longrightarrow \{P(f_Y(a))^F\}$$
$$\{P(Z)^T\}[f_Y(a)/Z] + \{P(f_Y(a))^F\} \Longrightarrow \{\}$$

Problem 10.3 (First-Order Tableaux) 35 pt

Prove the following formula using the first-order free variable tableaux calculus.
We have $P \in \Sigma_1^p$.

$$\exists X.(P(X) \Rightarrow \forall Y.P(Y))$$

Solution:

(1)	$\exists X.(P(X) \Rightarrow \forall Y.P(Y))^F$	
(2)	$P(V_X) \Rightarrow \forall Y.P(Y)^F$	(from 1)
(3)	$P(V_X)^T$	(from 2)
(4)	$\forall Y.P(Y)^F$	(from 2)
(5)	$P(c_Y)^F$	(from 4)
(6)	$\bot[c_Y/V_X]$	

AI 1 2022/23
Assignment11: Knowledge Representation
– Given Jan. 26, Due Feb. 5 –

Problem 11.1 (CompLog Students in ALC) 30 pt

Using ALC, first give a list of suitable primitive concepts and roles and then use them to represent the following:

1. the concept of students that are registered for at least one course
2. the concept of students that are only registered for courses taught by a woman
3. the fact that only students are registered for courses

Give the result of translating the ALC formulas into first-order logic. (Recall that a fact/concept is translated into a first-order formula with 0/1 free variables.)

Problem 11.2 (ALC Semantics) 30 pt

Consider the ALC concepts $\forall R.(C \sqcap D)$ and $(\forall R.C) \sqcap (\forall R.D)$.

1. By applying the semantics of ALC, show that the two are equivalent.
2. Translate both formulas to first-order logic and state which FOL formula we would need to prove (e.g., with the ND calculus) to show that the two are equivalent.

Problem 11.3 (ALC TBox) 40 pt

Consider ALC with the following

- primitive concepts: `woman`, `man`
- roles: `has_child`, `has_parent`, `has_sibling`, `has_spouse`

Give an ALC TBox that defines the concepts `person`, `parent`, `mother`, `father`, `grandmother`, `aunt`, `uncle`, `sister`, `brother`, `onlychild`, `cousin`, `nephew`, `niece`, `fatherinlaw`, `motherinlaw`.

AI 1 2022/23
Assignment11: Knowledge Representation
– Given Jan. 26, Due Feb. 5 –

Problem 11.1 (CompLog Students in ALC) 30 pt

Using ALC, first give a list of suitable primitive concepts and roles and then use them to represent the following:

1. the concept of students that are registered for at least one course
2. the concept of students that are only registered for courses taught by a woman
3. the fact that only students are registered for courses

Give the result of translating the ALC formulas into first-order logic. (Recall that a fact/concept is translated into a first-order formula with $0/1$ free variables.)

Solution: primitive concepts: `student`, `woman`, `course`
roles: `registeredfor`, `taughtby`

ALC representations and first-order translations:

1. $\texttt{student} \sqcap \exists \texttt{registeredfor}.\texttt{course}$
   $\texttt{student}(x) \wedge \exists y.\texttt{registeredfor}(x,y) \wedge \texttt{course}(y)$
2. $\texttt{student} \sqcap \forall \texttt{registeredfor}.(\texttt{course} \sqcap \exists \texttt{taughtby}.\texttt{woman})$
   $\texttt{student}(x) \wedge \forall y.\texttt{registeredfor}(x,y) \Rightarrow (\texttt{course}(y) \wedge \exists z.\texttt{taughtby}(y,z) \wedge \texttt{woman}(z))$
3. $\exists \texttt{registeredfor}.\texttt{course} \sqsubseteq \texttt{student}$
   $\forall x.(\exists y.\texttt{registeredfor}(x,y) \wedge \texttt{course}(y)) \Rightarrow \texttt{student}(x)$

Problem 11.2 (ALC Semantics) 30 pt

Consider the ALC concepts $\forall R.(C \sqcap D)$ and $(\forall R.C) \sqcap (\forall R.D)$.

1. By applying the semantics of ALC, show that the two are equivalent.
2. Translate both formulas to first-order logic and state which FOL formula we would need to prove (e.g., with the ND calculus) to show that the two are equivalent.

