

# CP468 Assignments 1, 2, & 3 Report

Daniel Crha | 190891000

Mera Fares | 190322170

Ruben Halanen | 180573480

Samuel Tessema | 170535150

## Abstract

Artificial intelligence is rapidly becoming an essential tool for solving complex problems in a variety of fields. Its ability to learn and improve over time, analyze vast amounts of data, and automate tasks previously done by humans is making it indispensable in healthcare, finance, transportation, logistics, and other industries. However, it is important to approach each problem with a critical eye and to ensure that AI systems are designed and used in a responsible and ethical manner. By integrating AI into our existing workflows and processes, we can unlock new insights and achieve breakthroughs in areas that were previously thought to be unsolvable. In this report we will show how we used PROLOG to solve several problems that require automated reasoning and logical inference.

## Assignment 1

In assignment one we practice representation of knowledge which is a fundamental concept in artificial intelligence. The problem requires us to represent the problem domain, including the animals, the boat, and the river, in a way that can be understood by a computer program. We also learned about problem formulation by defining the states, operators, and a goal state. This involves breaking down the problem into its essential components and deciding what information needs to be tracked.

## Code

```

1  % To run, use the query: go(state(west,3,3,1),state(east,0,0,0)).
2
3  % CLAUSES
4
5  member(X,[X|_]).
6  member(X,[_|T]):-
7  member(X,T).
8
9  move(state(X,L,W,R),state(Y,L1,W1,R1)):-
10 opposite(X,Y),
11 ((W>0, L1 is L+1, W1 is W-1, R1 is R) ;
12 (R>0, L1 is L, W1 is W, R1 is R-1) ;
13 (W>0, R>0, L1 is L+1, W1 is W-1, R1 is R-1) ;
14 (L>0, L1 is L-1, W1 is W, R1 is R)),
15 not((L1<W1, Y == east)).
16
17 opposite(east, west). % The opposite of east is west
18 opposite(west, east).
19
20 path(S,G,L,L1) :-
21 move(S,S1),
22 not(member(S1,L)),
23 path(S1, G, [S1|L], L1).
24 % Comment out the following line to get more than one solution
25 %path(G,G,T,T).
26
27 path(G,G,T,T).
28
29 go(S,G):-
30 path(S,G,[S],L),
31 write("A solution is: "),
32 write(L).

```

## Code Explanation

This PROLOG program solves a puzzle involving three lions and three wildebeest who are trying to cross a river on a boat that can carry up to two of them at a time. The goal is to get all six animals safely across the river without allowing the lions to outnumber the wildebeest at any time on either side of the river.

The program defines the states of the puzzle, represented by the current location of the boat, the number of lions, the number of wildebeest, and the number of trips across the river. It also defines the operators, or moves, that can be made from one state to another by carrying animals across the river on the boat. The program then defines a path predicate that finds a safe path from the initial state to the goal state, using the move predicate to generate valid moves and the opposite predicate to ensure that the boat always crosses from one side to the other. Finally, the program defines a go predicate that starts the search for a safe path and prints out the first solution that it finds.

To use the program, you can call the go predicate with the initial state and the goal state as arguments, like this: go(state(west,3,3,1),state(east,0,0,0)). This will find and print out the first safe path that it finds from the starting state to the goal state.

**States** - The starting state (S) is defined as three lions and three wildebeests on the east side of the river bank. All other states are possible as long as the lions do not outnumber the wildebeests on either side of the river bank. The boat can carry up to two animals and needs at least one lion or wildebeest to cross the river.

**Operators** - The path function checks if the next state is safe and that it has not been visited before. The possible move are 2 lions cross, 2 wildebeests cross, 1 lion cross, 1 wildebeest cross, and 1 lion and 1 wildebeest cross.

**Goal** - The goal state (G) is defined as three lions and three wildebeests on the west side of the river bank. The main function "go" is used to run the program and will output a solution if one is found.

