Artificial Intelligence 1
Lab 3

Eduardo Bier s3065979 & Auke Roorda s2973782
CS1

June 2, 2016

Theory Lab 3a

1. Solving a small set of equations

The solver is able to find the one and only solution to the set of equations: A = 3, B = 7, C = 2. The best propagation technique to be used for this problem is arc consistency. Since every variable is connected to each other, making the problem arc consistent has a huge impact on reducing the number of visited states. The available heuristics do not improve the performance of the solver, since all variables appear the same amount of times on each constraint and have the same amount of possible values. The code can be found on Listing 1.

Listing 1: 42.csp

```
#The set of variables of the CSP
variables:
        A,B,C : integer;

#Here the domains are defined
domains:
        A,B,C <- [0..250];

#Here are the constraints:
constraints:
        A + B = 5*C;
        B + C = 3*A;
        A * B * C = 42;

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

2. Market

The problem was formulated taking into account the fact that the solver only supports integers. As so, instead of using euros to represent the money related constraints, cents were used. Note that this problem is very similar to the previous problem, since it boils down to solving a small set of (5) equations. As so, the heuristics, propagation techniques and reasons for using them were exactly the same. There are 5 different solutions to this problem, as shown below:

Solution 1: O = 1, G = 62, M = 37

Solution 2: O = 4, G = 48, M = 48

Solution 3: O = 7, G = 34, M = 59

Solution 4: O = 10, G = 20, M = 70

Solution 5: O = 13, G = 6, M = 81

The code can be found on Listing 2.

Listing 2: market.csp

```
#The set of variables of the CSP
variables:
	O,G,M : integer;

#Here the domains are defined
domains:
	O,G,M <- [0..250];

#Here are the constraints:
constraints:
	O + G + M = 100;
	O*88 + G*99 + M*102 = 10000;

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

3. Chain of trivial equations

As expected, the number of solutions of the equations is 100, one for each value in the variable's domain. By adding the new constraint A = 42, the system now only has 1 solution (every variable equals 42) and the program visits 2502 states. By changing that constraint to Z = 42, the same solution is reached. The number of states, however, is greatly reduced to 27. This happens because

Z does not depend on any other variable, like A does. So when we write A = 42, we still have to check the values of B, C, D, ..., Z before finding a solution, whereas with Z = 42, we know right away that Z is actually 42 and therefore all other letters are too. No heuristics and propagation techniques help improve the performance of this problem because all variables appear on the same number of constraints. The code can be found on Listing 3.

Listing 3: chain.csp

```
#The set of variables of the CSP
variables:
	A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z : integer;

#Here the domains are defined
domains:
	A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z <- [0..99];

#Here are the constraints:
constraints:
	A=B;
	B=C;
	C=D;
	D=E;
	E=F;
	F=G;
	G=H;
	H=I;
	I=J;
	J=K;
	K=L;
	L=M;
	M=N;
	N=O;
	O=P;
	P=Q;
	Q=R;
	R=S;
	S=T;
	T=U;
	U=V;
	V=W;
	W=X;
	X=Y;
	Y=Z;
	N = 42;
# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

4. Constraint Graph

Both forward checking and arc consistency do a great job at solving the problem, but arc consistency ends up being better because of the structure of the problem. Every variable is connected to another one on each constraint. The solutions for the constraint graph are:

Solution 1: A = 3, B = 3, C = 4, D = 2, E = 1

Solution 2: A = 4, B = 4, C = 2, D = 3, E = 1

The code can be found on Listing 4.

Listing 4: graph.csp

```
#The set of variables of the CSP
variables:
       A,B,C,D,E : integer;

#Here the domains are defined
domains:
       A,B,C,D,E <- [1..4];

#Here are the constraints:
constraints:
       B >= A;
       C <> A;
       A > D;
       D > E;
       C > E;
       B <> C;
       C <> D;
       C <> D + 1;
# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

5. Cryptarithmetic Puzzle

Like in the previous problems, arc consistency is the best propagation technique to be used in this problem. The minimum remaining values heuristic also improves the performance of the solver, since now the number of possibilities of values of the variables is very different. By picking the one with the least values the solver ends up visiting less states, since it'll need to test less values. The other problems have 1, 1 and 0 solutions, respectively. The code can be found on Listings 5, 6, 7 and 8.

Listing 5: cryptarithmetic.csp

```
#The set of variables of the CSP
variables:
       S,E,N,D,M,O,R,Y,carry[4] : integer;

#Here the domains are defined
domains:
       S,E,N,D,M,O,R,N,Y <- [0..9];
       carry <- [0, 1];

#Here are the constraints:
constraints:
       D + E = Y + 10 * carry[0];
       carry[0] + N + R = E + 10 * carry[1];
       carry[1] + E + O = N + 10 * carry[2];
       carry[2] + S + M = O + 10 * carry[3];
       carry[3] = M;
       M <> 0;
       S <> 0;
       alldiff(S,E,N,D,M,O,R,Y);

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

Listing 6: onze.csp

```
#The set of variables of the CSP
variables:
       U,N,E,F,O,Z,carry[4] : integer;

#Here the domains are defined
domains:
       U,N,E,F,O,Z <- [0..9];
       carry <- [0..2];

#Here are the constraints:
constraints:
       F + N + N = E + 10 * carry[0];
       carry[0] + U + U + U = Z + 10 * carry[1];
       carry[1] + E = N + 10 * carry[2];
       carry[2] + N mod 10 = O + 10 * carry[3];
       N <> 0;
       O <> 0;
       U <> 0;
       alldiff(U,N,E,F,O,Z);

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
```

```
solutions: all
```

Listing 7: eighty.csp

```
#The set of variables of the CSP
variables:
       T,W,E,N,Y,F,I,O,G,H,carry[4] : integer;

#Here the domains are defined
domains:
       T,W,E,N,Y,F,I,O,G,H <- [0..9];
       carry[0],carry[1],carry[2] <- [0..3];
       carry[3] <- [0..2];