Solution:

1. We have:

$$\begin{aligned}
&[\![\forall R.(C \sqcap D)]\!] \\
&= \{x \in \mathcal{D} | \text{for all } y \in \mathcal{D}, \text{ if } (x,y) \in [\![R]\!], \text{then } y \in [\![C \sqcap D]\!]\} \\
&= \{x \in \mathcal{D} | \text{for all } y \in \mathcal{D}, \text{ if } (x,y) \in [\![R]\!], \text{then } y \in [\![C]\!] \cap [\![D]\!]\}
\end{aligned}$$

$$\begin{aligned}
&[\![(\forall R.C) \sqcap (\forall R.D)]\!] \\
&= [\![\forall R.C]\!] \cap [\![\forall R.D]\!] \\
&= \{x \in \mathcal{D} | \text{for all } y \in \mathcal{D}, \text{ if } (x,y) \in [\![R]\!], \text{then } y \in [\![C]\!] \cap \{x \in \mathcal{D} | \text{for all } y \in \mathcal{D}, \text{ if } (x,y) \in [\![R]\!], \text{then } y \in [\![D]\!]\}\}
\end{aligned}$$

   Now to prove that sets are equal, consider an $x \in \mathcal{D}$ and see that both conditions are equivalent to

$$\text{for all } y \in \mathcal{D}, \text{ if } (x,y) \in [\![R]\!], \text{then } y \in [\![C]\!] \text{and } y \in [\![D]\!]$$

2. The translation yields

$$C_1(x) = \forall y.R(x,y) \Rightarrow (C(y) \wedge D(y))$$

$$C_2(x) = (\forall y.R(x,y) \Rightarrow C(y)) \wedge (\forall y.R(x,y) \Rightarrow D(y))$$

   We need to show

$$\forall x.C_1(x) \Leftrightarrow C_2(x)$$

Problem 11.3 (ALC TBox) 40 pt

Consider ALC with the following

- primitive concepts: `woman`, `man`
- roles: `has_child`, `has_parent`, `has_sibling`, `has_spouse`

Give an ALC TBox that defines the concepts `person`, `parent`, `mother`, `father`, `grandmother`, `aunt`, `uncle`, `sister`, `brother`, `onlychild`, `cousin`, `nephew`, `niece`, `fatherinlaw`, `motherinlaw`.

Solution: $\texttt{person} = \texttt{man} \sqcup \texttt{woman}$
$\texttt{parent} = \texttt{person} \sqcap \exists \texttt{has\_child}.\texttt{person}$
$\texttt{mother} = \texttt{woman} \sqcap \texttt{parent}$
$\texttt{father} = \texttt{man} \sqcap \texttt{parent}$
$\texttt{grandmother} = \texttt{woman} \sqcap \exists \texttt{has\_child}.\texttt{parent}$
$\texttt{aunt} = \texttt{woman} \sqcap \exists \texttt{has\_sibling}.\texttt{parent}$
$\texttt{uncle} = \texttt{man} \sqcap \exists \texttt{has\_sibling}.\texttt{parent}$
$\texttt{sister} = \texttt{woman} \sqcap \exists \texttt{has\_sibling}.\texttt{person}$
$\texttt{brother} = \texttt{man} \sqcap \exists \texttt{has\_sibling}.\texttt{person}$
$\texttt{onlychild} = \texttt{person} \sqcap \neg\texttt{brother} \sqcup \texttt{sister}$
$\texttt{cousin} = \texttt{person} \sqcap \exists \texttt{has\_parent}.\exists \texttt{has\_sibling}.\texttt{parent}$
$\texttt{nephew} = \texttt{man} \sqcap \exists \texttt{has\_parent}.\exists \texttt{has\_sibling}.\texttt{person}$
$\texttt{niece} = \texttt{woman} \sqcap \exists \texttt{has\_parent}.\exists \texttt{has\_sibling}.\texttt{person}$
$\texttt{fatherinlaw} = \texttt{man} \sqcap \exists \texttt{has\_child}.\exists \texttt{has\_spouse}.\texttt{person}$
$\texttt{motherinlaw} = \texttt{woman} \sqcap \exists \texttt{has\_child}.\exists \texttt{has\_spouse}.\texttt{person}$

AI 1 2022/23
Assignment12: Planning
– Given Feb 2 –

Problem 12.1 (STRIPS Planning) 0 pt

Consider the road map of Australia given below. The task here is to visit Darwin, Brisbane and Perth, starting from Sydney.

