

Prolog

Andrew Rice
Michealmas 2007

Course Aims

- Introduce a declarative style of programming
- Fundamental elements of Prolog: terms, clauses, lists, arithmetic, cut, negation
- Write programs in Prolog and understand their execution
- Learn to use difference structures
- Understand the basic principles of constraint programming

2

Assessment

- 1 Exam question in Paper 4
- Tickable Assessed Exercise
 - you must get a tick for either Prolog or C & C++ (next term)
 - more information to follow

3

Supervision Work

- Separate set of questions at the end of the lecture handout
- I will give some pointers and outline solutions during the lectures

4

Recommended Text

“PROLOG Programming for Artificial Intelligence”,
Ivan Bratko

5

Lecture 1

- Logic programming and declarative programs
- Introduction to Prolog
- How to use it
- Prolog syntax: Terms
- Unification
- Solving a logic puzzle

6

Imperative Programming

/ to compute the sum of the list, go through the
* list adding each value to the accumulator */*

```
int sum(int[] list) {  
    int result = 0;  
    for(int i=0; i<list.length; ++i) {  
        result += list[i];  
    }  
    return result;  
}
```

7

Functional Programming

(The sum of the empty list is zero and the sum of
the list with head h and tail t is h plus the sum of the
tail *)*

```
fun sum([]) = 0  
| sum(h::t) = h + sum(t);
```

8

Logic Programming

% the sum of the empty list is zero

sum([],0).

% the sum of the list with head H and tail T is N

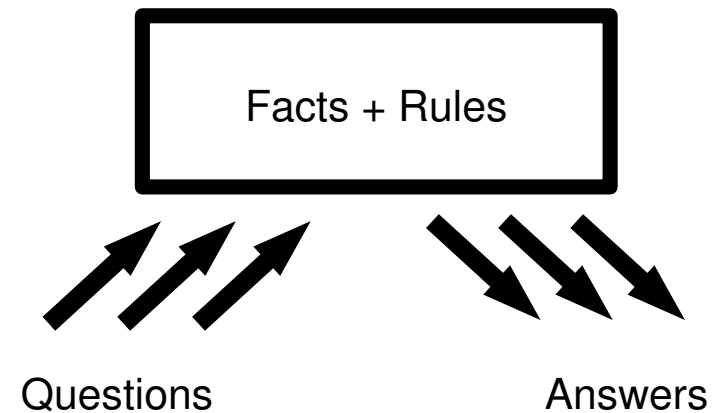
% if the sum of the list T is M and N is M + H

sum([H|T],N) :- sum(T,M), N is M+H.

This is a declarative reading of a program
- it describes the result not the procedure

9

Prolog Programs Answer Questions



10

Prolog came from the field of Natural Language Processing

PROgramming en **LOG**ique

Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R. "Un systeme de communication homme-machine en français", Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille. 1973.

11

Modern Prolog Interpreters Use the Warren Abstract Machine

David H. D. Warren. "An abstract Prolog instruction set." Technical Note 309, SRI International, Menlo Park, CA, October 1983.

12

You are expected to use SWI-Prolog

- Open-source Prolog environment.
- Development began in 1987
- Available for Linux, MacOS X and Windows.
- <http://www.swi-prolog.org/>

We will use SWI-Prolog throughout this course
Get a copy!

13

Prolog's database can answer simple questions

```
> prolog .....or maybe pl on your system
?- [user]. .....get ready to enter a new program
|: milestone(rousell,1972).
|: milestone(warren,1983).
|: milestone(swiprolog,1987).
|: milestone(yourcourse,2007). .....type [CTRL]-D when done
|: % user://1 compiled 0.01 sec, 764 bytes
Yes
?- milestone(warren,1983). .....ask it a question
Yes .....the answer is "yes"
?- milestone(swiprolog,X). .....let it find an answer
X=1987 .....the answer is 1987
Yes
?- milestone(yourcourse,2000).....ask it a question
No .....the answer is "no"
?- halt. ....exit the interpreter
```

14

We will usually write programs to a source file on disk

```
> cat milestone.pl .....enter your program in a text file
milestone(rousell,1972). .....it must have a .pl extension
milestone(warren,1983).
milestone(swiprolog,1987).
milestone(yourcourse,2007).

> prolog
?- [milestone]. .....instruct prolog to load the program
?- milestone(warren,1983).
Yes
?- milestone(X,Y). .....find answers
X = rousell
Y = 1972 ; .....you type a semi-colon (;) for more

X= warren
Y = 1983 .....you press enter when you've had enough
Yes
?- halt.
```

15

Our program is composed of facts and queries

```
> cat milestone.pl
milestone(rousell,1972).
milestone(warren,1983).
milestone(swiprolog,1987).
milestone(yourcourse,2007).
```

These are *facts*
(a particular type of *clause*)

```
> prolog
?- [milestone].
?- milestone(warren,1983).
Yes
?- milestone(X,Y).
X = rousell
Y = 1972 ;
```

These are *queries*

```
X= warren
Y = 1983
```

16

Using the prolog shell

- The Prolog shell accepts only queries at the top-level.
- Press enter to accept the answer and return to the top level
- Type a semi-colon (;) to request the next answer
- Type w to fully display a long result which Prolog has abbreviated

17

Terms can be Constants, Compounds or Variables

Constants

Numbers: 1 -2

3.14

Atoms: tigger

'100 Acre Wood'

Compound

likes(pooh_bear,honey)

plus(4,mult(3,plus(1,9)))

Variables

X

Sticks

18

Unification is Prolog's fundamental operation

unify(A,A).

Do these unify:

a	a
a	A
tree(l,r)	A
tree(A,r)	tree(l,C)
A	a(A)

a	b
a	B
tree(l,r)	tree(B,C)
tree(A,r)	tree(A,B)
a	a(A)

19

Zebra Puzzle

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Who drinks water? Who owns the zebra?

20

Model the situation

Represent each house with the clause:

`house(Nationality,Pet,Smokes,Drinks,Colour)`

Represent the row of houses as follows:

`(H1,H2,H3,H4,H5)`

21

Question

What sort of a term is:

`house(Nationality,Pet,Smokes,Drinks,Colour)`

- a) number
- b) atom
- c) compound
- d) variable

22

Question

What sort of a term is:

`Nationality`

- a) number
- b) atom
- c) compound
- d) variable

23

Question

What sort of a term is:

`(H1,H2,H3,H4,H5)`

- a) number
- b) atom
- c) compound
- d) variable

24

Define relevant facts

exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

Read this as “exists(A,(A,_,_,_,_)) is true if the details of house A unifies with the state of the first house”

25

More Facts

6. The green house is immediately **to the right of** the ivory house.

rightOf(A,B,(B,A,_,_,_)).

rightOf(A,B,(_,B,A,_,_)).

rightOf(A,B,(_,_,B,A,_)).

rightOf(A,B,(_,_,_,B,A)).

26

More Facts

9. Milk is drunk **in the middle** house.

middleHouse(A,(_,_,A,_,_)).

10. The Norwegian lives **in the first** house.

firstHouse(A,(A,_,_,_,_)).

27

More facts

11. The man who smokes Chesterfields lives in the house **next to the** man with the fox.

nextTo(A,B,(A,B,_,_,_)).

nextTo(A,B,(_,A,B,_,_)).

nextTo(A,B,(_,_,A,B,_)).

nextTo(A,B,(_,_,_,A,B)).

nextTo(A,B,(B,A,_,_,_)).

nextTo(A,B,(_,B,A,_,_)).

nextTo(A,B,(_,_,B,A,_)).

nextTo(A,B,(_,_,_,B,A)).

28

Express the puzzle as a query

```
exists(house(british,_,_,red),Houses),
exists(house(spanish,dog,_,_),Houses),
exists(house(_,_,coffee,green),Houses),
exists(house(ukranian,_,_,tea,_),Houses),
rightOf(house(_,_,_,green),house(_,_,_,ivory),Houses),
exists(house(_,snail,oldgold,_,_),Houses),
exists(house(_,kools,_,yellow),Houses),
middleHouse(house(_,_,_,milk,_),Houses),
firstHouse(house(norwegian,_,_,_),Houses),
nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
exists(house(_,_,luckystrike,orangejuice,_),Houses),
exists(house(japanese,_,_,parliaments,_,_),Houses),
nextTo(house(norwegian,_,_,_),house(_,_,_,blue),Houses),
exists(house(WaterDrinker,_,_,water,_),Houses),
exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

2. The Englishman lives in the red house.

29

Express the puzzle as a query

```
exists(house(british,_,_,red),Houses),
exists(house(spanish,dog,_,_),Houses),
exists(house(_,_,coffee,green),Houses),
exists(house(ukranian,_,_,tea,_),Houses),
rightOf(house(_,_,_,green),house(_,_,_,ivory),Houses),
exists(house(_,snail,oldgold,_,_),Houses),
exists(house(_,kools,_,yellow),Houses),
middleHouse(house(_,_,_,milk,_),Houses),
firstHouse(house(norwegian,_,_,_),Houses),
nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
exists(house(_,_,luckystrike,orangejuice,_),Houses),
exists(house(japanese,_,_,parliaments,_,_),Houses),
nextTo(house(norwegian,_,_,_),house(_,_,_,blue),Houses),
exists(house(WaterDrinker,_,_,water,_),Houses),
exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

3. The Spaniard owns the dog.

30

Express the puzzle as a query

```
exists(house(british,_,_,red),Houses),
exists(house(spanish,dog,_,_),Houses),
exists(house(_,_,coffee,green),Houses),
exists(house(ukranian,_,_,tea,_),Houses),
rightOf(house(_,_,_,green),house(_,_,_,ivory),Houses),
exists(house(_,snail,oldgold,_,_),Houses),
exists(house(_,kools,_,yellow),Houses),
middleHouse(house(_,_,_,milk,_),Houses),
firstHouse(house(norwegian,_,_,_),Houses),
nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
exists(house(_,_,luckystrike,orangejuice,_),Houses),
exists(house(japanese,_,_,parliaments,_,_),Houses),
nextTo(house(norwegian,_,_,_),house(_,_,_,blue),Houses),
exists(house(WaterDrinker,_,_,water,_),Houses),
exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

6. The green house is immediately to the right of the ivory house.

31

You can include queries in your source file

- Normal lines in the source file define new clauses
- Lines beginning with :- (colon followed by hyphen) are queries that Prolog will execute immediately
- Use the print() query to print the results

32

Zebra Puzzle

> prolog

?- [zebra].

japanese

norwegian

% zebra compiled 0.00 sec, 6,384 bytes

Yes

?- halt.

We use print(WaterDrinker),
print(ZebraOwner) in our query
for this output

Lecture 2

- Rules
- Lists
- Arithmetic
- Last-call optimisation
- Backtracking
- Generate and Test

1

Rules have a head which is true if
the body is true

head **body**
rule(X,Y) :- part1(X), part2(X,Y).

Read this as: “rule(X,Y) is true if part1(X) is true and
part2(X,Y) is true”

2

Internal variables are common

rule2(X) :- something(X,Z), else(Z).

Read this as “rule2(X) is true if there is a Z such that
something(X,Z) is true and else(Z) is true”

3

Prolog identifies clauses by name
and arity

The clause

rule.

is referred to as rule/0 and is different to:

rule(A).

which is referred to as rule/1

4

We sometimes identify the way to use parameters of a rule

```
myrule(+A,+B,-C,-D)
```

means the clause “myrule” should be queried with two ground (input) terms A and B and two variable (output) terms C and D

5

Rules can be recursive

```
rule3(ground).
```

```
rule3(In) :- anotherRule(In,Out), rule3(Out).
```

6

Prolog has builtin support for lists

Notated with square brackets e.g. [1,2,3,4]

The empty list is denoted []

Use a pipe symbol to refer to the tail of a list e.g. [H|T] and [1|T] and [1,2,3|T]

7

We can write a rule to find the last element of a list

```
last([H],H).
```

```
last(_|T,H) :- last(T,H).
```

8

Question

What happens if I ask: `last([],X).` ?

- a) pattern-match exception
- b) Prolog says no
- c) Prolog says yes, `X = []`
- d) Prolog says yes, `X = ???`

9

You should include tests for your clauses in your source code

Example `last.pl`:

