Exercises for Artificial Intelligence I

Dr Sean B Holden, 2010-14

1 Introduction

These notes provide some extra exercises for *Artificial Intelligence I* and, depending on when you download them, their solutions. If you are reading this, are a student, and haven't yet attempted the exercises without the benefit of the solutions, then you are about to commit a monumental waste of time. DO THEM WITHOUT LOOKING AT THE SOLUTIONS FIRST!!!

2 Introduction and Agents

It is notoriously difficult to predict what will be possible in the future, so your answers might well be amusing to you when you find them in twenty years time.

1. If you haven't seen it already, watch the film *A.I. Artificial Intelligence* paying particular attention to the character *"Teddy"*.

2. A large number of subjects are covered in the initial lectures in terms of how they've influenced AI: for example philosophy, mathematics, economics and so on. How do these show up in Teddy's design?

3. What aspects of Teddy are within our current capabilities to design?

4. What aspects of Teddy would you expect to be able to implement within the next fifteen years. How about the next fifty years?

5. Are there aspects of Teddy that you would expect to elude us for one hundred years or more?

6. To what extent does the "natural basic structure" for an agent, as described in the notes, form a useful basis for implementing Teddy's internals? What is missing?

Answer: This is of course very open-ended, and intended primarily as a promoter of discussion. Any and all reasonable contributions are acceptable. For example:

- Amongst other possibilities, the logicist approach and the more recent probabilistic approaches appear in Teddy's ability to reason, and he has a definite human-like mode of action necessary for him to be successful as a toy.
- His speech production (although not the mechanism by which he chooses the words) would be an example, as would some aspects of his locomotion.
- Inside the next fifteen years, perhaps his full physical abilities. In fifty years his ability to recognise and interpret speech.
- At one point in the film Teddy wrestles with the decision as to which of his owners to go to. I expect that this level of cognition may well take in excess of one hundred years further research.
- It actually fits quite well, although it is geared somewhat towards a mode of operation ordered according to `perceive -> update -> act` whereas Teddy clearly has a more flexible behaviour.

3 Search

1. Explain why breadth-first search is optimal if path-cost is a non-decreasing function of node-depth.

   Answer: As path cost is a *function* of node-depth, all nodes at any given depth in the graph have the same path-cost. As the function is increasing, goals at depth d have higher cost than those at d' if $d > d'$. Breadth-first search finds the shallowest, and hence lowest-cost, goal first.

2. In the graph search algorithm, assume a node is taken from the fringe and found *not* to be a goal and *not* to be in `closed`. We then add it to `closed` and add its descendants to `fringe`. Why do we *not* check the descendants first to see if they are in `closed`?

   Answer: One reason is that this is unnecessary for the algorithm to work. In graph search as presented new child nodes are placed in the queue. If they ever get to the head of the queue they are selected and checked for inclusion in the closed list. So ultimately their membership in the closed list will be checked in any case.

   Another is that one can argue that it might in fact increase the computational complexity of the search. If you check all child nodes for inclusion in the closed list at the time they're generated then you potentially perform the check on nodes *that would have remained in the queue without being checked* because a goal is found first. Thus you save on inclusion checks in the closed list at the expense of insertions into the queue.

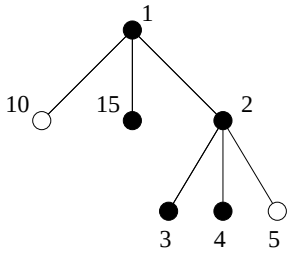
   In particular, we tend to be interested in informed rather than uninformed search procedures. Say we're using a monotonic heuristic. This implies that we search all nodes with f-cost less than the optimal goal's, before selecting an optimal goal. In this case any child nodes generated for a second time and placed in the queue will have a *higher* f-cost than the first time around (as f by definition increases). Thus it's reasonable to expect that in some circumstances a goal will be found before such a node is tested for inclusion in the closed list, potentially saving some unnecessary checks.

   There is potentially an issue here regarding the relative complexity of the inclusion check versus the insertion in the queue. So while this is food for thought, the costs and benefits are not entirely clear. If the queue has constant time insertion and extraction (for example, it's actually a stack or a non-priority queue) all is well. However if it's a priority queue the situation may be more complex.

3. The $A^\star$ algorithm does not perform a goal test on any state *until it has selected it for expansion*. We might consider a slightly different approach: namely, each time a node is expanded check all of its descendants to see if they include a goal.

   Give two reasons why this is a misguided idea, where possible illustrating your answer using a specific example of a search tree for which it would be problematic.

   Answer: The critical idea here is that the proof of the optimality of $A^\star$ search rests on the fact that no suboptimal goal can be *chosen for expansion* before an optimal goal. If you goal test all descendants of a node immediately on expanding it then you stand a chance of finding a suboptimal goal first, thus breaking the optimality of the algorithm. For example:



In this case the numbers are f-costs and the white nodes are goals. If you choose a node according to f-cost, *then* test it to see if it's a goal you expand 1, find it's not a goal and put 10, 15 and 2 in the fringe. You then expand 2, find it's not a goal and put 3, 4, and 5 in the fringe. Then as 3 and 4 have no children you select 5 and find the optimal goal. If you goal-test children as soon as they are generated, then you find the sub-optimal goal at the first expansion. So: this is misguided as it breaks optimality, and also because it will tend to force you to do unnecessary goal tests, which is problematic if the goal test is time-consuming.

4. The f-cost is defined in the usual way as

$$f(n) = p(n) + h(n)$$

where n is any node, p denotes path cost and h denotes the heuristic. An admissible heuristic is one for which, for any n

$$h(n) \leq \text{actual distance from } n \text{ to the goal}$$

and a heuristic is monotonic if for consecutive nodes n and n' it is always the case that

$$f(n') \geq f(n).$$

- Prove that h is monotonic if and only if it obeys the triangle inequality, which states that for any consecutive nodes n and n'

$$h(n) \leq c_{n \to n'} + h(n')$$

where $c_{n \to n'}$ is the cost of moving from n to n'.
Answer: Part I (only if): If h is monotonic then for any n and n' we know that $f(n') \geq f(n)$. Then

$$\begin{aligned} f(n') &= p(n') + h(n') \\ &= p(n) + c_{n \to n'} + h(n') \\ &\geq f(n) = p(n) + h(n). \end{aligned}$$

Cancelling $p(n)$ gives us

$$h(n) \leq c_{n \to n'} + h(n')$$

Part II (if): Now assume that $h(n) \leq c_{n \to n'} + h(n')$. The same argument as for part I can be reversed.

- Prove that if a heuristic is monotonic then it is also admissible.
Answer: This is essentially induction, working backwards from the goal (base case). Let $q(n)$ denote the *actual* distance from n to the goal and let n_G denote the goal. Thus $h(n_G) = 0$, $q(n_G) = 0$ and

$$c_{n \to n_G} = q(n). \tag{1}$$

Also, as h is monotonic $f(n_G) \geq f(n)$. Now

$$\begin{aligned} f(n_G) &= p(n_G) + h(n_G) \\ &= p(n_G) \\ &= p(n) + c_{n \to n_G} \\ &\geq f(n) = p(n) + h(n) \end{aligned}$$

and cancelling the p(n) gives $h(n) \leq q(n)$ using equation 1. Thus, starting from a goal, we have established that for the node n preceding the goal it is true that $h(n) \leq q(n)$. Now for some consecutive nodes n and n' assume that

$$h(n') \leq q(n'). \tag{2}$$

Then

$$\begin{aligned} f(n') &= p(n') + h(n') \\ &= p(n) + c_{n \to n'} + h(n') \\ &\geq f(n) \\ &= p(n) + h(n) \end{aligned}$$

where the inequality comes from the fact that h is monotonic. Cancelling the $p(n)$ then gives $h(n) \leq c_{n \to n'} + h(n')$, and using equation 2 we have $h(n) \leq c_{n \to n'} + q(n') = q(n)$ as required.

- Is the converse true? (That is, are all admissible heuristics also monotonic?) Either prove that this is the case or provide a counterexample.

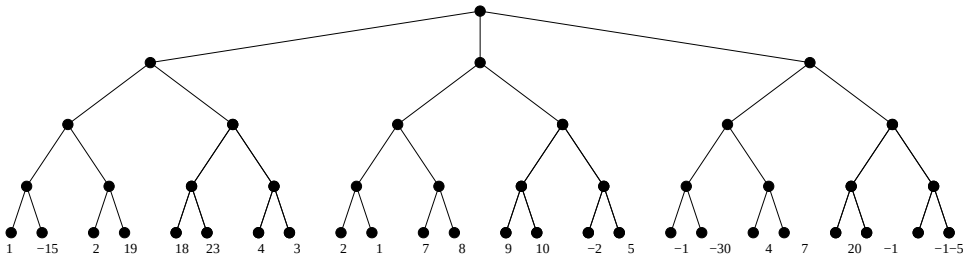
  Answer: No. An example is given in the lecture notes.

5. In RBFS we are replacing f values every time we backtrack to explore the current best alternative. This seems to imply a need to remember the new f values for all the nodes in the path we're discarding, and this in turn suggests a potentially exponential memory requirement. Why is this not the case?

   Answer: We don't actually have to remember the f values for all the nodes in the discarded path—only the one nearest the root.

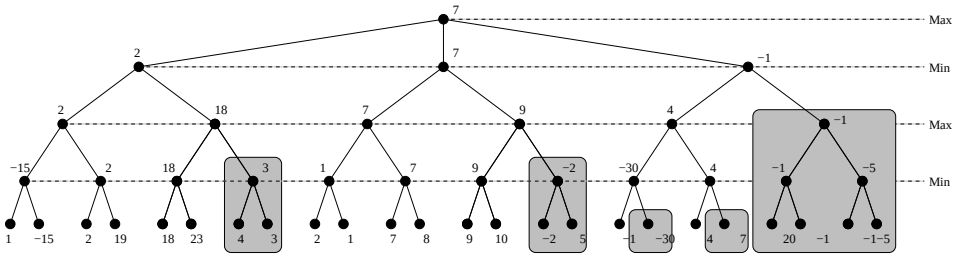
4 Games

1. Consider the following game tree:

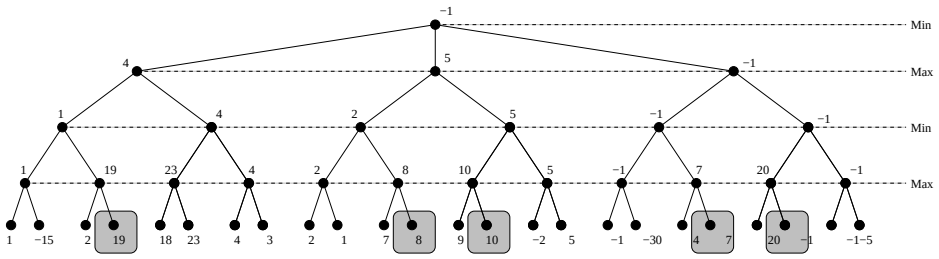


   Large outcomes are beneficial for Max. How is this tree pruned by $\alpha - \beta$ minimax if Max moves first? (That is, Max is the root.) How is it pruned if Min is the root, and therefore moves first?