#Here are the constraints:
constraints:
       Y + Y + E + E = Y + 10 * carry[0];
       carry[0] + T + T + N + N = T + 10 * carry[1];
       carry[1] + N + F + I + O = H + 10 * carry[2];
       carry[2] + E + I + N = G + 10 * carry[3];
       carry[3] + W + F = I + 10;
       1 + T = E;
       T <> 0;
       F <> 0;
       N <> 0;
       O <> 0;
       E <> 0;
       alldiff(T,W,E,N,Y,F,I,O,G,H);

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

Listing 8: truth.csp

```
#The set of variables of the CSP
variables:
       T,R,U,H,G,E,S,I,carry[4] : integer;

#Here the domains are defined
domains:
       T,R,U,H,G,E,S,I <- [0..9];
       carry <- [0, 3];

#Here are the constraints:
constraints:
       H + S + E + I = S + 10 * carry[0];
       carry[0] + T + S + H = T + 10 * carry[1];
       carry[1] + U + E + T = R + 10 * carry[2];
```

```
        carry[2] + R + U = U + 10 * carry[3];
        carry[3] + T + G = H;
        T <> 0;
        G <> 0;
        H <> 0;
        alldiff(T,R,U,H,G,E,S,in);

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

6. 20 First Primes

By creating an array to put the prime numbers in descending order, no flags were needed to be set for the problem to be solved in a reasonable amount of time. The solution given solves the problem visiting just 21 states and even works (with the proper adjustments) for a higher number of prime numbers. The reason it works so well is because of how the csp solver tries to assign values to the variables: it starts at the end of the array with the lowest value of the domain. It's therefore able to assign the right value for the smallest primes really quickly and finds the prime numbers in order. As so, the algorithm performs extremely well even with larger domains. Note that this only works because we only want to find 1 solution to the problem.The code can be found on Listing 9.

Listing 9: primes.csp

```
variables:
prime[20] : integer;

domains:
prime <- [2..74];

constraints:
forall(i in [0..18])
        prime[i] > prime[i + 1];
        forall(j in [i + 1..19])
                prime[i] mod prime[j] <> 0;
        end
end

solutions:1
```

7. Sudoku

The solver performed well on both solutions, but visited almost 5 times the amount of states for the second puzzle. This is probably due to the fact that there are more numbers already in place for the first puzzle. Sudoku puzzles work well with th e minimum remaining value heuristic, since you really limit your search space by choosing the right tile. The arc consistency was once again the best propagation method, although forward chaining also did a pretty good job at it. The performance is further improved by using the flag -iconst 2. The solutions for the Sudoku puzzles are give by tables 1 and 2

The code can be found on Listing 10.

Listing 10: sudoku.csp

```
variables:
sudoku[9][9] : integer;

domains:
sudoku <- [1..9];

constraints:
#Sudoku Rules

forall(i in [0..8])
	#Rows
	alldiff(sudoku[i]);

	#Columns
	alldiff(sudoku[0][i],
			sudoku[1][i],
			sudoku[2][i],
			sudoku[3][i],
			sudoku[4][i],
			sudoku[5][i],
			sudoku[6][i],
			sudoku[7][i],
			sudoku[8][i]);

	#Squares
	alldiff(sudoku[0 + 3 * (i div 3)][0 + 3 * (i mod 3)],
			sudoku[0 + 3 * (i div 3)][1 + 3 * (i mod 3)],
			sudoku[0 + 3 * (i div 3)][2 + 3 * (i mod 3)],
			sudoku[1 + 3 * (i div 3)][0 + 3 * (i mod 3)],
			sudoku[1 + 3 * (i div 3)][1 + 3 * (i mod 3)],
			sudoku[1 + 3 * (i div 3)][2 + 3 * (i mod 3)],
			sudoku[2 + 3 * (i div 3)][0 + 3 * (i mod 3)],
			sudoku[2 + 3 * (i div 3)][1 + 3 * (i mod 3)],
			sudoku[2 + 3 * (i div 3)][2 + 3 * (i mod 3)]);
end

#Initial Board
sudoku[0][2] = 8;
sudoku[0][3] = 6;
```

```
sudoku[0][4] = 3;
sudoku[0][5] = 2;
sudoku[0][6] = 4;

sudoku[1][1] = 4;
sudoku[1][7] = 1;

sudoku[2][0] = 5;
sudoku[2][3] = 9;
sudoku[2][5] = 4;
sudoku[2][8] = 6;

sudoku[3][0] = 8;
sudoku[3][8] = 5;

sudoku[4][0] = 6;
sudoku[4][8] = 4;

sudoku[5][0] = 1;
sudoku[5][2] = 7;
sudoku[5][6] = 9;
sudoku[5][8] = 2;

sudoku[6][0] = 4;
sudoku[6][3] = 7;
sudoku[6][4] = 5;
sudoku[6][5] = 1;
sudoku[6][8] = 3;

sudoku[7][1] = 6;
sudoku[7][7] = 2;

sudoku[8][2] = 5;
sudoku[8][3] = 8;
sudoku[8][4] = 2;
sudoku[8][5] = 6;
sudoku[8][6] = 7;

solutions:1
```

8. N Queens problem

The N queens problem is best solved using the arc consistency propagation technique because of the high degree of connection between the variables. The number of solutions for each problem is given by table 3.

The code can be found on Listings 11, 12, 13, 14 and 15.

9	1	8	6	3	2	4	5	7
2	4	6	5	8	7	3	1	9
5	7	3	9	1	4	2	8	6
8	3	4	2	6	9	1	7	5
6	2	9	1	7	5	8	3	4
1	5	7	3	4	8	9	6	2
4	8	2	7	5	1	6	9	3
7	6	1	4	9	3	5	2	8
3	9	5	8	2	6	7	4	1

Table 1: First sudoku puzzle solved

8	3	9	4	6	5	7	1	2
1	4	6	7	8	2	9	5	3
7	5	2	3	9	1	4	8	6
3	9	1	8	2	4	6	7	5
5	6	4	1	7	3	8	2	9
2	8	7	6	5	9	3	4	1
6	2	8	5	3	7	1	9	4
9	1	3	2	4	8	5	6	7
4	7	5	9	1	6	2	3	8

Table 2: Second sudoku puzzle solved

Listing 11: 4queens.csp

```
#The set of variables of the CSP
variables:
        queens[4] : integer;

#Here the domains are defined
domains:
        queens <- [0..3];

#Here are the constraints:
constraints:
        alldiff(queens); #different rows
        forall(i in [0..3])
                forall(j in [i + 1..3])
                        abs(queens[i] - queens[j]) <> abs(i - j);
                end
        end


# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

nQueens	Solutions
4	2
5	16
6	4
7	40
8	92

Table 3: Number of solutions for varying values of n.

