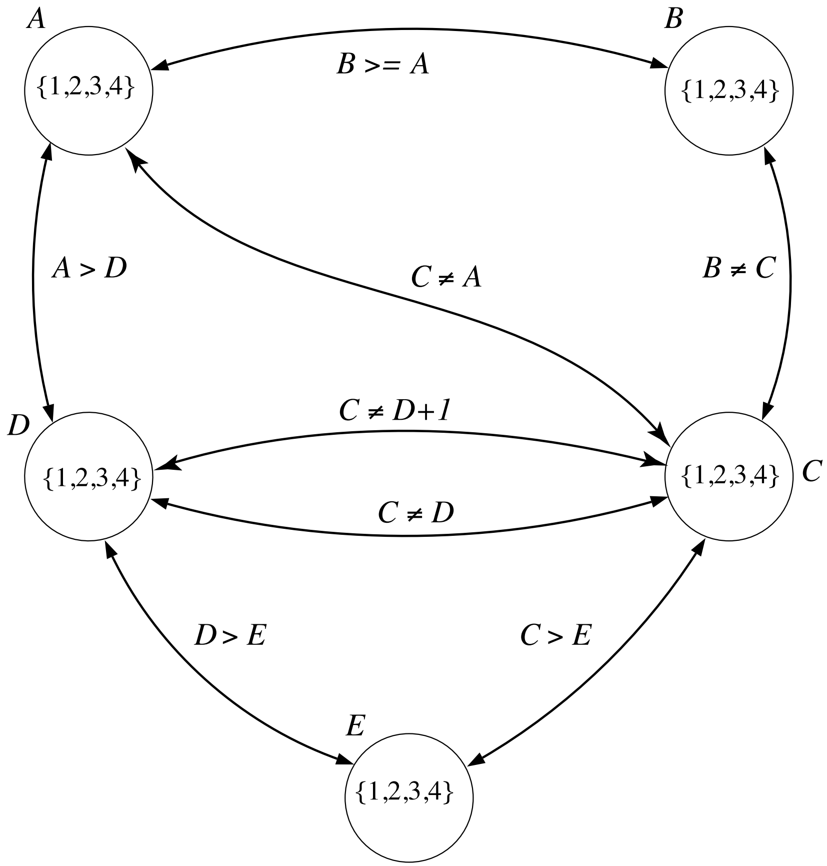
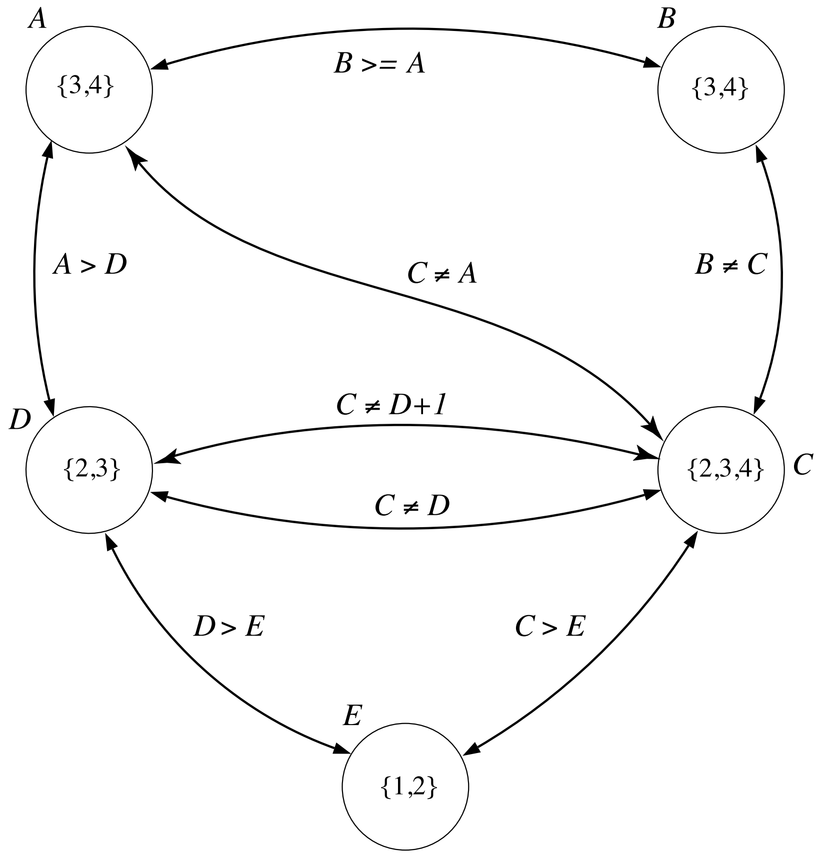
Constraint Programming

Finite Domain



Arc Consistency

Arc	Relation	Value(s) Removed
$\langle D, E\rangle$	$D > E$	$D = 1$
$\langle E, D\rangle$	$D > E$	$E = 4$
$\langle C, E\rangle$	$C > E$	$C = 1$
$\langle D, A\rangle$	$A > D$	$D = 4$
$\langle A, D\rangle$	$A > D$	$A = 1 \& A = 2$
$\langle B, A\rangle$	$B \geq A$	$B = 1 \& B = 2$
$\langle E, D\rangle$	$D > E$	$E = 3$



Constraint Ordering is Important

Much faster !

```
solve(A, B, C, D, E) :-
    domain(C),
    domain(D),
    domain(A),
    domain(B),
    domain(E),
    A > D,
    D > E,
    C =\= A,
    C > E,
    C =\= D,
    B >= A,
    B =\= C,
    C =\= D + 1.
```

```
solve(A, B, C, D, E) :-
    domain(C),
    domain(D),
    C =\= D,
    C =\= D + 1,
    domain(A),
    A > D,
    C =\= A,
    domain(B),
    B >= A,
    B =\= C,
    domain(E),
    C > E,
    D > E.
```

```
domain(1).
domain(2).
domain(3).
domain(4).
```

CLP(FD)

- SWI Prolog (and others) include constraint programming libraries
  - Others: ECLiPSe, YAP, GNU-Prolog, Ciao, …
- Non-standard extensions, so beware!
- They change Prolog’s normal depth-first search for variable bindings to incorporate constraint solving methods (including arc consistency, etc).

