Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Macean Marius
Group: 30433
Email: marius.macean@gmail.com

Lab. Prof. dr. eng. Adrian Groza
Adrian.Groza@cs.utcluj.ro

Chapter 1

(A_1) Search

The main objectives for this assignment are:

1. To implement 2 searching algorithms (DFS and BFS).
2. To implement one or more new challenging layouts.
3. To test and compare these algorithms on these grids.

The layouts and the comparisons results will be presented in this chapter.

1.1 Choose the algorithms (BFS, DFS)

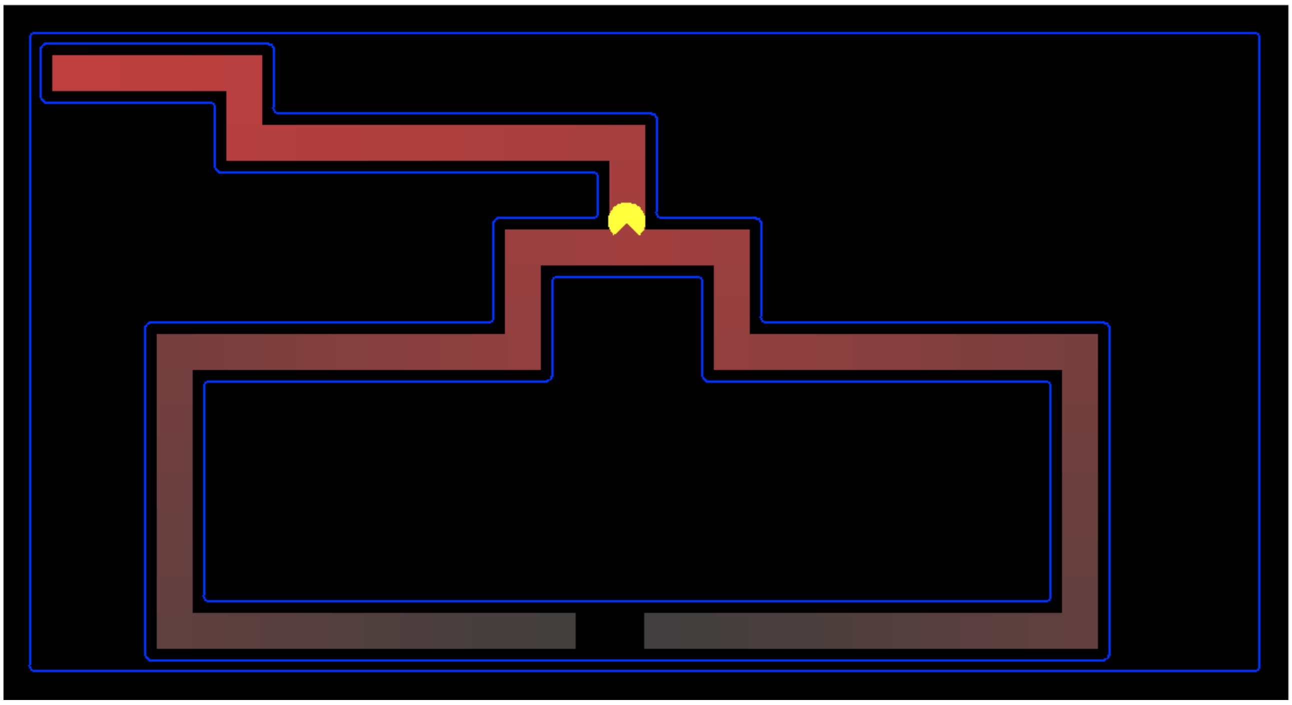
BFS (Breadth First Search). It practically means to visit all the parent nodes then visit all the children nodes. It uses a Queue to store the nodes. This algorithm is better when the goal point closer to the source.

DFS (Depth First Search). It practically means to visit all the children nodes first, until it reaches the end (a leaf) and then goes back to the last parent node. It uses a Stack to store the nodes. This algorithm is better when the goal point is far from the source.