   Answer: Max to play first, with the greyed out areas denoting pruned subtrees:



Min to play first, with the greyed out areas denoting pruned subtrees:



2. Implement the $\alpha - \beta$ pruning algorithm and use it to verify your answer to the previous problem.
   Answer: The following is a working solution:

```
//-------------------------------------------------------------
// Simple implementation of game-playing with alpha-beta pruning
// to verify the answer to the corresponding exercise in AI 1.
//
// With the usual apologies for the verbosity and horrid uglyness
// resulting from the use of Java. Honestly, I'd much rather use
// ML and do it in about 20 lines, but then everyone just
// complains :-(
//
// Copyright (C) Sean Holden 2010-12.
//-------------------------------------------------------------

import java.lang.Math;
import java.util.LinkedList;

//-------------------------------------------------------------
// There are two kinds of game tree nodes: leaves and non-leaves.
// Pruning is slightly different, depending on which kind we have.
//-------------------------------------------------------------

abstract class GameTreeNode {
    abstract int alphaBetaMax(int alpha, int beta);
    abstract int alphaBetaMin(int alpha, int beta);
}

//-------------------------------------------------------------
// Leaf nodes.
//-------------------------------------------------------------

class Leaf extends GameTreeNode {

private int value;

public Leaf(int leafValue) {
    value = leafValue;
}

public int alphaBetaMax (int alpha, int beta) {
    System.out.println("Leaf node with value: " + value);
    return value;
}

public int alphaBetaMin (int alpha, int beta) {
    System.out.println("Leaf node with value: " + value);
    return value;
}
}

//-------------------------------------------------------------
// Non-leaf nodes. alphaBetaMax and alphaBetaMin are essentially
```

```
// as presented in the lecture notes.
//-----------------------------------------------------------------

class Move extends GameTreeNode {

private LinkedList<GameTreeNode> moves;
private int id = 0;

public Move(int newId) {
    moves = new LinkedList<GameTreeNode>();
    id = newId;
}

public void addMove(GameTreeNode newMove) {
    moves.addLast(newMove);
}

public int alphaBetaMax (int alpha, int beta) {
    System.out.println("");
    System.out.println("Max called on node: " + this.id);
    int value = Integer.MIN_VALUE;
    for (GameTreeNode move : moves) {
        System.out.println("Calling min from node: " + this.id);
        value = Math.max(value, move.alphaBetaMin(alpha, beta));
        System.out.println("Value = " + value);
        if (value > beta) {
            System.out.println("--------- Pruned! Value = " + value);
            return value;
        }
        if (value > alpha) alpha = value;
    }
    System.out.println("Nothing pruned in max. Value = " + value);
    return value;
}

public int alphaBetaMin (int alpha, int beta) {
    System.out.println("");
    System.out.println("Min called on node: " + this.id);
    int value = Integer.MAX_VALUE;
    for (GameTreeNode move : moves) {
        System.out.println("Calling max from node: " + this.id);
        value = Math.min(value, move.alphaBetaMax(alpha, beta));
        System.out.println("Value = " + value);
        if (value < alpha) {
            System.out.println("--------- Pruned! Value = " + value);
            return value;
        }
        if (value < beta) beta = value;
    }
    System.out.println("Nothing pruned in min. Value = " + value);
    return value;
}
}

//-----------------------------------------------------------------
// Construct the example and run it for Max to play first.
//-----------------------------------------------------------------

public class AlphaBeta {

public static GameTreeNode makeExerciseExample() {
    Move three = new Move(3); three.addMove(new Leaf(1)); three.addMove(new Leaf(-15));
    Move four = new Move(4); four.addMove(new Leaf(2)); four.addMove(new Leaf(19));
    Move two = new Move(2); two.addMove(three); two.addMove(four);

    Move six = new Move(6); six.addMove(new Leaf(18)); six.addMove(new Leaf(23));
    Move seven = new Move(7); seven.addMove(new Leaf(4)); seven.addMove(new Leaf(3));
    Move five = new Move(5); five.addMove(six); five.addMove(seven);

    Move one = new Move(1); one.addMove(two); one.addMove(five);

    Move ten = new Move(10); ten.addMove(new Leaf(2)); ten.addMove(new Leaf(1));
    Move eleven = new Move(11); eleven.addMove(new Leaf(7)); eleven.addMove(new Leaf(8));
    Move nine = new Move(9); nine.addMove(ten); nine.addMove(eleven);

    Move thirteen = new Move(13); thirteen.addMove(new Leaf(9)); thirteen.addMove(new Leaf(10));
    Move fourteen = new Move(14); fourteen.addMove(new Leaf(-2)); fourteen.addMove(new Leaf(5));
    Move twelve = new Move(12); twelve.addMove(thirteen); twelve.addMove(fourteen);

    Move eight = new Move(8); eight.addMove(nine); eight.addMove(twelve);

    Move seventeen = new Move(17); seventeen.addMove(new Leaf(-1)); seventeen.addMove(new Leaf(-30));
    Move eighteen = new Move(18); eighteen.addMove(new Leaf(4)); eighteen.addMove(new Leaf(7));
    Move sixteen = new Move(16); sixteen.addMove(seventeen); sixteen.addMove(eighteen);

    Move twenty = new Move(20); twenty.addMove(new Leaf(20)); twenty.addMove(new Leaf(-1));
    Move twentyone = new Move(21); twentyone.addMove(new Leaf(-1)); twentyone.addMove(new Leaf(-5));
    Move nineteen = new Move(19); nineteen.addMove(twenty); nineteen.addMove(twentyone);

    Move fifteen = new Move(15); fifteen.addMove(sixteen); fifteen.addMove(nineteen);
```

```
    Move tree = new Move(0); tree.addMove(one); tree.addMove(eight); tree.addMove(fifteen);

    return tree;
}

public static void main(String[] args) {
    GameTreeNode game = makeExerciseExample();
    game.alphaBetaMax(Integer.MIN_VALUE, Integer.MAX_VALUE);
}

}
```

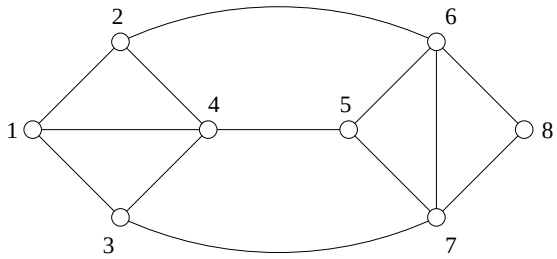
The corresponding output with Max moving first is given in the appendix.

3. Is the minimax approach to playing games optimal against an imperfect opponent? Either prove this is the case or give a counterexample.

   No, it is not optimal. Imagine you have two moves, where move 1 can force a draw but can not win. Move 2 can lead to either a win or a loss, but the loss is only likely to be forced by an excellent opponent. In this case, although minimax chooses move 1, we might in fact want to choose move 2 if we think the opponent is weak.[1]

5 Constraint satisfaction problems

1. Consider the following constraint satisfaction problem:



   We want to colour the nodes using the colours red (R), cyan (C) and black (B) in such a way that connected nodes have different colours.

   - Assume we attempt the assignments $1 = R$, $4 = C$, $5 = R$, $8 = C$, $6 = B$. Explain how *forward checking* operates in this example, and how it detects the need to backtrack.

     Answer: Forward checking proceeds as follows:

	1	2	3	4	5	6	7	8
	RCB	RCB	RCB	RCB	RCB	RCB	RCB	RCB
1=R	=R	CB	CB	CB	RCB	RCB	RCB	RCB
4=C		B	B	=C	RB	RCB	RCB	RCB
5=R		B	B		=R	CB	CB	RCB
8=C		B	B			B	B	=C
6=B		!	B			=B	!	

[1] This example is due to D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," *Artificial Intelligence* 6 (1975), 293–326.

- Will the AC-3 algorithm detect a problem earlier in this case? Explain the operation of the algorithm in this example.
  Answer: Yes, it will detect it at the third assignment. Using lexagraphic ordering on the arcs, that is $i \to j < i' \to j'$ if $i < i'$, or $i = i'$ and $j < j'$ the algorithm proceeds as follows:

Assignment	Arc identified	Domain change	Arcs added to queue
1=R	$2 \to 1$	Remove R from D_2	$1 \to 2, 4 \to 2, 6 \to 2$
	$3 \to 1$	Remove R from D_3	$1 \to 3, 4 \to 3, 7 \to 2$
	$4 \to 1$	Remove R from D_4	$1 \to 4, 2 \to 4, 3 \to 4, 5 \to 4$
4=C	$2 \to 4$	Remove C from D_2	$1 \to 2, 4 \to 2, 6 \to 2$
	$3 \to 4$	Remove C from D_3	$1 \to 3, 4 \to 3, 7 \to 3$
	$5 \to 4$	Remove C from D_5	$4 \to 5, 6 \to 5, 7 \to 5$
	$6 \to 2$	Remove B from D_6	$2 \to 6, 5 \to 6, 7 \to 6, 8 \to 6$
	$7 \to 3$	Remove B from D_7	$3 \to 7, 5 \to 8, 6 \to 8, 8 \to 7$
5=R	$6 \to 5$	Remove R from D_6	$2 \to 6, 5 \to 6, 7 \to 6, 8 \to 6$
	$7 \to 5$	Remove R from D_7	$3 \to 7, 5 \to 7, 6 \to 7, 8 \to 7$
	$7 \to 6$	Remove C from D_7	

  At the final line D_7 becomes empty.