Example

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

solve(A, B, C, D, E) :-
    [A, B, C, D, E] ins 1..4,
    A #> D,
    D #> E,
    C #\= A,
    C #> E,
    C #\= D,
    B #>= A,
    B #\= C,
    C #\= D + 1,
    labeling([], [A, B, C, D, E]).
```

Declare domain

'#' means operator is a constraint, satisfied by constraint solving rather than depth-first search

Assign values

Solution to FD Problem

```
?- solve(A, B, C, D, E).
```

A = 3,
B = 3,
C = 4,
D = 2,
E = 1 ;

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

Consistency Check

```
?- constraints(A, B, C, D, E).

A in 3..4,
C #\= A,
B #>= A,
D #=< A + -1,
C in 2..4,
C #\= D+1,
B #\= C,
C #\= D,
E #=< C + -1,
D in 2..3,
E #=< D + -1,
E in 1..2,
B in 3..4
```

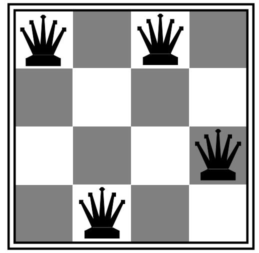
Cryptarithmetic

```
  D O N A L D
+ G E R A L D
  -----------
  R O B E R T
```

Cryptarithmetic

```
% Cryptarithmetic puzzle DONALD + GERALD = ROBERT in CLP(FD)

:- use_module(library(clpfd)).

solve([D,O,N,A,L,D],[G,E,R,A,L,D],[R,O,B,E,R,T]) :-
    Vars = [D,O,N,A,L,G,E,R,B,T],                     % All variables in the puzzle
    Vars ins 0..9,                                     % They are all decimal digits
    all_different(Vars),                               % They are all different
    100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
    100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
    100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
    labeling([], Vars).

?- solve(X, Y, Z).

X = [5, 2, 6, 4, 8, 5],
Y = [1, 9, 7, 4, 8, 5],
Z = [7, 2, 3, 9, 7, 0]
```

N-Queens



[1, 4, 1, 3]

```
% The k-th element of Cols is the column number of the queen in row k.

:- use_module(library(clpfd)).

n_queens(N, Qs) :-
    length(Qs, N),
    Qs ins 1..N,
    safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :-
    safe_queens(Qs, Q, 1),
    safe_queens(Qs).
```

```
safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
    Q0 #\= Q,
    abs(Q0 - Q) #\= D0,
    D1 #= D0 + 1,
    safe_queens(Qs, Q0, D1).

?- n_queens(8, Qs), labeling([ff], Qs).
```

CLP(R) - constraints over reals

Mortgage relation between the following arguments:

- *P* is the balance at *T0*
- *T* is the number of interest periods (e.g., years)
- *I* is the interest ratio where e.g., `0.1` means 10%
- *B* is the balance at the end of the period
- *MP* is the withdrawal amount for each interest period.

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

mg(P, T, I, B, MP):-
    { T = 1,
      B + MP = P * (1 + I)
    }.
mg(P, T, I, B, MP):-
    { T > 1,
      P1 = P * (1 + I) - MP,
      T1 = T - 1
    },
    mg(P1, T1, I, B, MP).

?- mg(1000, 30, 5/100, B, 0).

B = 4321.942375150665
```

Back to Standard Prolog:

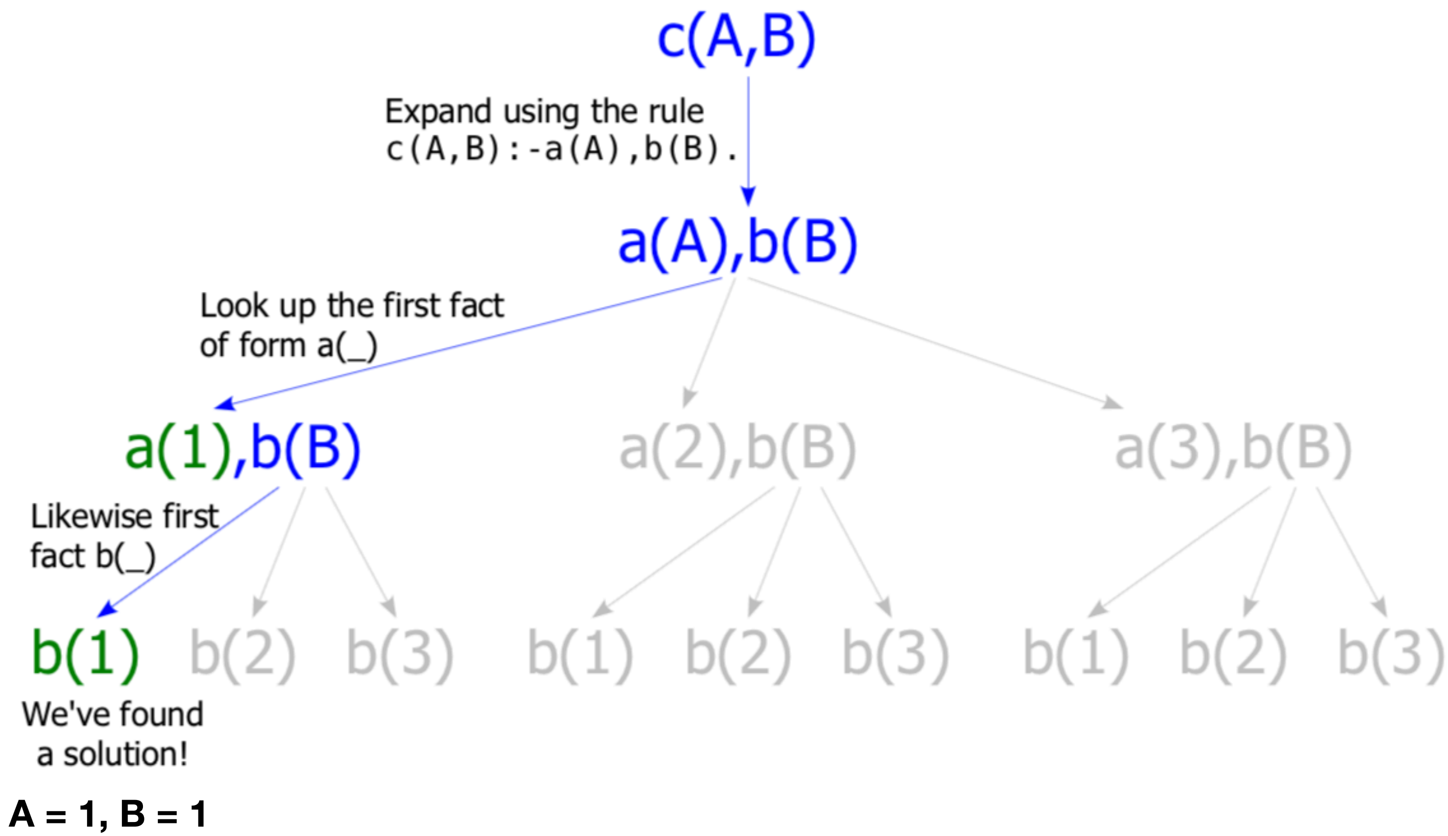
Controlling Execution

Prolog – Finding Answers

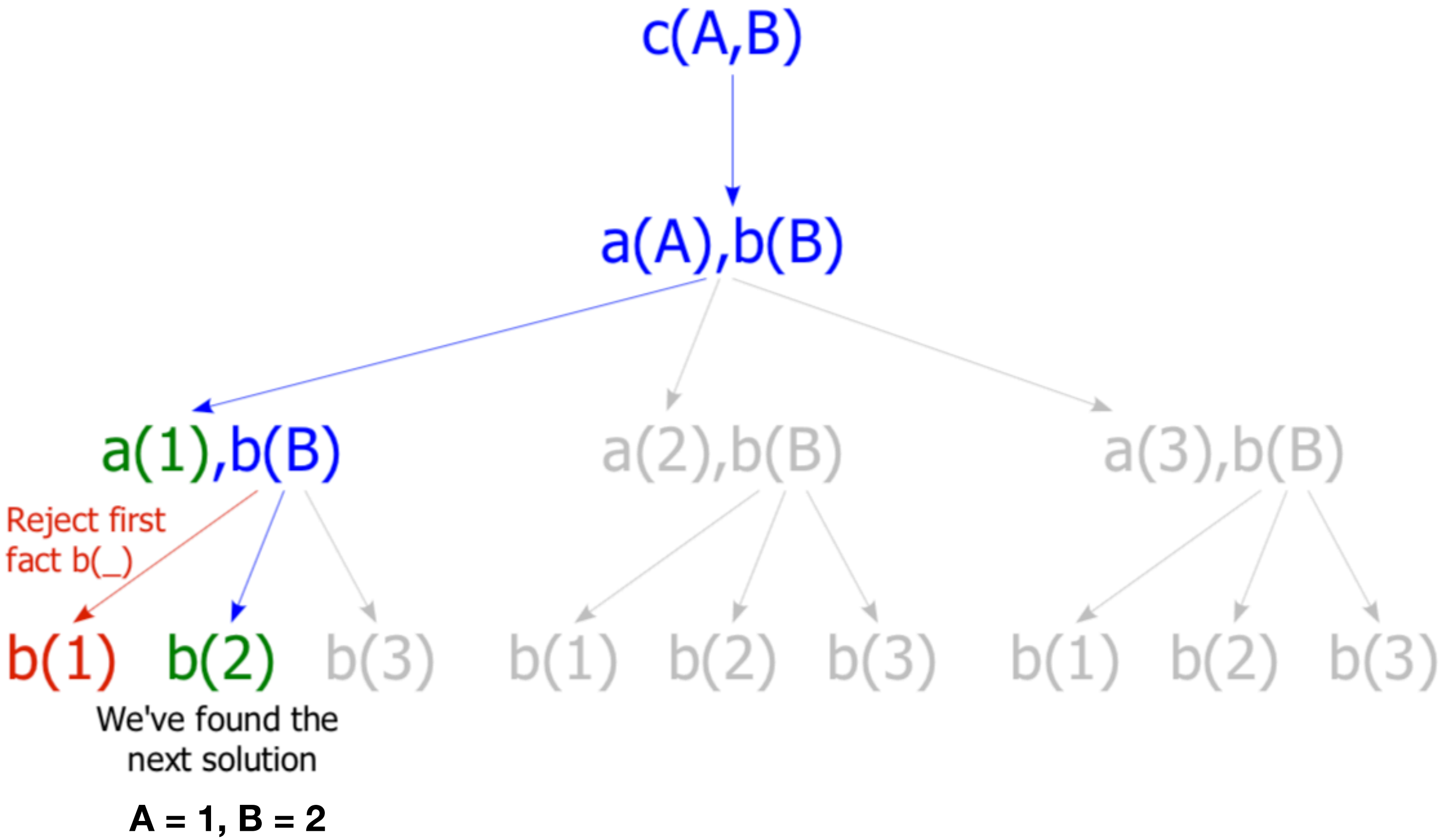
Prolog uses depth first search to find answers

```
a(1).
a(2).
a(3).
b(1).
b(2).
b(3).

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