The task is formalized in STRIPS as follows. Facts are $at(x)$ and $visited(x)$ where $x \in \{Adelaide, Brisbane, Darwin, Perth, Sydney\}$. The initial state is $\{at(Sydney), visited(Sydney)\}$, the goal is $\{visited(Brisbane), visited(Darwin), visited(Perth)\}$. The actions move along the roads, i.e., they are of the form

$$drive(x, y) : (\{at(x)\}, \{at(y), visited(y)\}, \{at(x)\})$$

where x and y have a direct connection according to the road map. **Each road can be driven in both directions, except for the road between Adelaide and Perth, which can only be driven from Adelaide to Perth, not in the opposite direction.** In your answers to the following questions, use the abbreviations "v" for "visited", and "Ad", "Br", "Da", "Pe", "Sy" for the cities.

(a) Give an optimal (shortest) plan for the initial state, if one exists; if no plan exists, argue why that is the case. Give an optimal (shortest) relaxed plan for the initial state, if one exists; if no relaxed plan exists, argue why that is the case. What is the h^* value and the h^+ value of the initial state? (When writing up a plan or relaxed plan, it suffices to give the sequence of action names.)

(b) Do the same as in (a) in the modified task where the road between Sydney and Brisbane is also one-way, i.e., it can only be driven from Sydney to Brisbane, not in the opposite direction.

(c) Write up, in STRIPS notation, all states reachable from the initial state in at most *two* steps. Start at the initial state, and insert successors. Indicate successor states by edges. Annotate the states with their h^* values as well as their h^+ values.

(d) Do the same as in (c) in the modified task where the road between Sydney and Brisbane is one-way, i.e., it can only be driven from Sydney to Brisbane, not in the opposite direction.

Problem 12.2 (Admissible Heuristics in Gripper) 0 pt

Consider a problem where we have two rooms, A and B, one robot initially located in room A, and n balls that are also initially located in room A. The goal demands that all balls be located in room B. The robot can move between the rooms, it can pick up balls provided its gripper hand is free (see below), and it can drop a ball it is currently holding.

Answer the following questions with yes/no. Justify your answer.

(a) Say that the robot has only one gripper, so that it can only hold one ball at a time. Is the number of balls not yet in room B an admissible heuristic function?

(b) Say that the robot has only one gripper, so that it can only hold one ball at a time. Is the number of balls still in room A, multiplied by 4, an admissible heuristic function?

(c) Say now the robot has two grippers, and it takes only one action to pick up two balls, and only one action to drop two balls. Is the number of balls not yet in room B an admissible heuristic function?

(d) Say now the robot has two grippers, but picks up/drops each ball individually, so that it needs two actions to take two balls, and two actions to drop two balls. Is the number of balls not yet in room B an admissible heuristic function?

Problem 12.3 (Partial Order Planning) 0 pt

Consider partial-order planning.

1. Given a STRIPS task $\Pi := \langle P, A, I, G \rangle$, what are the components of a partially ordered plan?
2. What are the conditions on a partially ordered plan to be complete and consistent?
3. How can we turn such a plan into a solution of the original planning task?

AI 1 2022/23
Assignment12: Planning
– Given Feb 2 –

Problem 12.1 (STRIPS Planning) 0 pt

Consider the road map of Australia given below. The task here is to visit Darwin, Brisbane and Perth, starting from Sydney.

The task is formalized in STRIPS as follows. Facts are $at(x)$ and $visited(x)$ where $x \in \{Adelaide, Brisbane, Darwin, Perth, Sydney\}$. The initial state is $\{at(Sydney), visited(Sydney)\}$, the goal is $\{visited(Brisbane), visited(Darwin), visited(Perth)\}$. The actions move along the roads, i.e., they are of the form

$$drive(x, y) : (\{at(x)\}, \{at(y), visited(y)\}, \{at(x)\})$$

where x and y have a direct connection according to the road map. **Each road can be driven in both directions, except for the road between Adelaide and Perth, which can only be driven from Adelaide to Perth, not in the opposite direction.** In your answers to the following questions, use the abbreviations "v" for "visited", and "Ad", "Br", "Da", "Pe", "Sy" for the cities.

