Prolog basics

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis
Alastair Beresford, Andy Rice

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;
}
```


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);
```


Logic Programming

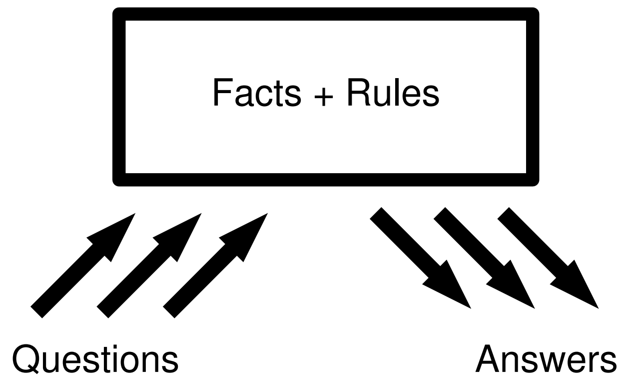
```
% the sum of the empty list is zero
sum([],0).

% 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

Prolog basics: Learning goals

- Identify programming styles
- Structure of a Prolog program
- Types of Prolog term
- Unifying terms

Prolog Programs Answer Questions



Prolog's database can answer simple questions

```
> prolog                          ......................or maybe swipl on your system
?- [user].                        .............get ready to enter a new program
|: milestone(rousell,1972).
|: milestone(warren,1983).
|: milestone(swiprolog,1987).
|: milestone(yourcourse,2012).    ........................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
```


The program is composed of facts and queries

```
> prolog
?- [user].
|: milestone(rousell,1972).
|: milestone(warren,1983).
|: milestone(swiprolog,1987).
|: milestone(yourcourse,2012).
|: % user://1 compiled 0.01 sec, 764 bytes
Yes
?- milestone(warren,1983).
Yes
?- milestone(swiprolog,X).
X=1987
Yes
?- milestone(yourcourse,2000).
No
?- halt.
```

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,2012).

> 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.
```


Terms can be Constants,Compounds or Variables

Constants	Variables

Compound

Unification is Prolog's fundamental operation

- Atoms unify if they are identical
- Variables unify with anything
- Compound terms unify if their top function symbols and arities match and all parameters unify recursively

?

Question 1

Which of these unify?

1	a	with a
2	a	with b
3	a	with A
4	a	with B
5	tree(l,r)	with A
6	tree(l,r)	with tree(B,C)
7	tree(A,r)	with tree(l,C)
8	tree(A,r)	with tree(A,B)
9	A	with a(A)
10	a	with a(A)

Prolog basics: Learning goals

- Identify programming styles
- Structure of a Prolog program
- Types of Prolog term
- Unifying terms

Solving a logic puzzle

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

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?

Solving a logic puzzle: Learning goals

- Try out the presented solution
- Understand the rationale behind it
- Develop an informal intuition about how Prolog finds the answer
- We don't expect you **yet** to be able to solve these problems yourself

Model the situation

Represent each house with the term:

Represent the row of houses as follows:

? Identifying types of term

Question 1

What sort of a term is:

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

1 number
2 atom
3 compound
4 variable

? Identifying types of term

Question 2

What sort of a term is:

Nationality

1 number
2 atom
3 compound
4 variable

? Identifying types of term

Question 3

What sort of a term is:

(H1,H2,H3,H4,H5)

1 number
2 atom
3 compound
4 variable

Define house-existence facts

?

Which queries return 'true'?

Question 4

1 exists(dog,(fly,spider,bird,cat,dog)).
2 exists(dog,(fly,spider,bird,cat)).
3 exists(dog).
4 exists(house(english,red),
(house(spanish,green),
house(french,orange),
house(dutch,yellow),
house(german,blue),
house(english,_)).

Define facts for 'to the right of'

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

Facts for the middle and first house

9. Milk is drunk in the middle house.

10. The Norwegian lives in the first house.

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)).
```


Express the puzzle as a query

```
1  exists(house(british,_,_,_,red),Houses),
2  exists(house(spanish,dog,_,_,_),Houses),
3  exists(house(_,_,_,coffee,green),Houses),
4  exists(house(ukranian,_,_,tea,_),Houses),
5  rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
6  exists(house(_,snail,oldgold,_,_),Houses),
7  exists(house(_,_,kools,_,yellow),Houses),
8  middleHouse(house(_,_,_,milk,_),Houses),
9  firstHouse(house(norwegian,_,_,_,_),Houses),
10 nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
11 nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
12 exists(house(_,_,luckystrike,orangejuice,_),Houses),
13 exists(house(japanese,_,parliaments,_,_),Houses),
14 nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
15 exists(house(WaterDrinker,_,_,water,_),Houses),
16 exists(house(ZebraOwner,zebra,_,_,_),Houses).
```


Finding clues

Question 5

```
1  exists(house(british,_,_,_,red),Houses),
2  exists(house(spanish,dog,_,_,_),Houses),
3  exists(house(_,_,_,coffee,green),Houses),
4  exists(house(ukranian,_,_,tea,_),Houses),
5  rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
6  exists(house(_,snail,oldgold,_,_),Houses),
7  exists(house(_,_,kools,_,yellow),Houses),
8  middleHouse(house(_,_,_,milk,_),Houses),
9  firstHouse(house(norwegian,_,_,_,_),Houses),
10 nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
11 nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
12 exists(house(_,_,luckystrike,orangejuice,_),Houses),
13 exists(house(japanese,_,parliaments,_,_),Houses),
14 nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
15 exists(house(WaterDrinker,_,_,water,_),Houses),
16 exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

The clue 'The Ukranian drinks Tea' is at line:

Finding clues

Question 6

```
1  exists(house(british,_,_,_,red),Houses),
2  exists(house(spanish,dog,_,_,_),Houses),
3  exists(house(_,_,_,coffee,green),Houses),
4  exists(house(ukranian,_,_,tea,_),Houses),
5  rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
6  exists(house(_,snail,oldgold,_,_),Houses),
7  exists(house(_,_,kools,_,yellow),Houses),
8  middleHouse(house(_,_,_,milk,_),Houses),
9  firstHouse(house(norwegian,_,_,_,_),Houses),
10 nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
11 nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
12 exists(house(_,_,luckystrike,orangejuice,_),Houses),
13 exists(house(japanese,_,parliaments,_,_),Houses),
14 nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
15 exists(house(WaterDrinker,_,_,water,_),Houses),
16 exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

The clue 'The Spaniard owns the dog' is at line:

?

Finding clues

Question 7