- Implement the AC-3 algorithm and use it to verify your answer to the preceding problem.
  Answer: The following is a working solution. The output is given in the appendix:

```
//-------------------------------------------------------------
// Simple implementation of the AC-3 algorithm to verify the
// answer to the corresponding exercise in AI 1.
//
// Again, it's in Java. Sorry sorry and sorry again to those
// who like code that's elegant and succinct.
// Personally, I REALLY would prefer to use ML...
//
// Copyright (C) Sean Holden 2010-12.
//-------------------------------------------------------------

import java.util.*;

//-------------------------------------------------------------
// CSP domains. Here only for the special case of 3-colouring
// graphs.
//-------------------------------------------------------------

class Domain {
    private enum Colours {R, C, B};
    private Set<Colours> d;

    public Domain() { d = EnumSet.allOf(Colours.class); }

    public String toString() {
String result = "{ ";
if (d.contains(Colours.R)) result += "red ";
if (d.contains(Colours.C)) result += "cyan ";
if (d.contains(Colours.B)) result += "black ";
return result + "}";
    }

    public void red() { d = EnumSet.of(Colours.R); }

    public void cyan() { d = EnumSet.of(Colours.C); }

    public void black() { d = EnumSet.of(Colours.B); }

    // Make a domain consistent with the argument. Return true if any
    // modification is made, false otherwise.
    public boolean removeInconsistencies(Domain right) {
boolean result = false;
for (Colours leftColour : d) {
    boolean consistent = false;
    for (Colours rightColour : right.d) {
if (!(rightColour.equals(leftColour))) {
    consistent = true;
    break;
}
    }
    if (!consistent) {
d.remove(leftColour);
result = true;
    }
```

```
}
return result;
    }
}

//---------------------------------------------------------------
// CSP arcs for the AC-3 algorithm.
//---------------------------------------------------------------

class Arc {
    private int l;
    private int r;

    public Arc(int lid, int rid) {
l = lid;
r = rid;
    }

    public int getLID() { return l; }

    public int getRID() { return r; }

    public String toString() { return (l + " -> " + r); }
}

//---------------------------------------------------------------
// Actual implementation of AC-3. Also constructs the relevant
// example and produces some output.
//---------------------------------------------------------------

public class AC3 {

public static LinkedList<Arc> allArcs(int[][] graph, int size) {
    LinkedList<Arc> arcs = new LinkedList<Arc>();
    for (int i = 0; i < size; i++) {
for  (int j = 0; j < size; j++) {
    if (graph[i][j] == 1) arcs.add(new Arc(i+1,j+1));
}
    }
    return arcs;
}

public static void ac3(Domain[] domains, int[][] graph, int size) {
    LinkedList<Arc> arcs = allArcs(graph,size);
    int iteration = 1;
    while (!arcs.isEmpty()) {
System.out.println("-----------------------------------------");
System.out.println("Iteration = " + iteration);
Arc currentArc = arcs.removeFirst();
System.out.println("  Dealing with arc " + currentArc);
int leftID = currentArc.getLID();
Domain l = domains[leftID-1];
Domain r = domains[currentArc.getRID()-1];
System.out.println("  Domains are " + l + r);
if (l.removeInconsistencies(r)) {
    System.out.println("  Found inconsistency in " + currentArc);
    System.out.println("  New left domain is " + l);
    for (int i = 0; i < size; i++) {
if (graph[i][leftID-1] == 1) {
    Arc newArc = new Arc (i+1,leftID);
    if (!arcs.contains(newArc)) {
arcs.addLast(newArc);
System.out.println("    Adding " + newArc);
    }
}
    }
}
System.out.println("Finished iteration " + iteration++);
System.out.println("New domains: ");
for (int i = 0; i < size; i++) System.out.println(domains[i]);
    }
}

public static void main(String[] args) {
    int size = 8;
    int[][] graph = {{0,1,1,1,0,0,0,0},
     {1,0,0,1,0,1,0,0},
     {1,0,0,1,0,0,1,0},
     {1,1,1,0,1,0,0,0},
     {0,0,0,1,0,1,1,0},
     {0,1,0,0,1,0,1,1},
     {0,0,1,0,1,1,0,1},
     {0,0,0,0,0,1,1,0}};

    Domain[] domains = new Domain[size];
    for (int i = 0; i < size; i++) {
domains[i] = new Domain();
System.out.println(domains[i]);
    }
```

```
        domains[0].red(); ac3(domains, graph, size);
        domains[3].cyan(); ac3(domains, graph, size);
        domains[4].red(); ac3(domains, graph, size);
    }
}
```

6 Knowledge representation and reasoning

1. There have in fact been *two* queries suggested in the notes for obtaining a sequence of actions. The details for

$$\exists a\ \exists s\ .\ \text{Sequence}(a, s_0, s) \wedge \text{Goal}(s)$$

were given on the last slide, but earlier in the notes the format

$$\exists \texttt{actionList}.\texttt{Goal}(\ldots\ \ \texttt{actionList}\ \ldots)$$

was suggested. Explain how this alternative form of query might be made to work.

Answer: Assuming that the theorem prover being used incorporates the usual syntactic sugar for lists, we can use

$$\forall s\ .\ \text{resultSequence}([], s) = s$$
$$\forall s\ \forall a\ \forall \text{as}\ .\ \text{resultSequence}(a :: \text{as}, s) = \text{resultSequence}(\text{as}, \text{result}(a, s))$$

Then

$$\exists \texttt{actionList}\ .\ \text{Goal}(\text{resultSequence}(\texttt{actionList}, s_0)).$$

2. Making correct use of the situation calculus, write the sentences in FOL required to implement the `Shoot` action in Wumpus World.

Answer: This is quite an open ended question with no single correct answer. Elements of a solution might include the following.

Initially, we'll want to know that the agent has an arrow and that the wumpus is alive

$$\texttt{Have}(\texttt{Arrow}, S_0)$$
$$\texttt{Alive}(\texttt{Wumpus}, S_0)$$

and we also need to make sure that we only shoot the arrow if we have it

$$\texttt{Have}(\texttt{Arrow}, s) \Longrightarrow \texttt{Poss}(\texttt{Shoot}, s)$$

and once shot the arrow can not be picked up again

$$\neg\texttt{Available}(\texttt{Arrow}, s)$$

where the latter works in conjunction with the successor-state axiom

$$\begin{aligned}&\texttt{Poss}(a, s) \Longrightarrow \\ &\quad \texttt{Have}(x, \texttt{Result}(a, s)) \iff \\ &\qquad (\texttt{Portable}(x) \wedge \texttt{Available}(x, s) \wedge a = \texttt{Grab}). \\ &\qquad \vee (\texttt{Have}(x, s) \wedge ((x \neq \texttt{Arrow} \wedge a \neq \texttt{Release}) \\ &\qquad \vee (x = \texttt{Arrow} \wedge a \neq \texttt{Shoot}))\end{aligned}$$

We need to try to infer whether or not the wumpus is ahead of us

$$\begin{aligned}\texttt{WumpusAhead}(s) \iff\ & \texttt{Alive}(\texttt{Wumpus}, s) \\ & \wedge \texttt{At}(\texttt{Agent}, [l_1, l_2], s) \\ & \wedge ((\texttt{Facing}(s) = 0 \wedge \texttt{At}(\texttt{Wumpus}, [l_1, l_3], s) \wedge l_3 > l_2) \\ & \vee \texttt{Facing}(s) = 90 \wedge \texttt{At}(\texttt{Wumpus}, [l_3, l_2], s) \wedge l_3 > l_2) \\ & \vee \texttt{Facing}(s) = 180 \wedge \texttt{At}(\texttt{Wumpus}, [l_1, l_3], s) \wedge l_3 < l_2) \\ & \vee \texttt{Facing}(s) = 270 \wedge \texttt{At}(\texttt{Wumpus}, [l_3, l_2], s) \wedge l_3 < l_2))\end{aligned}$$

We need to keep track of whether or not the wumpus is alive

$$\begin{aligned}&\texttt{Poss}(a, s) \implies \\ &\quad \neg\texttt{Alive}(\texttt{Wumpus}, \texttt{Result}(a, s)) \iff \\ &\qquad \neg\texttt{Alive}(\texttt{Wumpus}, s) \\ &\qquad \vee (a = \texttt{Shoot} \wedge \texttt{Have}(\texttt{Arrow}, s) \\ &\qquad \wedge \texttt{WumpusAhead}(s))\end{aligned}$$

3. Download and install a copy of *Prover9* from `www.cs.unm.edu/~mccune/prover9/`. (Hint: if you're Linux-based then you'll probably find it's already packaged. For instance, at the time of writing `yum install prover9.x86_64` works under Fedora 20.)

   Referring to exam question 2003, paper 9, question 8 assume that initially both owner and cat are in the living room. The cat can make its owner move to the kitchen by going to its food bowl in the kitchen and meeowing. It can then of course return to the living room and scratch something valuable.

   Implement sufficient knowledge in the situation calculus to allow an action sequence to be derived allowing the cat to achieve this, and use Prover9 to derive such an action sequence.

   In order to do this you need to know how to extract an answer from the theorem prover. Taking an easy example from the lecture notes:

```
formulas(assumptions).

wife(x,y) <-> (female(x) & married(x,y)).
female(Violet).
married(Violet,Bill).

end_of_list.

formulas(goals).

exists x wife(Violet,x).

end_of_list.
```

   Extracting the value of `x` requires two things: we need to move the goal into the assumptions (which is just like converting $\neg(A \rightarrow B)$ to $A \wedge \neg B$ when negating and converting to clauses) and we need to add a command to the knowledge base to get:

```
formulas(assumptions).
```

```
wife(x,y) <-> (female(x) & married(x,y)).
female(Violet).
married(Violet,Bill).

-wife(Violet,x) # answer(x).

end_of_list.

formulas(goals).
end_of_list.
```

Here, the addition of `# answer(x)` causes the prover to output the value of `x` as part of the solution.

Answer: The following is a working solution:

```
% Use prolog-style variables, so upper case denotes a variable and
% lower case denotes a constant.

assign(max_megs,2000).
set(prolog_style_variables).

%----------------------------------------------------------

formulas(assumptions).

% Axioms allowing us to deal with sequences of actions. The list
% notation is built in to this theorem prover.

result_sequence([],S) = S.
result_sequence([A:Rest],S) = result_sequence(Rest,result(A,S)).

% Uniqueness axioms

cat != owner.
cat != food_bowl.
cat != sofa.
cat != stereo.
cat != scratching_post.
owner != food_bowl.
owner != sofa.
owner != stereo.
owner != scratching_post.
food_bowl != sofa.
food_bowl != stereo.
food_bowl != scratching_post.
sofa != stereo.
sofa != scratching_post.
stereo != scratching_post.

% Uniqueness axioms for the actions "go", "scratch" and "meeow".

go(X) != scratch(Y).
go(X) != meeow.
scratch(X) != meeow.
```

```
go(X) = go(Y) -> X=Y.
scratch(X) = scratch(Y) -> X=Y.

% What do we know in the initial state s0?

s0 != result(A,S).
result(A1,S1) = result(A2,S2) -> A1=A2 & S1=S2.

at(owner,sofa,s0).
at(cat,sofa,s0).

valuable(sofa,s0).
valuable(stereo,s0).

-valuable(scratching_post,s0).
-valuable(food_bowl,s0).

% Axioms describing when actions are possible.

poss(go(X),S).
poss(meeow,S).

at(cat,X,S) & at(owner,food_bowl,S) & valuable(X,S) -> poss(scratch(X),S).

% We need a successor-state axiom for each fluent.
% The fluents are "at" and "valuable".
% The actions are "go", "scratch" and "meeow".

poss(A,S) ->
  (at(X,Y,result(A,S))
     <-> (X=cat & A=go(Y)) |
         (A=meeow & X=owner & Y=food_bowl & at(cat,food_bowl,S)) |
         (X=cat & at(X,Y,S) & (-(exists Z (A=go(Z) & Z!=Y)))) |
         (X=owner & at(X,Y,S))).

poss(A,S) ->
  (valuable(X,result(A,S)) <-> valuable(X,S) & A!=scratch(X)).

% This is the goal.

(S1 != result_sequence(Actions1,s0)) |
(valuable(stereo,S1)) |
(valuable(sofa,result_sequence(Actions2,S1))) # answer([Actions1,Actions2]).

end_of_list.

%----------------------------------------------------------

formulas(goals).
end_of_list.
```

When run it will (eventually) produce the following plan of action for the cat:

```
[[go(food_bowl),meeow,go(stereo),scratch(stereo)],[go(sofa),scratch(sofa)]]
```

7 Planning

1. We've seen how heuristics can be used to speed up the process of searching. Planning has much in common with search. Can you devise any general heuristics that you might expect to speed up the planning process?