```
last([H],H).
```

```
last([_|T],H) :- last(T,H).
```

```
:- last([1,2,3],A), A=3
```

10

We use trace to see the execution

?- `[last].` **path**
% last compiled 0.01 sec, 604 bytes

Yes

?- `trace,last([1,2],A).`

Call: (8) `last([1, 2], _G187) ? creep`

Call: (9) `last([2], _G187) ? creep`

Exit: (9) `last([2], 2) ? creep`

Exit: (8) `last([1, 2], 2) ? creep`

`A = 2`

Yes

Press enter to “creep”
to the next level

Press s to skip and
jump straight to
the result of the call

11

Arithmetic Expressions

What happens if you ask prolog:

`A = 1+2.`

?

12

Arithmetic equality != Unification

?- A = 1+2.

A = 1+2

Yes

?- 1+2 = 3.

No

Equals (=) in Prolog means “unifies with”

13

Arithmetic equality != Unification

?- A = money+power.

A = money+power

Yes

Plus (+) just forms a compound term e.g. +(1,2)

14

Use the “is” operator

The “is” operator tells prolog to **evaluate** the right-hand expression numerically and unify with the left

?- A is 1+2.

A = 3

Yes

?- A is money+power.

ERROR: Arithmetic: `money/0' is not a function

15

The right hand side must be a ground term (no variables)

?- A is B+2.

ERROR: Arguments are not sufficiently instantiated

?- 3 is B+2.

ERROR: Arguments are not sufficiently instantiated

16

A rule can be written to compute the list length

List length:

$\text{len}([],0).$

$\text{len}([_|T],N) :- \text{len}(T,M), N \text{ is } M+1.$

This uses $O(N)$ stack space for a list of length N

17

List Length using $O(N)$ stack space

- Evaluate $\text{len}([1,2],A).$

- Apply $\text{len}([1|[2]],A) :- \text{len}([2],M), A \text{ is } M+1$

- Evaluate $\text{len}([2],M)$

- Apply $\text{len}([2|[]],M) :- \text{len}([],M1), M \text{ is } M1+1$

- Evaluate $\text{len}([],M1)$

- Apply $\text{len}([],0)$ so $M1 = 0$

- Evaluate $M \text{ is } M1+1$ so $M = 1$

- Evaluate $A \text{ is } M+1$ so $A = 2$

- Result $\text{len}([1,2],2)$

18

List length using $O(1)$ stack space

List length using an accumulator:

$\text{len2}([],\text{Acc},\text{Acc}).$

$\text{len2}([_|T],\text{Acc},\text{Result}) :- \text{Acc1 is Acc} + 1,$
 $\text{len2}(T,\text{Acc1},\text{Result}).$

$\text{len2}(\text{List},\text{Result}) :- \text{len2}(\text{List},0,\text{Result}).$

19

List length using $O(1)$ stack space

- Evaluate $\text{len2}([1,2],0,R)$

- Apply $\text{len2}([1|[2]],0,R) :- \text{Acc1 is } 0+1, \text{len2}([2],\text{Acc1},R).$

- Evaluate $\text{Acc1 is } 0+1$ so $\text{Acc1} = 1$

- Evaluate $\text{len2}([2],1,R)$

- Apply $\text{len2}([2|[]],1,R) :- \text{Acc1 is } 1+1, \text{len2}([],\text{Acc1},R).$

- Evaluate $\text{Acc1 is } 1+1$ so $\text{Acc1} = 2$

- Evaluate $\text{len2}([],2,R).$

- Apply $\text{len2}([],2,2)$ so $R = 2$

20

Last Call Optimisation

- This technique is applied by the prolog interpreter
- The last clause of the rule is executed as a branch – we can forget that we were ever interested in the head
- We can only do this if the rule is *determinate* up to that point

21

Prolog uses depth-first search to find answers

Here is a (boring) program

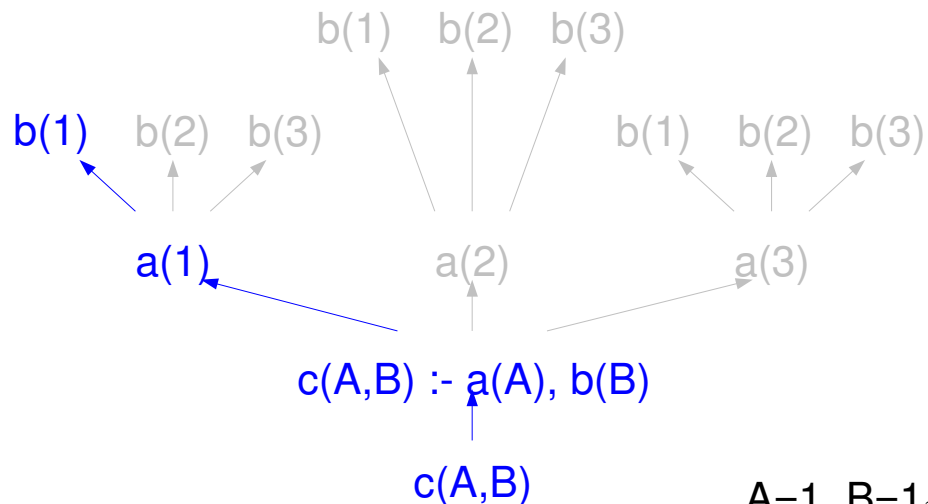
```
a(1).
a(2).
a(3).
b(1).
b(2).
b(3).
c(A,B) :- a(A), b(B).
```

What does prolog do when given the query:

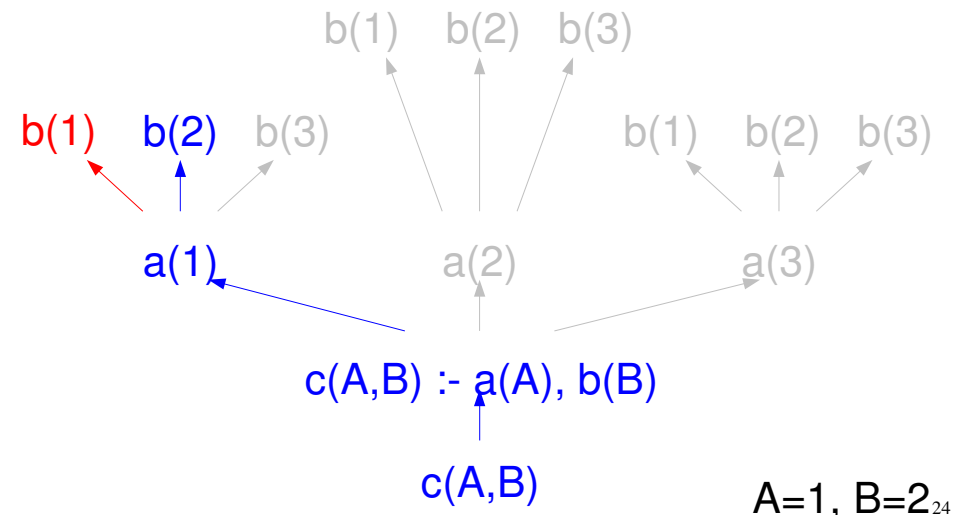
`c(A,B).`

22

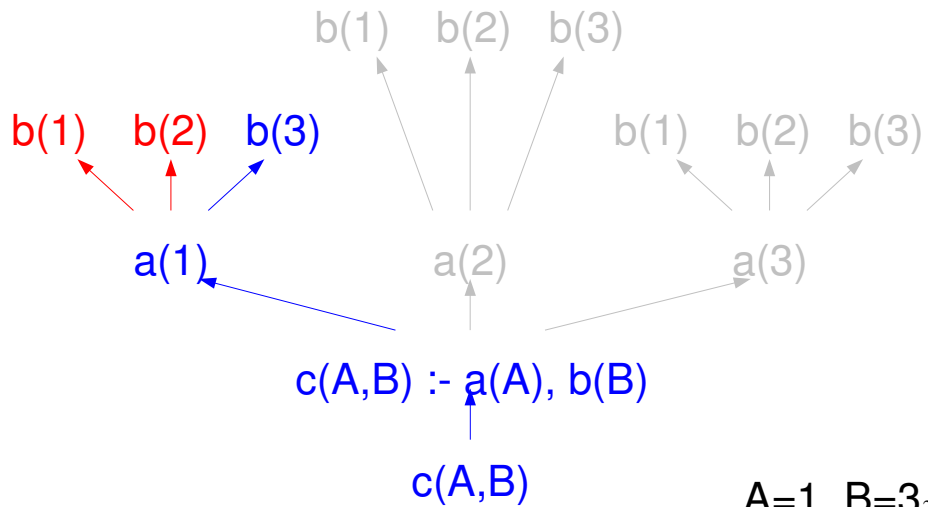
`c(A,B) :- a(A), b(B)`



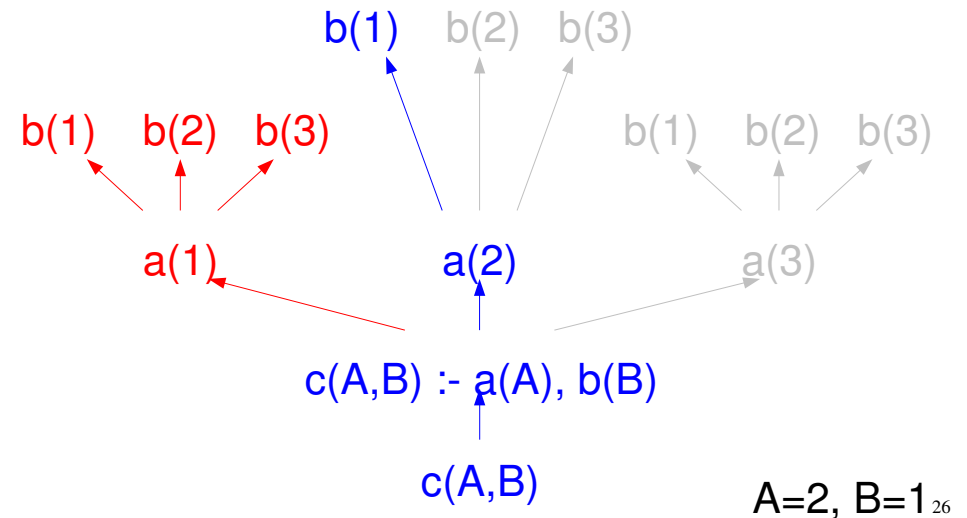
Backtracking is used to find other solutions



$c(A,B) \text{ :- } a(A), b(B)$



$c(A,B) \text{ :- } a(A), b(B)$



Take from a list

Here is a program which takes an element from a list

$\text{take}([H|T], H, T).$

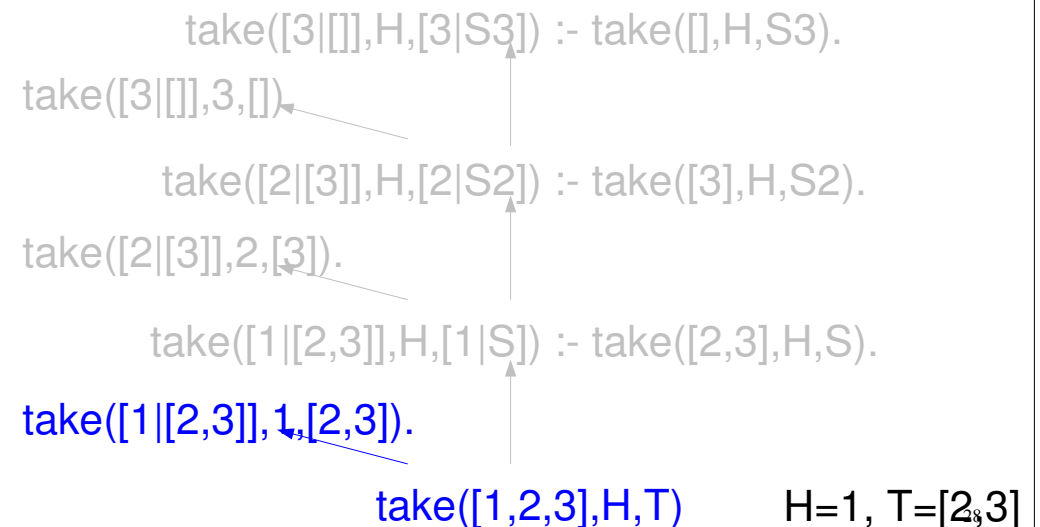
$\text{take}([H|T], R, [H|S]) \text{ :- } \text{take}(T, R, S).$

What does prolog do when given the query:

$\text{take}([1,2,3], H, T).$

27

$\text{take}([1,2,3], H, T).$



Backtrack to get the next answer

```

take([3|[]],H,[3|S3]) :- take([],H,S3).
take([3|[]],3,[])
take([2|[3]],H,[2|S2]) :- take([3],H,S2).
take([2|[3]],2,[3]).
take([1|[2,3]],H,[1|S]) :- take([2,3],H,S).  S=[3]
take([1|[2,3]],1,[2,3]).
take([1,2,3],H,T)  H=2, T=[1,3]

```

Backtrack again for another answer

```

take([3|[]],H,[3|S3]) :- take([],H,S3).
take([3|[]],3,[])
take([2|[3]],H,[2|S2]) :- take([3],H,S2).  S=[]
take([2|[3]],2,[3]).
take([1|[2,3]],H,[1|S]) :- take([2,3],H,S).  S=[2]
take([1|[2,3]],1,[2,3]).
take([1,2,3],H,T)  H=3, T=[1,2]

```

Prolog says “no”

```

take([3|[]],H,[3|S3]) :- take([],H,S3).
take([3|[]],3,[])
take([2|[3]],H,[2|S2]) :- take([3],H,S2).
take([2|[3]],2,[3]).
take([1|[2,3]],H,[1|S]) :- take([2,3],H,S).  No
take([1|[2,3]],1,[2,3]).
take([1,2,3],H,T)  No

```

List permutations is very elegant

```

perm([],[]).
perm(List,[H|T]) :- take(List,H,R), perm(R,T).

```

Dutch national flag

[red,white,blue,white,red]



[red,red,white,white,blue]

Take a list and re-order such that red precedes white precedes blue

33

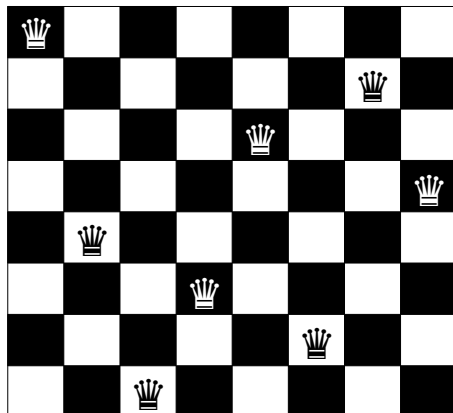
Generate and Test is a technique for solving problems like this

- 1) Generate a solution
- 2) Test if its valid
- 3) If not valid then backtrack to next solution

flag(In,Out) :- perm(In,Out), checkColour(Out).

34

Place 8 Queens so that none can take any other



[1 , 5 , 8 , 6 , 3 , 7 , 2 , 4]

35

Generate and Test works for 8 Queens too

8queens(R) :- perm([1,2,3,4,5,6,7,8],R),
checkDiagonals(R).

36

Anagrams

Load the dictionary into the prolog database e.g.:

`word([a,a,r,d,v,a,r,k]).`

Generate permutations of the input word and **test** if they are words from the dictionary

or

Generate words from the dictionary and **test** if they are a permutation!

<http://www.cl.cam.ac.uk/~acr31/anagram.pl>

Lecture 3

- Symbolic evaluation of arithmetic
- Controlling backtracking: cut
- Negation

1

Symbolic Evaluation

Let's write some prolog rules to evaluate
symbolic arithmetic expressions such as
`plus(1,mult(4,5))`

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.  
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 * B1.  
eval(A,A).
```

2

Evaluation starts with the first matching clause

Q: How does prolog evaluate:

`eval(plus(1,mult(4,5)),Ans)`

A: use the first matching clause to see if its true

`eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.`

In this case: $A = 1$, $B = \text{mult}(4,5)$ and $C = \text{Ans}$

3

Next it looks at the body of the rule

`eval(1,A1), eval(mult(4,5),B1), Ans is A1 + B1`

4

The body is checked from left to right
First part of the body: eval(1,A1)

Try: eval(plus(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 + B1.

Fail because 1 does not unify with plus(A,B)

Try: eval(mult(A,B),C) :- eval(A,A1), eval(B,B1), C is A1 * B1.

Fail because 1 does not unify with mult(A,B)

Try: eval(A,A).

Succeed: eval(1,A1) is true if $A1 = 1$

5

The body is checked from left to right

eval(1,1), eval(mult(4,5),B1), Ans is 1 + B1

6

The body is checked from left to right

eval(1,1), eval(mult(4,5),20), Ans is 1 + 20

7

The body is checked from left to right

eval(1,1), eval(mult(4,5),20), 21 is 1 + 20

8

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 * B1.
eval(A,A).
```

$\text{eval}(\text{plus}(1, \text{mult}(4, 5)), \text{Ans}) \text{ :- eval}(1, T1), \text{eval}(\text{mult}(4, 5), T2), \text{Ans is } T1 + T2$
 $\text{eval}(\text{plus}(A, B), C) \text{ :- eval}(A, A1), \text{eval}(B, B1), C \text{ is } A1 + B1$

9

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 * B1.
eval(A,A).
```

CHOICE POINT

$\text{eval}(\text{plus}(1, \text{mult}(4, \blacktriangledown)), \text{Ans}) \text{ :- eval}(1, T1), \text{eval}(\text{mult}(4, 5), T2), \text{Ans is } T1 + T2$
 $\text{eval}(\text{plus}(A, B), C) \text{ :- eval}(A, A1), \text{eval}(B, B1), C \text{ is } A1 + B1$

10

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 * B1.
eval(A,A).
```

$\text{eval}(1, 1).$
 $\text{eval}(A, A).$

$\text{eval}(\text{plus}(1, \text{mult}(4, 5)), \text{Ans}) \text{ :- eval}(1, T1), \text{eval}(\text{mult}(4, 5), T2), \text{Ans is } T1 + T2$
 $\text{eval}(\text{plus}(A, B), C) \text{ :- eval}(A, A1), \text{eval}(B, B1), C \text{ is } A1 + B1$

11

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                      C is A1 * B1.
eval(A,A).
```

$\text{eval}(\text{mult}(4, 5), T2) \text{ :- eval}(4, T3), \text{eval}(5, T4), T2 \text{ is } T3 * T4.$
 $\text{eval}(\text{mult}(A, B), C) \text{ :- eval}(A, A1), \text{eval}(B, B1), C \text{ is } A1 * B1.$

$\text{eval}(1, 1).$
 $\text{eval}(A, A).$

$\text{eval}(\text{plus}(1, \text{mult}(4, 5)), \text{Ans}) \text{ :- eval}(1, T1), \text{eval}(\text{mult}(4, 5), T2), \text{Ans is } T1 + T2$
 $\text{eval}(\text{plus}(A, B), C) \text{ :- eval}(A, A1), \text{eval}(B, B1), C \text{ is } A1 + B1$

12

```

eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).