```
1   exists(house(british,_,_,_,red),Houses),
2   exists(house(spanish,dog,_,_,_),Houses),
3   exists(house(_,_,_,coffee,green),Houses),
4   exists(house(ukranian,_,_,tea,_),Houses),
5   rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
6   exists(house(_,snail,oldgold,_,_),Houses),
7   exists(house(_,_,kools,_,yellow),Houses),
8   middleHouse(house(_,_,_,milk,_),Houses),
9   firstHouse(house(norwegian,_,_,_,_),Houses),
10  nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
11  nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
12  exists(house(_,_,luckystrike,orangejuice,_),Houses),
13  exists(house(japanese,_,parliaments,_,_),Houses),
14  nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
15  exists(house(WaterDrinker,_,_,water,_),Houses),
16  exists(house(ZebraOwner,zebra,_,_,_),Houses).
```

The clue 'Milk is drunk in the middle house' is at line:

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

Zebra Puzzle

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

```
> prolog
?- [zebra].
norwegian
japanese
% zebra compiled 0.00 sec, 6,384 bytes
Yes
?- halt.
```


Solving a logic puzzle: Learning goals

- Try out the presented solution
- Understand the rationale behind it
- Develop an informal intuition about how Prolog finds the answer
- We don't expect you **yet** to be able to solve these problems yourself

Prolog Rules

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

The Zebra solution with facts could be improved

```
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)).
```


Prolog Rules: Learning goals

- Syntax for rules
- Relation to First Order Logic
- Recursion

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

```
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”

Variables can be internal to a rule

```
rule2(X) :- thing(X,Z), thang(Z).
```

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

Rules can be recursive

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


? Which materials are valuable?

Question 1

```
material(gold).
material(aluminium).
process(bauxite,alumina).
process(alumina,aluminium).
process(copper,bronze).
valuable(X) :- material(X).
valuable(X) :- process(X,Y),valuable(Y).
```

1 gold
2 bauxite
3 bronze
4 copper

Rules are just a First Order Logic statement

“rule2(X) is true if there is a Z such that thing(X,Z) is true and thang(Z) is true”

Improving on nextTo

```
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)).
```


Prolog Rules: Learning goals

- Syntax for rules
- Relation to First Order Logic
- Recursion

Lists

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Our Zebra puzzle solution only works for 5 houses

- We would need to rewrite the whole program to change it
- The problem is (H1,H2,H3,H4,H5)
- What we really need is a list....

Lists: Learning goals

- Syntax for lists in Prolog
- Understand how Prolog finds the answer to a simple recursive query
- Use the Prolog debugger

Support for lists is built in to Prolog

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]

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

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



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

You can use 'trace' to follow execution

```
?- [last].
% 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

?

Question

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

Question 1

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

1 pattern-match exception
2 Prolog says no
3 Prolog says yes, X = []
4 Prolog says yes, X = ???

Lists: Learning goals

- Syntax for lists in Prolog
- Understand how Prolog finds the answer to a simple recursive query
- Use the Prolog debugger

Arithmetic

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Counting would be useful

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

What about if we want to know how many elements there are?

Arithmetic: Learning goals

- How to do arithmetic in Prolog
- More execution tracing
- Space efficiency and Last Call Optimisation

Arithmetic equality != Unification

```
?- A = 1+2.
A = 1+2
Yes

?- 1+2 = 3.
No
```

Equals (=) in Prolog means “unifies with”

Arithmetic equality != Unification

```
?- A = money+power.
A = money+power
Yes
```

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

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
```


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
```


?

Expressing expressions

Question 1

What is the result of the query: A is +(*(3,2),4) ?

1 Error - Not an arithmetic expression
2 10
3 18
4 20

We can now compute the list length

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

O(N) stack space

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


Use an accumulator for O(1) space

```
len2([],Acc,Acc).
len2([_|T],Acc,R) :-  B is Acc + 1,
                      len2(T,B,R).

len2(List,R) :- len2(List,0,R).
```


O(1) stack space

```
len2([],Acc,Acc).
len2([_|T],Acc,R) :-
   B is Acc + 1,len2(T,B,R).
```


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

Validate with a "test to destruction"

- The debugger would interfere with LCO
- Test to see if we run out of stack
  - Generate a big list
  - Show that len2 works
  - Show that len runs out of space

Arithmetic: Learning goals

- How to do arithmetic in Prolog
- More execution tracing
- Space efficiency and Last Call Optimisation

Backtracking

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Some queries have more than one answer

```
> 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,2012).

> 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.
```

Backtracking: Learning goals

- Rules to 'take' an item from a list
- Understand how Prolog searches for the next answer using 'choice points'
- How backtracking can cause our programs to misbehave

Repeatedly remove an element from a list

take([1,2,3],A,B) should give...

Use backtracking to 'take' each element from the list in turn

```
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).
```


Executing take

```
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).
```

```
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).
```

Backtracking take


```
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).
```

Backtracking take again

Prolog backtracks by recording 'choice points'

- Choice points are locations in the search where we could take another option
- If there are no choice points left then don't offer the user any more answers


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

'backwards' len

Question 1

What is the result of the query len(A,2). ?

1 Error due uninstantiated arithmetic expression
2 A = [_,_]
3 Query runs forever
4 Error due to invalid arguments

len([],0).
len([_|T],N) :- len(T,M), N is M+1.

'backwards' len

len([],0).
len([_|T],N) :- len(T,M), N is M+1.

? Backtracking len

Question 2

What happens if we ask len(A,2). for a second answer (press ';') ?

1 Error due uninstantiated arithmetic expression
2 A = [_,_]
3 Query runs forever
4 Error due to invalid arguments

len([],0).
len([_|T],N) :- len(T,M), N is M+1.

Backtracking len

Backtracking: Learning goals

- Rules to 'take' an item from a list
- Understand how Prolog searches for the next answer using 'choice points'
- How backtracking can cause our programs to misbehave

Generate and test

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

We're going to solve more puzzles

- The Zebra puzzle was pretty easy
- Apply our knowledge of lists and backtracking

Generate and Test: Learning goals

- New clauses for permutations of a list
- Solving problems with generate-and-test

Recall the 'take' predicate

```
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).
```

We can use 'take' to generate permutations of a list

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


'Dutch national flag' can be viewed as a permutations question

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

[red,white,blue,white,red]

↓

[red,red,white,white,blue]

'Dutch national flag' can be viewed as a permutations question