   Answer: Quite open ended and any reasonable suggestion is fine. This is discussed at the beginning of AI II; two possibilities are:

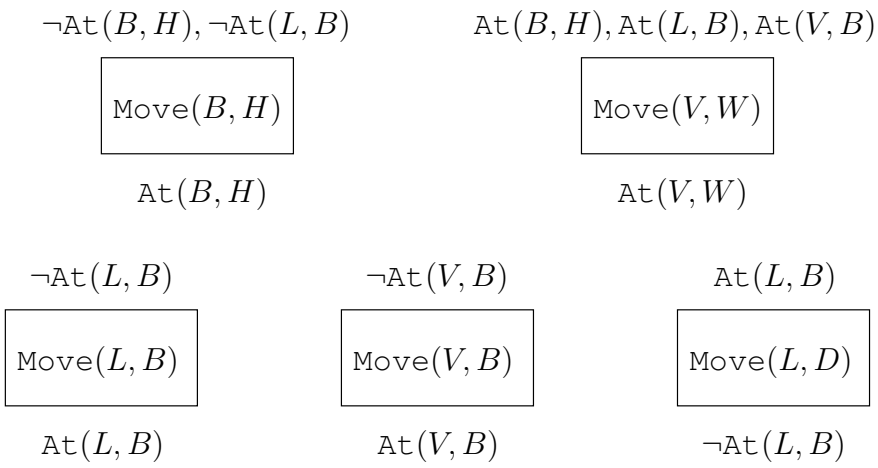
   - $h =$ number of unsatisfied preconditions
   - Prefer preconditions satisfiable in the smallest number of ways.

2. Violet Scroot is the cleverest student at Bibulous College. She has turned up at this term's Big Party, only to find that it is in the home of her arch rival, who has turned her away. She spies in the driveway a large box and a ladder, and hatches a plan to gatecrash by getting in through a second floor window. Party on!

   Here is the planning problem. She needs to move the box to the house, the ladder onto the box, then climb onto the box herself and at that point she can climb the ladder to the window. Using the abbreviations

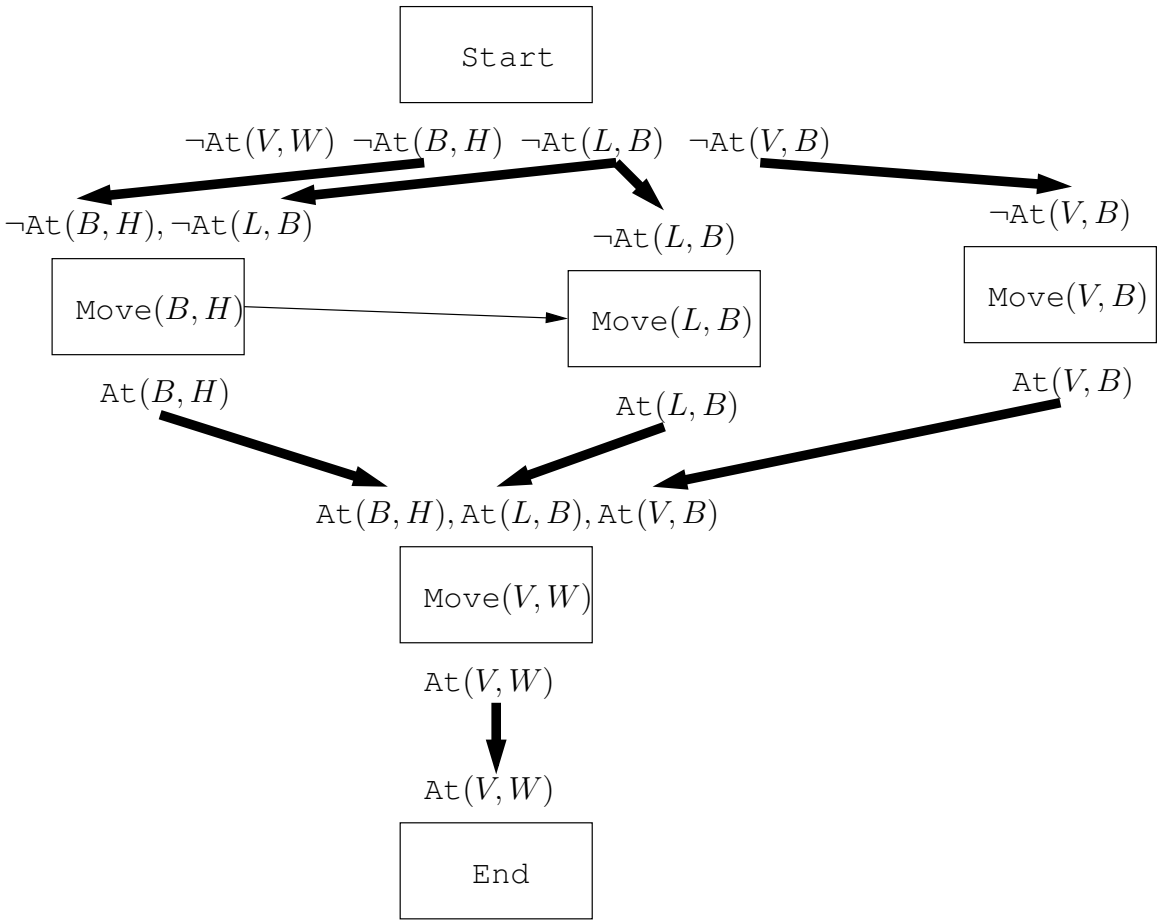
   - B - `Box`
   - L - `Ladder`
   - H - `House`
   - V - `Violet Scroot`
   - W - `Window`
   - D - `Driveway`

   The start state is $\neg\texttt{At}(B, H)$, $\neg\texttt{At}(L, B)$, $\neg\texttt{At}(V, W)$ and $\neg\texttt{At}(V, B)$. The goal is $\texttt{At}(V, W)$. The available actions are

   Construct a solution to the problem using the partial order planning algorithm.

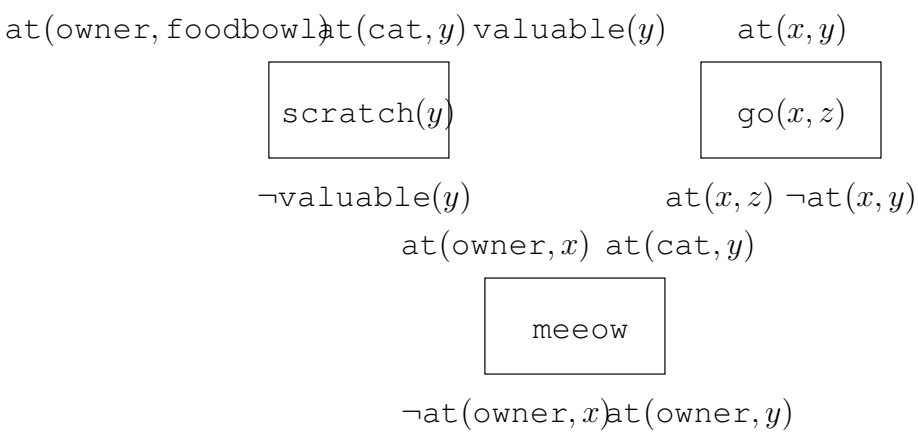
   Answer: This is rather straightforward because there is very little room for manoeuvre. The only mild complication is the need for an extra ordering constraint.

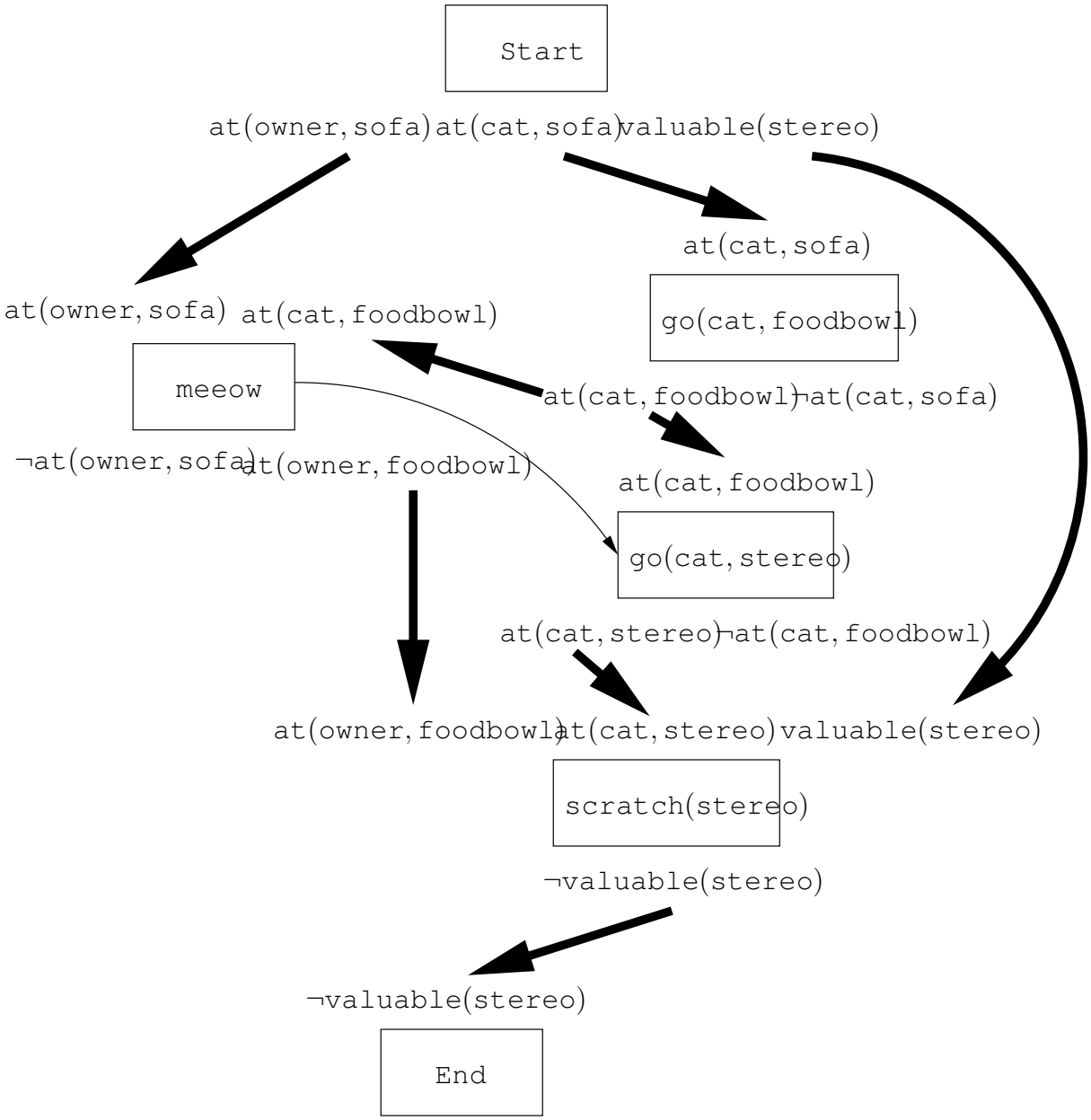


3. Return of the Evil Cat! Consider the problem involving the situation calculus and Prover9 above.

   - Represent this problem in the STRIPS format so that it could be given as input to the partial order planning algorithm.
   - Construct a solution to the problem using the partial order planning algorithm. How many specific plans can be extracted from the result?

   Answer: We can construct a plan from three actions:


at(owner, foodbowl)
at(cat, y)
valuable(y)
at(x, y)
scratch(y)
go(x, z)
¬valuable(y)
at(x, z)
¬at(x, y)
at(owner, x)
at(cat, y)
meeow
¬at(owner, x)
at(owner, y)



8 Learning

1. The purpose of this exercise is to gain some insight into the way in which the parameters of a basic, linear perceptron affect the position and orientation of its decision boundary. Recall that a linear perceptron is based on the function

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. The perceptron decides that a new input $\mathbf{x}$ is in class 1 if $f(\mathbf{x}) \geq 0$ and decides that the input is in class 2 otherwise. The decision boundary is therefore the collection of all points where $f(\mathbf{x}) = 0$.