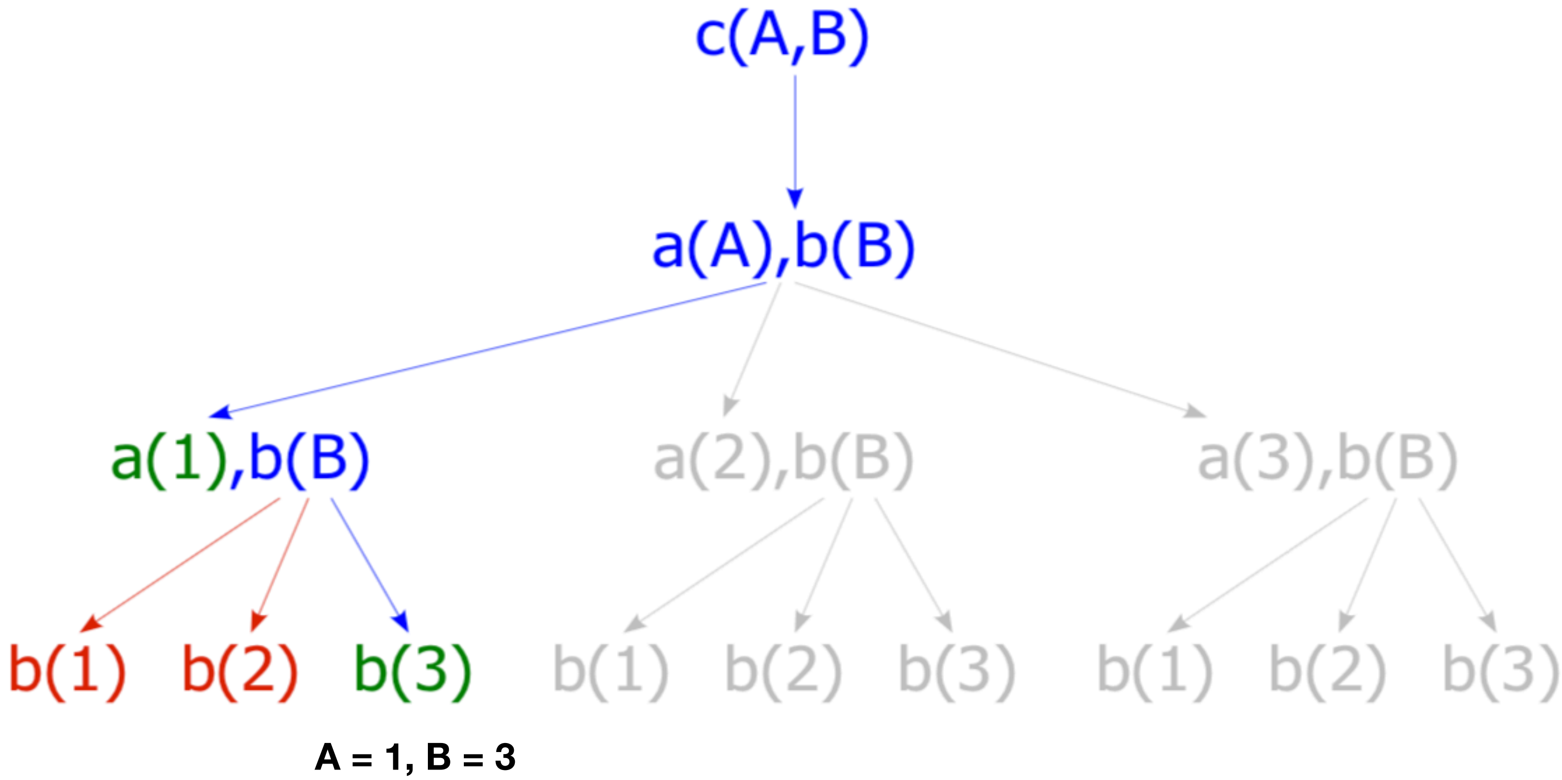
Depth-first solution of query c(A,B)



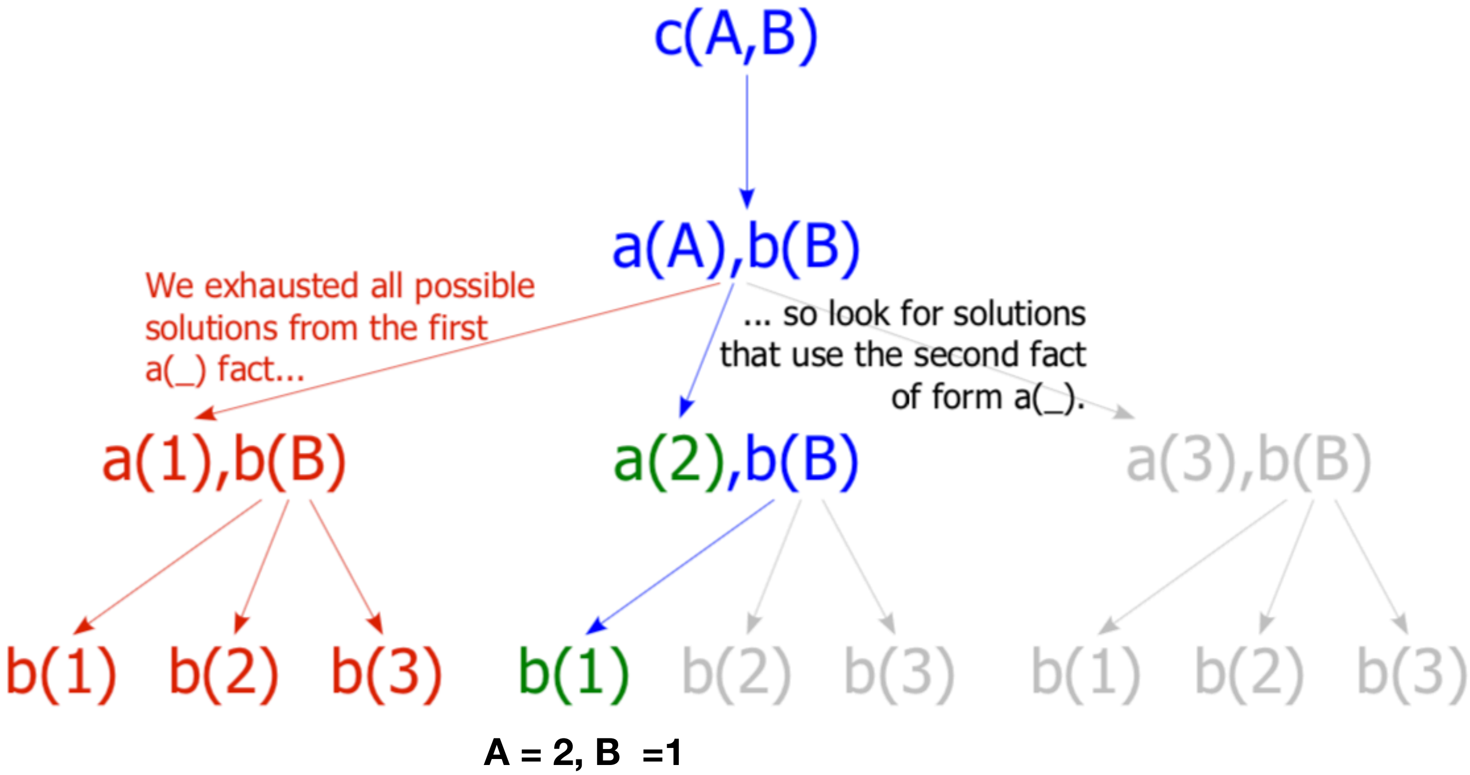
Backtrack to find another solution



Backtrack to find another solution



Backtrack to find another solution

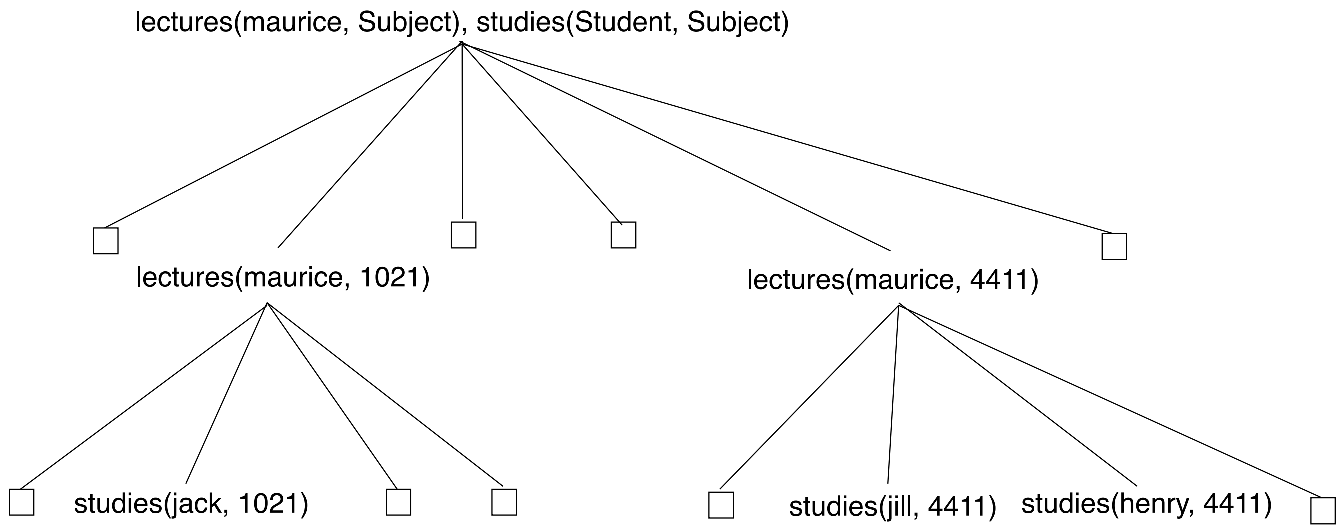


The Cut (!)