```
flag(In,Out) :-   perm(In,Out),
                  checkColours(Out).
```


Check that the colours are in order

- `checkRed` should be true
  - if Head = red & `checkRed`(Tail)
  - or if `checkWhite`(List)
- `checkWhite` should be true
  - if Head = `white` & `checkWhite`(Tail)
  - or if `checkBlue`(List)
- `checkBlue` should be true ....

?

Which is a correct 'checkRed'

Question 1

1 checkRed(L) :- Head = red,
    checkRed(Tail); checkWhite(Tail).

2 checkRed(L) :- Head = red, checkRed(Tail).
checkRed(L) :- checkWhite(Tail).

3 checkRed([H|T]) :- H = red, checkRed(T).
checkRed(L) :- checkWhite(L).

4 checkRed([red|T]) :- checkRed(T).
checkRed(L) :- checkWhite(L).

This is an example of Generate and Test

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

?

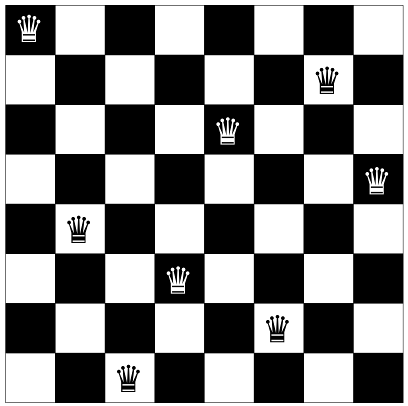
Which part of the flag solution is 'Generate'?

Question 2

```
flag(In,Out) :-    perm(In,Out),
                   checkColours(Out).
```

1 flag(In,Out)
2 perm(In,Out)
3 checkColours(Out)
4 No part of the solution is 'Generate'

Place 8 Queens so that none can take any other



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

Generate and Test works for 8 Queens too

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

(You can use X =\= Y which will succeed if X is not equal to Y)

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!

Generate and Test: Learning goals

- New clauses for permutations of a list
- Solving problems with generate-and-test

Symbolic evaluation

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Backtracking 'broke' len(A,3)

- One might argue we used it outside of its specification
- Bad backtracking can occur in legitimate programs too
- How can we fix it?

Symbolic evaluation – Learning goals

- New clauses for symbolic evaluation of arithmetic
- Understand why they cause a backtracking problem
- Know one strategy to fix it

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).
```


?

Question 1

```
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).
```

How many times is eval(A,A) satisfied in the evaluation of eval(plus(1,mult(4,5)),X)

1 3 times
2 0 times
3 1 time
4 4 times

Search tree

```
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).
```


Backtracking

```
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).
```


Eliminate spurious solutions by making your clauses orthogonal

- The problem occurs because we have too many choice points
- Instead make sure that only one clause matches

```
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).
```


Symbolic evaluation – Learning goals

- New clauses for symbolic evaluation of arithmetic
- Understand why they cause a backtracking problem
- Know one strategy to fix it

Cut!

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

We controlled backtracking by limiting choice points

- We did this with eval/2 by making the clauses orthogonal
- 'cut' gives us an alternative mechanism

Cut! - Learning goals

- Understand how the cut operator changes the search tree
- Recognise different types of cut

Cut tells Prolog to commit to its choice

```
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).
```

Search tree

```
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).
```


Cut closes choice points

?

What does split/3 do?

Question 1

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```


Cut can appear in the middle of a rule

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, !, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, !, split(T,L,R).
```

This is a **green** cut – it just helps execution go faster

What is the logical meaning of these clauses?

Question 2

```
p :- a,b.
p :- c.
```

1. $p \Leftrightarrow (a \wedge b) \vee c$
2. $p \Leftrightarrow a \wedge b \wedge c$
3. $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$
4. $p \Leftrightarrow a \wedge (b \vee c)$

What is the logical meaning of these clauses?

Question 3

```
p :- a,!,b.
p :- c.
```

1. $p \Leftrightarrow (a \wedge b) \vee c$
2. $p \Leftrightarrow a \wedge b \wedge c$
3. $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$
4. $p \Leftrightarrow a \wedge (b \vee c)$

Cut! - Learning goals

- Understand how the cut operator changes the search tree
- Recognise different types of cut

Negation

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Pruning the search tree gives us more expressive power

- cut lets us control backtracking by removing choice points
- this opens up a whole range of new programs (and problems)

Negation – Learning goals

- How to use cut to implement negation-by-failure
- Understand the dangers

What does this do?

Question 1

```
a :- !,1=2.
```

1 unifies 1 with 2
2 throws an exception
3 always succeeds
4 always fails

What does this do?

Question 2

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

1 unifies the two arguments
2 succeeds if the arguments unify
3 succeeds if the arguments don't unify
4 always fails

Pay attention to whether the unification is undone

- Clauses such as 'fail' and 'isDifferent' can cause us to backtrack in unusual ways
- This will undo any variable bindings along the way

We can now implement 'not' using negation-by-failure

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

You may also write not(A) as \+A

?

What sort of cut is this?

Question 3

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

1 red
2 amber
3 green

'not' is based on the closed world assumption

Everything that is true in the “world” is stated (or can be derived from) the clauses in the program.

A simple negation example

```
goodFood(theWrestlers).
goodFood(midsummerHouse).
expensive(midsummerHouse).

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


Negation query

Question 4

```
goodFood(theWrestlers).
goodFood(midsummerHouse).
expensive(midsummerHouse).
bargain(R) :-    goodFood(R),
                 not(expensive(R)).
```

What happens if we ask bargain(R). ?