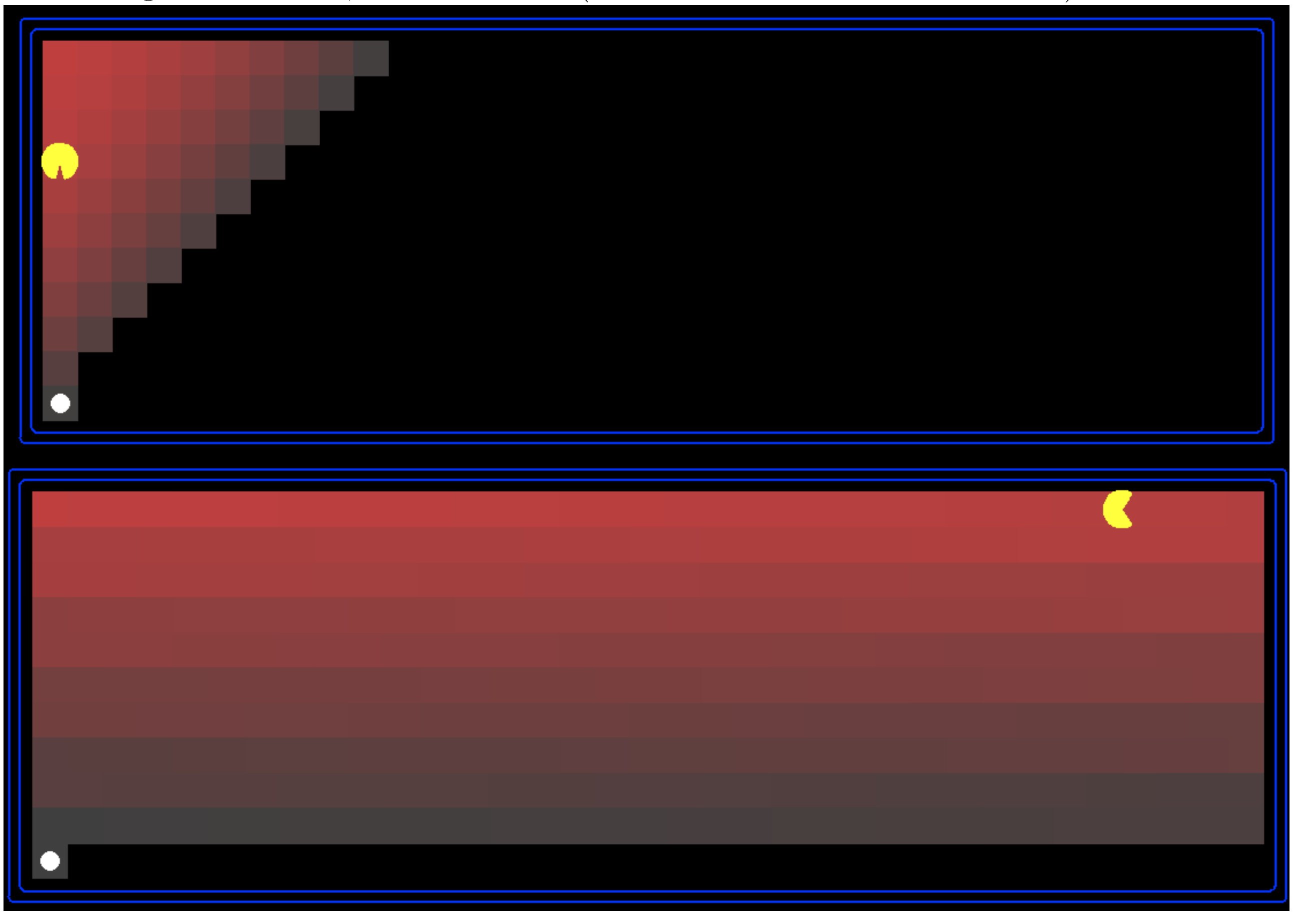
In the average case, the DFS search is faster than the BFS. However, we will find out which of them gives the better results on the test cases presented in the next sections.

1.2 Implement the grids

The first grid: twoPathMaze. The goal position will be (17, 1), witch is at the same distance from the left and right arms of the maze.(BFS is better in this situation)



The first grid: cleanMaze. The goal position will be (1, 1). Here we see how the algorithms are working. First is BFS, second is DFS. (BFS is also better in this situation)



1.3 Let's test them!

1.3.1 The results

Below are the results for the path length, respectively the results for the total number of search nodes:

	DFS	BFS
twoPathMaze	61	55
OtherMaze	350	10

	DFS	BFS
twoPathMaze	61	89
OtherMaze	350	56

1.3.2 The interpretation

In therms of the moves the Pacman has to make, the BFS algorithm is better in moth situation ($55 < 61$, $10 < 350$). However, when it comes to the total number of nodes taken into account when searching, BFS is only better in the second scenario ($56 < 350$, but $89 > 61$).