- Sometimes we need a way of preventing Prolog from finding all solutions
- The *cut* operator is a built-in predicate that prevents backtracking
- It violates the declarative reading of a Prolog programming
- Use it *VERY sparingly!!!*

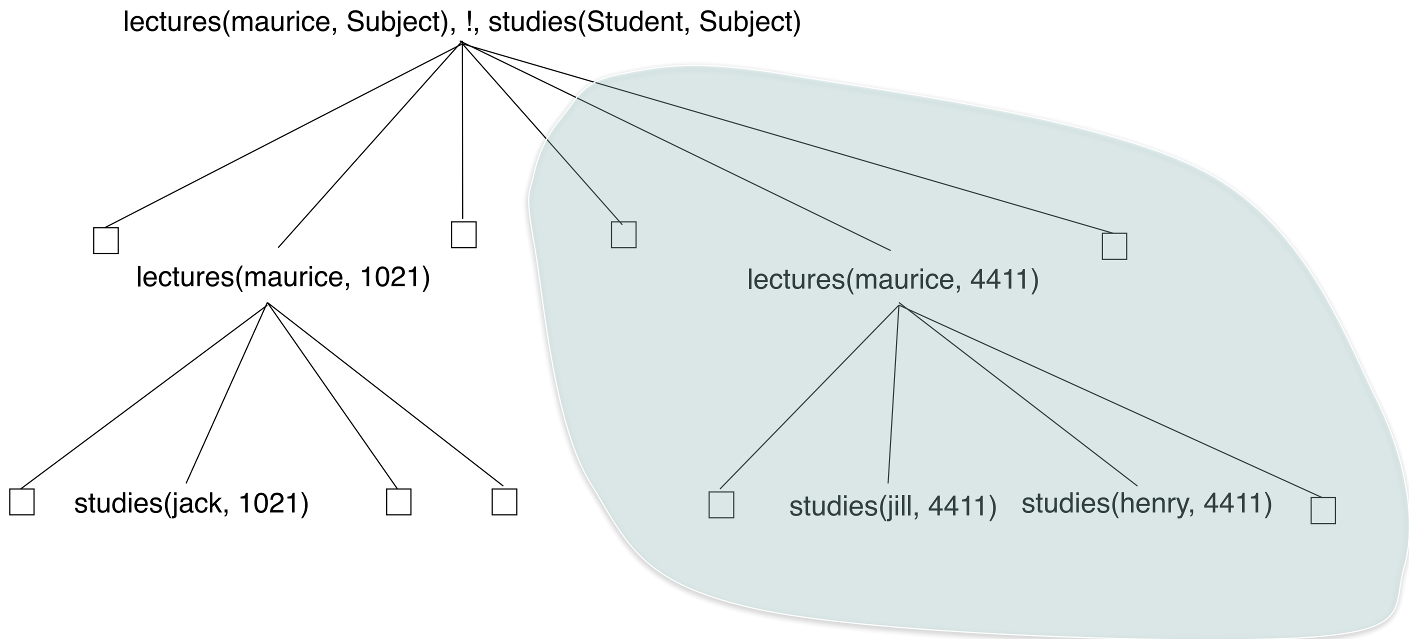
Backtracking