1 R = theWrestlers (and then no more results)
2 R = theWrestlers and then loop forever
3 R = theWrestlers, R = midsummerHouse
4 False

A simple negation mistake

```
goodFood(theWrestlers).
goodFood(midsummerHouse).
expensive(midsummerHouse).

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

The query bargain(R) now fails immediately

When using negation remember the quantifiers

- expensive(R)
- not(expensive(R))

Negation – Learning goals

- How to use cut to implement negation-by-failure
- Understand the dangers

Databases

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog

Ian Lewis

We used cut to build negation

- This produces some more complicated behaviour
- Practice....

Databases – Learning goals

- Understand how Prolog clauses map onto relations
- Get a basic idea of relational databases and SQL

Relational databases allow us to store and query structured data

- Structured Query Language (SQL) is the standard way to interact with a relational database
- Much more detail in a later course

Databases – Learning goals

- Understand how Prolog clauses map onto relations
- Get a basic idea of relational databases and SQL

Playing Countdown

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Let's look at another generate and test problem

- We've already visited Dutch National Flag, 8-Queens and Anagrams
- Time for another example

Playing Countdown – Learning Goals

- How to encode a search-based game into Prolog
- See an example of Generate and Test which doesn't use 'perm'

Countdown Numbers

- Select 6 of 24 number 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

Countdown Numbers

- To see the game in progress have a look on YouTube.
- I recommend James Martin's numbers game from 1997...

Countdown Numbers

- Strategy – generate and test
  - maintain a list of symbolic arithmetic terms
  - initially this list consists of ground terms e.g.: [25,50,75,100,6,3]
  - 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

?

Will this strategy terminate?

Question 1

1 It will always terminate
2 It will always terminate if there is a correct solution
3 It will not terminate if there are no solutions
4 It will always be quicker than a person

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)).

Helper predicates for symbolic evaluation

```
isGreater(A,B) :- eval(A,Av), eval(B,Bv), Av>Bv.
notOne(A) :- eval(A,Av), Av =\= 1.
isFactor(A,B) :-  eval(A,Av), eval(B,Bv),
                  0 is Bv rem Av.
```


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)).
% no negative numbers allowed
arithop(A,B,minus(A,B)) :- isGreater(A,B).
arithop(A,B,minus(B,A)) :- isGreater(B,A).
% don't allow mult by 1
arithop(A,B,mult(A,B)) :- notOne(A), notOne(B).
% dont allow div by 1 and no fractions allowed
arithop(A,B,div(A,B)) :- notOne(B), isFactor(B,A).
arithop(A,B,div(B,A)) :- notOne(A), isFactor(A,B).
```

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).
```


Which part of the program is 'generate'

Question 2

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

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


Closest Solution

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

```
solve2([Soln|_],Target,Soln,D) :- eval(Soln,R), diff(Target,R,D).
solve2(L,Target,Soln,D) :- choose(2,L,[A,B],R),
                           arithop(A,B,C),
                           solve2([C|R],Target,Soln,D).

closest(L,Target,Soln,D) :- range(0,100,D), solve2(L,Target,Soln,D).
```

This is *iterative deepening*

Playing Countdown – Learning Goals

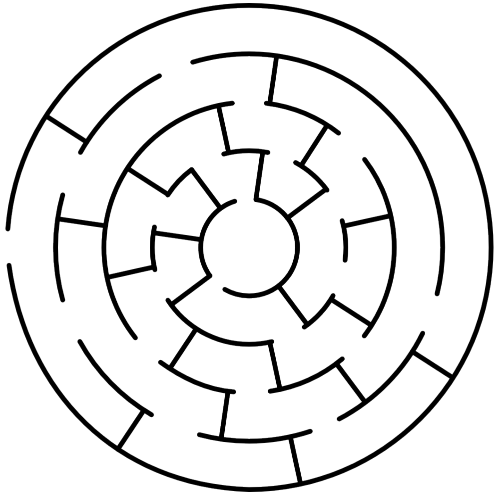
- How to encode a search-based game into Prolog
- See an example of Generate and Test which doesn't use 'perm'

Graph Search

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

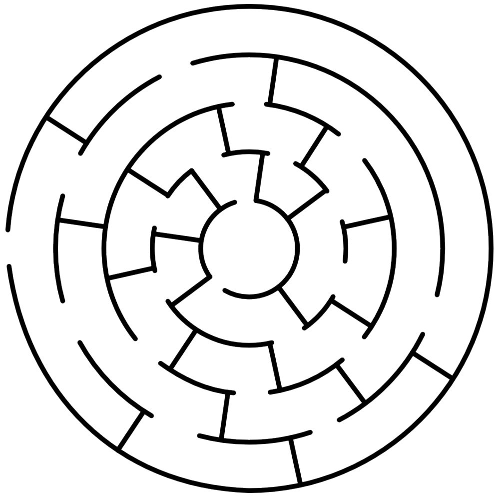
Generate and test is not the only approach...



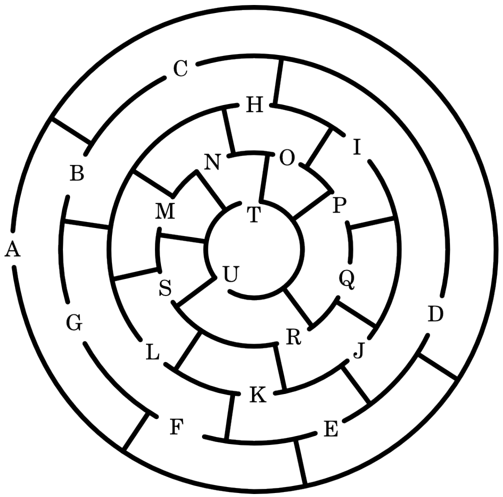
Graph Search – Learning goals

- Recognise a graph search problem
- Understand how to encode this in Prolog
- Searching cyclic graphs
- Know the basic building blocks of the graph pattern

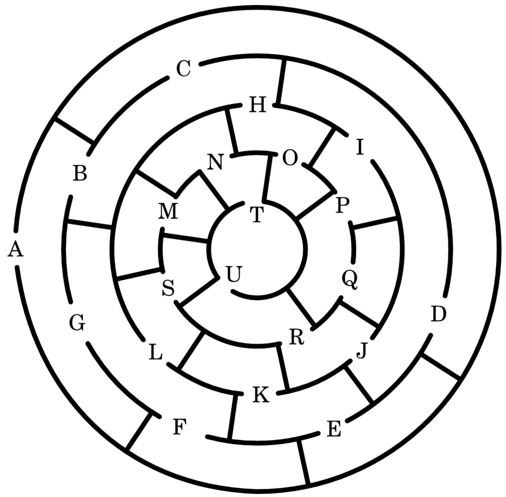
Solving a maze requires a graph search



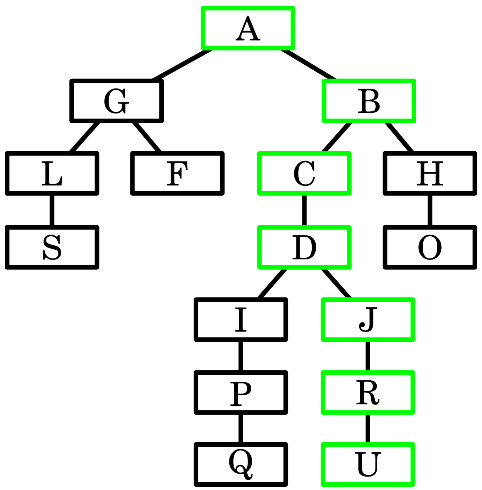
Each opening is a vertex



Edges connect adjacent openings



We have now abstracted the graph from the problem



We can encode the graph as Prolog facts

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

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