It's always easy to find n distinct points where $f(\mathbf{x}) = 0$ because for any $\mathbf{w}$ and b we just need to solve

$$\mathbf{w}^T\mathbf{x}' = -b$$

which is easy using

$$\mathbf{x}^T = (\ (-b/w_1)\quad 0\quad \cdots\quad 0\ )$$
$$\mathbf{x}^T = (\ 0\quad (-b/w_2)\quad \cdots\quad 0\ )$$

and so on. If any of the weights is 0 this is problematic but easy to fix. (I leave it as a warm-up exercise to work out how.) Let $\mathbf{x}'$ and $\mathbf{x}''$ be two points where $f(\mathbf{x}') = 0$ and $f(\mathbf{x}'') = 0$. Let's concentrate on the case where $n = 2$. Consider the vector

$$\mathbf{y} = \mathbf{x}' - \mathbf{x}''.$$

Now take any number $a \in \mathbb{R}$ and look at what happens if we evaluate

$$f(\mathbf{x}' + a\mathbf{y}).$$

We obtain

$$\begin{aligned} f(\mathbf{x}' + a\mathbf{y}) &= \mathbf{w}^T(\mathbf{x}' + a\mathbf{y}) + b \\ &= \mathbf{w}^T\mathbf{x}' + a\mathbf{w}^T\mathbf{y} + b \\ &= f(\mathbf{x}') + a\mathbf{w}^T(\mathbf{x}' - \mathbf{x}'') \\ &= a(\mathbf{w}^T\mathbf{x}' - \mathbf{w}^T\mathbf{x}'') \\ &= a(-b - (-b)) \\ &= 0. \end{aligned}$$

This works for *any* value $a \in \mathbb{R}$, and suggests that the decision boundary is a straight line in $\mathbb{R}^2$ as illustrated in figure 1. (Note however that we haven't yet demonstrated that $f(\mathbf{x}) \neq 0$ if $\mathbf{x}$ is not of the form $\mathbf{x} = \mathbf{x}' + a\mathbf{y}$.)

(a) Prove that the weight vector $\mathbf{w}$ is perpendicular to the line described by $\mathbf{x}' + a\mathbf{y}$; that is, the line corresponding to the set

$$\{\mathbf{x} | \mathbf{x} = \mathbf{x}' + a\mathbf{y} \text{ where } a \in \mathbb{R}\}.$$

(Hint: remember that vectors are perpendicular if their inner product is 0.) Note that this tells us that $\mathbf{w}$ describes the orientation of the decision boundary.
Answer: The line described by $\mathbf{x}' + a\mathbf{y}$ is parallel to the vector $\mathbf{x}' - \mathbf{x}''$.

$$\begin{aligned} \mathbf{w}^T(\mathbf{x}' - \mathbf{x}'') &= \mathbf{w}^T\mathbf{x}' - \mathbf{w}^T\mathbf{x}'' \\ &= f(\mathbf{x}') - b - (f(\mathbf{x}'') - b) \\ &= -b - (-b) \text{ (because } f(\mathbf{x}') = f(\mathbf{x}'') = 0 \\ &= 0. \end{aligned}$$

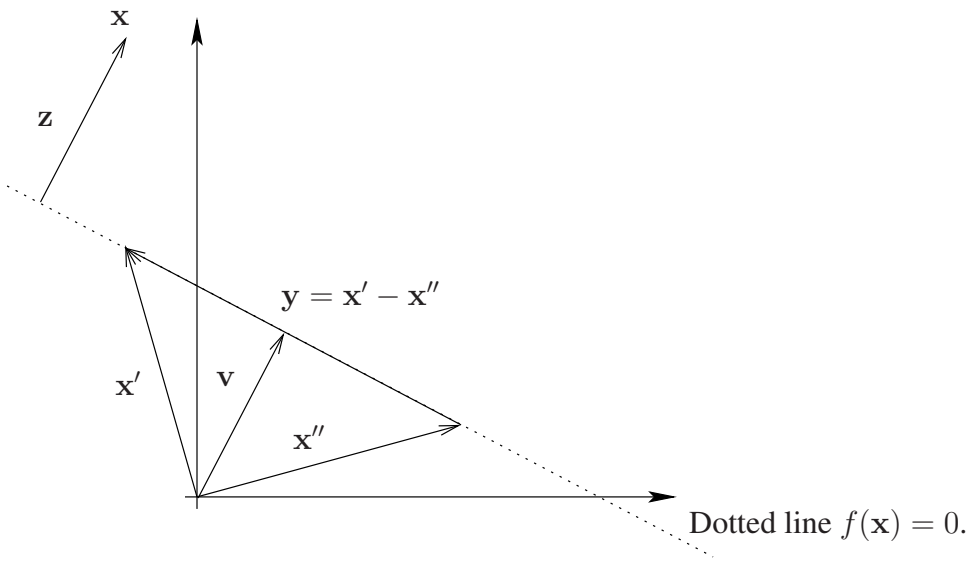


Figure 1: The decision boundary appears to be a straight line.

(b) Let $\mathbf{v}$ be the vector from the origin to the line described by $\mathbf{x}' + a\mathbf{y}$ and perpendicular to it as illustrated in figure 1. Prove that

$$||\mathbf{v}|| = \frac{|b|}{||\mathbf{w}||}.$$

Note that this tells us the following: if $||\mathbf{w}|| = 1$ then $|b|$ tells us the distance from the origin to the decision boundary.

Answer: As $\mathbf{v}$ and $\mathbf{w}$ are both perpendicular to the same line

$$\mathbf{v} = a\mathbf{w}$$

for some constant a. So

$$\mathbf{v}^T\mathbf{v} = ||\mathbf{v}||^2 = a^2\mathbf{w}^T\mathbf{w} = a^2||\mathbf{w}||^2$$

and

$$||\mathbf{v}|| = |a| \times ||\mathbf{w}||. \tag{3}$$

As $\mathbf{v}$ is on the line $f(\mathbf{x}) = 0$ we have

$$f(\mathbf{v}) = \mathbf{w}^T\mathbf{v} + b = 0$$

so

$$\mathbf{w}^T\mathbf{v} = a\mathbf{w}^T\mathbf{w} = a||\mathbf{w}||^2 = -b \tag{4}$$

and using (3) and (4)

$$|a| \times ||\mathbf{w}||^2 = ||\mathbf{v}|| \times ||\mathbf{w}|| = |b|.$$

(c) Let $\mathbf{x}$ be any point *not* on the line described by $\mathbf{x}' + a\mathbf{y}$. Let $\mathbf{z}$ be the vector from the line to $\mathbf{x}$ and perpendicular to the line as illustrated in figure 1. Prove that

$$||\mathbf{z}|| = \frac{|f(\mathbf{x})|}{||\mathbf{w}||}.$$

This tells us that points not on the line do not obey $f(\mathbf{x}) = 0$ and that the value of $f(\mathbf{x})$ tells us the distance from the decision boundary to $\mathbf{x}$.

Answer: As $\mathbf{w}$ and $\mathbf{z}$ are both perpendicular to the same line

$$\mathbf{z} = a\mathbf{w} \text{ for a constant } a$$

so

$$||\mathbf{z}|| = |a| \times ||\mathbf{w}||.$$

Also, the vector $\mathbf{x} - \mathbf{z}$ is on the dotted line so

$$\begin{aligned} f(\mathbf{x} - \mathbf{z}) &= \mathbf{w}^T(\mathbf{x} - \mathbf{z}) + b = \mathbf{w}^T\mathbf{x} + b - \mathbf{w}^T\mathbf{z} \\ &= f(\mathbf{x}) - \mathbf{w}^T\mathbf{z} = f(\mathbf{x}) - a||\mathbf{w}||^2 = 0 \end{aligned}$$

so

$$|f(\mathbf{x})| = |a| \times ||\mathbf{w}||^2 = ||\mathbf{z}|| \times ||\mathbf{w}||.$$

(d) Prove that replacing $\mathbf{w}$ with $\mathbf{w}/||\mathbf{w}||$ and b with $b/||\mathbf{w}||$ does not alter the decision boundary.

Answer: For any $\mathbf{x}$

$$\begin{aligned} f(\mathbf{x}) \geq 0 &\longleftrightarrow \mathbf{w}^T\mathbf{x} + b \geq 0 \\ &\longleftrightarrow c\mathbf{w}^T\mathbf{x} + cb \geq 0 \text{ for any } c > 0. \end{aligned}$$

As

$$c = \frac{1}{||\mathbf{w}||} > 0$$

the suggested replacement works.

2. In the application of neural networks to *pattern classification*—where we wish to assign any input vector $\mathbf{x}$ to membership in a specific *class*—it makes sense to attempt to interpret network outputs as probabilities of class membership.

   For example, in the medical diagnosis scenario presented in the lectures, where we try to map an input $\mathbf{x}$ to either class A (patient has the disease) or class B (patient is free of the disease) it makes sense to use a network with a single output producing values constrained between 0 and 1 such that the output $h(\mathbf{w}; \mathbf{x})$ of a network using weights $\mathbf{w}$ is interpreted as

   $$h(\mathbf{w}; \mathbf{x}) = \Pr(\mathbf{x} \text{ is in class } A).$$

   Clearly we also have

   $$\Pr(\mathbf{x} \text{ is in class } B) = 1 - h(\mathbf{w}; \mathbf{x})$$

   and it follows that training examples should be labelled 1 and 0 for classes A and B respectively.

   Say we have a specific training example $(\mathbf{x}', 0)$. What does it tell us about how to choose a good $\mathbf{w}$? Clearly we might want to choose $\mathbf{w}$ to maximise[2]

   $$\begin{aligned} \Pr(\text{We see the example } (\mathbf{x}', 0)|\mathbf{w}) &= \Pr(\text{We see the label } 0|\mathbf{w}, \mathbf{x}') \times \Pr(\mathbf{x}') \\ &= \left\{1 - h(\mathbf{w}; \mathbf{x}')\right\} \times \Pr(\mathbf{x}') \end{aligned}$$

[2]The basic result in probability theory being used here is that $\Pr(A, B|C) = \Pr(A|B, C)\Pr(B|C)$. You might want at this point to review the relevant notes.

where the second step incorporates the assumption that $\mathbf{x}'$ and $\mathbf{w}$ are independent. This quantity is called the *likelihood* of $\mathbf{w}$. Given an entire training sequence

$$\mathbf{s} = ((\mathbf{x}_1, c_1), (\mathbf{x}_2, c_2), \ldots, (\mathbf{x}_m, c_m))$$

where the labels c_i take values 0 or 1 we can also consider choosing $\mathbf{w}$ to maximise the probability that the entire collection of m input vectors is labelled in the specified manner (the likelihood $\Pr(\mathbf{s}|\mathbf{w})$ of $\mathbf{w}$).

Assuming that the examples in $\mathbf{s}$ are independent, show that in order to achieve this we should choose $\mathbf{w}$ to maximise the expression

$$\sum_{i=1}^{m} c_i \log h(\mathbf{w}; \mathbf{x}_i) + (1 - c_i) \log(1 - h(\mathbf{w}; \mathbf{x}_i)).$$

(Hint: When independence is assumed, $\Pr(A, B) = \Pr(A) \Pr(B)$, and you can maximise an expression equally well by maximising its log.) What does this allow you to conclude about the version of the backpropagation algorithm presented in the lectures?