```
lectures(maurice, Subject), studies(Student, Subject)?
Subject = 1021
Student = jack ;

Subject = 4411
Student = Jill ;

Subject = 4411
Student = Henry
```

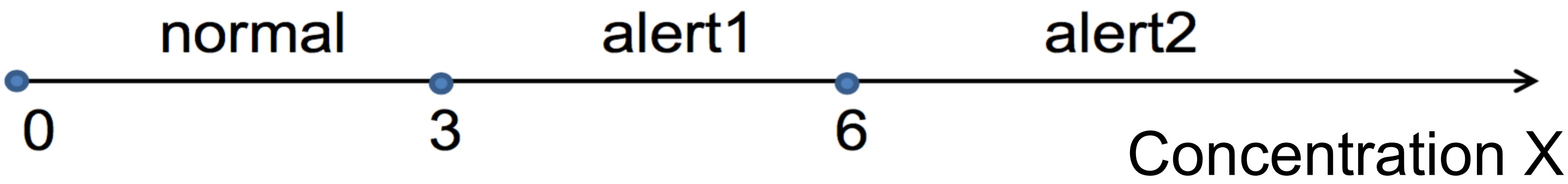


Cut prunes the search

```
lectures(maurice, Subject), !, studies(Student, Subject)?
Subject = 1021
Student = jack ;

Subject = 4411
Student = Jill ;

Subject = 4411
Student = Henry
```



Example



Rules for determining the degree of pollution

Rule 1: if $X < 3$ then $Y = \text{normal}$

Rule 2: if $3 \leq X$ and $X < 6$ then $Y = \text{alert1}$

Rule 3: if $6 \leq X$ then $Y = \text{alert2}$

In Prolog: **`f(Concentration, Pollution_Alert)`**

```
f(X, normal) :-  X < 3.              % Rule1
f(X, alert1) :-  3 =< X, X < 6.      % Rule2
f(X, alert2) :-  6 =< X.             % Rule3
```

Alternative Version

```
f(X, normal) :-  X < 3, !.          % Rule1
f(X, alert1) :-  X < 6, !.          % Rule2
f(X, alert2).                       % Rule3
```

Which version is easier to read?

Operators

Operator Notation

- Operators are just compound (i.e. functional) terms

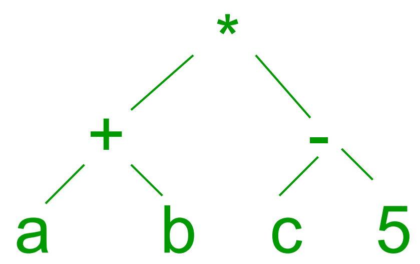
```
2*a + b*c = +(*(2,a), *(b,c))
```

- `+`, `*` are infix operators in Prolog
  - They are only interpreted as arithmetic expressions when the appear on the right-hand side of the ***is*** operator.

Operator Expressions are also Trees

- For example: (a + b) * (c - 5)
- Written as an expression with the functors:

*(+(a, b), -(c, 5))



Operators in Prolog

- You can define your own operators.

    :- op(Precedence, Type, Name).

- Precedence is a number between 0 and 1200.

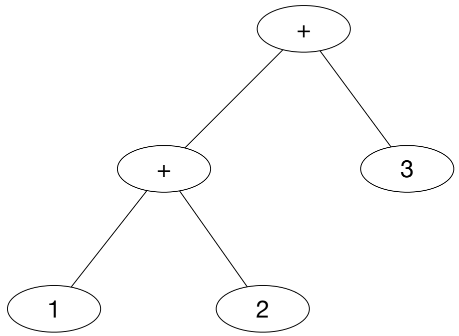
For example,

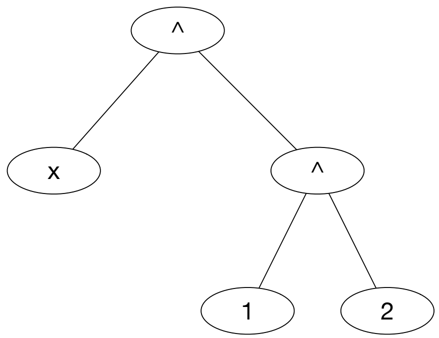
- the precedence of “ = “ is 700,
- the precedence of “ + ” is 500,
- the precedence of “ * “ is 400.

Operators in Prolog

:- op(Precedence, Type, Name).

- Type is an atom specifying the associativity of the operator.
- Infix operators:
  - yfx - left associate (e.g. 1 + 2 + 3 = ((1 + 2) + 3)
  - xfy - right associative (e.g. x ^ 2 ^ 2 = (x ^ (2 ^ 2))
  - xfx - non-associative (e.g. wa = green; a = b = c is not valid)
- Prefix operators
  - fy, fx (associative, non-associative)
- Postfix opertators
  - yf, xf (associative, non-associative)





Predefined Operators

- Operators with the same properties can be specified in one statement by giving a list of their names instead of a single name as third argument of op.
- Operator definitions don't specify the meaning of an operator, only how it can be used syntactically.
- Operator definition doesn't say how a query involving operator is evaluated to true.

```
:- op(1200, xfx, [:-, —>]).
:- op(1200,  fx, [:-, ?-]).
:- op(1100, xfy, [ ; ]).
:- op(1000, xfy, [‘,’]).
:- op( 700, xfx, [=, is, =.., ==. \==, \==, =:=, =\=, <, >, =<, >=]).
:- op( 500, yfx, [+, -]).
:- op( 500,  fx, [+, -]).
:- op( 300, xfx, [ mod ]).
:- op( 200, xfy, [ ^ ]).
```

User Defined Operators

Relations can be defined as operators, e.g.