```

```

eval(4,4).
eval(A,A).

```

```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4.
eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.

```

```

eval(A,A).

```

```

eval(1,1).
eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1

```

```

eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans)

```

13

```

eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).

```

```

eval(5,5).
eval(A,A).

```

```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4.
eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.

```

```

eval(A,A).

```

```

eval(1,1).
eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1

```

```

eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans)

```

14

```

eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).

```

```

20 is 5 * 4.
eval(5,5).
eval(A,A).

```

```

eval(4,4).
eval(A,A).

```

```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4.
eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.

```

```

eval(A,A).

```

```

eval(1,1).
eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1

```

```

eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans)

```

15

```

eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).

```

```

21 is 1 + 20.
20 is 5 * 4.
eval(5,5).
eval(A,A).

```

```

eval(4,4).
eval(A,A).

```

```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4.
eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.

```

```

eval(A,A).

```

```

eval(1,1).
eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1

```

```

eval(A,A).

```

```

eval(plus(1,mult(4,5)),Ans)

```

16

What happens if we use backtracking and ask Prolog for the next solution?

17

21 is 1 + 20.

20 is 5 * 4.

eval(5,5).

eval(A,A).

eval(4,4).

eval(A,A).

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).
```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4

eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

eval(1,1).

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2

eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1

eval(A,A).

eval(plus(1,mult(4,5)),Ans)

18

21 is 1 + 20.

20 is 5 * 4.

eval(5,5).

eval(A,A).

eval(4,4).

eval(A,A).

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).
```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4

eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

eval(1,1).

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2

eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1

eval(A,A).

eval(plus(1,mult(4,5)),Ans)

19

21 is 1 + 20.

20 is 5 * 4.

eval(5,5).

eval(A,A).

eval(4,4).

eval(A,A).

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).
```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4

eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.

eval(A,A).

eval(1,1).

eval(A,A).

eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2

eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1

eval(A,A).

eval(plus(1,mult(4,5)),Ans)

20

21 is 1 + 20.
 20 is 5 * 4.
 eval(5,5).
 eval(A,A).
 eval(4,4).
 eval(A,A).

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).
```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4.
 eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.
 eval(1,1).
 eval(A,A).
 eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
 eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1
 eval(plus(1,mult(4,5)),Ans)

21

21 is 1 + 20.
 20 is 5 * 4.
 eval(5,5).
 eval(A,A).
 eval(4,4).
 eval(A,A).

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).
```

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4.
 eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.
 eval(1,1).
 eval(A,A).
 eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
 eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1
 eval(plus(1,mult(4,5)),Ans)

22

21 is 1 + 20.
 20 is 5 * 4.
 eval(5,5).
 eval(A,A).
 eval(4,4).
 eval(A,A).

Ans is 1 + mult(4,5)

eval(mult(4,5),T2) :- eval(4,T3),eval(5,T4), T2 is T3 * T4.
 eval(mult(A,B),C) :- eval(A,A1),eval(B,B1), C is A1 * B1.
 eval(1,1).
 eval(A,A).
 eval(plus(1,mult(4,5)),Ans) :- eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
 eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),C is A1 + B1
 eval(plus(1,mult(4,5)),Ans)

23

Eliminate spurious solutions by making your clauses orthogonal

- We need to eliminate the choice point
- The first way to do this is to make sure only one clause matches: **eval(A,A)** becomes **eval(gnd(A),A)**.

```
eval(plus(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(gnd(A),A).
```

24

Eliminate spurious solutions by explicitly discarding choice points

- Alternatively we can tell Prolog to commit to its first choice and discard the choice point
- We do this with the cut operator. Written: !

```
eval(plus(A,B),C) :- !,eval(A,A1), eval(B,B1),
                    C is A1 + B1.
eval(mult(A,B),C) :- !,eval(A,A1), eval(B,B1),
                    C is A1 * B1.
eval(A,A).
```

25

21 is 1 + 20.

20 is 5 * 4.

eval(5,5).

eval(A,A).

eval(4,4).

eval(A,A).

Cut removes choice points

These choices are eliminated

eval(mult(4,5),T2) :- !,eval(4,T3),eval(5,T4), T2 is T3 * T4.
eval(mult(A,B),C) :- !,eval(A,A1),eval(B,B1), C is A1 * B1.

CUT

eval(1,1).
eval(A,A).

eval(plus(1,mult(4,5)),Ans) :- !,eval(1,T1), eval(mult(4,5),T2), Ans is T1 + T2
eval(plus(A,B),C) :- !,eval(A,A1), eval(B,B1),C is A1 + B1

eval(plus(1,mult(4,5)),Ans)

26

Cutting out Choice

Whenever Prolog evaluates a cut it discards all choice points back to the parent clause

An example:

```
a(1).          c(A,B,C) :- a(A),d(B,C).
a(2).          c(A,B,C) :- b(A),d(B,C).
a(3).          d(B,C) :- a(B),!,a(C).
b(apple).      d(B,_ ) :- b(B).
b(orange).
```

27

```
a(1).
a(2).
a(3).
b(apple).
b(orange).
c(A,B,C) :- a(A),d(B,C).
c(A,B,C) :- b(A),d(B,C).
d(B,C) :- a(B),!,a(C).
d(B,_ ) :- b(B).
```

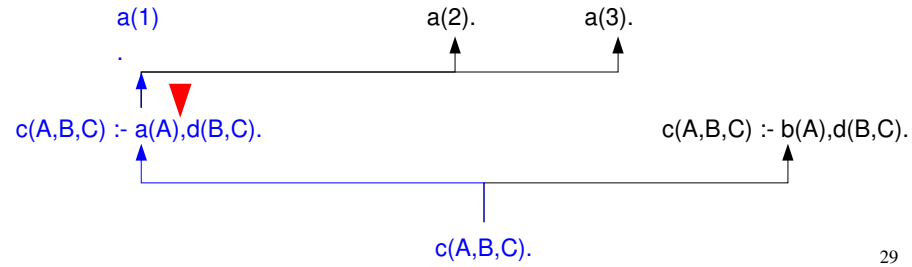
c(A,B,C) :- a(A),d(B,C).

c(A,B,C) :- b(A),d(B,C).

c(A,B,C).

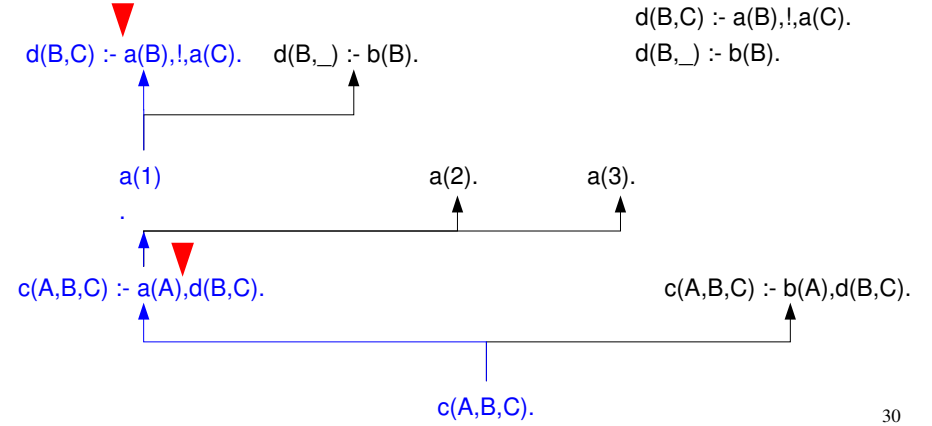
28

a(1).
 a(2).
 a(3).
 b(apple).
 b(orange).
 c(A,B,C) :- a(A),d(B,C).
 c(A,B,C) :- b(A),d(B,C).
 d(B,C) :- a(B),!,a(C).
 d(B,_) :- b(B).



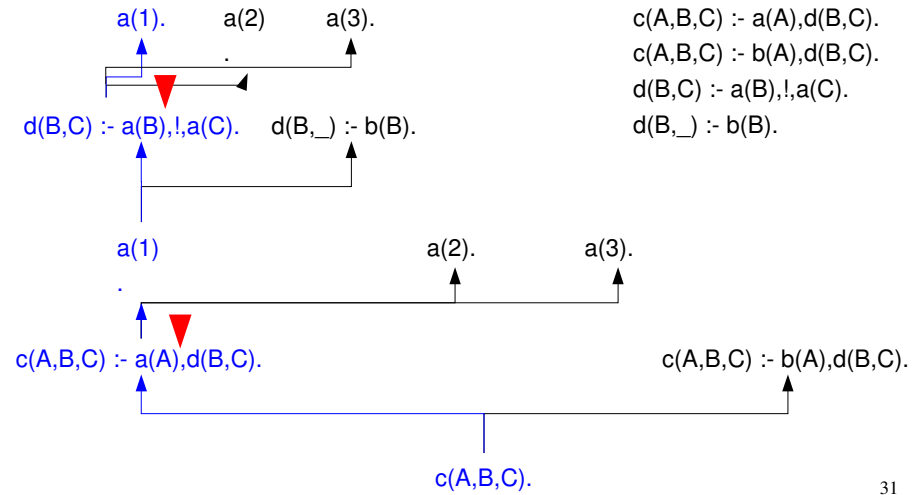
29

a(1).
 a(2).
 a(3).
 b(apple).
 b(orange).
 c(A,B,C) :- a(A),d(B,C).
 c(A,B,C) :- b(A),d(B,C).
 d(B,C) :- a(B),!,a(C).
 d(B,_) :- b(B).



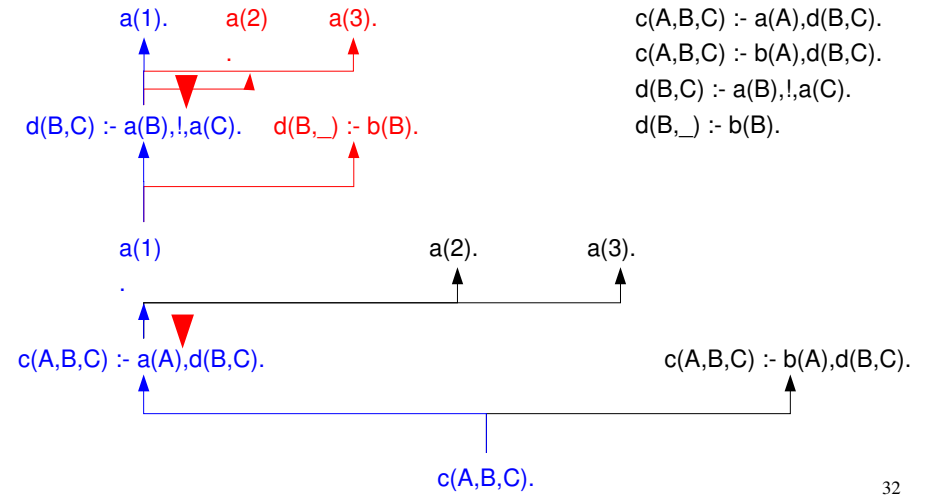
30

a(1).
 a(2).
 a(3).
 b(apple).
 b(orange).
 c(A,B,C) :- a(A),d(B,C).
 c(A,B,C) :- b(A),d(B,C).
 d(B,C) :- a(B),!,a(C).
 d(B,_) :- b(B).

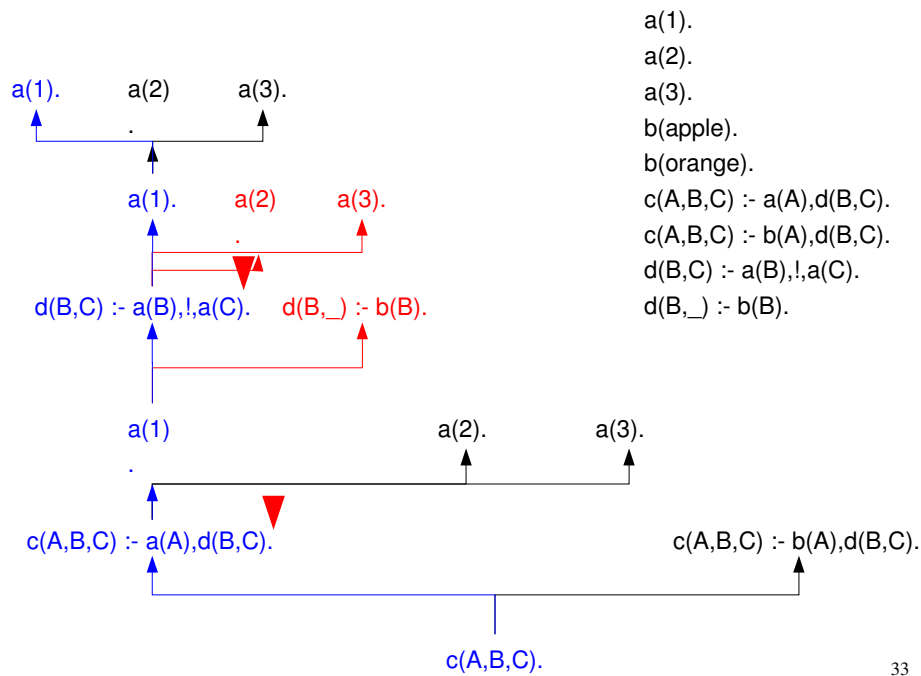


31

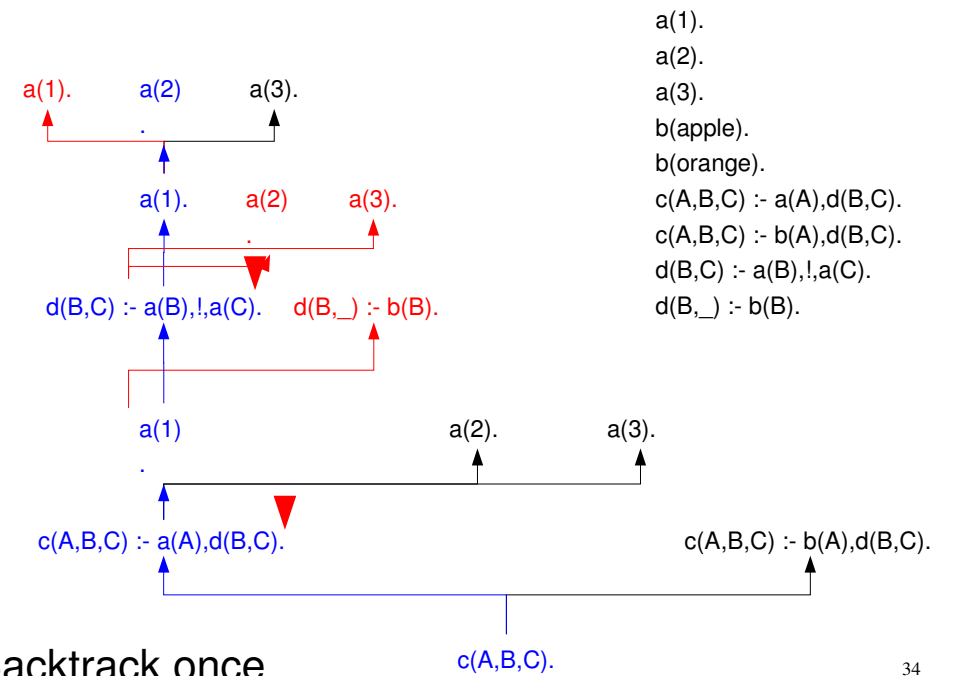
a(1).
 a(2).
 a(3).
 b(apple).
 b(orange).
 c(A,B,C) :- a(A),d(B,C).
 c(A,B,C) :- b(A),d(B,C).
 d(B,C) :- a(B),!,a(C).
 d(B,_) :- b(B).



32

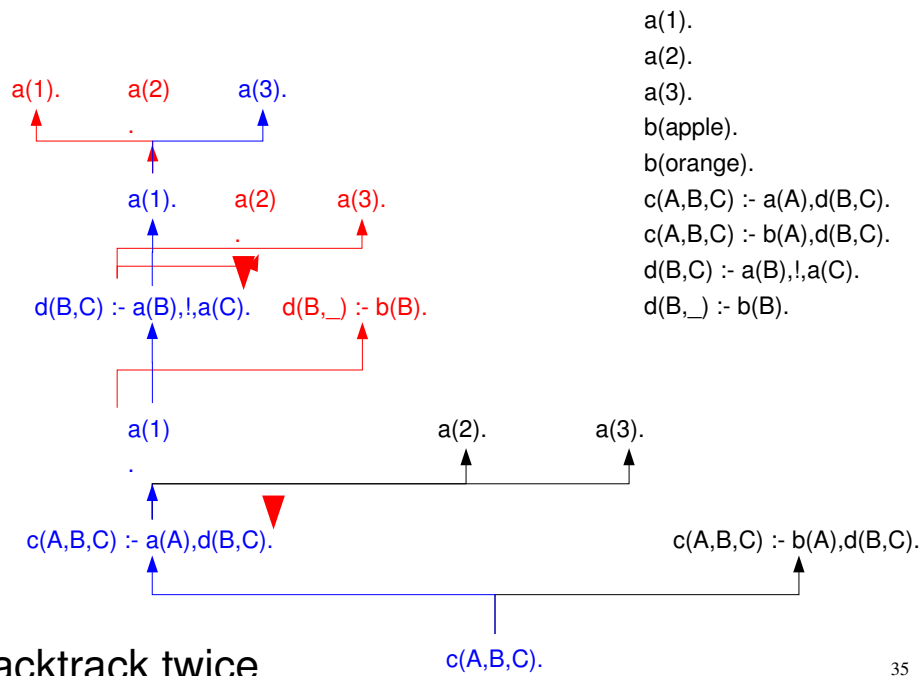


33



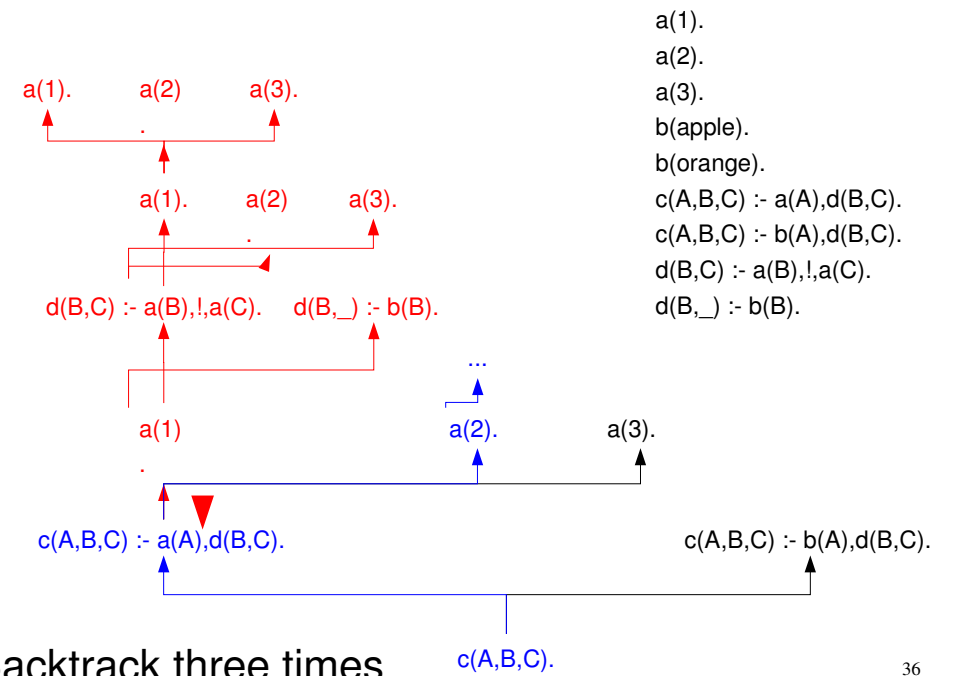
Backtrack once

34



Backtrack twice

35



Backtrack three times

36

Cut can change the logical meaning of your program

$p \text{ :- } a, b.$
 $p \text{ :- } c.$
 $p \Leftrightarrow (a \wedge b) \vee c$

$p \text{ :- } a, !, b.$
 $p \text{ :- } c.$
 $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

This is a **red** cut - **DANGER!**

37

Cut can be used for efficiency reasons

$\text{split}([], [], []).$
 $\text{split}([H|T], [H|L], R) \text{ :- } H < 5, \text{split}(T, L, R).$
 $\text{split}([H|T], L, [H|R]) \text{ :- } H \geq 5, \text{split}(T, L, R).$

If the second clause succeeds the third cannot
-> we don't need to keep a choice point
-> the interpreter cannot infer this

38

Cut can be used for efficiency reasons

$\text{split}([], [], []).$
 $\text{split}([H|T], [H|L], R) \text{ :- } H < 5, !, \text{split}(T, L, R).$
 $\text{split}([H|T], L, [H|R]) \text{ :- } H \geq 5, \text{split}(T, L, R).$

Add a cut to make the orthogonality explicit

This is a **green** cut – it makes things go better

39

We could go one step further at the expense of readability

$\text{split}([], [], []).$
 $\text{split}([H|T], [H|L], R) \text{ :- } H < 5, !, \text{split}(T, L, R).$
 $\text{split}([H|T], L, [H|R]) \text{ :- } \text{split}(T, L, R).$

The comparison in the third clause is no longer necessary

- but each clause no longer stands on its own
- stylistic preference – I avoid doing this

40

Cut gives us more expressive power