Answer: For an example $(\mathbf{x}_i, c_i)$ in $\mathbf{s}$ we know that

$$\Pr(c_i|\mathbf{w}, \mathbf{x}_i) = \begin{cases} h(\mathbf{w}; \mathbf{x}_i) & \text{if } c_i = 1 \\ (1 - h(\mathbf{w}; \mathbf{x}_i)) & \text{if } c_i = 0 \end{cases}.$$

This means we can write

$$\Pr((\mathbf{x}_i, c_i)|\mathbf{w}) = \{h(\mathbf{w}; \mathbf{x}_i)\}^{c_i} \{1 - h(\mathbf{w}; \mathbf{x}_i)\}^{1-c_i} \times \Pr(\mathbf{x}_i)$$

and as the examples are independent

$$\Pr(\mathbf{s}|\mathbf{w}) = \prod_{i=1}^{m} \{h(\mathbf{w}; \mathbf{x}_i)\}^{c_i} \{1 - h(\mathbf{w}; \mathbf{x}_i)\}^{1-c_i} \Pr(\mathbf{x}_i).$$

This equation can be maximised by maximising its log, so we maximise

$$\begin{aligned} \log \Pr(\mathbf{s}|\mathbf{w}) &= \sum_{i=1}^{m} \log \left\{ \{h(\mathbf{w}; \mathbf{x}_i)\}^{c_i} \{1 - h(\mathbf{w}; \mathbf{x}_i)\}^{1-c_i} \times \Pr(\mathbf{x}_i) \right\} \\ &= \sum_{i=1}^{m} c_i \log h(\mathbf{w}; \mathbf{x}_i) + (1 - c_i) \log(1 - h(\mathbf{w}; \mathbf{x}_i)) + \sum_{i=1}^{m} \Pr(\mathbf{x}_i) \end{aligned}$$

and as for all i, $\Pr(\mathbf{x}_i)$ is independent of $\mathbf{w}$ we obtain the stated equation.

This allows us to conclude that the version of backpropagation presented in the lectures is not necessarily appropriate for use with two-class classification problems.

3. We now return to the case of regression. As in the previous question, the *likelihood* of a hypothesis h can be thought of as the probability of obtaining a training sequence $\mathbf{s}$ given that h is a perfect mapping from attribute vectors to classifications. Assume that $\mathcal{H}$ contains functions $h : X \to \mathbb{R}$ and examples are labelled using a specific target function $f \in \mathcal{H}$ but corrupted by noise, so

$$\mathbf{s} = ((\mathbf{x}_1, o_1), (\mathbf{x}_2, o_2), \ldots, (\mathbf{x}_m, o_m))$$

and

$$o_i = f(\mathbf{x}_i) + e_i$$

for $i = 1, 2, \ldots, m$ where e_i denotes noise. If the attribute vectors are fixed, and the e_i are independent and identically distributed with the Gaussian distribution

$$p(e_i) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(e_i - \mu)^2}{2\sigma^2}\right)$$

where μ is the noise mean and σ^2 the noise variance, then the likelihood of any hypothesis is

$$p(\mathbf{s}|h) = p((o_1, o_2, \ldots, o_m)|h) = \prod_{i=1}^{m} p(o_i|h)$$

where the last step follows because the e_i are independent. Assume in the following that $\mu = 0$.

(a) Show that the mean of o_i is $f(\mathbf{x}_i)$ and the variance of o_i is σ^2.
Answer: The mean of o_i is

$$\begin{aligned} E(o_i) &= E\left\{f(\mathbf{x}_i) + e_i\right\} \\ &= E\left\{f(\mathbf{x}_i)\right\} + E(e_i) \\ &= f(\mathbf{x}_i). \end{aligned}$$

The variance of o_i is

$$\begin{aligned} E\left\{(o_i - E(o_i))^2\right\} &= E\left\{(o_i - f(\mathbf{x}_i))^2\right\} \\ &= E\left\{(f(\mathbf{x}_i) + e_i - f(\mathbf{x}_i))^2\right\} \\ &= E(e_i^2) \\ &= \sigma^2. \end{aligned}$$

(b) Show that

$$p(o_i|h) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(o_i - h(\mathbf{x}_i))^2}{2\sigma^2}\right).$$

(Hint: what happens to data having a Gaussian density if you linearly transform it?)
Answer:

- Instead of fixing $f(\mathbf{x}_i)$ and assuming it was used, we're introducing an arbitrary h so

$$o_i = h(\mathbf{x}_i) + e_i \tag{5}$$

and as before

$$E(o_i) = h(\mathbf{x}_i)$$

and

$$\text{Var}(o_i) = \sigma^2.$$

- As (5) is a *linear* transformation of e_i, we known o_i must still have a Gaussian distribution.
- The equation for the distribution is therefore the usual equation for a Gaussian with mean $h(\mathbf{x}_i)$ and variance σ^2

$$p(o_i|h) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(o_i - h(\mathbf{x}_i))^2}{2\sigma^2}\right)$$

(c) Show that any hypothesis that *maximises* the likelihood is also one that *minimises* the quantity

$$\sum_{i=1}^{m}(o_i - h(\mathbf{x}_i))^2.$$

Answer:

$$\begin{aligned} h &= \operatorname{argmax} \prod_{i=1}^{m} p(o_i|h) \\ &= \operatorname{argmax} \sum_{i=1}^{m} \log p(o_i|h) \\ &= \operatorname{argmax} \sum_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(o_i - h(\mathbf{x}_i))^2}{2\sigma^2} \\ &= \operatorname{argmax} \sum_{i=1}^{m} -\frac{(o_i - h(\mathbf{x}_i))^2}{2\sigma^2} \\ &= \operatorname{argmin} \sum_{i=1}^{m} (o_i - h(\mathbf{x}_i))^2. \end{aligned}$$

(d) What does this tell you about the specific example of the backpropagation algorithm given in the lectures?

Answer: That it makes the hidden assumption that noise on the examples is Gaussian.

4. The demonstration of the backpropagation algorithm given in the lectures can be improved. In solving the parity problem what we really want to know is the *probability* that an example should be placed in class one, exactly as described above. Probabilities lie in the interval $[0, 1]$, but the output of the network used in the lectures is unbounded.

   - Derive the modification required to the algorithm if the activation function on the output node is changed from $g(x) = x$ to $g(x) = 1/(1+\exp(-x))$. (This is a function commonly used to produce probabilities as outputs, as it has range lying between 0 and 1.)
     Answer: The only modification required is to the computation of δ for the output node. This is now

     $$\delta = \sigma'(a_{\text{output}}) \frac{\partial E(\mathbf{w})}{\partial y}.$$

     We already know that $\sigma'(x) = \sigma(x)[1 - \sigma(x)]$. Also we have

     $$\begin{aligned} \frac{\partial E(\mathbf{w})}{\partial y} &= -\frac{\partial}{\partial y}\left(c \log y + (1-c)\log(1-y)\right) \\ &= \frac{y - c}{y(1-y)}. \end{aligned}$$

     Finally as $y = h(\mathbf{w}, \mathbf{x}) = \sigma(a_{\text{output}})$ we have

     $$\delta = h(\mathbf{w}, \mathbf{x}) - c.$$

   - Implement the modified algorithm. (Matlab is probably a good language to use.) Apply it to the parity data described in the lectures, and plot the results you obtain.

Answer: Here is a working solution. The following functions compute the function represented by a network, and the required gradients using backpropagation:

```
function [hidden_1, hidden_2, output] = simple_network2(x,W,w)
%-------------------------------------------------------------------------
%
% AI 1 exercise - modification of backpropagation.
%
% Copyright (C) Sean Holden 2010.
%
% Standard neural network with one hidden layer, sigmoid hidden
% nodes, and sigmoid output nodes.
%
% x is a column vector of inputs.
% W is a matrix of weights for the hidden nodes. Each row is a set of
% weights for a single node.
% w is a set of weights for the output nodes. Each row has the weights
% for one output.
%
% The first weight in a row is always for the bias.
%
% output is a vector of output values produced.
% hidden_1: hidden node outputs before the activation function.
% hidden_2: hidden node outputs after the activation function.
%
%-------------------------------------------------------------------------

number_of_inputs = size(x,1);
number_hidden = size(W,1) - 1;  % The -1 is for the bias weight.
number_output = size(w,1) - 1;

hidden_1 = W * [1;x];
hidden_2 = 1 ./ (1 + exp(-(hidden_1)));
output = 1 ./ (1 + exp(-(w * [1;hidden_2])));

function [new_W, new_w] = simple_backprop2(x,W,w,y)
%-------------------------------------------------------------------------
%
% AI 1 exercise - modification of backpropagation.
%
% Copyright (C) Sean Holden 2010.
%
% Do a forward propagation and compute the gradients.
%
%-------------------------------------------------------------------------

[hidden_1, hidden_2, output] = simple_network2(x,W,w);

output_delta = output - y;

hidden_deltas = hidden_2 .* (ones(size(hidden_2)) - hidden_2);
hidden_deltas = hidden_deltas .* (w(2:size(w,2))' * output_delta);
```

```
s1 = size(hidden_deltas,1);
s2 = size(x,1);

new_W = (diag(hidden_deltas) * ones(s1, size(W, 2))) ...
        .* [ones(s1,1) (diag(x) * ones(s2, s1))'];

new_w = output_delta * [1; hidden_2]';
```