```
has(peter, information).
supports(floor, table).
```

can be written with operators:

```
:- op(600, xfx, has).
:- op(600, xfx, supports).

peter has information.
floor supports table.
```

Example - IF statement

```
% Operator declarations
:- op(500, fx,  if).
:- op(400, xfx, then).
:- op(300, xfx, else).


% Interpreter

if Condition then S1 else S2 :-
        Condition, !, S1.
if Condition then S1 else S2 :-
        S2.
```

% Don’t allow backtracking if Condition is true

Built-in Predicates

- Testing the type of terms
- Construction and decomposition of terms: =. . , functor, arg, name
- Comparison
- *bagof*, *setof* and *findall*
- Input, output

Testing the type of terms

var(X)	true if X is unbound or instantiated to an unbound variable
nonvar(X)	X is not a variable or instantiated to an unbound variable
atom(X)	true if X is an atom
integer(X)	true if X is an integer
float(X)	true if X is a real number
number(X)	true if X is a number
atomic(X)	true if X is a number or an atom
compound(X)	true if X is a compound term (a structure)
is_list(X)	true if X is [] or a non-empty list

Example: Arithmetic Operations

```
…,
number( X),          % Value of  X number?

number( Y),          % Value of  Y number?

 Z is X + Y,         % Then addition is possible
…
```

Construction and decomposition of terms:

=.. , *functor, arg, name*

```
Term =.. [Functor, Arg1, Arg2, Arg3, ...]   % "univ"

Term =.. L
```

true if **`L`** is a list that conations the principal functor of **`Term`**, followed by its arguments.

Example:

```
?- f(a, b) =.. L.
L = [f, a, b]

?- T =.. [rectangle, 3,5].
T = rectangle(3, 5)
```

Construction and decomposition of terms:

=.. *, functor, arg, name*

```
?- functor(a(), N, A).
N = a, A = 0.

?- functor(T, a, 0).
T = a.

?- arg(2, f(a, b), X).
X = b.

?- arg(N, f(a, b), V).
N = 1, V = a ;
N = 2, V = b.
```

Substitute

```
substitute(Subterm, Term, Subterm1, Term1)
```

Find all occurrences of `Subterm` in `Term` and substitute with `Subterm1` to get `Term1`.

```
?- substitute(sin(x), 2*sin(x)*f(sin(x)), t, F).
F = 2*t*f(t)
```

Substitute

```
% Case 1: Substitute whole term
substitute(Term, Term, Term1, Term1) :- !.

% Case 2: Nothing to substitute if Term atomic
substitute(_, Term, _, Term) :-
           atomic(Term), !.              % Term is a constant

% Case 3: Do substitution on arguments
substitute(Sub, Term, Sub1, Term1) :-
    Term =.. [F | Args],                 % Get arguments
    substlist(Sub, Args, Sub1, Args1),   % Perform substitution on them
    Term1 =.. [F | Args1].               % Construct Term1

% substlist(SubTerm, Term_List, NewSubTerm, NewTerm_List)
```

Substitute

```
% substlist(SubTerm, Term_List, NewSubTerm, NewTerm_List)

% End of list, nothing to do
substlist(_, [], _, []).

% Otherwise, try substituting recursively
substlist(A, [X|List], B, [Y|List1]) :-
    substitute(A, X, B, Y),
    substlist(A, List, B, List1).
```

Example - Use of *substitute / 4*

```
?- E0 = (a+b) * (a-b),
      substitute(a, E0, 6, E1),
      substitute(b, E1, 3, E2),
      Value is E2.

E1 = (6+b) * (6-b)
E2 = (6+3) * (6-3)
Value = 27
```

Comparison

X = Y	true if X and Y match
X == Y	if X and Y are identical
X \== Y	if X and Y are not identical
X @< Y	X is lexicographically smaller then Y, term X precedes term Y by alphabetical or numerical ordering (e.g. paul @< peter)

findall, bagof, setof

% Find all values of **Object** that satisfy **Condition** and collect in **List**
findall(Object, Condition, List)

% Same as findall except only stores unique values and ***fails*** if no solution found
bagof(Object, Condition, List

% Find all values of **Object** that satisfy **Condition** and collect in ***sorted*** **List**
setof(Object, Condition, List)

Example: robot world

```
?- findall(B, on(B,_), L).                              % L is a List of all blocks
L = [a,b,c,d,e]
```

Procedure *findall*, *bagof, setof*

Examples:

```
child(joze, ana).     child(miha, ana).
child(lili, ana).     child(lili, andrej).

?- findall(X, child(X, ana), S).
S = [joze, miha, lili]

?- setof(X, child(X, ana), S).
S = [joze, lili, miha]

?- findall(X, child(X, Y), S).
S = [joze, miha, lili, lili]

?- bagof(X, child(X, Y), S).
S = [joze, miha, lili]
Y = ana;
```

Input / Output

```
consult(File)    % Load File into Prolog's database

see(File)        % File becomes the current input stream

see(user)        % user input (i.e. from terminal)

seen             % close the current input stream

seeing(X)        % binds X to the current input file

tell(File)       % File becomes the current output  stream

tell(user)       % user output (i.e. output to terminal)