Chapter 2

(A_2) Logics

The main objectives for this assignment are:

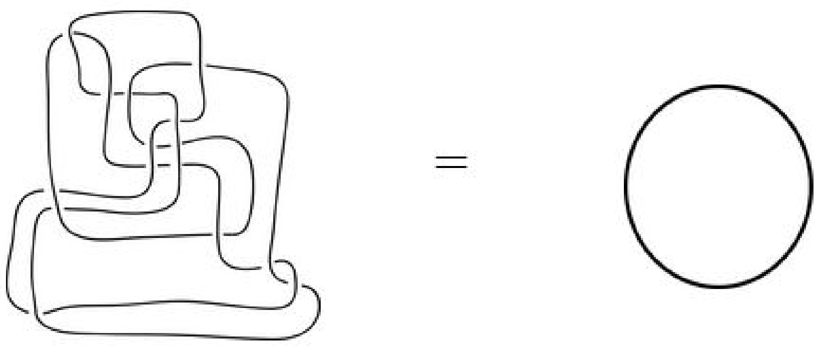
1. To analyze the detecting unknots problem as a logic puzzle.
2. To implement a codification for the crossings within the knot.
3. To understand what we want to prove in order to untie the knot (if possible).
4. To use Prover9 and Mace4 to solve the problem.

2.1 Analyze the problem of KNOTS.

2.1.1 Definitions

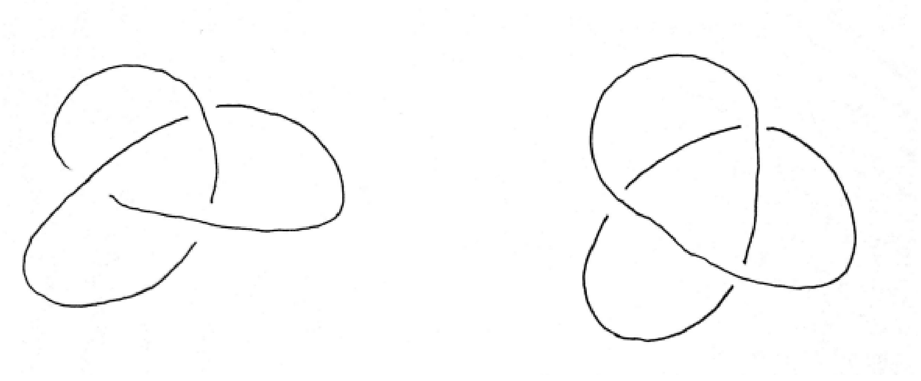
What is a knot? It is fastening made by looping a piece of string. More scientifically: a closed loop without self-intersection.

What is a unknot? In the mathematical theory of knots, the unknot, or trivial knot, is the least knotted of all knots. Therefore, a particular knot is an unknot if and only if it can be untied as the boundary of a disk (as shown in the figure below).



2.1.2 Knot / Unknot

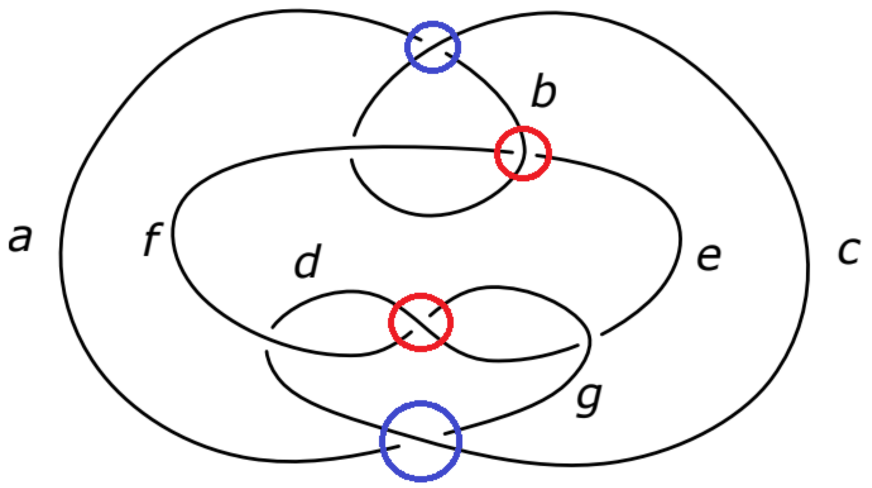
Now let's see the difference between a non-trivial knot (on the left) and an unknot (on the right). The first one cannot be solved in a boundary of a disk, it is untieable. Meanwhile, it can be easily noticed that the second one can be unknotted.