### Modified for 4 lions and 5 wildebeest

There is a solution if there were 4 lions and 5 wildebeest, but it would require more crossings than the previous scenario. Here's one possible solution:

1. One lion and one wildebeest cross to the left bank.
2. The lion goes back to the right bank, leaving one wildebeest on the left bank.
3. Two lions cross to the left bank.
4. One lion goes back to the right bank, leaving two lions on the left bank.
5. Two wildebeest cross to the left bank.
6. One lion goes back to the right bank, leaving one lion and two wildebeest on the left bank.
7. One wildebeest goes back to the right bank, leaving one lion and one wildebeest on the left bank.
8. Two lions cross to the left bank.
9. One lion goes back to the right bank, leaving two lions on the left bank.
10. Two wildebeest cross to the left bank.
11. One lion goes back to the right bank, leaving one lion and two wildebeest on the left bank.
12. One lion and one wildebeest cross to the left bank.

This is the minimum number of crossings required to ensure the safety of all animals in this scenario.

### Results

```
go(state(west,3,3,1),state(east,0,0,0)).
A solution is: [state(east,0,0,0), state(west,1,0,0), state(east,2,0,0), state(west,3,0,0), state(east,4,0,0), state(west,5,0,0), state(east,6,0,0), state(west,5,1,1), state(east,4,2,1), state(west,3,3,1)]
true
```

## Assignment 2

In assignment two we practice state representation, another fundamental concept in artificial intelligence. This program will solve the 8-puzzle problem. The 8-puzzle problem starts off with a set of tiles 3x3, with numbers 1-8, filling 8 of those tiles. The other tile is left blank. The goal is to rearrange the tiles by swapping the blank tile with one of its adjacent tiles, until the goal state is reached.

## Code -

```
s([Empty | Tiles], [Tile | Tiles1], 1) :-
    swap( Empty, Tile, Tiles, Tiles1).

swap( Empty, Tile, [Tile | Ts], [Empty | Ts] ) :-
    mandist( Empty, Tile, 1).           % Manhattan distance = 1.

swap( Empty, Tile, [T1 | Ts], [T1 | Ts1] ) :-
    swap( Empty, Tile, Ts, Ts1).

mandist( X/Y, X1/Y1, D) :-             % D is Manhattan dist. between two squares
    dif( X, X1, Dx),
    dif( Y, Y1, Dy),
    D is Dx + Dy.

dif( A, B, D) :-                       % D is |A-B|
    D is A-B, D >= 0, !
;
    D is B-A.

% Heuristic estimate h is the sum of distances of each tile
% from its "home" square plus 3 times "sequence" score

h( [Empty | Tiles], H) :-
    goal( [Empty | GoalSquares]),
    totdist( Tiles, GoalSquares, D),    % Total distance from home squares
    seq( Tiles, S),                   % Sequence score
    H is D + 3*S.

totdist( [], [], 0).

^totdist( [Tile | Tiles], [Square | Squares], D) :-
    mandist( Tile, Square, D1),
    totdist( Tiles, Squares, D2),
    D is D1 + D2.

% seq( TilePositions, Score): sequence score

seq( [First | OtherTiles], S) :-
    seq( [First | OtherTiles ], First, S).
```

```

seq( [Tile1, Tile2 | Tiles], First, S) :-
    score( Tile1, Tile2, S1),
    seq( [Tile2 | Tiles], First, S2),
    S is S1 + S2.

seq( [Last], First, S) :-
    score( Last, First, S).

score( 2/2, _, 1) :- !.                % Tile in centre scores 1

score( 3/3, 2/3, 0) :- !.              % Proper successor scores 0
score( 2/3, 1/3, 0) :- !.
score( 1/3, 1/2, 0) :- !.
score( 1/2, 1/1, 0) :- !.
score( 1/1, 2/1, 0) :- !.
score( 2/1, 3/1, 0) :- !.
score( 3/1, 3/2, 0) :- !.
score( 3/2, 3/3, 0) :- !.

score( _, _, 2).                        % Tiles out of sequence score 2

goal( [2/2,3/3,2/3,1/3,1/2,1/1,2/1,3/1,3/2] ). % Goal squares for tiles
% goal( [2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2] ). % Goal squares for tiles

% Display a solution path as a list of board positions

showsol( [] ).

showsol( [P | L] ) :-
    showsol( L),
    nl, write( '---'),
    showpos( P).

% Display a board position

showpos( [S0,S1,S2,S3,S4,S5,S6,S7,S8] ) :-
    member( Y, [3,2,1] ),                % Order of Y-coordinates
    nl, member( X, [1,2,3] ),            % Order of X-coordinates
    member( Tile-X/Y,                     % Tile on square X/Y
        [ ' '-S0,1-S1,2-S2,3-S3,4-S4,5-S5,6-S6,7-S7,8-S8 ] ),
    format( "~w ", [Tile] ),
    fail                                  % Backtrack to next square
;
true.                                     % All squares done

% Starting positions for some puzzles

start1( [2/2,1/3,3/2,2/3,3/3,3/1,2/1,1/1,1/2] ). % Requires 4 steps
start2( [2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3] ). % Requires 5 steps
start3( [2/2,2/3,1/3,3/1,1/2,2/1,3/3,1/1,3/2] ). % Requires 18 steps
start4( [2/2,1/2,3/3,1/3,1/1,2/1,3/1,3/2,2/3] ).

% An example query: ?-
start4( Pos),bestfirst( Pos, Sol), showsol(Sol).