told             % close the current output stream

telling(X)       % binds X to the current output file
```

Input / Output

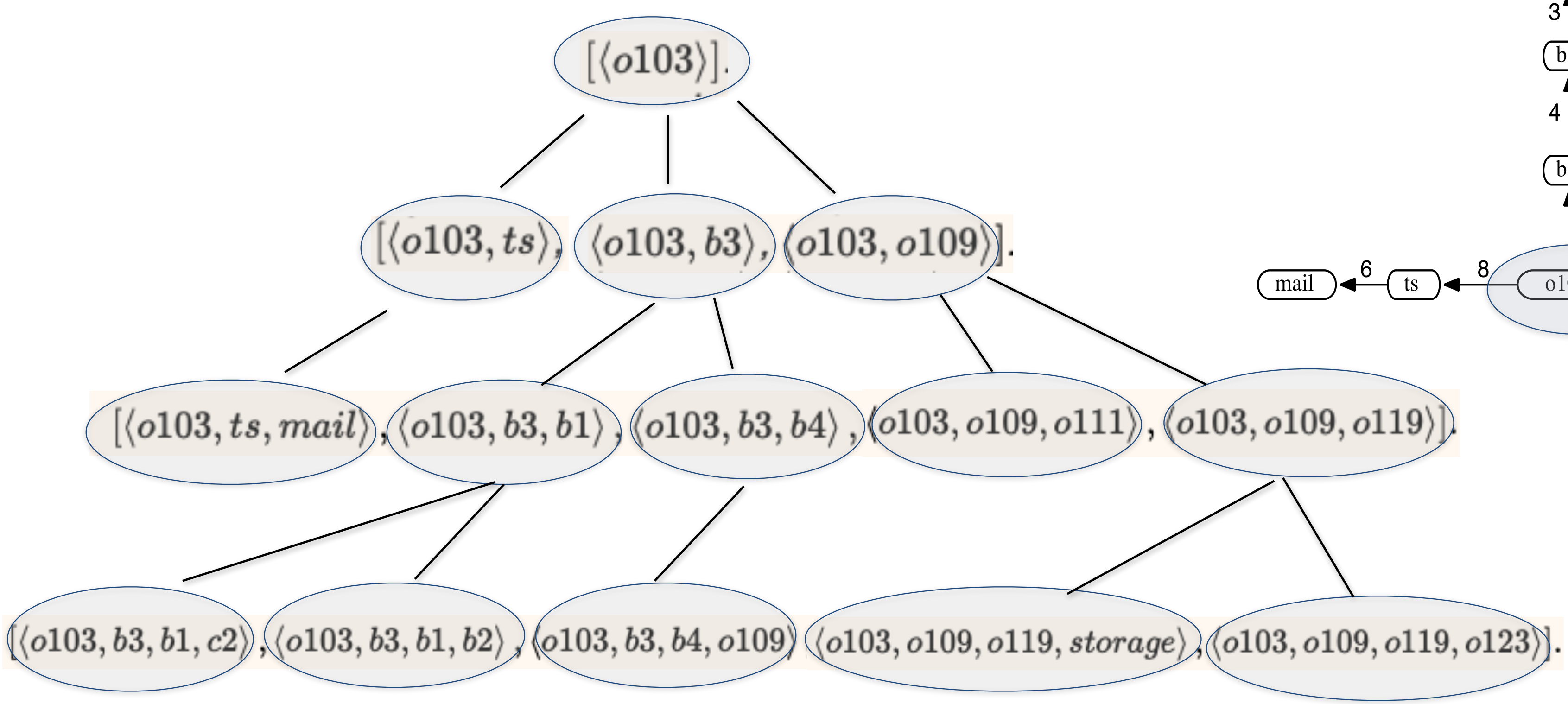
```
write(Term)    % write Term to current output stream

writeln(Term) % write Term and append newline

nl             % write newline to current output stream

read(Term)     % read Term from current input stream
```

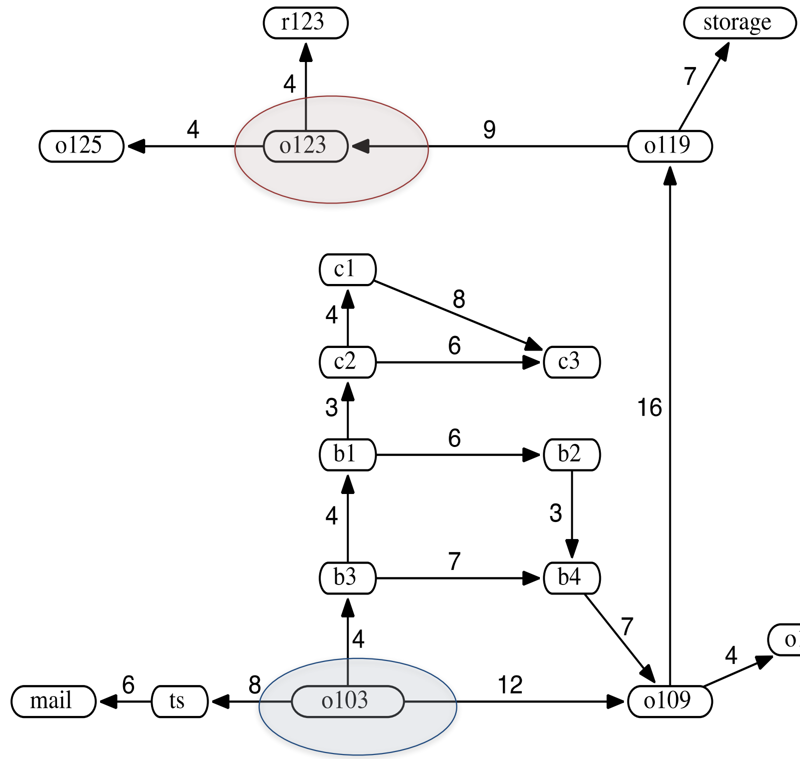
SWI Prolog Manual

There is a lot more to learn in the user manual:

https://www.swi-prolog.org/pldoc/doc_for?object=manual

Breadth-First Search





Each path on the frontier has either the same number of arcs or one more arc than the next element of the frontier that will be selected.

Breadth-First Search

```
solve(Start, Solution)  :- breadthfirst([[Start]], Solution).

breadthfirst([[Node|Path]|_], [Node|Path]) :- goal(Node).
breadthfirst([Path|Paths], Solution)  :-
    extend(Path, NewPaths),
    append(Paths, NewPaths, Paths1),
    breadthfirst(Paths1, Solution).

extend([Node|Path], NewPaths)  :-
    findall([Neighbour, Node|Path], new_neighbour([Neighbour, Node|Path]), NewPaths).

new_neighbour([Neighbour, Node|Path]) :-
    edge(Node, Neighbour),
    \+ member(Neighbour, [Node|Path]).
```