```
isDifferent(A,A) :- !,fail.  
isDifferent(_,_).
```

isDifferent(A,B) is true if A and B do not unify

41

Implementing “not”

```
not(A) :- A,!fail.  
not(_).
```

not(A) is true if A cannot be shown to be true
This is *negation by failure*

42

Negation by Failure

Negation by failure is based on the *closed world assumption*

Everything which is true is stated (or can be derived from) the clauses in the program

43

Negation Example

```
good_food(theWrestlers).  
good_food(theCambridgeLodge).  
expensive(theCambridgeLodge).
```

```
bargain(R) :- good_food(R), not(expensive(R)).
```

44

Negation Example


```
bargain(R) :- good_food(R), not(expensive(R)).
```

we can ask: bargain(R) and Prolog replies:
R = theWrestlers

45

Negation Gotcha!

swapped round



```
bargain(R) :- not(expensive(R)), good_food(R).
```

we can ask: bargain(R) and Prolog replies:
no

46

Why?

```
bargain(R) :- not(expensive(R)), good_food(R).
```

Prolog first tries to find an R such that
expensive(R) is true, and therefore
not(expensive(R)) will fail if there are *any*
expensive restaurants

47

Databases

Store information as tuples in the Prolog database

```
tName(acr31,'Andrew Rice').  
tName(arb33,'Alastair Beresford').  
  
tGrade(acr31,'IA',2.1).  
tGrade(acr31,'IB',1).  
tGrade(acr31,'II',1).  
tGrade(arb33,'IA',2.1).  
tGrade(arb33,'IB',1).  
tGrade(arb33,'II',1).
```

48

Databases

We can now write a program to find all names:

`qName(N) :- tName(_,N).`

Databases

Or a program to find the full name and all grades for
acr31.

`qGrades(F,G) :- tName(C,F), tGrade(C,G).`

Lecture 4

- Playing Countdown
- Iterative deepening
- Search

1

Countdown Numbers

- Select 6 of 24 numbers tiles
 - large numbers: 25,50,75,100
 - small numbers: 1,2,3...10 (two of each)
- Contestant chooses how many large and small
- Randomly chosen 3-digit target number
- Get as close as possible using each of the 6 numbers at most once and the operations of addition, subtraction, multiplication and division
- No floats or fractions allowed

2

Countdown Numbers

- Strategy – generate and test
 - maintain a list of symbolic arithmetic terms
 - initially this list consists of ground terms e.g.:
[gnd(25),gnd(6),gnd(3),gnd(3),gnd(7),gnd(50)]
 - if the head of the list evaluates to the total then succeed
 - otherwise pick two of the elements, combine them using one of the available arithmetic operations, put the result on the head of the list, and repeat

3

Countdown Numbers

- Prerequisites
 - **eval(A,B)** – true if the symbolic expression A evaluates to B
 - **choose(N,L,R,S)** – true if R is the result of choosing N items from L and S is the remaining items left in L
 - **arithop(A,B,C)** – true if C is a valid combination of A and B
 - e.g. arithop(A,B,plus(A,B)).

4

Countdown Numbers

%%% arith_op(+A, +B, -C)

%%% unify C with a valid binary operation of expressions A and B

arithop(A,B,plus(A,B)).

% minus is not commutative

arithop(A,B,minus(A,B)) :- eval(A,D), eval(B,E), D>E.

arithop(B,A,minus(A,B)) :- eval(A,D), eval(B,E), D>E.

% don't allow mult by 1

arithop(A,B,mult(A,B)) :- eval(A,D), D \== 1, eval(B,E), E \== 1.

% div is not commutative and dont allow div by 0 or 1

arithop(A,B,div(A,B)) :- eval(B,E), E \== 1, E \== 0,
eval(A,D), 0 is D rem E.

arithop(B,A,div(A,B)) :- eval(B,E), E \== 1, E \== 0,
eval(A,D), 0 is D rem E.

5

Countdown Numbers

countdown([Soln|_],Target,Soln) :-
eval(Soln,Target).

countdown(L,Target,Soln) :-
choose(2,L,[A,B],R),
arithop(A,B,C),
countdown([C|R],Target,Soln).

6

Closest Solution

If there are no solutions we want to find the closest solution

solve([Soln|_],Target,Soln,D) :- eval(Soln,R), diff(Target,R,D).

solve(L,Target,Soln,D) :- choose(2,L,[A,B],R),

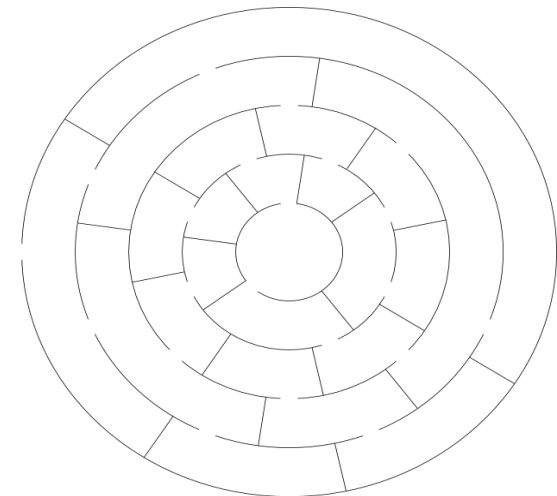
arithop(A,B,C),

solve([C|R],Target,Soln,D).

solve(L,Target,Soln) :- range(0,100,D), solve(L,Target,Soln,D).

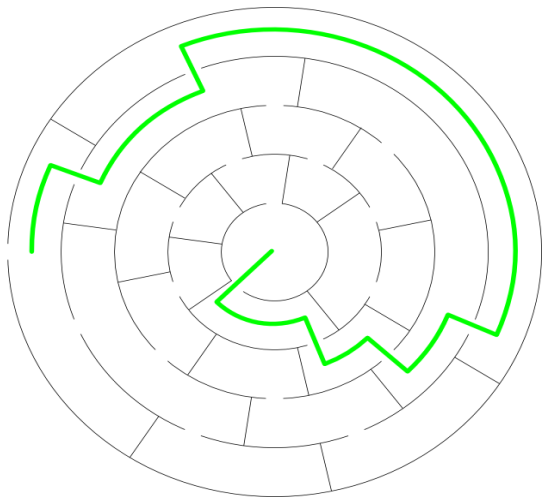
This is *iterative deepening* – wait until the Artificial Intelligence course⁷

Searching



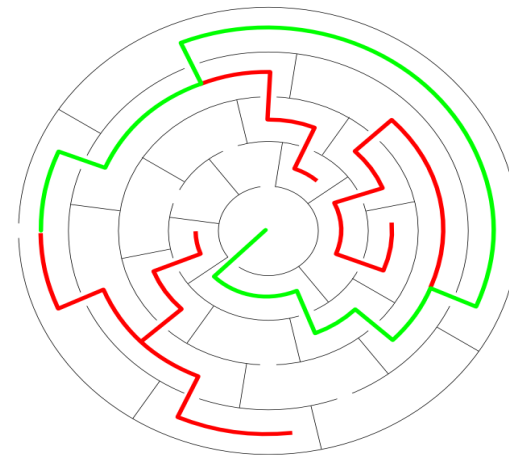
8

Searching



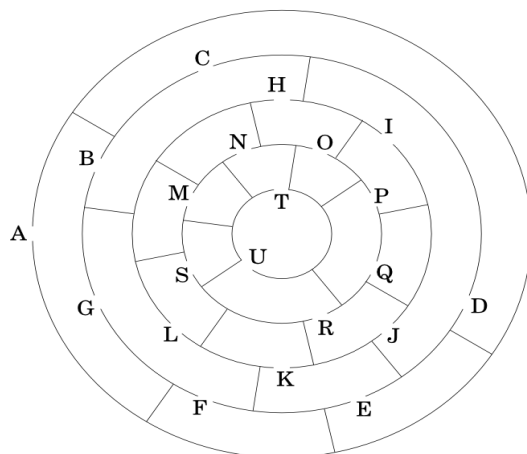
9

Searching



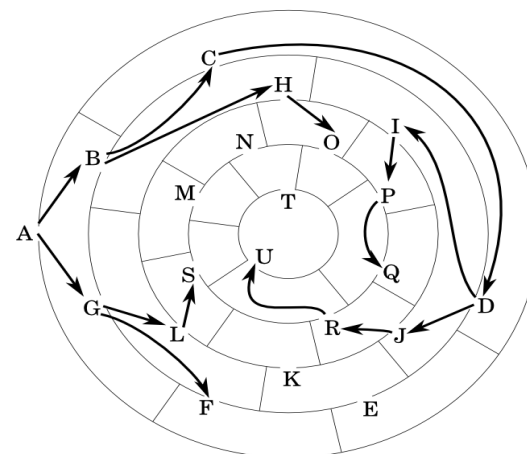
10

Searching



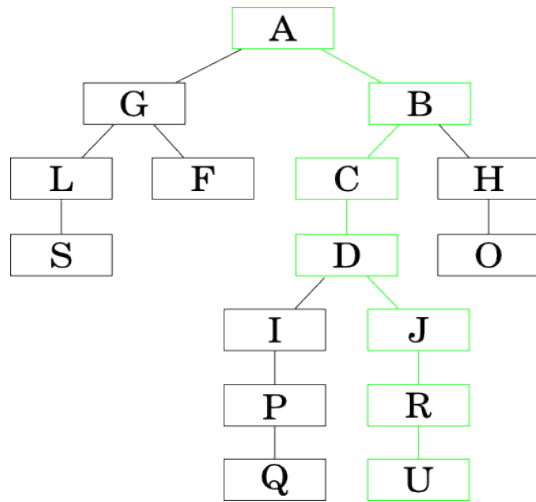
11

Searching



12

Searching



13

Searching

```
route(a,g).
route(g,l).                start(a).
route(l,s).                finish(u).
...
travel(A,A).
travel(A,C) :- route(A,B),travel(B,C).