Listing 12: 5queens.csp

```
#The set of variables of the CSP
variables:
	queens[5] : integer;

#Here the domains are defined
domains:
	queens <- [0..4];

#Here are the constraints:
constraints:
	alldiff(queens); #different rows
	forall(i in [0..4])
		forall(j in [i + 1..4])
			abs(queens[i] - queens[j]) <> abs(i - j);
		end
	end


# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

Listing 13: 6queens.csp

```
#The set of variables of the CSP
variables:
	queens[6] : integer;

#Here the domains are defined
domains:
	queens <- [0..5];

#Here are the constraints:
constraints:
	alldiff(queens); #different rows
	forall(i in [0..5])
		forall(j in [i + 1..5])
			abs(queens[i] - queens[j]) <> abs(i - j);
```

```
		end
	end


# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

Listing 14: 7queens.csp

```
#The set of variables of the CSP
variables:
	queens[7] : integer;

#Here the domains are defined
domains:
	queens <- [0..6];

#Here are the constraints:
constraints:
	alldiff(queens); #different rows
	forall(i in [0..6])
		forall(j in [i + 1..6])
			abs(queens[i] - queens[j]) <> abs(i - j);
		end
	end


# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

Listing 15: 8queens.csp

```
#The set of variables of the CSP
variables:
	queens[8] : integer;

#Here the domains are defined
domains:
	queens <- [0..7];

#Here are the constraints:
constraints:
	alldiff(queens); #different rows
	forall(i in [0..7])
		forall(j in [i + 1..7])
			abs(queens[i] - queens[j]) <> abs(i - j);
		end
```

```
        end


# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

9. Magic Square

The magic square problem is best solved using the -arc and -mrv flags because of the high degree of connection between the variables and because this limits the search space, since by picking the variables with fewest possibilities first we find dead-ends quickly. There are 7040 different solutions for the magic square of size 4. The code can be found on Listing 16.

Listing 16: magicSquares.csp

```
#The set of variables of the CSP
variables:
        square[4][4] : integer;

#Here the domains are defined
domains:
        square <- [1..16];

#Here are the constraints:
constraints:
        alldiff(square);
        forall(i in [0..3])
                sum(square[i]) = 34; #rows
                square[0][i] + square[1][i] + square[2][i] + square[3][i] = 34; #columns
        end
        square[0][0] + square[1][1] + square[2][2] + square[3][3] = 34; #diagonal 1
        square[0][3] + square[1][2] + square[2][1] + square[3][0] = 34; #diagonal 2

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

10. Boolean satisfiability

In order to represent boolean values, the domains of the variables is [0,1]. To make the operation $\vee$ the operation max() was used, while the operation $\wedge$ didn't have to be explicitly described on the .csp file, since each constraint would have to be true anyway. Any of the propagation techniques aligned with any of the heuristics proved to improve the solvers performance in the same way. There are 9 solutions to this problem:

Solution 1
x = 0 1 0 0 0

Solution 2
x = 1 1 0 0 0

Solution 3
x = 1 1 1 0 0

Solution 4
x = 1 0 1 1 0

Solution 5
x = 0 0 0 0 1

Solution 6
x = 1 0 0 0 1

Solution 7
x = 1 1 0 0 1

Solution 8
x = 0 0 0 1 1

Solution 9
x = 1 0 1 1 1

The code can be found on Listing 17.

Listing 17: bool.csp

```
#The set of variables of the CSP
variables:
       x[5] : integer;

#Here the domains are defined
domains:
       x <- [0..1];

#Here are the constraints:
constraints:
       max(x[0],max(x[1],(x[2] + 1) mod 2)) = 1; # x1 v x2 v ~x3
       max((x[0] + 1) mod 2, max((x[1] + 1) mod 2,(x[3] + 1) mod 2)) =
           1; #~x1 v ~x2 v ~x4
```

```
        max(x[0], max((x[1] + 1) mod 2,(x[4] + 1) mod 2)) = 1; # x1 v ~x2 v ~x5
        max((x[0] + 1) mod 2, max(x[2],(x[3] + 1) mod 2)) = 1; # ~x1 v x3 v ~x4
        max(x[0], max((x[2] + 1) mod 2,x[4])) = 1; # x1 v ~x3 v x5
        max(x[0], max((x[3] + 1) mod 2,x[4])) = 1; # x1 v ~x4 v x5
        max(x[1], max(x[3],x[4])) = 1; # x2 v x4 v x5
        max((x[2] + 1) mod 2, max(x[3],(x[4] + 1) mod 2)) = 1; # ~x3 v x4 v ~x5

# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

Theory Lab 3b

Model Checking

In order to generate all the possible model values, the model array was viewed as a binary number. The model checking started with the value 0 and, after checking that model, 1 was added to that number. For example, if the model had 5 variables, the initial model would be represented by 00000. After checking that model, we would move to the model 00001, followed by 00010, 00011 and so on, until 00000 was reached again. This ensures that every possible combination is tested by the model checker. The checking itself was done by the already implemented function evaluateExpressionSet(). Note that the inferred expressions were only checked if the model was consistent with the KB. The code can be found on Listing 18.