2.2 Codifications

2.2.1 The pieces of the string

Take a string, the one in the picture below, for example. We will denote every piece of it (that holds the part of the string between 2 consecutive crossings, where this certain piece goes UNDER other pieces) with a letter. As you can notice, the piece of string "a" goes between



the 2 blue crossings, which are consecutive crossings where the piece "a" goes beneath other pieces. Same applies to all of them. For instance, the "f" piece goes between the red crossings.

2.2.2 The crossings of the string

Now we define an operation (let's call it "multiplication"), labelled: "*". For example, take the same image presented above. At each crossing, we apply this operation as follows: a first peace that goes beneath a second one multiplied by the second one equals the piece that comes from beneath the second one (which is practically a continuation of the first piece). For the first blue crossing, the operation will be applied like this: a*c = b. In order for this codification to work, we also need some axioms for the operation (the prover needs to know these trivial relations):

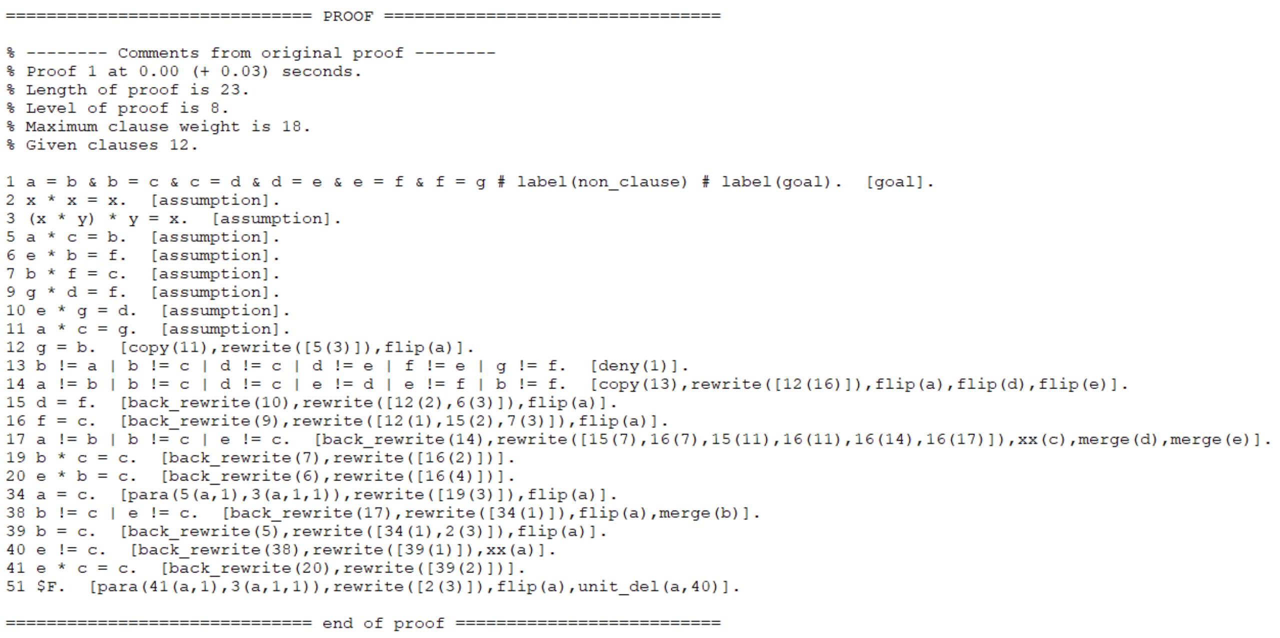
1. x*x = x for all x in Q.
2. For all x, y in Q, there is a unique z in Q such that z*y = x (that is z = x*y).
3. For all x, y, z in Q, we have (x*y)*z = (x*z)*(y*z).

2.3 Solve the Problem

In this section, we will use Prover9 (an automated theorem prover) and Mace4 in order to prove whether or not a knot is an unknot. The Prover9 program will help us check if a the string is an unknot. However, if it is not, the Prover9 will crash and we will have to use the help of Mace4.