```
start(a).
finish(u).
```

We need to remember the route too

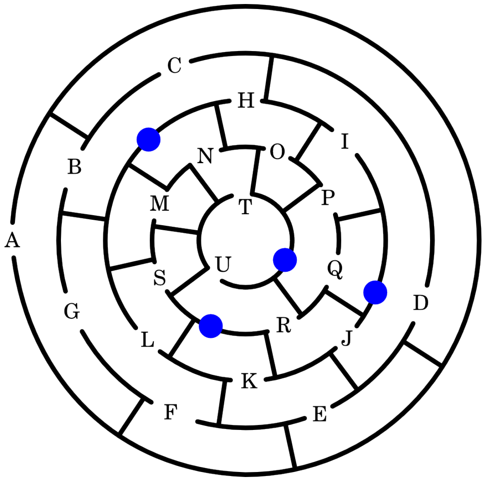
```
travellog(A,A,[]).
travellog(A,C,[A-B|Steps]) :-
      route(A,B), travellog(B,C,Steps).

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


What happens if our graph has a loop?

?

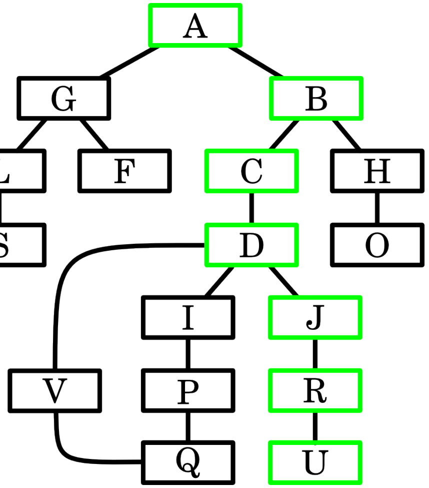
Question 1

Which new opening would create a cycle in the graph?



Cyclic Graphs

route(q,v).
route(v,d).



Avoiding revisiting any node will beat the cycle

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

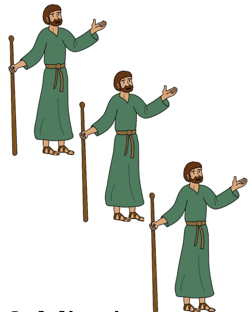

Graph searching fits a general pattern

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

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

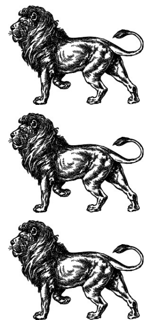

State space representation is important

- Maze searching is a straight-forward mapping
- Other problems are not so obvious
- Choose a representation with as little redundancy as possible
- This will shorten your rules for which transitions are possible

Missionaries and Cannibals



3 Missionaries

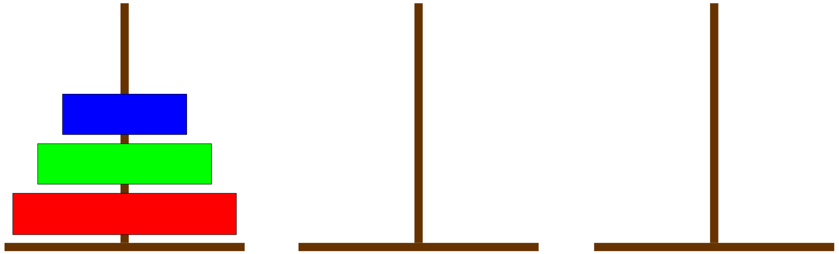


3 Cannibals

1 boat

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?

Towers of Hanoi



Graph Search – Learning goals

- Recognise a graph search problem
- Understand how to encode this in Prolog
- Searching cyclic graphs
- Know the basic building blocks of the graph pattern

Difference Lists

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Appending two lists

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

Difference lists – Learning goals

- How to append normal lists
- Difference list append and how to derive it
- Why they are called difference lists
- How to write an empty difference list

We'd like to append without copying

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

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

We'd like to append without copying

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

4 :: (5 :: (6 :: B))

?

Question 1

Prolog syntax

What is the Prolog syntax for

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

1 [1,2,3,A]
2 ::(1,::(2,::(3,A))
3 [1,2,3|A]
4 There is no way to express this

Reimplementing append

app(L1,T1,L2,T2,L3,T3) :- ….

Reimplementing append

$[\ l1_0, l1_1, l1_2, l1_3\ |\ T1]$

$[\ l2_0, l2_1, l2_2, l2_3\ |\ T2]$

app(L1,T1,L2,T2,L3,T3) :-

Reimplementing append

```
app(L1,T1,L2,T2,L3,T3) :-   T1 = L2,
                            L3 = L1,
                            T3 = T2
```


Reimplementing append

```
app(L1,T1,T1,T2,L3,T3) :-   T1 = T1,
                            L3 = L1,
                            T3 = T2
```

Reimplementing append

```
app(L1,T1,T1,T2,L3,T3) :-  L3 = L1,
                           T3 = T2
```


Reimplementing append

```
app(L3,T1,T1,T2,L3,T3) :-  L3 = L3,
                           T3 = T2
```


Reimplementing append

```
app(L3,T1,T1,T2,L3,T3) :-  T3 = T2
```


Reimplementing append

```
app(L3,T1,T1,T3,L3,T3) :-  T3 = T3
```

Reimplementing append

app(L3,T1,T1,T3,L3,T3).

Reimplementing append

app(A,B,B,C,A,C).

Reimplementing append

app(A-B,B-C,A-C).

These are commonly called Difference Lists

[1,2,3|A]-A is:

[1,2,3|A] minus A = [1,2,3]

(I think this is just confusing)

Its just a convention

If you see A-B then you should imagine that you actually have something of the form […..|B]-B

But its up to you to make sure that your program preserves this....

?

Empty difference list

Question 2

How should you write an empty difference list?

1 []
2 []-[]
3 A-A
4 [A]

Consider append onto an empty list

dapp(A-B,B-C,A-C).

dapp(P-P,[1,2,3|Q]-Q,R-S).

vs

dapp([]-[],[1,2,3|Q]-Q,R-S).