We also need something that calls these in order to learn and to produce some results. The following was used to produce the diagrams that follow; most of it is in fact required only to plot the figures.

```
%--------------------------------------------------------------------------
%
% AI 1 exercise - modification of backpropagation.
%
% Copyright (C) Sean Holden 2010.
%
%--------------------------------------------------------------------------

clear
close all

number_of_examples = 40;
number_of_hidden_nodes = 5;

% y is a column vector containing targets.

y = [ones(number_of_examples/2, 1) ; zeros(number_of_examples/2, 1)];

% X is a matrix containing inputs. Each row is an input.

X = [zeros(number_of_examples/4, 2)];
X = [X ; ones(number_of_examples/4, 2)];
X = [X ; [zeros(number_of_examples/4, 1) ones(number_of_examples/4, 1)]];
X = [X ; [ones(number_of_examples/4, 1) zeros(number_of_examples/4, 1)]];
X = X + (randn(size(X)) * 0.05);

% Plot the training examples

figure(1)
subplot(1,2,1)
hold on

for i = 1:number_of_examples
  if i <= number_of_examples/2
    plot(X(i,1), X(i,2), '+r')
  else
    plot(X(i,1), X(i,2), 'oc')
  end
end

% These are the initial weights.
```

```
W = randn(number_of_hidden_nodes, 3) * 0.5;     % Hidden layer.
w = randn(1, number_of_hidden_nodes + 1) * 0.5; % Single output node.

% Plot the initial situation

x = 1;
z = 1;
surface = [];

for i = -1:0.05:2
  for j = -1:0.05:2;
    [hidden_1, hidden_2, output] = simple_network2([i; j],W,w);
    surface(x, z) = output;
    if output > 0.5
      h = plot(i,j,'.');
      set(h,'Markersize',5);
    end
    x = x + 1;
  end
  z = z + 1;
  x = 1;
end

axis([-1 2 -1 2])
axis square
title('Before training','Interpreter','latex')
xlabel('$x_1$','Interpreter','latex')
ylabel('$x_2$','Interpreter','latex')
hold off

figure(2)
subplot(1,2,1)
surf([-1:0.05:2],[-1:0.05:2],surface)
hold on

for i = 1:number_of_examples
  if i <= number_of_examples/2
    plot3(X(i,1), X(i,2), 0.5, '+r')
  else
    plot3(X(i,1), X(i,2), 0.5, 'oc')
  end
end

axis([-1 2 -1 2 0 1])
title('Before training','Interpreter','latex')
xlabel('$x_1$','Interpreter','latex')
ylabel('$x_2$','Interpreter','latex')
zlabel('Network output','Interpreter','latex')
axis square

% The following implements backprop

figure(3)
```

```
step_size = 0.01;
Error = [0];
x_axis = [0];
h = plot(x_axis, Error);
set(h, 'Erasemode', 'xor')

for main_loop = 1:3000
  % Calculate the current error for the entire training sequence.

  error = 0;
  for j = 1:number_of_examples
    [hidden_1, hidden_2, output] = simple_network2(X(j,:)',W,w);
    error = error -((y(j) * log(output)) + ((1-y(j))*log(1-output)));
  end
  disp(['Iteration: ' num2str(main_loop) ' Error: ' num2str(error)])
  Error = [Error error];
  x_axis = [x_axis main_loop];

  drawnow
  set(h, 'XData', x_axis, 'YData', Error)

  % Now do some training.

  for i = 1:number_of_examples
    [W_update, w_update] = simple_backprop2(X(i,:)',W,w,y(i));
    W = W - (step_size * W_update);
    w = w - (step_size * w_update);
  end
end

title('Error during training','Interpreter','latex')

% Now visualize what the network actually does.

figure(1)
subplot(1,2,2)
hold on

for i = 1:number_of_examples
  if i <= number_of_examples/2
    plot(X(i,1), X(i,2), '+r')
  else
    plot(X(i,1), X(i,2), 'oc')
  end
end

x = 1;
z = 1;
surface = [];

for i = -1:0.05:2
  for j = -1:0.05:2;
```

```
    [hidden_1, hidden_2, output] = simple_network2([i; j],W,w);
    surface(x, z) = output;
    if output > 0.5
      h = plot(i,j,'.');
      set(h,'Markersize',5);
    end
    x = x + 1;
  end
  z = z + 1;
  x = 1;
end

axis([-1 2 -1 2])
axis square
title('After training','Interpreter','latex')
xlabel('$x_1$','Interpreter','latex')
ylabel('$x_2$','Interpreter','latex')
figure(2)
subplot(1,2,2)
surf([-1:0.05:2],[-1:0.05:2],surface)
hold on

for i = 1:number_of_examples
  if i <= number_of_examples/2
    plot3(X(i,1), X(i,2), 0.5, '+r')
  else
    plot3(X(i,1), X(i,2), 0.5, 'oc')
  end
end

axis([-1 2 -1 2 0 1])
axis square
title('After training','Interpreter','latex')
xlabel('$x_1$','Interpreter','latex')
ylabel('$x_2$','Interpreter','latex')
zlabel('Network output','Interpreter','latex')
```

Finally, running this code produces the results shown in figures 2, 3 and 4.

9 Appendix

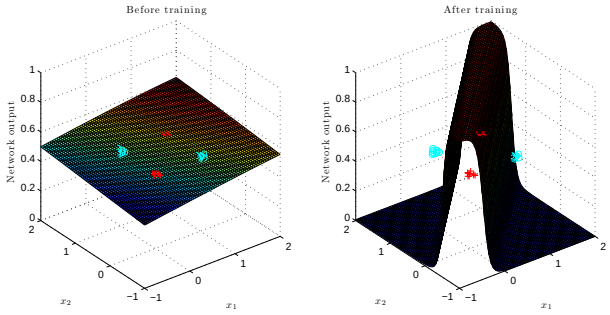


Figure 2: Network output before and after training

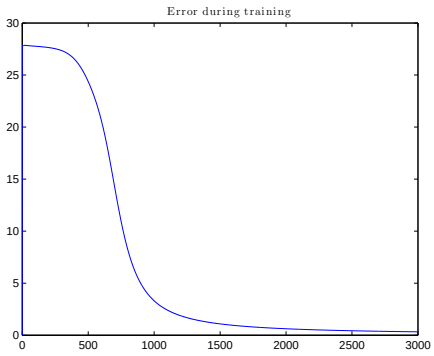


Figure 3: Error during training

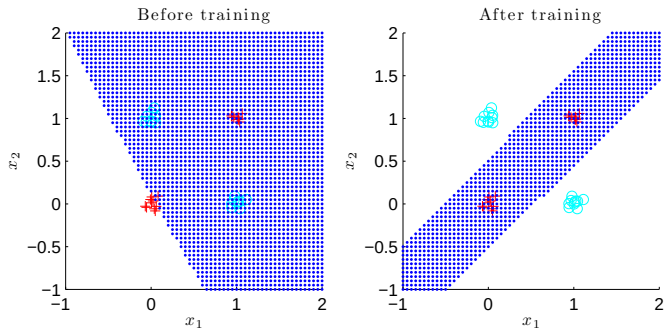


Figure 4: Decision boundary before and after training

10 Output for $\alpha - \beta$ pruning: max first

```
Max called on node: 0
Calling min from node: 0

Min called on node: 1
Calling max from node: 1

Max called on node: 2
Calling min from node: 2

Min called on node: 3
Calling max from node: 3
Leaf node with value: 1
Value = 1
Calling max from node: 3
Leaf node with value: -15
Value = -15
Nothing pruned in min. Value = -15
Value = -15
Calling min from node: 2

Min called on node: 4
Calling max from node: 4
Leaf node with value: 2
Value = 2
Calling max from node: 4
Leaf node with value: 19
Value = 2
Nothing pruned in min. Value = 2
Value = 2
Nothing pruned in max. Value = 2
Value = 2
Calling max from node: 1

Max called on node: 5
Calling min from node: 5

Min called on node: 6
Calling max from node: 6
Leaf node with value: 18
Value = 18
Calling max from node: 6
Leaf node with value: 23
Value = 18
Nothing pruned in min. Value = 18
Value = 18
--------- Pruned! Value = 18
Value = 2
Nothing pruned in min. Value = 2
Value = 2
Calling min from node: 0

Min called on node: 8
Calling max from node: 8

Max called on node: 9
Calling min from node: 9

Min called on node: 10
Calling max from node: 10
Leaf node with value: 2
Value = 2
Calling max from node: 10
Leaf node with value: 1
Value = 1
--------- Pruned! Value = 1
Value = 1
Calling min from node: 9

Min called on node: 11
Calling max from node: 11
Leaf node with value: 7
Value = 7
Calling max from node: 11
Leaf node with value: 8
Value = 7
Nothing pruned in min. Value = 7
Value = 7
Nothing pruned in max. Value = 7
Value = 7
Calling max from node: 8

Max called on node: 12
Calling min from node: 12

Min called on node: 13
Calling max from node: 13
Leaf node with value: 9
Value = 9
Calling max from node: 13
Leaf node with value: 10
Value = 9
Nothing pruned in min. Value = 9
Value = 9
--------- Pruned! Value = 9
Value = 7
Nothing pruned in min. Value = 7
Value = 7
Calling min from node: 0

Min called on node: 15
Calling max from node: 15

Max called on node: 16
Calling min from node: 16

Min called on node: 17
Calling max from node: 17
Leaf node with value: -1
Value = -1
--------- Pruned! Value = -1
Value = -1
Calling min from node: 16

Min called on node: 18
Calling max from node: 18
Leaf node with value: 4
Value = 4
--------- Pruned! Value = 4
Value = 4
Nothing pruned in max. Value = 4
Value = 4
--------- Pruned! Value = 4
Value = 7
Nothing pruned in max. Value = 7
```