Resolution

The recursivePrintProof function was implemented by saving the parents of each clause, which were easy to keep track of, since they never change locations on the kb array. As so, only their indexes needed to be kept.
The KB created and query were:

```
KB=[[a],[~a,b],[~b,c],[~c,d],[~d,e],[~e,f],[~f,g],[~g,h],[~h,i],[~i,j],
    [~j,k],[~k,l],[~l,m],[~m,n],[~n,o]]
query = [o]
```

Which proved to be true, as shown below:

```
Proof:
[b] is inferred from [a] and [~a,b].
[~b,d] is inferred from [~b,c] and [~c,d].
[d] is inferred from [b] and [~b,d].
```

```
[~d,f] is inferred from [~d,e] and [~e,f].
[~f,h] is inferred from [~f,g] and [~g,h].
[~d,h] is inferred from [~d,f] and [~f,h].
[h] is inferred from [d] and [~d,h].
[~h,j] is inferred from [~h,i] and [~i,j].
[~j,l] is inferred from [~j,k] and [~k,l].
[~h,l] is inferred from [~h,j] and [~j,l].
[~l,n] is inferred from [~l,m] and [~m,n].
[~n] is inferred from [~n,o] and [~o].
[~l] is inferred from [~l,n] and [~n].
[~h] is inferred from [~h,l] and [~l].
[]=FALSE is inferred from [h] and [~h].
```

The code can be found on Listing 19.

Prolog

1. **Biblical Family**

   (a) The prolog query that determines who's Lot's grandfather is:

   ```
   ?- grandfather(X,lot).
   ```

   The answer is:

   ```
   X = terach .
   ```

   (b) The prolog query that determines all of Terach's grandsons is:

   ```
   ?- findall(X,grandfather(terach,X),Grandsons).
   ```

   which returns the result in the list Grandsons:

   ```
   Grandsons = [isaac, milcah, yiscah, lot].
   ```

2. **Arithmetic with natural numbers**

   (a) The query should be:

   ```
   plus(s(s(s(0))),s(s(0)),s(s(s(s(s(0)))))).
   ```

   The program returns true.

   (b) The query should be:

```
plus(s(s(s(0))),s(s(0)),s(s(s(s(s(0)))))).
```

The program returns false.

(c) To add the even and odd predicates, the same recursive approach was taken as with the other predicates. The following code was added to arith.pl:

```
% Even numbers
even(0).
even(s(X)) :- odd(X).

% Odd numbers
odd(0) :- false.
odd(s(X)) :- even(X).
```

The code was tested for the values 0, 1, 2 and 3, which returned the expected result.

(d) To add the even and odd predicates using the times predicate, the following code was added to arith.pl:

```
% divi2(N,D) is true if D = N / 2
divi2(N,D) :- times(D,s(s(0)),N).
divi2(s(N),D) :- times(D,s(s(0)),N).
```

The code was tested for the values [0,0], [2,1], [3, 1], [1,0] and [5,2], which returned the expected result.

(e) The query should be

```
pow(s(s(0)),X,s(s(s(s(s(s(s(s(0)))))))))
```

which returns

```
X = s(s(s(0))).
```

The predicate log was implemented by the following code:

```
log(X,B,N) :- pow(B,N,X).
```

(f) The predicate fib(X,Y) was implemented by the following code:

```
fib(0,0).
fib(s(0),s(0)).
fib(s(s(X)),Y) :- fib(s(X),F1), fib(X,F2), plus(F1,F2,Y).
```

(g) The predicate power was implemented by the following code:

```
power(X,0,s(0)).
power(X,s(s(0)),Y) :- times(X, X, Y).
power(X,N,Y) :- even(N), div2(N, D1), power(X,s(s(0)),P1),
    power(P1,D1,Y).
power(X,s(N),Y) :- odd(s(N)), power(X,N,P1), times(X,P1,Y).
```

This is not an improvement over the direct computation because when the program is looking for a wrong result it ends up getting stuck in a very long (albeit finite) loop, since it calls itself several times and has to check whether the values check out or not. For the cases that are true, however, the code seems to work pretty well.

The code can be found on Listing 20.

3. **Lists**
The list operations member and concat were implemented in a straightforward way, by using recursion and the head/tail feature of prolog to go through the list. To implement reverse, the same strategy was used, but an extra list had to be created in order to actually reverse the list, since there was no way to get the last element of the list. The new list was built using concat. To determine whether or not a list was a palindrome, all that had to be done was use the reverse operator with the same list. The code can be found on Listing 21.

4. **Maze**
In order to represent the maze in prolog, the predicates north(X,Y), south(X,Y), west(X,Y) and east(X,Y) were created, which were true whenever X was north/south/west/east of Y. Finding a path from X to Y boiled down to finding a path from one of the adjacent squares of X to Y. The recursion ends when the goal is reached. To avoid infinite loops (going back and forth between two opposite directions), a list containing the path so far was kept using the list operations created on the previous exercise.
The query path(a,p) was indeed successful, but path(a,m) returned false, which is in fact the desired behavior. As so, the programs seems to work well. The code can be found on Listing 22.

1 Code

model.c

Listing 18: model.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

```

```c
#define MAXKBSIZE       1024
#define MAXIDENTIFIERS  1024
#define MAXIDENTNAMELENGTH 30

#define FALSE    0
#define TRUE     1

#define CONSTANT 0
#define IDENTIFIER 1
#define EQUIV    2
#define IMPLIES  3
#define AND      4
#define OR       5
#define NEG      6

typedef struct Expression {
  int operator;
  int atom;        /* only used if operator is CONSTANT or IDENTIFIER */
  struct Expression *operand1; /* only used if expression is not an atom */
  struct Expression *operand2; /* only used if operator is binary */
} *Expression;

int linenr = 1;
int colnr = 0;
int curchar;

int kbSize, inferSize;
Expression kb[MAXKBSIZE], infer[MAXKBSIZE];
int model[MAXIDENTIFIERS];
int inferred;

int cntidents = 0;
char identifiers[MAXIDENTIFIERS][MAXIDENTNAMELENGTH+1];

Expression makeConstantExpression(int value) {
  Expression e = malloc(sizeof(struct Expression));
  e->operator = CONSTANT;
  e->atom = value;
  e->operand1 = e->operand2 = NULL; /* not used */
  return e;
}

Expression makeIdentifier(char *ident) {
  int i = 0;
  Expression e;
  if (strcmp("false", ident)==0) return makeConstantExpression(FALSE);
  if (strcmp("true", ident)==0) return makeConstantExpression(TRUE);