2.3.1 Prover9

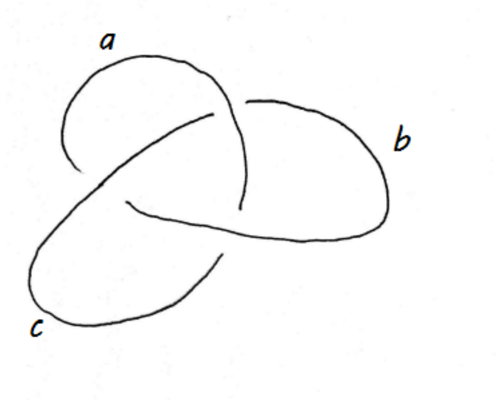
We will use the string/ rope shown in the last section. All the codification (the string separated into pieces, the crossings and the axioms) are translated into equational logic language, that Prover9 will understand. The code is presented in the Appendix at the end of the documentation. You can see the proof in the following picture. Therefore, YES, the knot is an unknot!



```
============================== PROOF =================================

% -------- Comments from original proof --------
% Proof 1 at 0.00 (+ 0.03) seconds.
% Length of proof is 23.
% Level of proof is 8.
% Maximum clause weight is 18.
% Given clauses 12.

1 a = b & b = c & c = d & d = e & e = f & f = g # label(non_clause) # label(goal).  [goal].
2 x * x = x.  [assumption].
3 (x * y) * y = x.  [assumption].
5 a * c = b.  [assumption].
6 e * b = f.  [assumption].
7 b * f = c.  [assumption].
9 g * d = f.  [assumption].
10 e * g = d.  [assumption].
11 a * c = g.  [assumption].
12 g = b.  [copy(11),rewrite([5(3)]),flip(a)].
13 b != a | b != c | d != c | d != e | f != e | g != f.  [deny(1)].
14 a != b | b != c | d != c | e != d | e != f | b != f.  [copy(13),rewrite([12(16)]),flip(a),flip(d),flip(e)].
15 d = f.  [back_rewrite(10),rewrite([12(2),6(3)]),flip(a)].
16 f = c.  [back_rewrite(9),rewrite([12(1),15(2),7(3)]),flip(a)].
17 a != b | b != c | e != c.  [back_rewrite(14),rewrite([15(7),16(7),15(11),16(11),16(14),16(17)]),xx(c),merge(d),merge(e)].
19 b * c = c.  [back_rewrite(7),rewrite([16(2)])].
20 e * b = c.  [back_rewrite(6),rewrite([16(4)])].
34 a = c.  [para(5(a,1),3(a,1,1)),rewrite([19(3)]),flip(a)].
38 b != c | e != c.  [back_rewrite(17),rewrite([34(1)]),flip(a),merge(b)].
39 b = c.  [back_rewrite(5),rewrite([34(1),2(3)]),flip(a)].
40 e != c.  [back_rewrite(38),rewrite([39(1)]),xx(a)].
41 e * c = c.  [back_rewrite(20),rewrite([39(2)])].
51 $F.  [para(41(a,1),3(a,1,1)),rewrite([2(3)]),flip(a),unit_del(a,40)].

============================== end of proof ==========================
```

2.3.2 Mace4

It is possible to solve the same problem (the knot is an unknot) using Mace4. We only have to negate the goal relation and move it to the assumptions part. The result is: "Number of models = 0". That means that the opposite is true. Therefore, we solved the problem again! Let's take, for example, a non-trivial knot, as shown in the following picture. The Mace4 will give us 6 models of it (2 of them presented below), which means what we expected. This is not an unknot!



```
============================== MODEL =================================

interpretation( 3, [number=5, seconds=0], [

        function(a, [ 0 ]),

        function(b, [ 0 ]),

        function(c, [ 0 ]),

        function(*(_,_), [
                 0, 2, 1,
                 2, 1, 0,
                 1, 0, 2 ])
]).

============================== end of model ==========================

============================== MODEL =================================

interpretation( 3, [number=6, seconds=0], [

        function(a, [ 0 ]),

        function(b, [ 1 ]),

        function(c, [ 2 ]),

        function(*(_,_), [
                 0, 2, 1,
                 2, 1, 0,
                 1, 0, 2 ])
]).

============================== end of model ==========================
```