solve :- start(A),finish(B), travel(A,B).
```

14

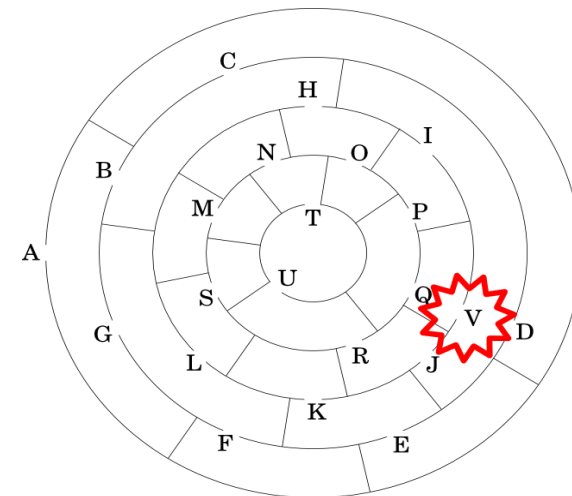
We need to remember the route

```
travellog(A,A,[]).
travellog(A,C,[A-B|Steps]) :-
    route(A,B), travellog(B,C,Steps).

solve(L) :- start(A), finish(B), travellog(A,B,L).
```

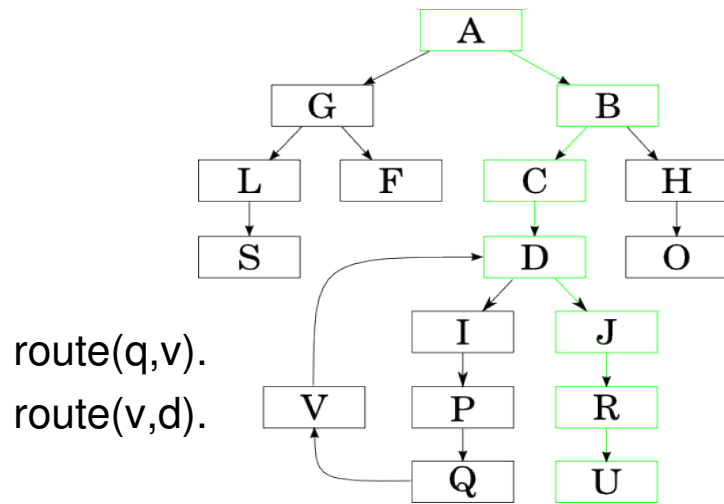
15

Cyclic Graphs



16

Cyclic Graphs



17

Searching

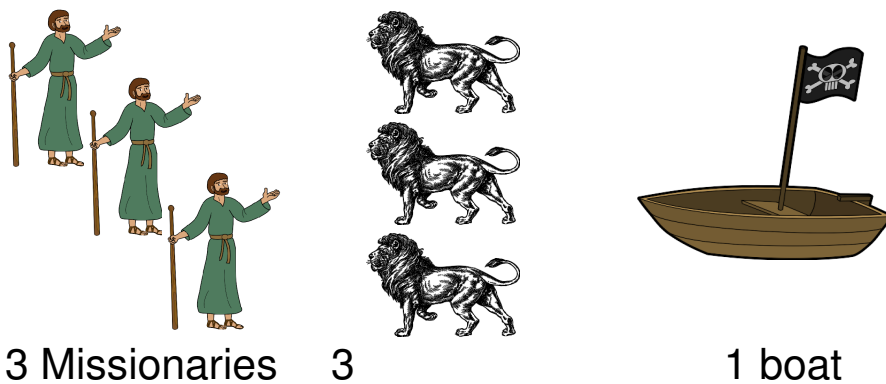
Solution: maintain a set of places we've already been
– the closed set

```

travelsafe(A,A,_).
travelsafe(A,C,Closed) :-
    route(A,B),
    \+member(B,Closed),
    travelsafe(B,C,[B|Closed]).
  
```

18

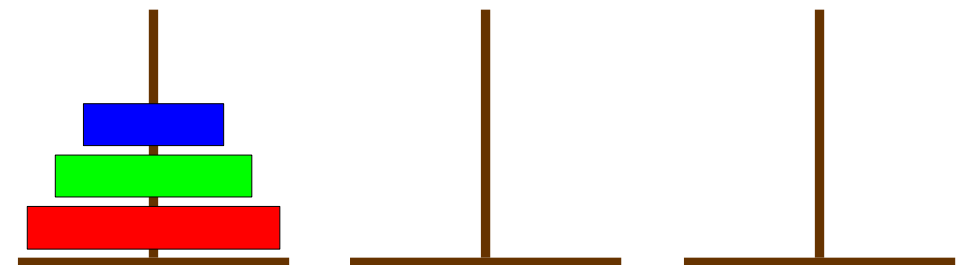
Missionaries and Cannibals



The boat carries 2 people
If the Cannibals outnumber the Missionaries they will eat them
Get them all from one side of the river to the other?

19

Towers of Hanoi



20



Umbrella problem

A group of 4 people, Andy, Brenda, Carl, & Dana, arrive in a car near a friend's house, who is having a large party. It is raining heavily, & the group was forced to park around the block from the house because of the lack of available parking spaces due to the large number of people at the party. The group has only 1 umbrella, & agrees to share it by having Andy, the fastest, walk with each person into the house, & then return each time. It takes Andy 1 minute to walk each way, 2 minutes for Brenda, 5 minutes for Carl, & 10 minutes for Dana. It thus appears that it will take a total of 19 minutes to get everyone into the house. However, Dana indicates that everyone can get into the house in 17 minutes by a different method. How? The individuals must use the umbrella to get to & from the house, & only 2 people can go at a time (& no funny stuff like riding on someone's back, throwing the umbrella, etc.).

Lecture 5

- Difference structures
- Difference lists
- Notational fun
- Appendless append

Appending two Lists

```
append([],L,L).
append([X|T],L,[X|R]) :- append(T,L,R).
```

1

2

```
append([],L,L).
append([X|T],L,[X|R]) :- append(T,L,R).
```

```

append([], [3,4], [3,4]).
  ↑
append([2|[]], [3,4], [2|T2]) :- append([], [3,4], T2).
  ↑
append([1|[2]], [3,4], [1|T1]) :- append([2], [3,4], T1).
  ↑
append([1,2], [3,4], A).

```

T2=[3,4]
T1=[2|T2]
A = [1|T1]

3

Difference Lists

Store two lists, the Difference List is the difference between them

We might represent **[1,2,3]** as

[1,2,3,4,5]-[4,5] or [1,2,3,acr]-[acr] or [1,2,3|X]-X

4

Difference Lists are useful as Partial Structures

Store two lists, the Difference List is the difference between them

We might represent **[1,2,3]** as

~~[1,2,3,4,5]-[4,5]~~ or ~~[1,2,3,acr]-[acr]~~ or [1,2,3|X]-X

5

Difference List Append

$1 :: (2 :: (3 :: []))$

$4 :: (5 :: (6 :: []))$

6

Difference List Append

$1 :: (2 :: (3 :: []))$

↙
 $4 :: (5 :: (6 :: []))$

7

Difference List Append

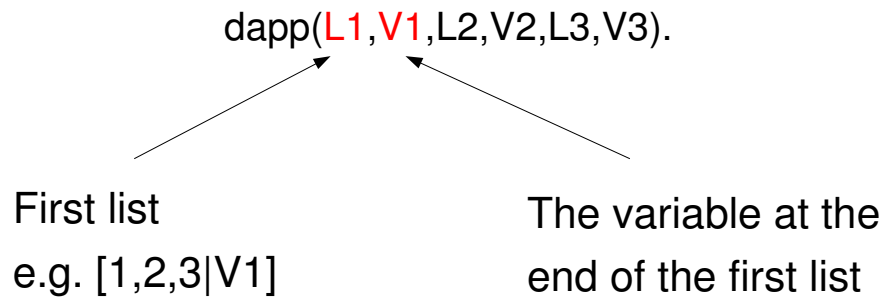
$1 :: (2 :: (3 :: A))$

$4 :: (5 :: (6 :: B))$
A

Prolog syntax for the first list is [1,2,3|A]

8

Difference List Append



9

Difference List Append

dapp(L1-V1, L2-V2, L3-V3)

By convention we write our difference list pair as

A-B

But we could also write

differenceList(A,B) or **A+B** or **A*B**

10

Difference List Append (implementation)

L1 $L1_0 :: L1_1 :: \dots :: L1_n :: [V1]$

L2 $L2_0 :: L2_1 :: \dots :: L2_n :: [V2]$

L3 $L1_0 :: \dots :: L1_n :: L2_0 :: \dots :: L2_n :: [V2]$

dapp(L1-V1, L2-V2, L3-V3) :- V1=L2, L3=L1, V3=V2.

11

Difference List Append (implementation)

dapp(L1-V1, L2-V2, L3-V3) :- V1=L2, L3=L1, V3=V2.

We know that **V1** and **L2** must be the same so
replace all instances of **V1** and **L2** with a new
variable **B**

dapp(L1-B, B-V2, L3-V3) :- B=B, L3=L1, V3=V2.

12

Difference List Append (implementation)

$\text{dapp}(\text{L1-B}, \text{B-V2}, \text{L3-V3}) :- \text{L3}=\text{L1}, \text{V3}=\text{V2}.$

We know that **L3** and **L1** must be the same so replace all instances of **L3** and **L1** with a new variable **A**

$\text{dapp}(\text{A-B}, \text{B-V2}, \text{A-V3}) :- \text{A}=\text{A}, \text{V3}=\text{V2}.$

13

Difference List Append (implementation)

$\text{dapp}(\text{A-B}, \text{B-V2}, \text{A-V3}) :- \text{V3}=\text{V2}.$

We know that **V3** and **V2** must be the same so replace all instances of **V3** and **V2** with a new variable **C**

$\text{dapp}(\text{A-B}, \text{B-C}, \text{A-C}) :- \text{C}=\text{C}.$

14

Difference List Append

$\text{dapp}(\text{A-B}, \text{B-C}, \text{A-C}).$

The technique of making simple substitutions when we know two variables must be equal turns out to apply in lots of situations (and supervision work)

15

Empty Difference Lists

The empty difference list is an empty list with a variable appended to the end for later use:

$[] @ A$

which is equivalent to (simply):

A

we write this in the conventional notation as:

$A-A$

16

Converting to difference lists

```
double([],[]).  
double([H|T],[R|S]) :- R is H*2, double(T,S).
```

17

Convert to difference lists

```
double(A-A,B-B).  
double([H|T]-T1,[R|S]-S1) :-  
    R is H*2,  
    double(T-T1,S-S1).
```

18

Question

What does `double([1,2,3|T]-T,R)` produce?

- a) yes, $R = [2,4,6|X]-X$
- b) no
- c) yes, $R = X-X$
- d) an exception

19

Another Difference List Example

Define a procedure `rotate(X,Y)` where both X and Y are represented by difference lists, and Y is formed by rotating X to the left by one element.

[14 marks]
1996-6-7

20

Write the answer first without Difference Lists

Take the first element off the first list and append it to
the end

```
rotate([H|T],R) :- append(T,[H],R).
```

21

Rewrite with Difference Lists

```
rotate([H|T]-T1,R-S) :- append(T-T1,[H|A]-A,R-S).
```

22

Rename Variables To Get Rid Of Append

```
rotate([H|T]-T1,R-S) :- append(T-T1,[H|A]-A,R-S).
```

rename T1 to be [H|A]

```
rotate([H|T]-[H|A],R-S) :-  
    append(T-[H|A],[H|A]-A,R-S).
```

23

Rename Variables To Get Rid Of Append

```
rotate([H|T]-[H|A],R-S) :-  
    append(T-[H|A],[H|A]-A,R-S).
```

Rename R to be T

```
rotate([H|T]-[H|A],T-S) :-  
    append(T-[H|A],[H|A]-A,T-S).
```

24

Rename Variables To Get Rid Of Append

```
rotate([H|T]-[H|A],T-S) :-  
    append(T-[H|A],[H|A]-A,T-S).
```

Rename S to be A

```
rotate([H|T]-[H|A],T-A) :-  
    append(T-[H|A],[H|A]-A,T-A).
```

25

Final Answer

```
rotate([H|T]-[H|A],T-A).
```

If you have code like this I suggest you comment
it really well!

27

The call to append/3 is now
redundant and we can remove it

```
% difference list append  
append(A-B,B-C,A-C).
```

```
rotate([H|T]-[H|A],T-A) :-  
    append(T-[H|A],[H|A]-A,T-A).
```

26

Towers of Hanoi Revisited

- Move n rings from Src to Dest
 - move n-1 rings from Src to Aux
 - move the nth ring from Src to Dest
 - move n-1 tiles from Aux to Dest
- Base case: move 0 rings from Src to Dest

28

Lecture 6

- A Sudoku solver
- Constraint Logic Programming

1

Playing Sudoku

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

2

Make the problem easier

		4	
	2		
		1	
	3		

3

List permutations model this problem in Prolog

- Each row must be a permutation of [1,2,3,4]
- Each column must be a permutation of [1,2,3,4]
- Each 2x2 box must be a permutation of [1,2,3,4]

4

Represent the board as a list of lists

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

[[A,B,C,D],
[E,F,G,H],
[I,J,K,L],
[M,N,O,P]]

5

Specify constraints with permutation