  /* is this a new identifier? */
```

```c
  while ((i < cntidents) && (strcmp(identifiers[i], ident))) i++;
  if (i == cntidents) {
    /* new identifier, insert in symbol table */
    strncpy(identifiers[i], ident, MAXIDENTNAMELENGTH);
    identifiers[i][MAXIDENTNAMELENGTH-1]='\0';
    cntidents++;
  }
  e = malloc(sizeof(struct Expression));
  e->operator = IDENTIFIER;
  e->atom = i;
  e->operand1 = e->operand2 = NULL; /* not used */
  return e;
}

Expression makeNegation(Expression arg) {
  Expression e = malloc(sizeof(struct Expression));
  e->operator = NEG;
  e->atom = 99; /* not used */
  e->operand1 = arg;
  e->operand2 = NULL; /* not used */
  return e;
}

Expression makeBinaryExpr(int op, Expression arg1, Expression arg2) {
  Expression e = malloc(sizeof(struct Expression));
  e->operator = op;
  e->atom = 99; /* not used */
  e->operand1 = arg1;
  e->operand2 = arg2;
  return e;
}

void printExpression(Expression e) {
  if (e->operator == CONSTANT) {
    printf ("%s", (e->atom==TRUE ? "true" :"false"));
    return;
  }
  if (e->operator == IDENTIFIER) {
    printf("%s", identifiers[e->atom]);
    return;
  }
  if (e->operator == NEG) {
    printf ("!");
    printExpression(e->operand1);
    return;
  }

  printf("(");
  printExpression(e->operand1);

```

```c
  switch(e->operator) {
  case EQUIV:
    printf (" <=> ");
    break;
  case IMPLIES:
    printf (" => ");
    break;
  case AND:
    printf ("*");
    break;
  case OR:
    printf (" + ");
    break;
  }
  printExpression(e->operand2);
  printf(")");
}

void showExpSet(char *name, int size, Expression expset[]) {
  int i;
  printf("%s = [\n", name);
  for (i=0; i + 1 < size; i++) {
    printf(" ");
    printExpression(expset[i]);
    printf(" ;\n");
  }
  printf(" ");
  printExpression(expset[i]);
  printf("\n]\n");
}

int evaluateExpression(Expression e) {
  switch(e->operator) {
  case CONSTANT:
    return (e->atom==TRUE ? 1 : 0);
  case IDENTIFIER:
    return model[e->atom];
  case EQUIV:
    return (evaluateExpression(e->operand1) ==
         evaluateExpression(e->operand2));
  case IMPLIES:
    return ((!evaluateExpression(e->operand1)) ||
         evaluateExpression(e->operand2));
  case AND:
    return (evaluateExpression(e->operand1) &&
         evaluateExpression(e->operand2));
  case OR:
    return (evaluateExpression(e->operand1) ||
         evaluateExpression(e->operand2));
  case NEG:
```

```c
    return (!evaluateExpression(e->operand1));
  }
  /* we should never get here */
  printf("Houston, we've got a problem\n");
  return 0;
}

int evaluateExpressionSet(int size, Expression expset[]) {
  int i;
  for (i=0; i < size; i++) {
    if (evaluateExpression(expset[i]) == 0) {
      return 0;
    }
  }
  return 1;
}

int nextchar(int skipspaces) {
  do {
    curchar = getchar();
    colnr++;
    if (curchar == '\n') {
      linenr++;
      colnr = 0;
    }
  } while ((skipspaces) &&
         ((curchar == ' ') || (curchar == '\t') || (curchar == '\n')));
  curchar = tolower(curchar);
  return curchar;
}

void showLocation() {
  printf("line %d (column %d): ", linenr, colnr);
}

void matchFailure(char *str) {
  showLocation();
  printf("parsing failed, expected '%s' ", str);
  printf("[failed on character '");
  if (curchar > ' ') putchar(curchar);
  else if (curchar == EOF) printf("EOF");
  else printf("chr(%d)", (int)(curchar));
  printf("'].\n");
  exit(EXIT_FAILURE);
}

void match(char *str) {
  int i;
  if (curchar != str[0]) matchFailure(str);
  for (i=1; str[i] != '\0'; i++) {
```

```c
    if (nextchar(FALSE) != str[i]) matchFailure(str);
  }
  nextchar(TRUE);
}

void matchIdentifier(char *ident) {
  match(ident);
  if (isalnum(curchar)) matchFailure(ident);
}

Expression parseEquivalence(); /* forward decl. (mutual recursion) */

Expression parseAtom() {
  char id[MAXIDENTNAMELENGTH];
  int i=0;

  /* identifier/true/false */
  if (!isalpha(curchar)) {
    showLocation();
    printf("parse error, expected false, true, identifier or
        (expression)\n");
    exit(EXIT_FAILURE);
  }

  while ((i < MAXIDENTNAMELENGTH) && (isalnum(curchar))) {
    id[i++] = curchar;
    nextchar(FALSE);
  }
  id[i] = '\0';
  if (i > MAXIDENTNAMELENGTH) {
    printf("Error: identifier (%s..) too long ", id);
    printf("(maximum length is %d characters)\n", MAXIDENTNAMELENGTH);
    exit(EXIT_FAILURE);
  }
  id[i] = '\0';
  if ((curchar == ' ') || (curchar == '\t') || (curchar == '\n')) {
    nextchar(TRUE);
  }
  return makeIdentifier(id);
}

Expression parseTerm() {
  if (curchar == '!') {
    Expression e;
    match("!");
    e = parseTerm();
    return makeNegation(e);
  }
  if (curchar == '(') {
    Expression e;
```

```
    match("(");
    e = parseEquivalence();
    match(")");
    return e;
  }
  return parseAtom();
}

Expression parseConjunction() {
  Expression e0;
  e0 = parseTerm();
  if (curchar == '*') {
    Expression e1;
    match("*");
    e1 = parseTerm();
    return makeBinaryExpr(AND, e0, e1);
  }
  return e0;
}

Expression parseDisjunction() {
  Expression e0;
  e0 = parseConjunction();
  if (curchar == '+') {
    Expression e1;
    match("+");
    e1 = parseConjunction();
    return makeBinaryExpr(OR, e0, e1);
  }
  return e0;
}