Difference lists – Learning goals

- How to append normal lists
- Difference list append and how to derive it
- Why they are called difference lists
- How to write an empty difference list

Difference List Example

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

We saw how to derive append for difference lists

- The technique of substituting variables and then simplifying them can be applied to many difference list problems

Difference List Example – Learning goals

- Translate a program into difference lists
- Simplify using substitution

Previous exam question

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.

1996-6-7

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).
```


Rewrite with Difference Lists

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

becomes

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


Rename Variables To Get Rid Of Append

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


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).
```

Final Answer

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

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

Difference List Example – Learning goals

- Translate a program into difference lists
- Simplify using substitution

Empty difference lists

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

We've seen that the correct way to write an empty difference list is A-A

This can cause problems sometimes

Empty difference lists – Learning Goals

- Understand why the use of A-A can be problematic
- Have a repertoire of options for fixing these problems

We've looked at possible behaviours for A = a(A)

What about A-A = [1,2,3|B]-B?

Example: length of a list

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

Example: length of a list

```
lend(A-A,0).
lend([_|T]-T1,N) :- lend(T-T1,M), N is M+1.
```


?

```
lend(A-A,0).
lend([_|T]-T1,N) :- lend(T-T1,M), N is M+1.
```

Using len

Question 1

What is the result of:

lend([1,2,3|A]-A,B).

1 A = _, B = 3
2 Error: Arguments not sufficiently instantiated
3 A = infinite term, B = 0
4 false.

You might 'ground' the list

```
lend2([]-[],0).
lend2([_|T]-T1,N) :- lend2(T-T1,M), N is M+1.
```


Or force an occurs check

```
lend3(A-A1,0) :- unify_with_occurs_check(A,A1).
lend3([_|T]-T1,N) :- lend3(T-T1,M), N is M+1.
```

You can force SWIProlog to do this

:- set_prolog_flag(occurs_check,true).

Empty difference lists – Learning Goals

- Understand why the use of A-A can be problematic
- Have a repertoire of options for fixing these problems

Playing Sudoku

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

Playing Sudoku

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

Playing Sudoku – Learning goals

- Another example of how to encode a problem in Prolog
- Understand how to improve performance by controlling the search space

Make the problem easier

A	B	4	D
E	2	G	H
I	J	1	L
M	3	O	P

[A,B,4,D,
E,2,G,H,
I,J,1,L,
M,3,O,P]


```
range([]).
range([H|T]) :- range(1,5,H), range(T).

diff([A,B,C,D]) :-   A =\= B, A =\= C, A =\= D,
                     B =\= C, B =\= D,
                     C =\= D.

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,C,D]),diff([E,F,G,H]),
                                           diff([I,J,K,L]),diff([M,N,O,P]).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,E,I,M]),diff([B,F,J,N]),
                                           diff([C,G,K,O]),diff([D,H,L,P]).

box([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :-  diff([A,B,E,F]),diff([C,D,G,H]),
                                           diff([I,J,M,N]),diff([K,L,O,P]).

sudoku(L) :- range(L), rows(L), cols(L), box(L).
```


?

Question 1

Solution strategy

What strategy did we adopt to build our solution?

1 generate and test
2 graph search
3 ad-hoc program

Our program generates lots of implausible answers

- The first call to range generates a board of all 1's
- We can do better by reducing the search space
- Use list permutations:
  - all rows are a permutation of [1,2,3,4]
  - all columns are a permutation of [1,2,3,4]
  - all boxes are a permutation of [1,2,3,4]


```
diff(L) :- perm([1,2,3,4],L).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,C,D]),diff([E,F,G,H]),
                                          diff([I,J,K,L]),diff([M,N,O,P]).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,E,I,M]),diff([B,F,J,N]),
                                          diff([C,G,K,O]),diff([D,H,L,P]).

box([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,E,F]),diff([C,D,G,H]),
                                         diff([I,J,M,N]),diff([K,L,O,P]).

sudoku(L) :- rows(L), cols(L), box(L).
```


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

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/

We need a smarter solving strategy....

Playing Sudoku – Learning goals

- Another example of how to encode a problem in Prolog
- Understand how to improve performance by controlling the search space

Constraint solving

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming in Prolog
Ian Lewis

The Sudoku search space was too big

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

Constraint solving – Learning goals

- Unification can be seen as a specific instance of constraint solving
- Understand how constraint propagation works
- Be able to solve simple constraint problems

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)