Chapter 3

(A_3) Planning

Appendix A

Original code - Search

```
import util

class SearchProblem:

    def getStartState(self):
        """
        Returns the start state for the search problem.
        """
        util.raiseNotDefined()


    def isGoalState(self, state):
        util.raiseNotDefined()

    def getSuccessors(self, state):

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH,
        Directions.EAST, Directions.WEST]:
          x,y = state
          dx, dy = Actions.directionToVector(action)
          nextx, nexty = int(x + dx), int(y + dy)
          if not self.walls[nextx][nexty]:
            nextState = (nextx, nexty)
            cost = self.costFn(nextState)
            successors.append( ( nextState, action, cost) )

        self._expanded += 1
        if state not in self._visited:
          self._visited[state] = True
          self._visitedlist.append(state)

        return successors

    def expand(self, state):
        util.raiseNotDefined()
    def getActions(self, state):
        util.raiseNotDefined()

    def getActionCost(self, state, action, next_state):
        util.raiseNotDefined()

    def getNextState(self, state, action):
        util.raiseNotDefined()

    def getCostOfActionSequence(self, actions):
        util.raiseNotDefined()


def tinyMazeSearch(problem):
    from game import Directions
    s = Directions.SOUTH
    w = Directions.WEST
    return  [s, s, w, s, w, w, s, w]


def depthFirstSearch(problem):
    visited = {}
    moves = []
    my_stack = util.Stack()
    parents = {}

    start = problem.getStartState()
    my_stack.push((start, 'Undefined', 0))
    visited[start] = 'Undefined'

    if problem.isGoalState(start):
        return moves

    isGoal = False;
    while(my_stack.isEmpty() != True and isGoal != True):
        node = my_stack.pop()
        visited[node[0]] = node[1]
        if problem.isGoalState(node[0]):
            node_sol = node[0]
            isGoal = True
            break
        for next_node in problem.getSuccessors(node[0]):
            if next_node[0] not in visited.keys():
                parents[next_node[0]] = node[0]
                my_stack.push(next_node)

    while(node_sol in parents.keys()):
        node_sol_prev = parents[node_sol]
        moves.insert(0, visited[node_sol])
```

```
            node_sol = node_sol_prev

    return moves


def breadthFirstSearch(problem):

    my_queue = util.Queue()
    my_queue.push(((problem.getStartState()), []))

    visited = []

    while(not my_queue.isEmpty()):
        (state, moves) = my_queue.pop()
        if(problem.isGoalState(state)):
            break

        for next_node in problem.getSuccessors(state):
            if(next_node[0] not in visited):
                visited.append(next_node[0])
                my_queue.push((next_node[0], moves + [next_node[1]]))

    return moves

def nullHeuristic(state, problem=None):
    util.raiseNotDefined()


# Abbreviations
bfs = breadthFirstSearch
dfs = depthFirstSearch
astar = aStarSearch
```

Appendix B

Original code - Logics

UNKNOT:

```
assign(report_stderr, 2).
set(ignore_option_dependencies). % GUI handles dependencies

if(Prover9). % Options for Prover9
  assign(max_seconds, 60).
end_if.

if(Mace4).   % Options for Mace4
  assign(max_seconds, 60).
end_if.

formulas(assumptions).

x * x = x.
(x * y) * y = x.
(x * z) * (y * z) = (x * y) * z.

a * c = b.
e * b = f.
b * f = c.
d * f = c.
g * d = f.
e * g = d.
a * c = g.

end_of_list.

formulas(goals).

(a = b) & (b = c) &
(c = d) & (d = e) &
(e = f) & (f = g).

end_of_list.
```

NON–TRIVIAL–KNOT:
```
assign(report_stderr, 2).
set(ignore_option_dependencies). % GUI handles dependencies

if(Prover9). % Options for Prover9
  assign(max_seconds, 60).
end_if.

if(Mace4).   % Options for Mace4
  assign(domain_size, 3).
  assign(start_size, 3).
  assign(end_size, 3).
  assign(max_models, -1).
  assign(max_seconds, 60).
end_if.

formulas(assumptions).

x * x = x.
(x * y) * y = x.
(x * z) * (y * z) = (x * y) * z.

a * c = b.
c * b = a.
b * a = c.

end_of_list.

formulas(goals).

end_of_list.
```

Bibliography

Intelligent Systems Group