Expression parseImplication() {
  Expression e0;
  e0 = parseDisjunction();
  if (curchar == '=') {
    Expression e1;
    match("=>");
    e1 = parseImplication();
    return makeBinaryExpr(IMPLIES, e0, e1);
  }
  return e0;
}

Expression parseEquivalence() {
  Expression e0;
  e0 = parseImplication();
  if (curchar == '<') {
    Expression e1;
    match("<=>");
```

```
    e1 = parseEquivalence();
    return makeBinaryExpr(EQUIV, e0, e1);
  }
  return e0;
}

int parseSentences(Expression expset[]) {
  int numberOfSentences = 0;
  Expression e;
  e = parseEquivalence();
  expset[numberOfSentences] = e;
  numberOfSentences++;
  while (curchar == ';') {
    match(";");
    e = parseEquivalence();
    expset[numberOfSentences] = e;
    numberOfSentences++;
  }
  return numberOfSentences;
}

int parseSentenceSet(char *setname, Expression expset[]) {
  int numberOfSentences;
  matchIdentifier(setname);
  match("=");
  match("[");
  numberOfSentences = parseSentences(expset);
  match("]");
  return numberOfSentences;
}

void parseInput() {
  nextchar(TRUE);
  kbSize = parseSentenceSet("kb", kb);
  inferSize = parseSentenceSet("infer", infer);
}

void showBoolean(int val) {
  if (val == 0) printf ("false");
  else printf("true");
}

void showModel(int modelSize) {
  int i;
  printf("[");
  for (i=0; i < modelSize; i++) {
    if (i > 0) printf(",");
    printf("%s=", identifiers[i]);
    showBoolean(model[i]);
  }
```

```c
  printf("]\n");
}

void evaluateRandomModel(int modelSize) {
  int i;
  /* make a random assignment/model */
  for (i=0; i < modelSize; i++) {
    model[i] = random()%2; /* 0 or 1 (False or True) */
  }

  /* print model */
  printf("Random chosen model: ");
  showModel(modelSize);

  /* evaluate KB */
  printf(" KB evaluates to ");
  showBoolean(evaluateExpressionSet(kbSize, kb));
  printf("\n");

  /* evaluate INFER */
  printf(" INFER evaluates to ");
  showBoolean(evaluateExpressionSet(inferSize, infer));
  printf("\n");
}

/*** You should not need to change any code above this line ****/

void addOne(int modelSize){
  if (modelSize > 0){
    model[modelSize - 1] = (model[modelSize - 1] + 1) % 2;
    if (!model[modelSize - 1])
      addOne(modelSize - 1);
  }
}

void nextModel(int modelSize){
  addOne(modelSize);
  // showModel(modelSize);
}

int checkAllModels(int modelSize) {
  /* return 1 if KB entails INFER, otherwise 0 */
  int nModel = 1;
  inferred = 1;

  for (int i = 0; i < modelSize; i++)
    nModel *= 2;

  for (int i = 0; i < modelSize; i++)
    model[i] = 0;
```

```
  for (int i = 0; i < nModel; i++){
    if (evaluateExpressionSet(kbSize, kb) &&
         !evaluateExpressionSet(inferSize, infer)){
      inferred = 0;
      break;
    }
    nextModel(modelSize);
  }
  return inferred;
}


int main(int argc, char *argv[]) {
  parseInput();

  showExpSet("KB", kbSize, kb);
  printf("\n");
  showExpSet("INFER", inferSize, infer);
  printf("\n");

  // evaluateRandomModel(cntidents);
  //printf("\n");

  if (checkAllModels(cntidents)) {
    printf("KB entails INFER\n");
  } else {
    printf("KB does not entail INFER:\n");
    printf("Counter example: ");
    showModel(cntidents);
  }

  return 0;
}
```

resolution.c

Listing 19: resolution.c

```
#include <stdio.h>
#include <stdlib.h>

/* Assumption: the propositional symbols are a, b, .., z.
 * So, each propositional formula contains at most 26
 * different variables. Sets of variables can therefore
 * be represented by 26 bits, hence an integer (bitstring
 * of 32 bits) suffices.
 */

#define FALSE 0
```

```c
#define TRUE 1
#define CLAUSEMAXSIZE 1024

typedef unsigned int bitset;

typedef struct clause {
  bitset positive; /* bitset of positive symbols in clause */
  bitset negative; /* bitset of negative symbols in clause */
  int parents[2];
} clause;

typedef struct clauseSet {
  int size;     /* number of clauses in set                */
  int allocated; /* maximum number of clauses (allocated space) */
  clause *clauses;
} clauseSet;

/******** ADT for clauses ***************************************/

int isEmptyClause(clause c) {
  /* returns TRUE if and only if the clauses c is empty */
  return ((c.positive == 0) && (c.negative == 0) ? TRUE : FALSE);
}

void makeEmptyClause(clause *c) {
  /* makes c the empty clause (i.e. false) */
  c->positive = 0;
  c->negative = 0;
}

int expectLetter(char c) {
  /* helper routine for parsing clause strings in makeClause() */
  if ((c < 'a') || (c > 'z')) {
    fprintf(stderr, "Syntax error: expected a letter ('a'..'z')\n");
    exit(EXIT_FAILURE);
  }
  return c - 'a';
}

void makeClause(clause *c, char *cstr) {
  /* Converts the string cstr into a clause.
   * For example, the clause {a,b,~c} is represented
   * by the string "a,b,~c".
   */
  int n, idx = 0;
  int pow2[] = {1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,
                32768,65536,131072,262144,524288,1048576,2097152,4194304,
                8388608,16777216,33554432};
  makeEmptyClause(c);
  while (cstr[idx] != '\0') {
```