```
sudoku( [[X11,X12,X13,X14],[X21,X22,X23,X24],  
        [X31,X32,X33,X34],[X41,X42,X43,X44]]) :-  
    %rows  
    perm([X11,X12,X13,X14],[1,2,3,4]), perm([X21,X22,X23,X24],[1,2,3,4]),  
    perm([X31,X32,X33,X34],[1,2,3,4]), perm([X41,X42,X43,X44],[1,2,3,4]),  
    %cols  
    perm([X11,X21,X31,X41],[1,2,3,4]), perm([X12,X22,X32,X42],[1,2,3,4]),  
    perm([X13,X23,X33,X43],[1,2,3,4]), perm([X14,X24,X34,X44],[1,2,3,4]),  
    %boxes  
    perm([X11,X12,X21,X22],[1,2,3,4]), perm([X13,X14,X23,X24],[1,2,3,4]),  
    perm([X31,X32,X41,X42],[1,2,3,4]), perm([X33,X34,X43,X44],[1,2,3,4]).
```

6

Scale up in the obvious way to 3x3

X11	X12	X13	X14	X15	X16	X17	X18	X19
X21	X22	X23	X24	X25	X26	X27	X28	X29
X31	X32	X33	X34	X35	X36	X37	X38	X39
X41	X42	X43	X44	X45	X46	X47	X48	X49
X51	X52	X53	X54	X55	X56	X57	X58	X59
X61	X62	X63	X64	X65	X66	X67	X68	X69
X71	X72	X73	X74	X75	X76	X77	X78	X79
X81	X82	X83	X84	X85	X86	X87	X88	X89
X91	X92	X93	X94	X95	X96	X97	X98	X99

7

Brute-force is impractically slow for this problem

There are very many valid grids:
 $6670903752021072936960 \approx 6.671 \times 10^{21}$

See: <http://www.afjarvis.staff.shef.ac.uk/sudoku/>

8

Prolog programs can be viewed as constraint satisfaction problems

Prolog is limited to the **single equality constraint**
that two terms must unify

We can generalise this to include other types of
constraint

This gives us **Constraint Logic Programming**
(and a means to solve Sudoku problems)

9

Consider variables over domains with constraints

Given:

the set of **variables**

the **domains** of each variable

constraints on these variables

Find:

an **assignment** of values to variables satisfying
the constraints

10

Sudoku can be expressed as constraints

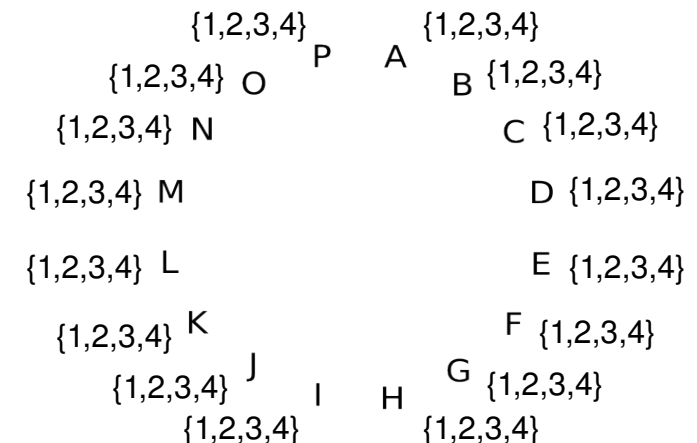
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Variables and Domains

$A \in \{1,2,3,4\}$	$B \in \{1,2,3,4\}$
$C \in \{1,2,3,4\}$	$D \in \{1,2,3,4\}$
$E \in \{1,2,3,4\}$	$F \in \{1,2,3,4\}$
$G \in \{1,2,3,4\}$	$H \in \{1,2,3,4\}$
$I \in \{1,2,3,4\}$	$J \in \{1,2,3,4\}$
$K \in \{1,2,3,4\}$	$L \in \{1,2,3,4\}$
$M \in \{1,2,3,4\}$	$N \in \{1,2,3,4\}$
$O \in \{1,2,3,4\}$	$P \in \{1,2,3,4\}$

11

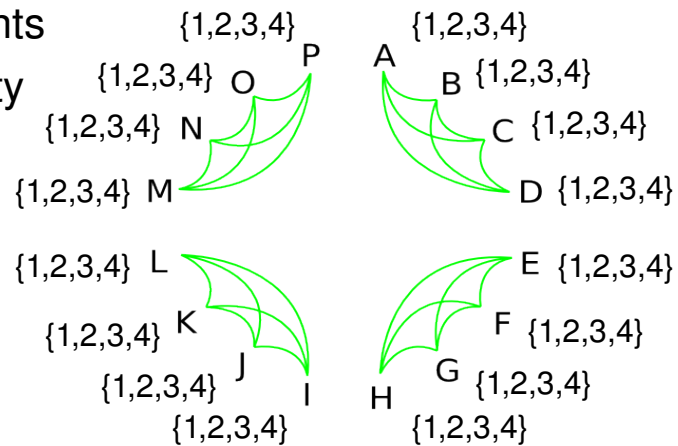
Express Sudoku as a Constraint Graph



12

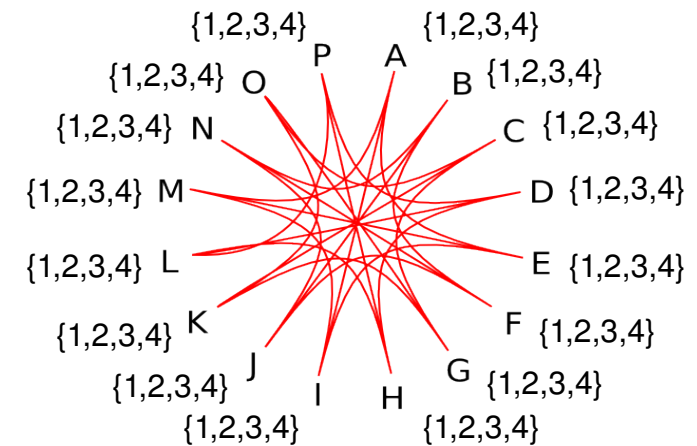
Constraints: All variables in rows are different

Each edge
represents
inequality



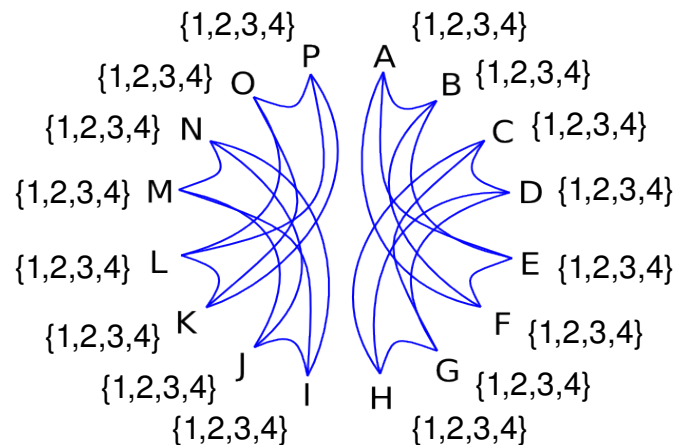
13

Constraints: All variables in columns are different



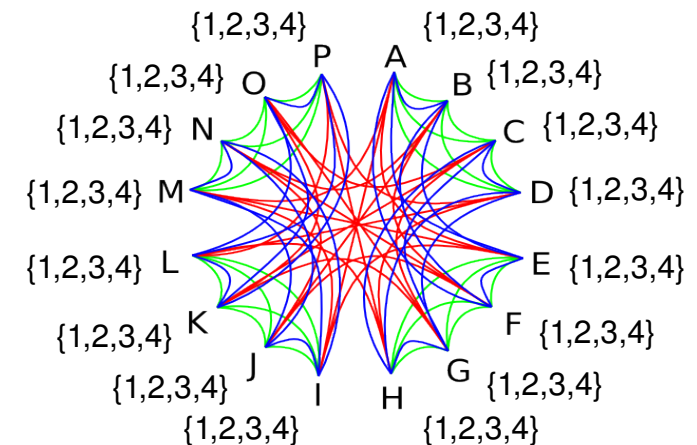
14

Constraints: All variables in boxes are different



15

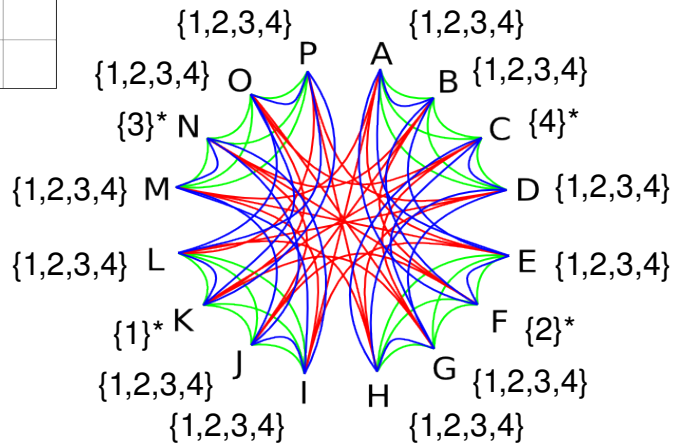
All constraints



16

Reduce domains according to initial values

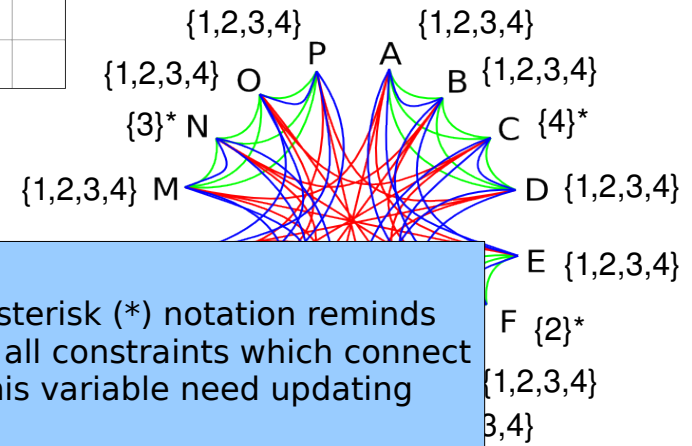
		4	
	2		
		1	
	3		



17

When a domain changes we update its constraints

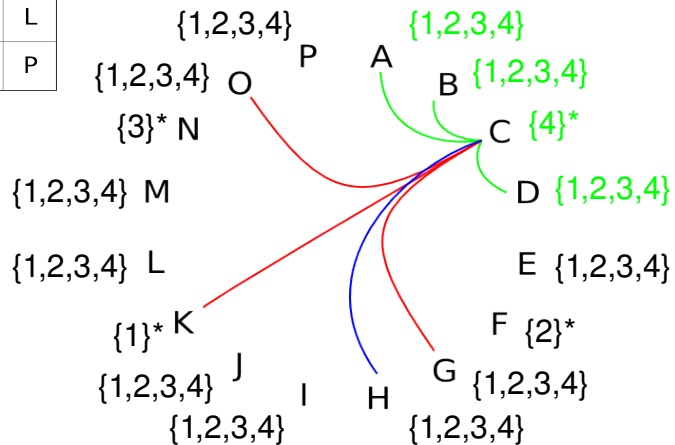
		4	
	2		
		1	
	3		



18

Update constraints connected to C

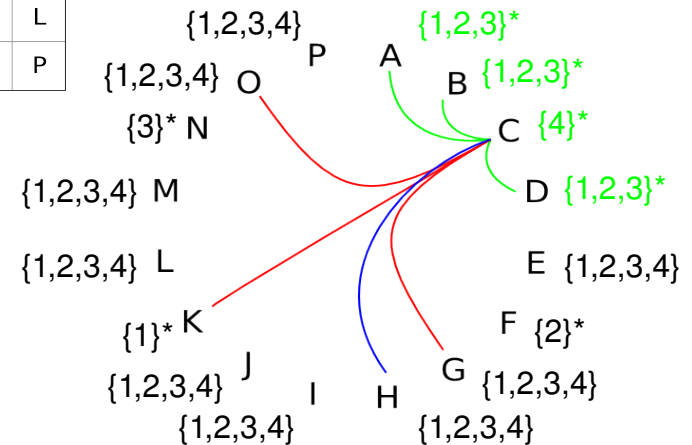
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



19

Update constraints connected to C

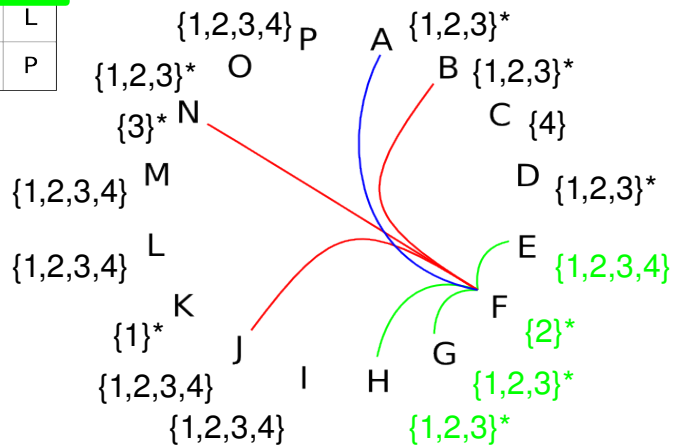
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



20

Update constraints connected to F

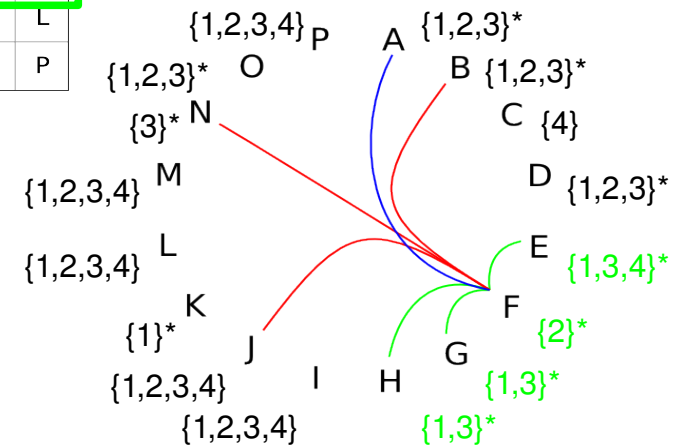
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



25

Update constraints connected to F

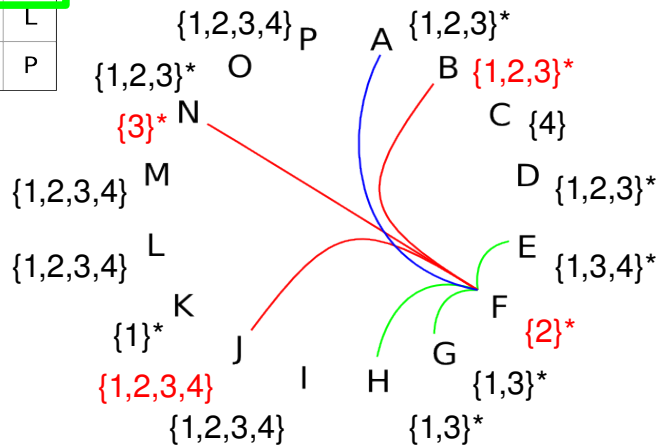
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



26

Update constraints connected to F

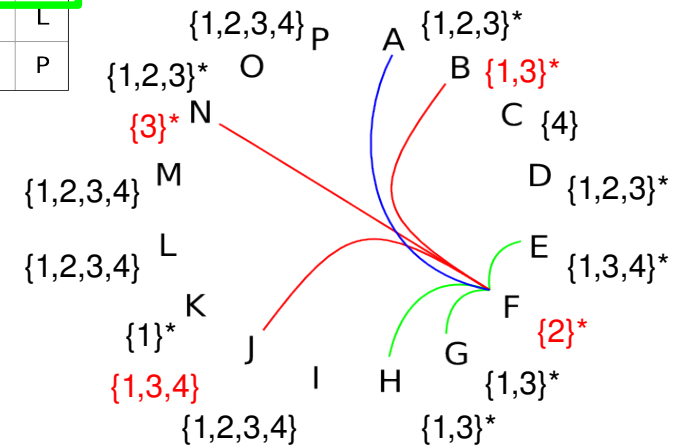
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



27

Update constraints connected to F

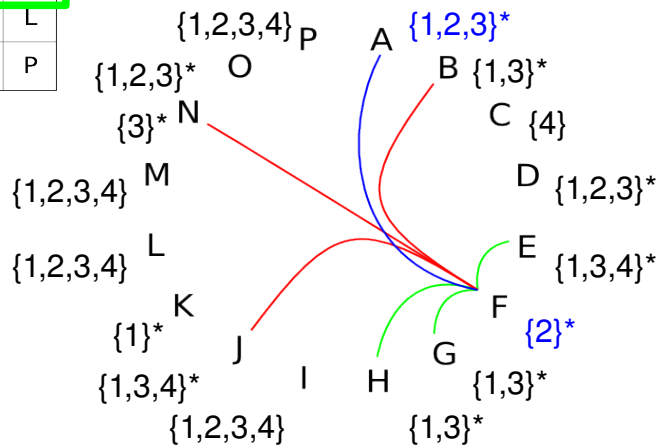
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



28

Update constraints connected to F

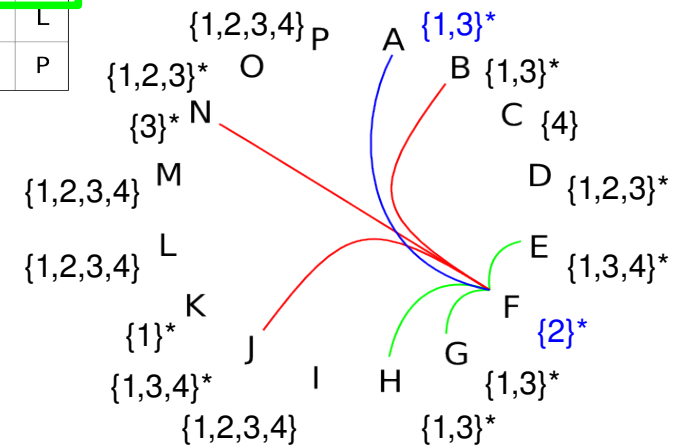
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



29

Update constraints connected to F

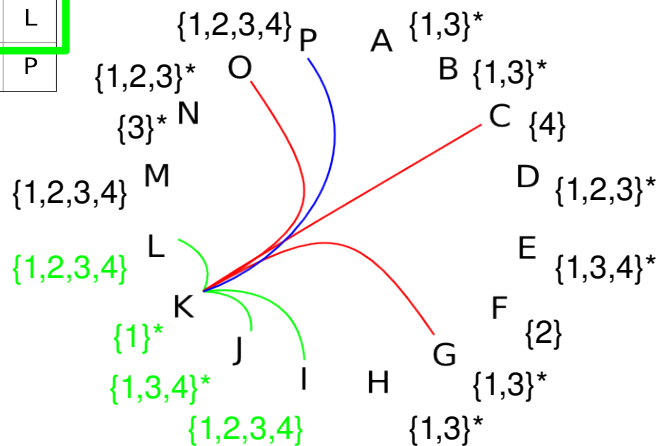
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



30

Update constraints connected to K

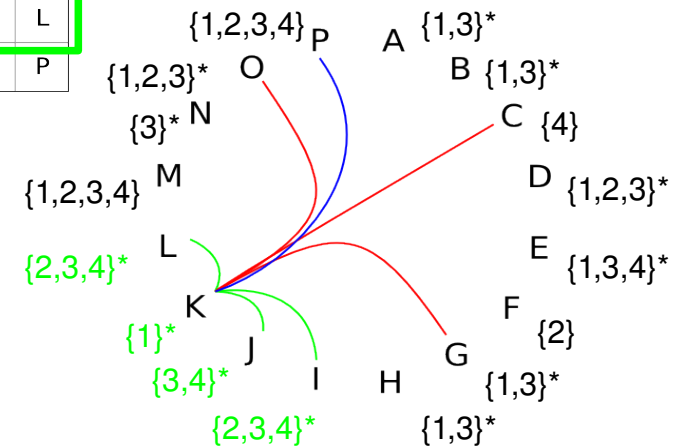
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



31

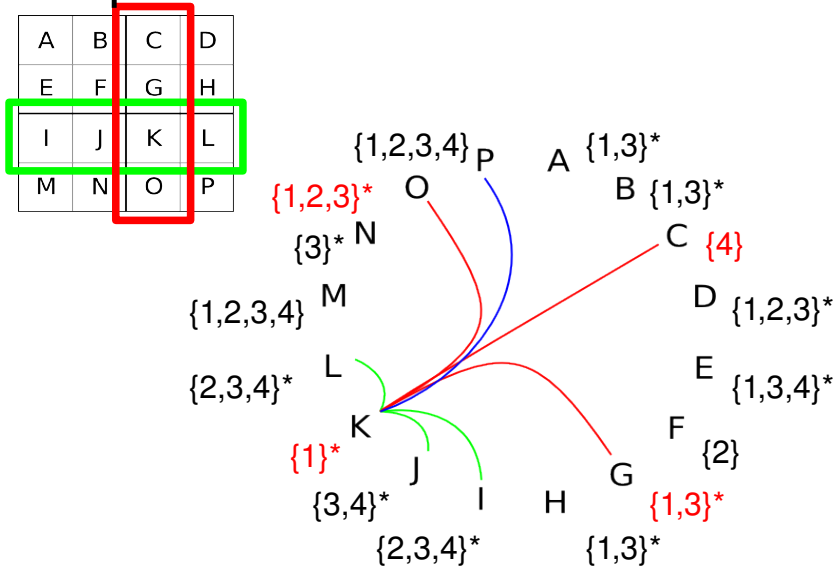
Update constraints connected to K

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



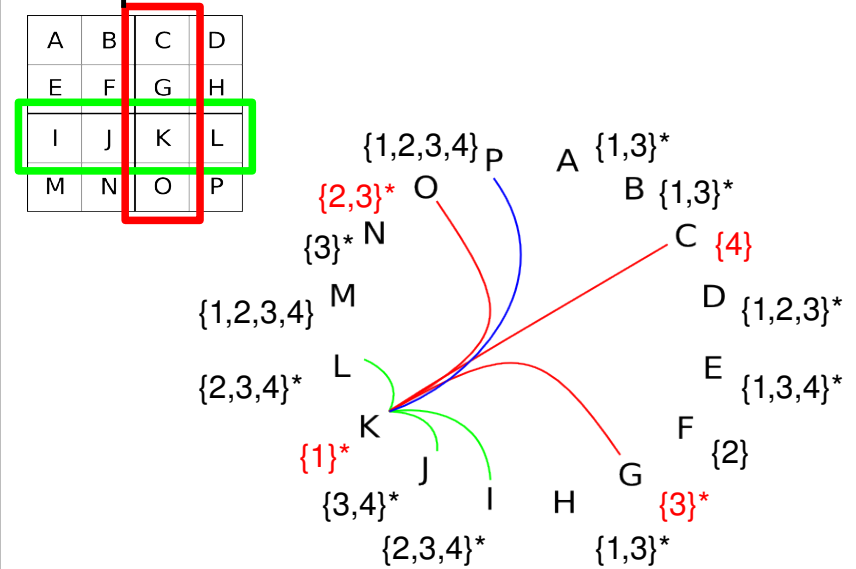
32

Update constraints connected to K



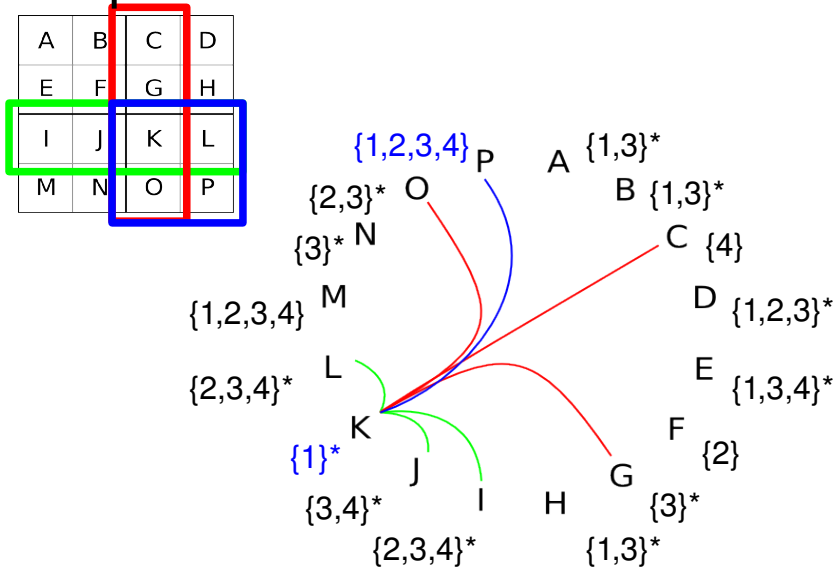
33

Update constraints connected to K



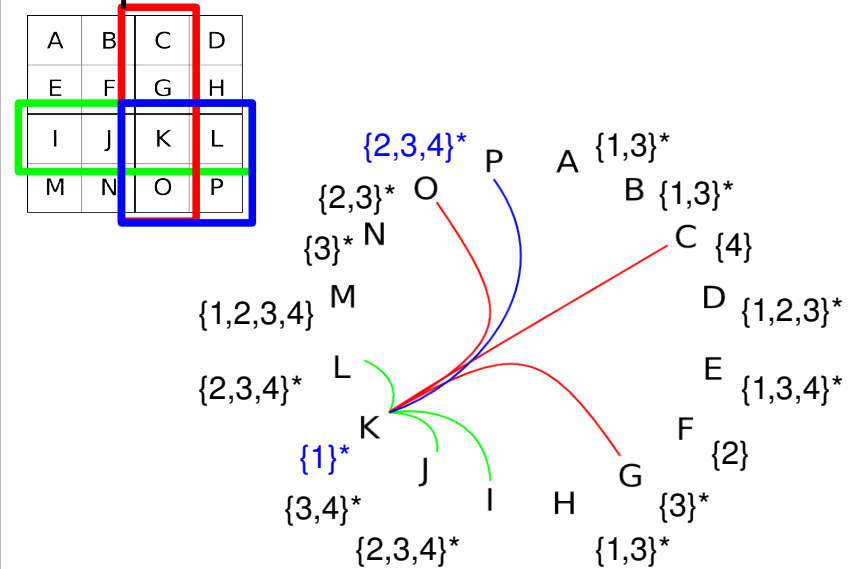
34

Update constraints connected to K



35

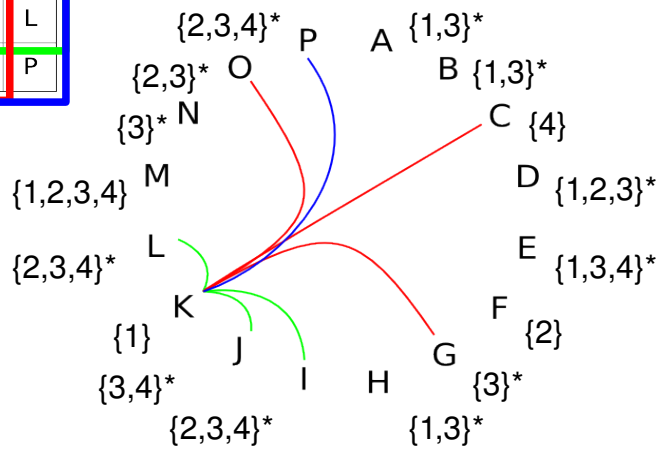
Update constraints connected to K



36

Update constraints connected to K

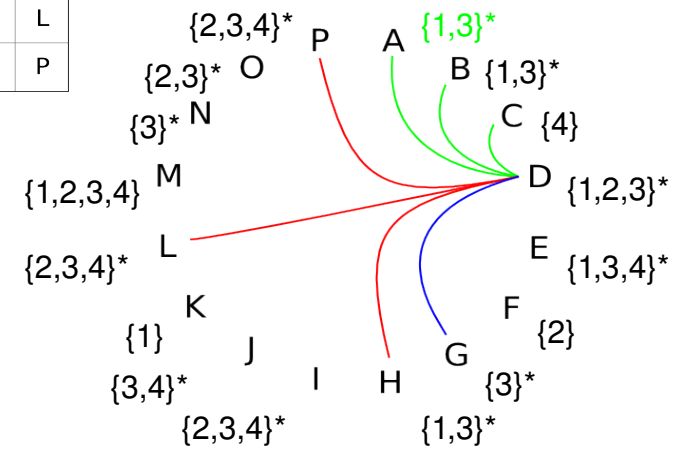
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



37

Update constraints connected to D

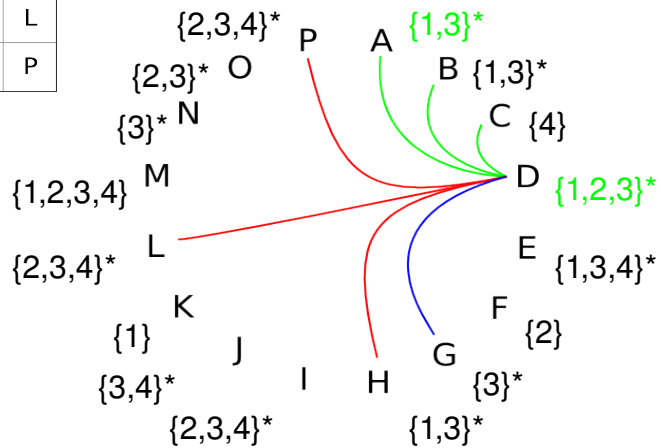
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



38

Sometimes no change occurs in the domain

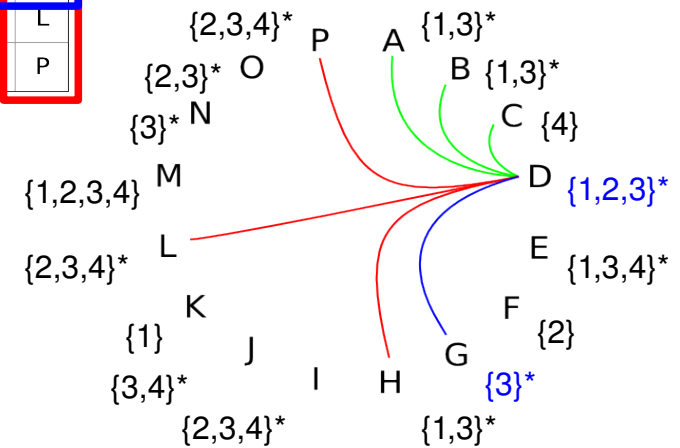
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



39

Sometimes a change occurs in the source domain

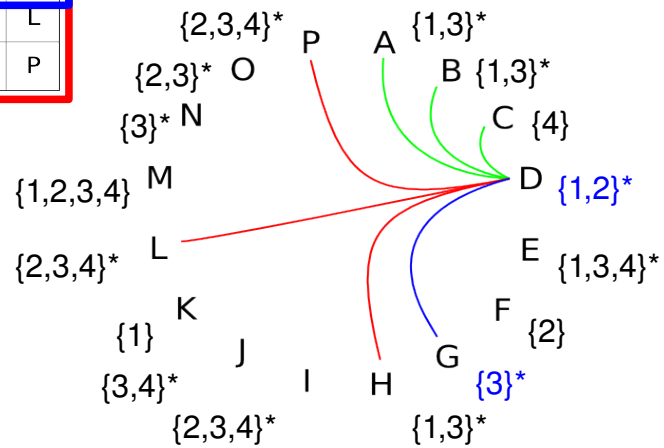
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



40

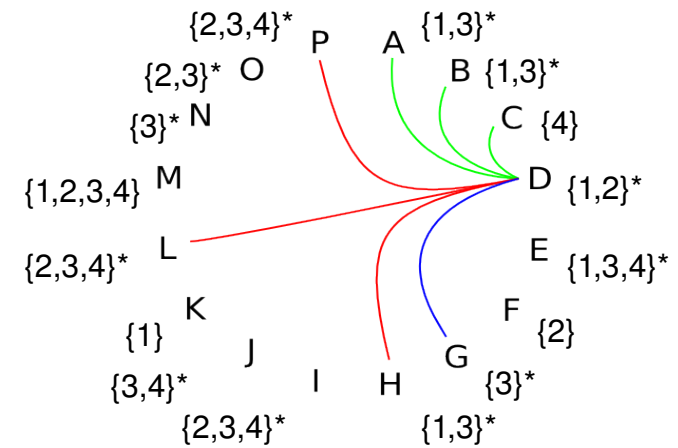
Sometimes a change occurs in the source domain

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P



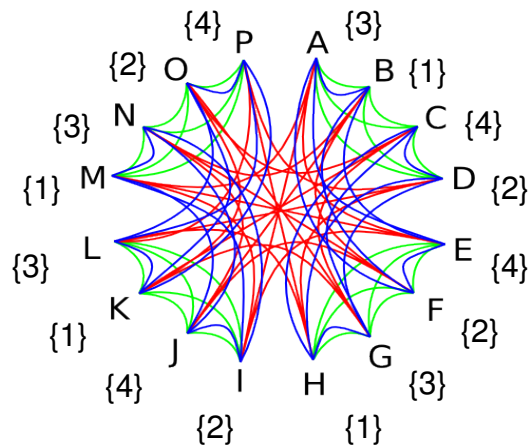
41

If the source domain changes we mark all its constraints for update again



42

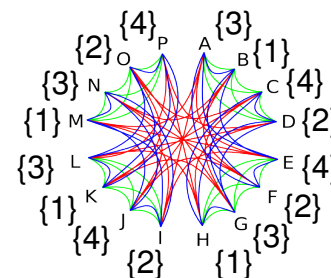
Eventually the algorithm will converge and no further changes occur



43

Outcome 1: Single valued domains

We have found a unique solution to the problem



3	1	4	2
4	2	3	1
2	4	1	3
1	3	2	4

44

Outcome 2: Some empty domains

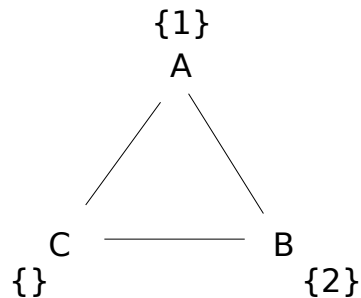
Our constraints are inconsistent – there is no solution to this problem

Variables

$A \in \{1\}$
 $B \in \{1,2\}$
 $C \in \{1,2\}$

Constraints

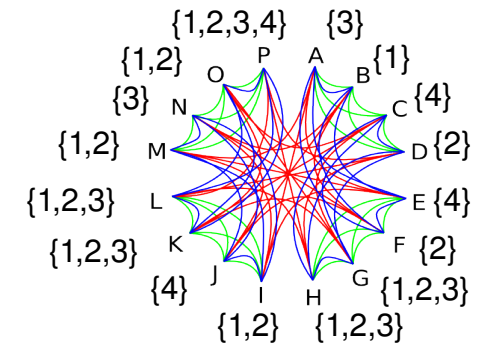
$A \neq B, A \neq C, B \neq C$



45

Outcome 3: Some multivalued domains

		4	
	2		
	3		



Not all combinations of the remaining possibilities are global solutions

46

Outcome 3: Hypothesise labellings

- To find global solutions from the narrowed domains we hypothesise a solution in a domain and propagate the changes
- Backtrack if something goes wrong

47

Using CLP in Prolog