```

## Code Explanation -

The puzzle is represented by a list of positions in a 3x3 square. E.g. [2/2,1/2,3/3,1/3,1/1,2/1,3/1,3/2,2/3]. The first element in the list corresponds with the “Empty” square. The empty square can change places with an adjacent square. The goal is to switch positions with the empty square until we get the desired goal state. The “swap” predicate is what does this action.

Firstly, the program establishes the Manhattan distance formula between any two squares. The formula is  $|x1-x2| + |y1-y2|$ . Then, the swap predicate establishes which square the empty square can swap with, that being any square that has a Manhattan distance of 1.

The heuristic used in this program is the sum of distances of each tile from its goal square plus 3 times sequence score. Sequence score is defined as Manhattan distance + 3\*s. S is equal to the distance between 2 tiles that are supposed to be next to each other. E.g., 3/3 and 2/3. If the sum of the distances is lower, we know that we are getting closer to the goal state. The score predicate determines if a number is in the right position. If the score between two side by side nodes is 0, that means the number is in the right spot and will no longer be moved.

The program will continue to swap between the empty tile and adjacent tiles until the goal state is reached.

## Results -

```
?- start4( Pos),bestfirst( Pos, Sol), showsol(Sol).

-----
382
1 7
456
-----
382
17
456
-----
382
417
56
-----
382
417
5 6
-----
382
417
56
-----
382
41
567
-----
382
4 1
567
-----
3 2
481
567
-----
32
481
567
-----
321
48
567
-----
321
4 8
567
```

**State** - The state is represented as a list of positions of the tiles, with the first item in the list corresponding to the empty square. For this example, the starting position in the figure can be represented as [2/2,1/2,3/3,1/3,1/1,2/1,3/1,3/2,2/3]. Starting from the first element, each element

An initial state with no solution can look like this:

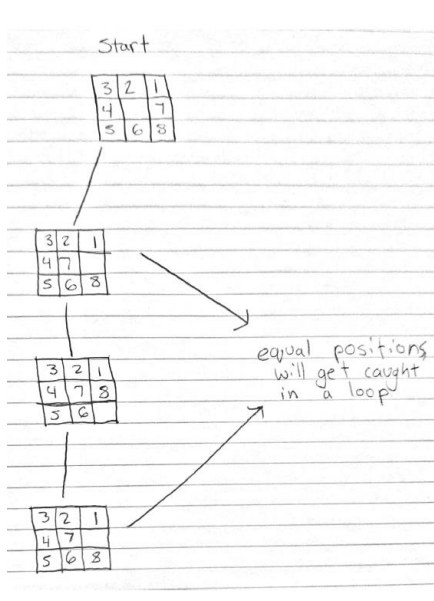
3	2	1
4		7
5	6	8

This position can be represented in the program as: [2/2, 1/3, 3/2, 2/3, 3/3, 3/1, 2/1, 1/2, 1/1].