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

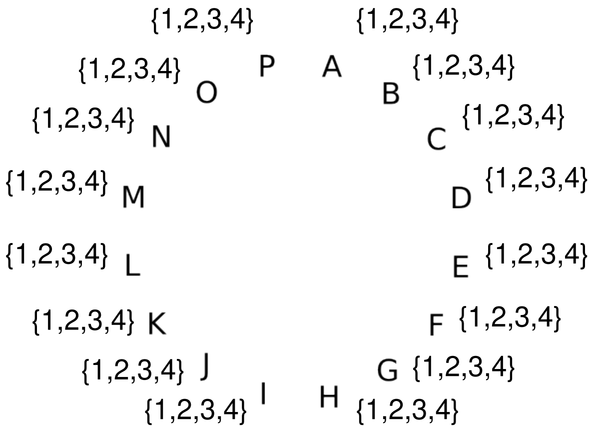
Sudoku can be expressed as constraints

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

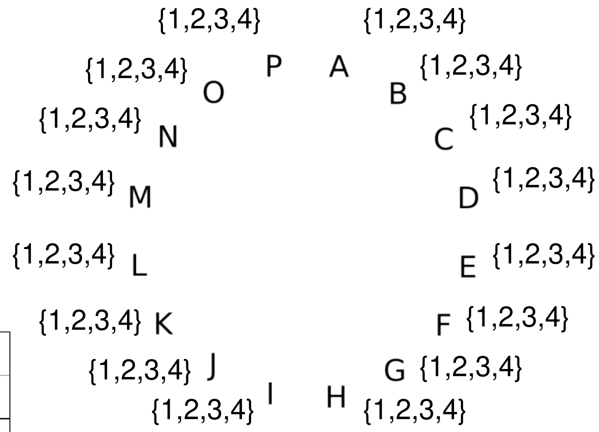
Variables and Domains

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

Express Sudoku as a Constraint Graph

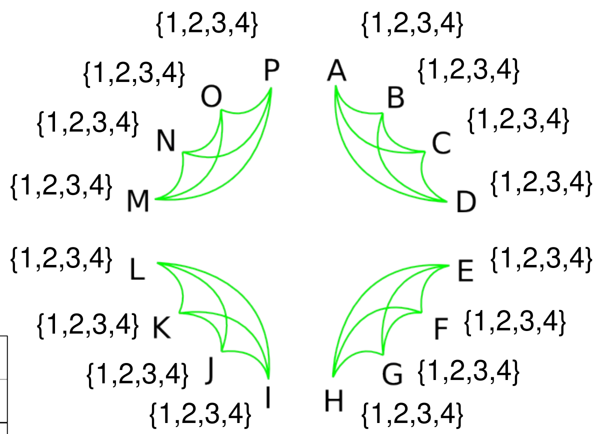


Constraints: All variables in rows are different



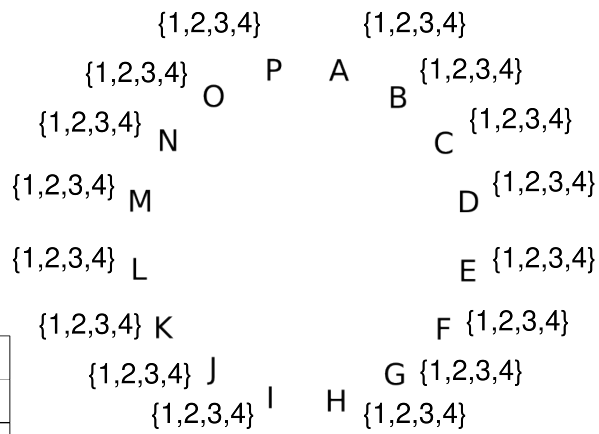
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints: All variables in rows are different



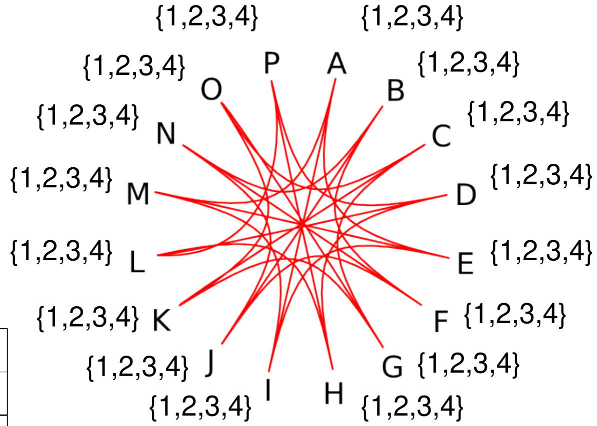
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints: All variables in columns are different



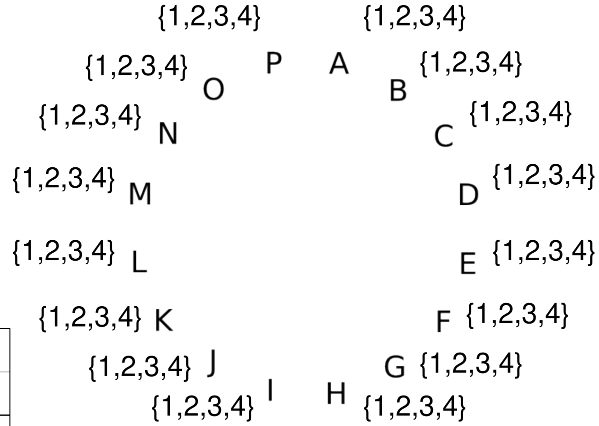
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints: All variables in columns are different



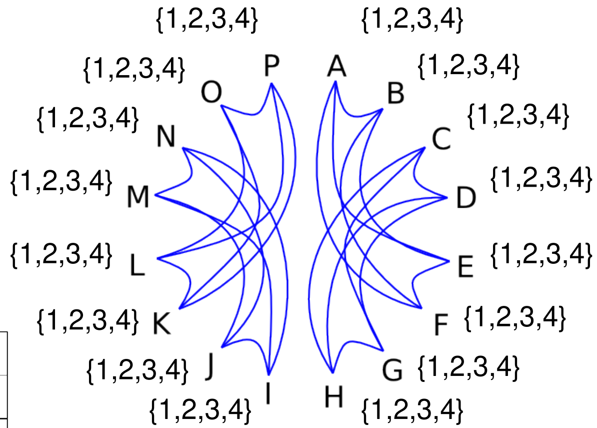
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints: All variables in boxes are different



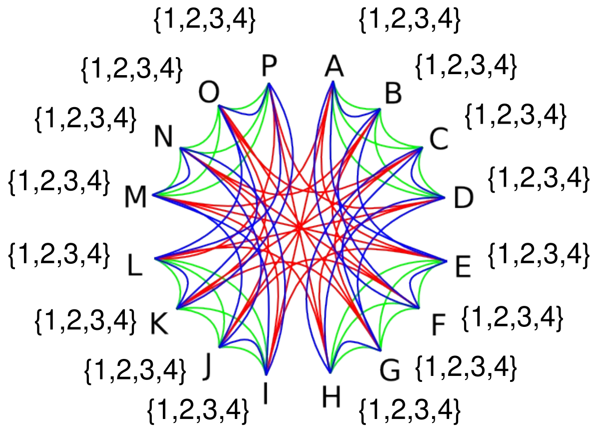
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints: All variables in boxes are different



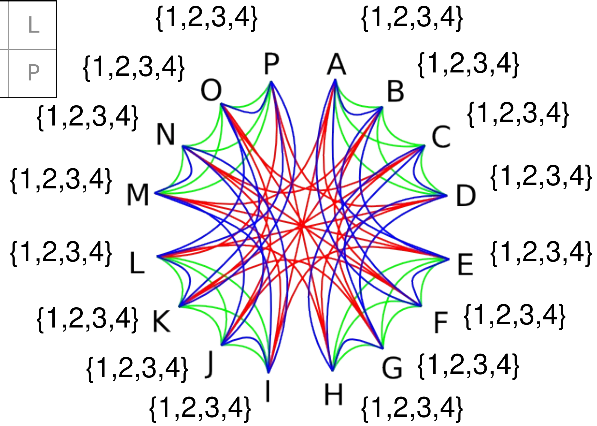
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