(a) Give an optimal (shortest) plan for the initial state, if one exists; if no plan exists, argue why that is the case. Give an optimal (shortest) relaxed plan for the initial state, if one exists; if no relaxed plan exists, argue why that is the case. What is the h^* value and the h^+ value of the initial state? (When writing up a plan or relaxed plan, it suffices to give the sequence of action names.)

(b) Do the same as in (a) in the modified task where the road between Sydney and Brisbane is also one-way, i.e., it can only be driven from Sydney to Brisbane, not in the opposite direction.

(c) Write up, in STRIPS notation, all states reachable from the initial state in at most *two* steps. Start at the initial state, and insert successors. Indicate successor states by edges. Annotate the states with their h^* values as well as their h^+ values.

(d) Do the same as in (c) in the modified task where the road between Sydney and Brisbane is one-way, i.e., it can only be driven from Sydney to Brisbane, not in the opposite direction.

Solution:

(a) Optimal plan: $drive(Sy, Br), drive(Br, Sy), drive(Sy, Ad), drive(Ad, Da), drive(Da, Ad), drive(Ad, Pe)$. Optimal relaxed plan: $drive(Sy, Br), drive(Sy, Ad), drive(Ad, Da), drive(Ad, Pe)$. $h^* = 6, h^+ = 4$.

(b) Optimal plan: Does not exist because we must visit both Brisbane and Perth, but once we moved to either of these two, we cannot get back out again. Optimal relaxed plan: $drive(Sy, Br), drive(Sy, Ad), drive(Ad, Da), drive(Ad, Pe)$. $h^* = \infty, h^+ = 4$.

Problem 12.2 (Admissible Heuristics in Gripper) 0 pt

Consider a problem where we have two rooms, A and B, one robot initially located in room A, and n balls that are also initially located in room A. The goal demands that all balls be located in room B. The robot can move between the rooms, it can pick up balls provided its gripper hand is free (see below), and it can drop a ball it is currently holding.

Answer the following questions with yes/no. Justify your answer.

(a) Say that the robot has only one gripper, so that it can only hold one ball at a time. Is the number of balls not yet in room B an admissible heuristic function?

(b) Say that the robot has only one gripper, so that it can only hold one ball at a time. Is the number of balls still in room A, multiplied by 4, an admissible heuristic function?

(c) Say now the robot has two grippers, and it takes only one action to pick up two balls, and only one action to drop two balls. Is the number of balls not yet in room B an admissible heuristic function?

(d) Say now the robot has two grippers, but picks up/drops each ball individually, so that it needs two actions to take two balls, and two actions to drop two balls. Is the number of balls not yet in room B an admissible heuristic function?

Solution:

(a) Yes: The solution must contain at least one separate drop action for each ball that is not yet currently in room B.

(b) No: For example, in the initial state for $n = 1$, the length of an optimal solution is 3 (pick, move A B, drop), whereas the value of this heuristic function is 4.

(c) No: For example, if all but 2 balls are already in room B, and the robot is in room B and holds the 2 remaining balls, then the length of an optimal solution is 1, whereas the value of this heuristic function is 2.

(d) Yes, for the same reason as in (a).

Problem 12.3 (Partial Order Planning) 0 pt

Consider partial-order planning.

1. Given a STRIPS task $\Pi := \langle P, A, I, G \rangle$, what are the components of a partially ordered plan?
2. What are the conditions on a partially ordered plan to be complete and consistent?
3. How can we turn such a plan into a solution of the original planning task?

Solution:

1. A partially ordered plan consists of
   - a start node which has the facts in I as a postcondition,
   - a finish node which has the facts in G as a precondition
   - causal links $S \xrightarrow{p} T$ where p is a precondition fulfilled by S
   - temporal ordering constraints $S \prec T$.
2. A partially ordered plan is complete iff all preconditions are achieved by a causal link; and consistent iff the relation induced by causal links and ordering relations is a partial ordering.
3. Any linearization of a complete partially ordered plan is a solution.