```
:- use_module(library(bounds)).
valid4(L) :- L in 1..4, all_different(L).
sudoku2([[X11,X12,X13,X14],[X21,X22,X23,X24],[X31,X32,X33,X34]
, [X41,X42,X43,X44]]) :-
    valid4([X11,X12,X13,X14]), valid4([X21,X22,X23,X24]),
    valid4([X31,X32,X33,X34]), valid4([X41,X42,X43,X44]),
    valid4([X11,X21,X31,X41]), valid4([X12,X22,X32,X42]),
    valid4([X13,X23,X33,X43]), valid4([X14,X24,X34,X44]),
    valid4([X11,X12,X21,X22]), valid4([X13,X14,X23,X24]),
    valid4([X31,X32,X41,X42]), valid4([X33,X34,X43,X44]),
    labeling([], [X11,X12,X13,X14,X21,X22,X23,X24,
    X31,X32,X33,X34,X41,X42,X43,X44]).
```

Rows

Cols

Boxes

48

More information on CLP

PROLOG Programming for Artificial Intelligence,
Ivan Bratko, Chapter 14

Prolog Supervision Work

Andrew Rice <andrew.rice@cl.cam.ac.uk>

November 5, 2007

1 Introduction

These questions form the suggested supervision work for the Prolog course. All students should attempt the basic questions. Those questions marked with an asterisk are more difficult although all students should be able to answer them with the help of their supervisor. Questions marked with a double asterisk are particularly challenging and are beyond the level required for producing a good answer in the exam.

Prolog contains a number of features and facilities not covered in the lectures such as: assert, findall and retract. Students should limit themselves to using only the features covered in the lecture course and are not expected to know about anything further. All questions can be successfully answered using only the lectured features.

Students are encouraged to contact me by email with bug-reports and solutions to double-asterisk questions.

2 Lecture 1

2.1 Unification

1. Unify these two terms by hand: $\text{tree}(\text{tree}(1,2),A,B), \text{tree}(C, \text{tree}(E,F,G))$ and $\text{tree}(C, \text{tree}(Z,C))$
2. Explain Prolog's behaviour when you unify $a(A)$ with A .
3. Relate unification with ML type inference

2.2 The Zebra Puzzle

1. Implement and test the Zebra Puzzle solution
2. Explain how Clue 1 has been expressed in the zebra query

3 Lecture 2

3.1 Encoding arithmetic in Prolog

The `is` operator in Prolog evaluates arithmetic expressions. This builtin functionality can also be modelled within Prolog's logical framework.

Let the atom `i` represent the identity (1) and the compound term $s(A)$ represent the successor of A . For example $4 = s(s(s(i)))$

Implement and test the following rules:

1. $\text{prim}(A,B)$ which is true if A is a number and B is its primitive representation
2. $\text{plus}(A,B,C)$ which is true if C is $A+B$ (all with primitive representations, A and B are both ground terms)
3. $\text{mult}(A,B,C)$ which is true if C is $A*B$ (all with primitive representations, A and B are both ground term)

The development of arithmetic (and general computation) from first principles is considered more formally in the Foundations of Functional Programming course.

3.2 List Operations

1. Explain the operation of the `append/3` clause