```
    if (cstr[idx] == '~') {
      /* negative literal */
      idx++;
      n = expectLetter(cstr[idx++]);
      c->negative |= pow2[n];
    } else {
      /* positive literal */
      n = expectLetter(cstr[idx++]);
      c->positive |= pow2[n];
    }
    if (cstr[idx] == ',') {
      idx++;
      if (cstr[idx] == '\0') {
        fprintf(stderr, "Syntax error: truncated clause\n");
        exit(EXIT_FAILURE);
      }
    }
  }
}

int areEqualClauses(clause a, clause b) {
  /* returns TRUE if and only if the clauses a and b are the same */
  if ((a.positive == b.positive) && (a.negative == b.negative)) {
    return TRUE;
  }
  return FALSE;
}

void printClause(clause c) {
  /* prints clause c on standard output */
  int i, mask, trueflag, comma=0;
  printf("[");
  trueflag = 0;
  for (i=0,mask=1; i < 26; i++,mask*=2) {
    int cnt = 0;
    if (c.positive & mask) {
      if (comma) putchar(',');
      putchar('a'+i);
      comma = 1;
      cnt++;
    }
    if (c.negative & mask) {
      if (comma) putchar(',');
      printf("~%c", 'a'+i);
      comma = 1;
      cnt++;
    }
    trueflag |= (cnt == 2);
  }
  printf("]");
```

```
  if (trueflag) {
    printf("=TRUE");
  } else {
    if (isEmptyClause(c)) {
      printf("=FALSE");
    }
  }
}

void printlnClause(clause c) {
  /* prints clause c followed by a newline on standard output */
  printClause(c);
  putchar('\n');
}

/******** ADT for clauseSets ************************************/

void freeClauseSet(clauseSet set) {
  /* releases the memory allocated for set */
  free(set.clauses);
}

void makeEmptyClauseSet(clauseSet *set) {
  /* makes an empty clause set */
  set->size = 0;
  set->allocated = 0;
  set->clauses = NULL;
}

int isEmptyClauseSet(clauseSet s) {
  /* returns TRUE if and only if s is and empty set of clauses */
  return (s.size == 0 ? TRUE : FALSE);
}

int findIndexOfClause(clause c, clauseSet s) {
  /* returns index of clause c in set s, or -1 if c is not found */
  int i = 0;
  while (i < s.size && !areEqualClauses(s.clauses[i], c)) {
    i++;
  }
  return (i < s.size ? i : -1);
}

int isElementOfClauseSet(clause c, clauseSet s) {
  /* returns TRUE if and only if c is in set s */
  return (findIndexOfClause(c, s) == -1 ? FALSE : TRUE);
}

int containsEmptyClause(clauseSet s) {
  /* returns TRUE if and only if the set s contains the empty clause */
```

```c
  clause empty;
  makeEmptyClause(&empty);
  return isElementOfClauseSet(empty, s);
}

int isClauseSubset(clauseSet a, clauseSet b) {
  /* returns TRUE if and only if a is a subset of b */
  int i;
  if (b.size < a.size) {
    return FALSE;
  }
  for (i=0; i < a.size; i++) {
    if (!isElementOfClauseSet(a.clauses[i], b)) {
      return FALSE;
    }
  }
  return TRUE;
}

void insertInClauseSet(clause c, clauseSet *s) {
  /* inserts clause s in set s */
  if (isElementOfClauseSet(c, *s)) {
    return;  /* clause was already in set */
  }
  if (s->size == s->allocated) {
    /* reallocation needed. */
    s->allocated += 128;
    s->clauses = realloc(s->clauses, s->allocated*sizeof(clause));
  }
  s->clauses[s->size++] = c;
}

void unionOfClauseSets(clauseSet *a, clauseSet b) {
  /* implements: a = union(a,b) */
  int i;
  for (i=0; i < b.size; i++) {
    insertInClauseSet(b.clauses[i], a);
  }
}

void crossClauses(clause a, clause b, clauseSet *rsv) {
  /* returns in rsv the set of clauses that are produced by
   * resolving the positive literals of a with the negative literals
   * of b. Note that rsv must be an empty set, before
   * calling this function.
   */
  int crossing = a.positive & b.negative;
  int mask = 1;
  while (crossing != 0) {
    if (crossing & mask) {
```

```
      clause c;
      c.positive = (a.positive | b.positive) & (~mask);
      c.negative = (a.negative | b.negative) & (~mask);
      insertInClauseSet(c, rsv);
    }
    crossing &= ~mask;
    mask *= 2;
  }
}

void printClauseSet(clauseSet s) {
  /* prints set of clauses s on standard output */
  int i;
  printf("{");
  if (s.size > 0) {
    printClause(s.clauses[0]);
    for (i=1; i < s.size; i++) {
      printf(", ");
      printClause(s.clauses[i]);
    }
  }
  printf("}");
}

void printlnClauseSet(clauseSet s) {
  /* prints set of clauses s followed by a newline on standard output */
  printClauseSet(s);
  putchar('\n');
}

/******** Main program ****************************************/

void resolveClauses(clause a, clause b, clauseSet *rsv) {
  /* returns the resolvents of the clauses a and b in the set rsv */
  makeEmptyClauseSet(rsv);
  crossClauses(a, b, rsv);
  crossClauses(b, a, rsv);
}

void setParents(clauseSet *set, int parent1, int parent2){
  for (int i = 0; i < set->size; i++){
    set->clauses[i].parents[0] = parent1;
    set->clauses[i].parents[1] = parent2;
  }

}

void resolution(clauseSet *kb) {
  /* Extends the kb with rules that can be inferred by resolution.
   * The function returns, as soon as it inferred the empty
```

```c
   * clause (i.e. false). The function also returns, if all possible
   * resolvents have been computed.
   */
  while (!containsEmptyClause(*kb)) {
    int i, j;
    clauseSet inferred;
    makeEmptyClauseSet(&inferred);
    for (i=0; i < kb->size; i++) {
      for (j=i+1; j < kb->size; j++) {
        clauseSet resolvents;
        resolveClauses(kb->clauses[i], kb->clauses[j], &resolvents);

        setParents(&resolvents, i, j);

        unionOfClauseSets(&inferred, resolvents);
        freeClauseSet(resolvents);
      }
    }
    if (isClauseSubset(inferred, *kb)) {
      break; /* No new clauses found */
    }
    unionOfClauseSets(kb, inferred);
    freeClauseSet(inferred);
  }
}

char* readClause(char *c){
  scanf("[");
  scanf("%[^]]", c);
  scanf("]");
  return c;
}

void init(clauseSet *s) {
  clause c;
  char *clauseString;
  char ch;

  clauseString = malloc(sizeof(char) * CLAUSEMAXSIZE);
  makeEmptyClauseSet(s);

  scanf("KB=[");
  ch = getchar();
  while (ch == ',' || ch == '['){
    readClause(clauseString);
    makeClause(&c, clauseString);
    insertInClauseSet(c, s);
    ch = getchar();
  }

```