Running this position into the program we get a false output:

```
?- start1([2/2,1/3,3/2,2/3,3/3,3/1,2/1,1/2,1/1]).
false.
```

This is because the seven and eight will constantly try and go to the correct position but will be caught in a loop, as represented by this diagram:



Since numbers 1-6 are on their “home squares (the correct position)” they will not move, leaving 7 and 8 stuck in a loop.

## Assignment 3

In assignment three we learned about the use of predicate logic by creating a function called "solve" that takes in two locations, "From" and "To", and returns a "Path" that goes from "From" to "To". The predicate "solve" is a statement that defines a relationship between two or more variables. This is a basic concept in predicate logic, which is the foundation of AI and forms the basis of reasoning.

### Code



```

1 % Define the maze
2 mazeSize(5, 9).
3 barrier(1, 8).
4 barrier(2, 2).
5 barrier(2, 4).
6 barrier(2, 5).
7 barrier(3, 4).
8 barrier(3, 7).
9 barrier(3, 9).
10 barrier(4, 4).
11 barrier(4, 7).
12 barrier(4, 9).
13 barrier(5, 2).
14
15 % Define valid moves from a given position
16 move([X, Y], [X, Y1]) :-
17     Y1 is Y + 1,
18     Y1 <= 9,
19     \+ barrier(X, Y1).
20 move([X, Y], [X, Y1]) :-
21     Y1 is Y - 1,
22     Y1 >= 1,
23     \+ barrier(X, Y1).
24 move([X, Y], [X1, Y]) :-
25     X1 is X + 1,
26     X1 <= 5,
27     \+ barrier(X1, Y).
28 move([X, Y], [X1, Y]) :-
29     X1 is X - 1,
30     X1 >= 1,
31     \+ barrier(X1, Y).
32
33 % Define a predicate to find a path from From to To
34 solve(From, To, Path) :-
35     solve_helper([From], To, [], Path).
36
37 % Define a helper predicate to find a path recursively
38 solve_helper([To|Path], To, _, Solution) :-
39     reverse([To|Path], Solution).
40 solve_helper([From|Path], To, Tried, Solution) :-
41     move(From, Next),
42     \+ member(Next, Tried),
43     solve_helper([Next,From|Path], To, [Next|Tried], Solution).
44

```

## Code explanation

This program defines a maze and provides a solution to find a path from a starting position to an end position in the maze using PROLOG. The following is a detailed explanation of the code:

First, the code defines the size of the maze and the locations of barriers. The maze is represented as a grid with five rows and nine columns. The barrier predicate is used to specify the row and column of each barrier in the maze.



Next, the move predicate is defined to specify the valid moves that can be made from a given position in the maze. A move is valid if it does not go beyond the maze boundaries and does not hit a barrier.

The solve predicate is then used to find a path from the starting position to the end position in the maze. This predicate calls a helper predicate, solve\_helper, with the starting position, end position, an empty list for tried positions, and an empty list for the solution path.

The `solve_helper` predicate is used to find a path recursively. It first checks if the current position is the end position. If it is, it returns the path taken so far as the solution. If the current position is not the end position, it checks for all valid moves from the current position and recursively calls `solve_helper` with the next position, the current position added to the path, the list of tried positions with the next position added, and the solution path.

Overall, the code uses recursion to find a path through the maze by exploring all possible paths from the starting position to the end position and avoiding any barriers.

## Results

 <code>solve([3,1], [2,6], Path).</code>	 <code>solve([3,1], [5,9], Path).</code>
<b>Path</b> = [ [3, 1], [3, 2], [3, 3], [4, 3], [4, 2], [4, 1], [3, 1], [2, 1], [1, 1], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [1, 7], [2, 7], [2, 8], [3, 8], [4, 8], [5, 8], [5, 7], [5, 6], [5, 5], [4, 5], [4, 6], [3, 6], [2, 6] ]	<b>Path</b> = [ [3, 1], [3, 2], [3, 3], [4, 3], [4, 2], [4, 1], [3, 1], [2, 1], [1, 1], [1, 2], [1, 3], [1, 4], [1, 5], [1, 6], [1, 7], [2, 7], [2, 8], [3, 8], [4, 8], [5, 8], [5, 9] ]