All constraints

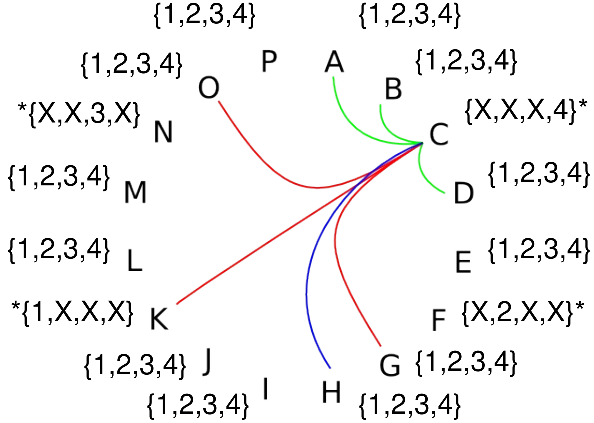


Reduce domains according to initial values

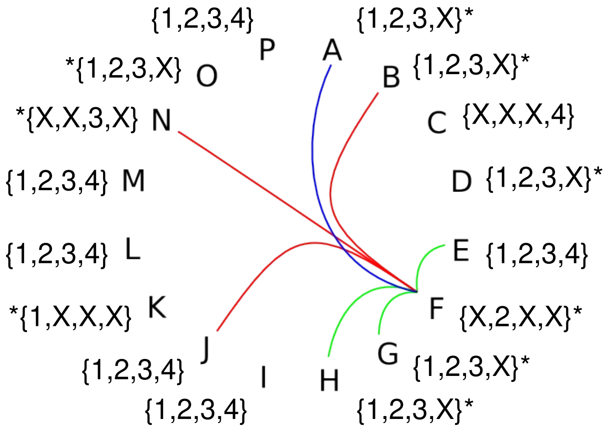
A	B	4	D
E	2	G	H
I	J	1	L
M	3	O	P



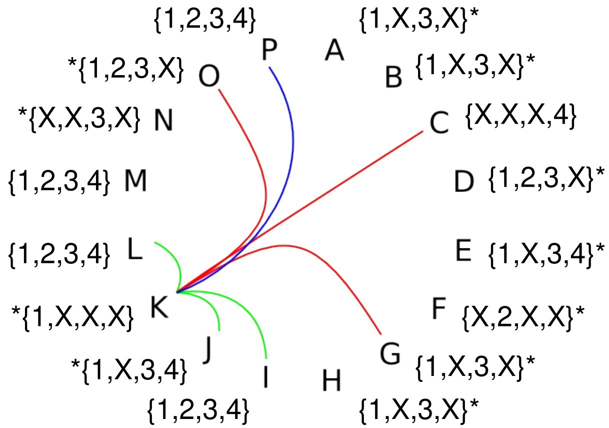
Update constraints connected to C



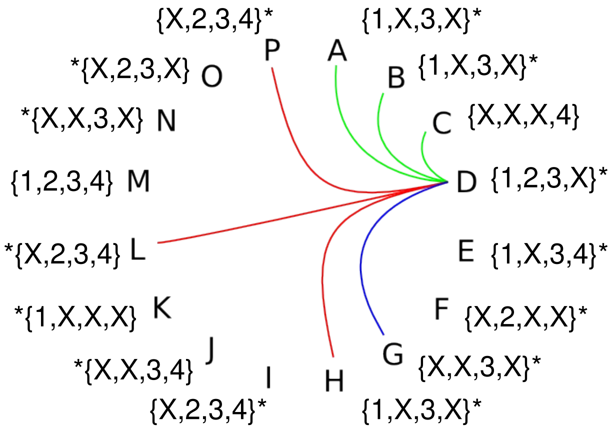
Update constraints connected to F



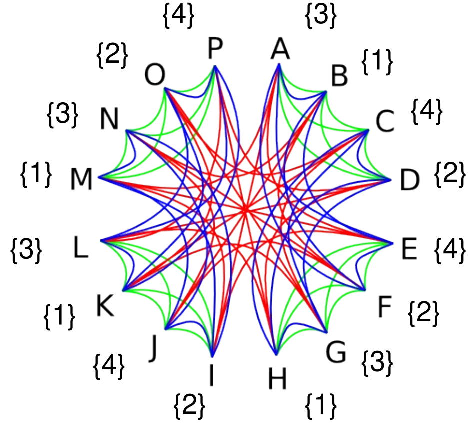
Update constraints connected to K



Update constraints connected to D

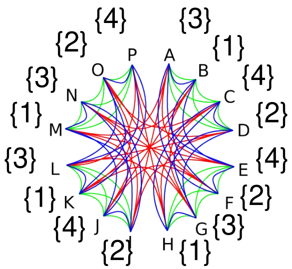


Eventually the algorithm will converge and no further changes occur



Outcome 1: Single valued domains

We have found a unique solution to the problem



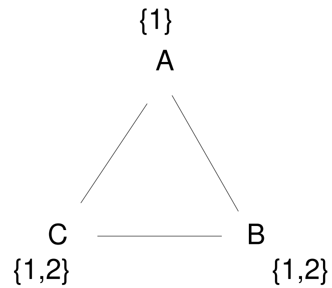
3	1	4	2
4	2	3	1
2	4	1	3
1	3	2	4

Outcome 2: Some empty domains

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

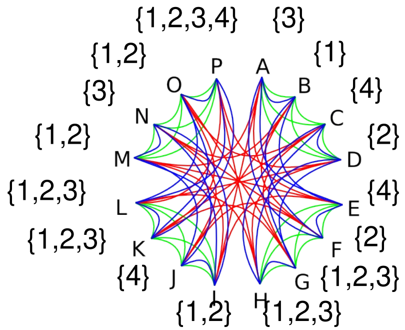
Variables
- A ∈ {1}
- B ∈ {1,2}
- C ∈ {1,2}

Constraints
- A ≠B, A ≠C, B ≠C



Outcome 3: Some multivalued domains

		4	
	2		
	3		



Not all combinations of the remaining possibilities are global solutions

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

Using CLP in Prolog

```
:- use_module(library(bounds)).

diff(L) :- L in 1..4, all_different(L).

rows([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,B,C,D]), diff([E,F,G,H]),
                                           diff([I,J,K,L]), diff([M,N,O,P]).

cols([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :- diff([A,E,I,M]), diff([B,F,J,N]),
                                           diff([C,G,K,O]), diff([D,H,L,P]).

box([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P]) :-  diff([A,B,E,F]), diff([C,D,G,H]),
                                           diff([I,J,M,N]), diff([K,L,O,P]).

sudoku(L) :- rows(L), cols(L), box(L), label(L).
```


Constraint solving – Learning goals

- Unification can be seen as a specific instance of constraint solving
- Understand how constraint propagation works
- Be able to solve simple constraint problems