11 Output for: AC-3

```
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 1
  Dealing with arc 1 -> 2
  Domains are { red }{ red cyan black }
Finished iteration 1
New domains:
{ red }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 2
  Dealing with arc 1 -> 3
  Domains are { red }{ red cyan black }
Finished iteration 2
New domains:
{ red }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 3
  Dealing with arc 1 -> 4
  Domains are { red }{ red cyan black }
Finished iteration 3
New domains:
{ red }
{ red cyan black }
{ red cyan black }
```

```
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 4
  Dealing with arc 2 -> 1
  Domains are { red cyan black }{ red }
  Found inconsistency in 2 -> 1
  New left domain is { cyan black }
    Adding 1 -> 2
    Adding 4 -> 2
    Adding 6 -> 2
Finished iteration 4
New domains:
{ red }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 5
  Dealing with arc 2 -> 4
  Domains are { cyan black }{ red cyan black }
Finished iteration 5
New domains:
{ red }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 6
  Dealing with arc 2 -> 6
  Domains are { cyan black }{ red cyan black }
Finished iteration 6
New domains:
{ red }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 7
  Dealing with arc 3 -> 1
  Domains are { red cyan black }{ red }
  Found inconsistency in 3 -> 1
  New left domain is { cyan black }
    Adding 1 -> 3
    Adding 4 -> 3
    Adding 7 -> 3
Finished iteration 7
New domains:
{ red }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 8
  Dealing with arc 3 -> 4
  Domains are { cyan black }{ red cyan black }
Finished iteration 8
New domains:
{ red }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 9
  Dealing with arc 3 -> 7
  Domains are { cyan black }{ red cyan black }
Finished iteration 9
New domains:
{ red }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 10
  Dealing with arc 4 -> 1
  Domains are { red cyan black }{ red }
  Found inconsistency in 4 -> 1
  New left domain is { cyan black }
    Adding 1 -> 4
    Adding 2 -> 4
    Adding 3 -> 4
    Adding 5 -> 4
Finished iteration 10
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 11
  Dealing with arc 4 -> 2
  Domains are { cyan black }{ cyan black }
Finished iteration 11
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 12
  Dealing with arc 4 -> 3
  Domains are { cyan black }{ cyan black }
Finished iteration 12
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 13
  Dealing with arc 4 -> 5
  Domains are { cyan black }{ red cyan black }
Finished iteration 13
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 14
  Dealing with arc 5 -> 4
  Domains are { red cyan black }{ cyan black }
Finished iteration 14
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 15
  Dealing with arc 5 -> 6
  Domains are { red cyan black }{ red cyan black }
Finished iteration 15
```

```
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 16
  Dealing with arc 5 -> 7
  Domains are { red cyan black }{ red cyan black }
Finished iteration 16
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 17
  Dealing with arc 6 -> 2
  Domains are { red cyan black }{ cyan black }
Finished iteration 17
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 18
  Dealing with arc 6 -> 5
  Domains are { red cyan black }{ red cyan black }
Finished iteration 18
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 19
  Dealing with arc 6 -> 7
  Domains are { red cyan black }{ red cyan black }
Finished iteration 19
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 20
  Dealing with arc 6 -> 8
  Domains are { red cyan black }{ red cyan black }
Finished iteration 20
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 21
  Dealing with arc 7 -> 3
  Domains are { red cyan black }{ cyan black }
Finished iteration 21
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 22
  Dealing with arc 7 -> 5
  Domains are { red cyan black }{ red cyan black }
Finished iteration 22
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 23
  Dealing with arc 7 -> 6
  Domains are { red cyan black }{ red cyan black }
Finished iteration 23
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 24
  Dealing with arc 7 -> 8
  Domains are { red cyan black }{ red cyan black }
Finished iteration 24
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 25
  Dealing with arc 8 -> 6
  Domains are { red cyan black }{ red cyan black }
Finished iteration 25
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 26
  Dealing with arc 8 -> 7
  Domains are { red cyan black }{ red cyan black }
Finished iteration 26
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 27
  Dealing with arc 1 -> 2
  Domains are { red }{ cyan black }
Finished iteration 27
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 28
  Dealing with arc 4 -> 2
```

```
  Domains are { cyan black }{ cyan black }
Finished iteration 28
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 29
  Dealing with arc 6 -> 2
  Domains are { red cyan black }{ cyan black }
Finished iteration 29
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 30
  Dealing with arc 1 -> 3
  Domains are { red }{ cyan black }
Finished iteration 30
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 31
  Dealing with arc 4 -> 3
  Domains are { cyan black }{ cyan black }
Finished iteration 31
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 32
  Dealing with arc 7 -> 3
  Domains are { red cyan black }{ cyan black }
Finished iteration 32
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 33
  Dealing with arc 1 -> 4
  Domains are { red }{ cyan black }
Finished iteration 33
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 34
  Dealing with arc 2 -> 4
  Domains are { cyan black }{ cyan black }
Finished iteration 34
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 35
  Dealing with arc 3 -> 4
  Domains are { cyan black }{ cyan black }
Finished iteration 35
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 36
  Dealing with arc 5 -> 4
  Domains are { red cyan black }{ cyan black }
Finished iteration 36
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 1
  Dealing with arc 1 -> 2
  Domains are { red }{ cyan black }
Finished iteration 1
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 2
  Dealing with arc 1 -> 3
  Domains are { red }{ cyan black }
Finished iteration 2
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 3
  Dealing with arc 1 -> 4
  Domains are { red }{ cyan }
Finished iteration 3
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 4
  Dealing with arc 2 -> 1
  Domains are { cyan black }{ red }
Finished iteration 4
New domains:
{ red }
{ cyan black }
{ cyan black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-----------------------------------------
```

```
Iteration = 5
  Dealing with arc 2 -> 4
  Domains are { cyan black }{ cyan }
  Found inconsistency in 2 -> 4
  New left domain is { black }
    Adding 1 -> 2
    Adding 4 -> 2
    Adding 6 -> 2
Finished iteration 5
New domains:
{ red }
{ black }
{ cyan black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 6
  Dealing with arc 2 -> 6
  Domains are { black }{ red cyan black }
Finished iteration 6
New domains:
{ red }
{ black }
{ cyan black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 7
  Dealing with arc 3 -> 1
  Domains are { cyan black }{ red }
Finished iteration 7
New domains:
{ red }
{ black }
{ cyan black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 8
  Dealing with arc 3 -> 4
  Domains are { cyan black }{ cyan }
  Found inconsistency in 3 -> 4
  New left domain is { black }
    Adding 1 -> 3
    Adding 4 -> 3
    Adding 7 -> 3
Finished iteration 8
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 9
  Dealing with arc 3 -> 7
  Domains are { black }{ red cyan black }
Finished iteration 9
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 10
  Dealing with arc 4 -> 1
  Domains are { cyan }{ red }
Finished iteration 10
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 11
  Dealing with arc 4 -> 2
  Domains are { cyan }{ black }
Finished iteration 11
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 12
  Dealing with arc 4 -> 3
  Domains are { cyan }{ black }
Finished iteration 12
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 13
  Dealing with arc 4 -> 5
  Domains are { cyan }{ red cyan black }
Finished iteration 13
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red cyan black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 14
  Dealing with arc 5 -> 4
  Domains are { red cyan black }{ cyan }
  Found inconsistency in 5 -> 4
  New left domain is { red black }
    Adding 4 -> 5
    Adding 6 -> 5
    Adding 7 -> 5
Finished iteration 14
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 15
  Dealing with arc 5 -> 6
  Domains are { red black }{ red cyan black }
Finished iteration 15
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan black }
{ red cyan black }
{ red cyan black }
-------------------------------------------
Iteration = 16
  Dealing with arc 5 -> 7
  Domains are { red black }{ red cyan black }
Finished iteration 16
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan black }
```

```
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 17
  Dealing with arc 6 -> 2
  Domains are { red cyan black }{ black }
  Found inconsistency in 6 -> 2
  New left domain is { red cyan }
    Adding 2 -> 6
    Adding 5 -> 6
    Adding 7 -> 6
    Adding 8 -> 6
Finished iteration 17
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 18
  Dealing with arc 6 -> 5
  Domains are { red cyan }{ red black }
Finished iteration 18
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 19
  Dealing with arc 6 -> 7
  Domains are { red cyan }{ red cyan black }
Finished iteration 19
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 20
  Dealing with arc 6 -> 8
  Domains are { red cyan }{ red cyan black }
Finished iteration 20
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan black }
{ red cyan black }
-----------------------------------------
Iteration = 21
  Dealing with arc 7 -> 3
  Domains are { red cyan black }{ black }
  Found inconsistency in 7 -> 3
  New left domain is { red cyan }
    Adding 3 -> 7
    Adding 5 -> 7
    Adding 6 -> 7
    Adding 8 -> 7
Finished iteration 21
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 22
  Dealing with arc 7 -> 5
  Domains are { red cyan }{ red black }
Finished iteration 22
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 23
  Dealing with arc 7 -> 6
  Domains are { red cyan }{ red cyan }
Finished iteration 23
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 24
  Dealing with arc 7 -> 8
  Domains are { red cyan }{ red cyan black }
Finished iteration 24
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 25
  Dealing with arc 8 -> 6
  Domains are { red cyan black }{ red cyan }
Finished iteration 25
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 26
  Dealing with arc 8 -> 7
  Domains are { red cyan black }{ red cyan }
Finished iteration 26
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 27
  Dealing with arc 1 -> 2
  Domains are { red }{ black }
Finished iteration 27
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 28
  Dealing with arc 4 -> 2
  Domains are { cyan }{ black }
Finished iteration 28
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
```

```
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 29
  Dealing with arc 6 -> 2
  Domains are { red cyan }{ black }
Finished iteration 29
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 30
  Dealing with arc 1 -> 3
  Domains are { red }{ black }
Finished iteration 30
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 31
  Dealing with arc 4 -> 3
  Domains are { cyan }{ black }
Finished iteration 31
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 32
  Dealing with arc 7 -> 3
  Domains are { red cyan }{ black }
Finished iteration 32
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 33
  Dealing with arc 4 -> 5
  Domains are { cyan }{ red black }
Finished iteration 33
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 34
  Dealing with arc 6 -> 5
  Domains are { red cyan }{ red black }
Finished iteration 34
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 35
  Dealing with arc 7 -> 5
  Domains are { red cyan }{ red black }
Finished iteration 35
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 36
  Dealing with arc 2 -> 6
  Domains are { black }{ red cyan }
Finished iteration 36
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 37
  Dealing with arc 5 -> 6
  Domains are { red black }{ red cyan }
Finished iteration 37
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 38
  Dealing with arc 7 -> 6
  Domains are { red cyan }{ red cyan }
Finished iteration 38
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 39
  Dealing with arc 8 -> 6
  Domains are { red cyan black }{ red cyan }
Finished iteration 39
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 40
  Dealing with arc 3 -> 7
  Domains are { black }{ red cyan }
Finished iteration 40
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 41
  Dealing with arc 5 -> 7
  Domains are { red black }{ red cyan }
Finished iteration 41
New domains:
{ red }
{ black }
{ black }
{ cyan }
```

```
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 42
  Dealing with arc 6 -> 7
  Domains are { red cyan }{ red cyan }
Finished iteration 42
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 43
  Dealing with arc 8 -> 7
  Domains are { red cyan black }{ red cyan }
Finished iteration 43
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red black }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 1
  Dealing with arc 1 -> 2
  Domains are { red }{ black }
Finished iteration 1
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 2
  Dealing with arc 1 -> 3
  Domains are { red }{ black }
Finished iteration 2
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 3
  Dealing with arc 1 -> 4
  Domains are { red }{ cyan }
Finished iteration 3
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 4
  Dealing with arc 2 -> 1
  Domains are { black }{ red }
Finished iteration 4
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 5
  Dealing with arc 2 -> 4
  Domains are { black }{ cyan }
Finished iteration 5
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 6
  Dealing with arc 2 -> 6
  Domains are { black }{ red cyan }
Finished iteration 6
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 7
  Dealing with arc 3 -> 1
  Domains are { black }{ red }
Finished iteration 7
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 8
  Dealing with arc 3 -> 4
  Domains are { black }{ cyan }
Finished iteration 8
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 9
  Dealing with arc 3 -> 7
  Domains are { black }{ red cyan }
Finished iteration 9
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 10
  Dealing with arc 4 -> 1
  Domains are { cyan }{ red }
Finished iteration 10
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-------------------------------------------
Iteration = 11
  Dealing with arc 4 -> 2
  Domains are { cyan }{ black }
Finished iteration 11
New domains:
{ red }
{ black }
```

```
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 12
  Dealing with arc 4 -> 3
  Domains are { cyan }{ black }
Finished iteration 12
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 13
  Dealing with arc 4 -> 5
  Domains are { cyan }{ red }
Finished iteration 13
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 14
  Dealing with arc 5 -> 4
  Domains are { red }{ cyan }
Finished iteration 14
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 15
  Dealing with arc 5 -> 6
  Domains are { red }{ red cyan }
Finished iteration 15
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 16
  Dealing with arc 5 -> 7
  Domains are { red }{ red cyan }
Finished iteration 16
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 17
  Dealing with arc 6 -> 2
  Domains are { red cyan }{ black }
Finished iteration 17
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ red cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 18
  Dealing with arc 6 -> 5
  Domains are { red cyan }{ red }
  Found inconsistency in 6 -> 5
  New left domain is { cyan }
    Adding 2 -> 6
    Adding 5 -> 6
    Adding 7 -> 6
    Adding 8 -> 6
Finished iteration 18
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 19
  Dealing with arc 6 -> 7
  Domains are { cyan }{ red cyan }
Finished iteration 19
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 20
  Dealing with arc 6 -> 8
  Domains are { cyan }{ red cyan black }
Finished iteration 20
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 21
  Dealing with arc 7 -> 3
  Domains are { red cyan }{ black }
Finished iteration 21
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ cyan }
{ red cyan }
{ red cyan black }
-----------------------------------------
Iteration = 22
  Dealing with arc 7 -> 5
  Domains are { red cyan }{ red }
  Found inconsistency in 7 -> 5
  New left domain is { cyan }
    Adding 3 -> 7
    Adding 5 -> 7
    Adding 6 -> 7
    Adding 8 -> 7
Finished iteration 22
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ cyan }
{ cyan }
{ red cyan black }
-----------------------------------------
Iteration = 23
  Dealing with arc 7 -> 6
  Domains are { cyan }{ cyan }
  Found inconsistency in 7 -> 6
  New left domain is { }
    Adding 3 -> 7
    Adding 5 -> 7
```

```
    Adding 6 -> 7
    Adding 8 -> 7
Finished iteration 23
New domains:
{ red }
{ black }
{ black }
{ cyan }
{ red }
{ cyan }
{ }
{ red cyan black }
```