```
  /* Set empty parents */
  setParents(s, -1, -1);
  free(clauseString);

}

void recursivePrintProof(int idx, clauseSet s) {
  int *parentsIdx = s.clauses[idx].parents;
  if (parentsIdx[0] != -1 && parentsIdx[1] != -1){
    recursivePrintProof(parentsIdx[0], s);
    recursivePrintProof(parentsIdx[1], s);

    printClause(s.clauses[idx]);
    printf(" is inferred from ");
    printClause(s.clauses[parentsIdx[0]]);
    printf(" and ");
    printClause(s.clauses[parentsIdx[1]]);
    printf(".\n");
  }
}

void printProof(clauseSet s) {
  int idx;
  clause empty;
  makeEmptyClause(&empty);
  idx = findIndexOfClause(empty, s);
  recursivePrintProof(idx, s);
}

int main(int argc, char *argv[]) {
  clauseSet kb;
  init(&kb);
  printf("KB=");
  printlnClauseSet(kb);
  resolution(&kb);
  printf("KB after resolution=");
  printlnClauseSet(kb);
  if (containsEmptyClause(kb)) {
    printf("Resolution proof completed.\n");
    printf("\nProof:\n");
    printProof(kb);
  } else {
    printf("Resolution proof failed.\n");
  }
  freeClauseSet(kb);
  return EXIT_SUCCESS;
}
```

arith.pl

Listing 20: arith.pl

```
% isnumber(X) is true if X is a isnumber

isnumber(0).
isnumber(s(X)) :- isnumber(X).

% plus(X,Y,Z) is true if X + Y = Z

plus(0,X,X) :- isnumber(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

% minus(X,Y,Z) is true if X - Y =Z

minus(X,0,X) :- isnumber(X).
minus(s(X),s(Y),Z) :- minus(X,Y,Z).

% times(X,Y,Z) is true if X * Y = Z

times(X,0,0) :- isnumber(X).
times(X,s(Y),Z) :- times(X,Y,Z1), plus(X,Z1,Z).

% pow(X,Y,Z) is true if X^Y = Z

pow(X,0,s(0)) :- isnumber(X).
pow(X,s(Y),Z) :- pow(X,Y,Z1), times(X,Z1,Z).

% Example queries:
% Isnumbers are represented as successors of 0. For example, 2 is
    s(s(0)).
% 2+2=4 is plus(s(s(0)), s(s(0)), s(s(s(s(0)))))
% 3*2=? is times(s(s(s(0))), s(s(0)), X)

% even(x) is true if x is an even number
even(0).
even(s(X)) :- odd(X).

% even(x) is true if x is an odd number
odd(0) :- false.
odd(s(X)) :- even(X).

% div2(N,D) is true if D = N / 2
div2(0,0).
div2(s(0), 0).
div2(s(s(N)),s(D)) :- div2(N, D).

% divi2(N,D) is true if D = N / 2
divi2(N,D) :- times(D,s(s(0)),N).
```

```
divi2(s(N),D) :- times(D,s(s(0)),N).

% log(X,B,N) is true if B^N = X
log(X,B,N) :- pow(B,N,X).

% fib(X,Y) is true if fib(X) = Y
fib(0,0).
fib(s(0),s(0)).
fib(s(s(X)),Y) :- fib(s(X),F1), fib(X,F2), plus(F1,F2,Y).

% power(X,N,Y) is true if X^N = Y
power(X,0,s(0)).
power(X,s(s(0)),Y) :- times(X, X, Y).
power(X,N,Y) :- even(N), div2(N, D1), power(X,s(s(0)),P1),
    power(P1,D1,Y).
power(X,s(N),Y) :- odd(s(N)), power(X,N,P1), times(X,P1,Y).
```

list.pl

Listing 21: list.pl

```
len([],0).
len([H|T],N) :- len(T,N1), N is N1+1.

member(X,[]) :- false.
member(X,[H|T]) :- X = H; member(X,T).

concat([HL|TL],[HX|TX],Y) :- HL = HX, concat(TL,TX,Y).
concat([HL|TL],[],[HY|TY]) :- HL = HY, concat(TL,[],TY).
concat([],[],[]).

reverse(L,R) :- reverse(L,[],R).
reverse([],R,R).
reverse([H|T],L,R) :- concat(L2,[H],L), reverse(T,L2,R).

palindrome(L) :- reverse(L,L).
```

maze.pl

Listing 22: maze.pl

```
% predicate north(x,y) is true when x is north of y
north(m,i).
north(j,f).
north(f,b).
north(k,g).
north(g,c).
north(p,l).
north(l,h).
```

```
north(h,d).

% predicate south(x,y) is true when x is south of y
south(X,Y) :- north(Y,X).

% predicate west(x,y) is true when x is west of y
west(n,o).
west(m,n).
west(j,k).
west(e,f).
west(a,b).
west(c,d).

% predicate east(x,y) is true when x is east of y
east(X,Y) :- west(Y,X).

% List operations
member(X,[]) :- false.
member(X,[H|T]) :- X = H; member(X,T).

concat([HL|TL],[HX|TX],Y) :- HL = HX, concat(TL,TX,Y).
concat([HL|TL],[],[HY|TY]) :- HL = HY, concat(TL,[],TY).
concat([],[],[]).

% path finding
path(X,X,L).
path(X,Y) :- path(X,Y,[]).
path(X,Y,Path) :- east(E,X), not(member(E,Path)),
    concat(NewPath,[E],Path), path(E,Y,NewPath);
                              north(N,X), not(member(N,Path)),
                                  concat(NewPath,[N],Path),
                                  path(N,Y,NewPath);
                              south(S,X), not(member(S,Path)),
                                  concat(NewPath,[S],Path),
                                  path(S,Y,NewPath);
                              west(W,X), not(member(W,Path)),
                                  concat(NewPath,[W],Path),
                                  path(W,Y,NewPath).
```