```
append([ ], A, A) .  
append([H|T], A, [H|R]) :- append(T, A, R) .
```

2. Draw the Prolog search tree for $\text{perm}([1,2,3],A)$.
3. Implement a clause $\text{choose}(N,L,R,S)$ which chooses N items from L and puts them in R with the remaining elements in L left in S
4. * What is the purpose of the following clauses:

```
a([H|T]) :- a([H|T], H) .  
a([ ], _) .  
a([H|T], Prev) :- H >= Prev, a(T, H) .
```

5. * What does the following do and how does it work?:

```
b(X, X) :- a(X) .  
b(X, Y) :- a(A, [H1, H2|B], X), H1 > H2, a(A, [H2, H1|B], X1), b(X1, Y) .
```

3.3 Generate and Test

The description of these problems will be given in the lecture.

1. Complete the Dutch National Flag solution
2. * Complete the 8-Queens solution
3. * Generalise 8-Queens to n -Queens
4. Complete the Anagram generator. In what situations is it more efficient to Test-and-Generate rather than Generate-and-Test?

4 Lecture 3

4.1 Symbolic Evaluation

1. Explain what happens when you put the clauses of the symbolic evaluator in a different order
2. Add additional clauses to the symbolic evaluator for subtraction and integer division (this is the `//` operator in Prolog i.e. 2 is $6/3$)

4.2 Negation

State and explain Prolog's response to the following queries:

1. `X=1.`
2. `not(X=1).`
3. `not(not(X=1)).`
4. `not(not(not(X=1))).`

In those cases where Prolog says 'yes' your answer should include the unified result for X.

4.3 Databases

We can use facts entered into Prolog as a general database for storing and querying information. This question considers the construction of a database containing information about students, their colleges and their grades in the various parts of the CS Tripos.

Each fact in our Prolog database corresponds to a row in a table of data. A table is constructed from rows produced by facts with the same name. The initial database of facts is as follows:

```
tName(acr31, 'Andrew Rice').
tName(arb33, 'Alastair Beresford').

tCollege(acr31, 'Churchill').
tCollege(arb33, 'Robinson').

tGrade(acr31, 'IA', 2.1).
tGrade(acr31, 'IB', 1).
tGrade(acr31, 'II', 1).
tGrade(arb33, 'IA', 2.1).
tGrade(arb33, 'IB', 1).
tGrade(arb33, 'II', 1).
```

As an example, this database contains a table called 'tName' which contains two rows of two columns. The first column is the CRSID of the individual and the second column is their full name.

4.3.1 Part 1

1. Add your own details to the database.
2. Add a new table tDOB which contains CRSID and DOB.
3. Alter the database such that for some users their college is not present (this final step is necessary for testing your answers to the questions in Part 2)

4.3.2 Part 2

The next task is to provide rules and show queries which implement various queries of the database. You should answer each question with the Prolog facts and rules required to implement the query and also an example invocation of these rules.

For example:

```
% The full name of each person in the database
qFullName(A) :- tName(_,A).

% Example invocation
% qFullName(A).
```

Each query should return one row of the answer at a time, subsequent rows should be returned by backtracking.

For the example above:

```
?- qFullName(A).

A = 'Andrew Rice' ;

A = 'Alastair Beresford'
```

Yes

- The descriptions that follow provide a plain English description of the query you should implement followed by the same query in SQL.
- SQL (Structured Query Language) is the industry standard language used to query relational databases—you will see more on this in the Databases course.
- The ? notation in the SQL statements derives from the use of prepared statements in relational databases where (for efficiency) a single statement is sent to the database server and repeatedly evaluated with different values replacing the ?. Interested students can consult the Java PreparedStatement documentation.

1. Full name and College attended.
SELECT name,college from tName, tCollege where tName.crsid = tCollege.crsid
2. Full name and College attended only including entries where the user can choose a single CRSID to include in the results.
SELECT name,college from tName, tCollege where tName.crsid = tCollege.crsid and tName.crsid = ?
3. Full name and College attended or blank if the college is unknown.
SELECT name,college from tName left outer join tCollege on tName.crsid = tCollege.crsid
4. Full name and College attended. The full name or the college should be blank if either is unknown.
SELECT name,dob from tName FULL OUTER JOIN tDOB ON tName.crsid = tDOB.crsid
5. * Find the lowest grade where the CRSID is specified by the user. Note that this predicate should only return one result even when backtracking.
SELECT min(grade) from tGrade WHERE crsid = ?

6. ** Find the number of people with a First class mark
SELECT count(grade) from tGrade WHERE grade = 1
Hint: This is not the number of rows with First class marks in the tGrade table. You will need build a list of First class CRSIDs by repeatedly querying tGrade and checking if the result is already in your list. Every time you find a new unique CRSID, increment an accumulator which will form the result
7. ** Find the number of First class marks awarded to each person. Your output should consist of a tuple (CRSID,NumFirsts) which iterates through all CRSIDs which have at least one First class mark upon backtracking
SELECT crsid,count(grade) from tGrade WHERE grade=1 GROUP BY crsid

5 Lecture 4

5.1 Countdown Numbers Game

1. Type in the example code which finds exact solutions from the lectures and test it;
2. Implement the predicate range(Min,Max,Val) which unifies Val with Min on the first evaluation, and then all values up to Max-1 when backtracking;
3. Get the iterative deepening version of the numbers game working.

5.2 Graph searching

Implement search-based solutions for:

1. Missionaries and Cannibals: there are three missionaries, three cannibals and who need to cross a river. They have one boat which can hold at most two people. If, at any point, the cannibals outnumber the missionaries then they will eat them. Discover the procedure for a safe crossing.
2. * Towers of Hanoi: you have N rings of increasing size and three pegs. Initially the three rings are stacked in order of decreasing size on the first peg. You can move them between pegs but you must never stack a big ring onto a smaller one. What is the sequence of moves to move from all the rings from the first to the third peg.
3. * Umbrella: A group of 4 people, Andy, Brenda, Carl, & Dana, arrive in a car near a friend's house, who is having a large party. It is raining heavily, & the group was forced to park around the block from the house because of the lack of available parking spaces due to the large number of people at the party. The group has only 1 umbrella, & agrees to share it by having Andy, the fastest, walk with each person into the house, & then return each time. It takes Andy 1 minute to walk each way, 2 minutes for Brenda, 5 minutes for Carl, & 10 minutes for Dana. It thus appears that it will take a total of 19 minutes to get everyone into the house. However, Dana indicates that everyone can get into the house in 17 minutes by a different method. How? The individuals must use the umbrella to get to & from the house, & only 2 people can go at a time (& no funny stuff like riding on someone's back, throwing the umbrella, etc.). (This puzzle included with kind permission from <http://www.puzz.com/>)

6 Lecture 5

6.1 Sorting

Implement the following sorting algorithms in Prolog:

1. Finding the minimum element from the list and recursively sorting the remainder.
2. Quicksort.
3. * Quicksort where the partitioning step divides the list into three groups: those items less than the pivot, those items equal to the pivot and those items greater than the pivot. Explain in what situations this additional complexity might be desirable.
4. * Quicksort with the append removed using difference lists. Note: you don't need to alter the partitioning clauses.
5. * Mergesort

6.2 Towers of Hanoi

The Towers of Hanoi problem can be solved without requiring an inefficient graph search. Discover the algorithm required to do this from the lecture notes or the web and implement it. Once you have a simple list-based implementation rewrite it to use difference lists.

6.3 Dutch Flag

Earlier in the course we solved the dutch flag problem using generate and test. A more efficient approach is to make one pass through the initial list collecting three separate lists (one for each colour). When you reach the end of the initial list you append the three separate collection lists together and you are done. Implement this algorithm using normal lists and then rewrite it to use difference lists.

7 Lecture 6

7.1 Sudoku Solver

Extend the CLP-based 2x2 Sudoku solver given in the lecture to a 3x3 grid and test it.

7.2 Cryptarithmic

Here is a classic example of a cryptarithmic puzzle:

$$\begin{array}{r} S E N D \\ + M O R E \\ \hline M O N E Y \end{array}$$

The problem is to find an assignment of the numbers 0-9 (inclusive) to the letters S,E,N,D,M,O,R,E,Y such that the arithmetic expression holds and the numeric value assigned to each letter is unique.

We can formulate this problem in CLP as follows:

```
solve1([S,E,N,D],[M,O,R,E],[M,O,N,E,Y]) :-
    Var = [S,E,N,D,M,O,R,E,Y],
    Var in 0..9, all_different(Var),
    1000*S + 100*E + 10*N + D +
    1000*M + 100*O + 10*R + E #=
    10000*M + 1000*O + 100*N + 10*E + Y,
    labeling([],Var).
```

1. Get the example above working in your Prolog interpreter. How many unique solutions are there?
2. A further requirement of these types of puzzle is that the leading digit of each number in the equation is not zero. Extend your program to incorporate this. The CLP operator for arithmetic not-equals is `#\=`. How many unique solutions remain now?
3. * Extend your program to work in an arbitrary base (rather than base 10), the domain of your variables should change to reflect this. How many solutions of the puzzle above are there in base 16?
4. * Consult the prolog documentation for the `findall` predicate. Use this to write a predicate `count(Base,N)` which unifies `N` with the number of solutions in base `Base`.
5. * Use the `range/3` predicate from Lecture 4 to extend your `count` predicate to try all values from 1 to 50 as the base.
6. * Plot a graph of the number of solutions against the chosen base.

7.3 **Findall

The `findall` predicate is an extra-logical predicate for backtracking and collecting the results into a list. The implementation within Prolog is something along the lines of:

```
findall(Template,Goal,Solutions) :- call(Goal),
                                   assertz(findallsol(Template)),
                                   fail.
findall(Template,Goal,Solutions) :- collect(Solutions).

collect([Template|RestSols]):- retract(findallsol(Template)),
                              !,
                              collect(RestSols).
collect([]).
```

1. ** Consult the prolog documentation and work out what the above is doing. The predicate `assertz` adds a new clause to the end of the running prolog program and the predicate `retract` removes a clause which unifies with its argument.
2. ** Develop an alternative to `findall` which doesn't use extra-logical predicates such as `assertz` and `retract`. This alternative will necessarily take the form of a pattern which you can apply to your clause to find all the possible results. The algorithm for the alternative `findall` proceeds as follows: maintain a list of solutions found so far, evaluate the target clause and repeatedly backtrack through it until it returns a value not in your list of results, add the result to the list and repeat.
3. ** Comment on the runtime complexity of the alternate `findall` compared to the builtin version.