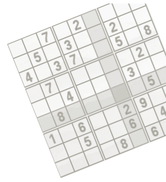


Constraint Programming

Practical Exercises - Prolog

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

How Is CSP Used in Practice?

- Exploiting the principles of constraint satisfaction, but **programming them ad-hoc** for a given problem.
  - flexibility (complete customisation to a given problem)
  - speed (for a given problem)
  - expensive in terms of initial development and maintenance
- Exploiting an **existing constraint solver**.
  - usually integrated to a host language as a library
  - contains core constraint satisfaction algorithms
  - the user can focus on problem modelling
  - It is hard to modify low-level implementation (domains,…)
  - Sometimes possible to implement own constraints
  - frequently possible to implement own search strategies

Constraint Solvers

`http://ktiml.mff.cuni.cz/~bartak/constraints/systems.html`

- Available Services:
  - Implementation of **data structures** for modelling variable domains and constraints
  - core framework for **constraint propagation**
  - **filtering algorithms** for many constraints (including global constraints)
  - core **search strategies** including variable and value ordering heuristics
  - **Interface** for writing own constraints
- Classification of solvers:
  - stand-alone solvers
    - Minion
  - own programming/modelling language
    - Mozart, OPL, Comet, CHR
  - host programming languge
    - **Prolog**: ECLiPSe, SICStus Prolog
    - **C/C++**: ILOG Solver, Gecode
    - **Java**: Choco, JaCoP

SICStus Prolog

`http://sicstus.sics.se/`

- A commercial product with students licence
- **Features**
  - ISO standard Prolog
  - support for many computer platforms (Win, MacOS X, Linux, Solaris)
  - development environment GNU Emacs/SPIDER(Eclipse)
  - many libraries including clpfd
  - possibility to build stand-alone and embedded applications
- **Why Prolog?**
  - simple syntax
  - compact - short programs can do a lot of things
  - natural integration of constraints
  - search alagorithm is core solving framework

Prolog is a deductive system that finds answers to **queries** using a knowledge base consisting of **facts** and **rules**.

Where is the programming?

– writing the database of facts and rules
– Prolog interpreter deduces the answer automatically
↳ **declarative programming**

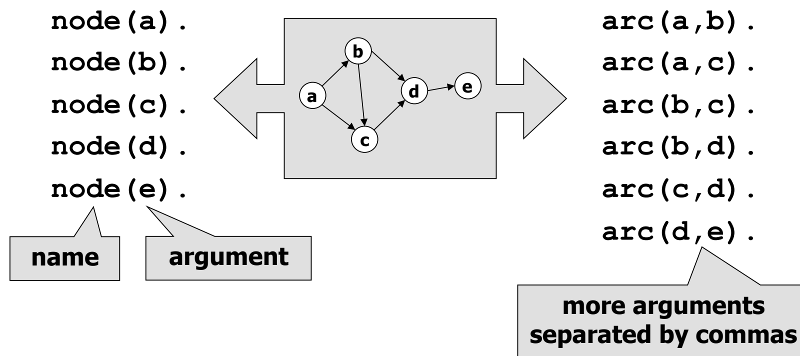
- Prolog source files
  - *.pl
- Prolog database
- Queries

```
member(X,[X|_]).
member(X,[_|T]):-
            member(X,T).

delete([],_X,[]).
delete([X|T],X,T).
delete([Y|T],X,[Y|NewT]):-
            X\=Y,
            delete(T,X,NewT).
```

```
arc(a,b).
arc(X,Y).
arc(Y,X).
arc(X,Y).
arc(X,Z),path(Z,Y).
```

```
SICStus 3.11.0 (x86-win32-nt-4):
Mon Oct 20 00:38:10 WEDT 2003
Licensed to visopt.com
| ?-
```


Prolog facts describe basic information about the problem.



It is possible to ask **queries** about the facts stored in the knowledge base:

Prolog prompt **query** **answer**

```
?-node(a).
yes
?-node(bla).
no
?-arc(a,c).
yes
?-arc(a,d).
no
?-path(a,d).
no
```

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

Open Queries

The query may contain **variables** whose values will be found using stored facts:

```
?-node(X).
X=a ;
X=b ;
X=c ;
X=d ;
X=e ;
no
```

a request for an alternative answer

no more answers

```
?-arc(a,X).
X=b ;
X=c ;
no
```

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

Compound Queries

- List of facts is nothing more than a simple database.
- Is it possible to generate an answer that is not stored directly as a fact but that can be combined from several facts?

Yes. It is possible to **query over a combination of facts** from the knowledge base:

```
?-arc(a,Y),arc(Y,Z).
Y=b
Z=c ;
Y=b
Z=d ;
Y=c
Z=d ;
no
```

variables can be shared between simple open queries

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

Syntax Break - Atoms vs Variables

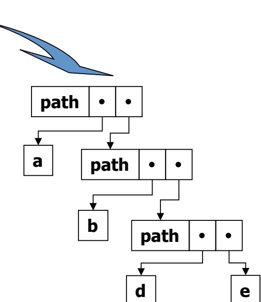
Data (and programs) are expressed using terms

- **Atoms**
  - words consisting of letters, numbers and underscores that start with a non-capital letter
    - `a, arc, john_123, …`
  - words enclosed in single quotas
    - `´Edinburgh´, …`
- **Variables**
  - words consisting of letters, numbers and underscores that start with a capital letter or underscore
    - `X, Node, _noname, …`
  - _ is an anonymous variable
    - two occurences of _ are assumed to be different variables
    - contents is not reported to the user

Syntax Break – Compound Terms

Compound terms express **structured information**

- atoms and variables are terms
- **functor(arg1,…,argn)** is a (compound) term, where functor is an atom and arg1, …, argn are terms
  - arc(a,c)
  - **path(a,path(b,path(d,e)))**
  - tree(tree(a,tree(b,c)),tree(d,e))
  - arc(a,X)
  - …



Deductive Rules

- We can give a name to the query so it can be used repeatedly

```
doubleArc(X,Z):-arc(X,Y),arc(Y,Z).
```

  – This is called a **rule**.
- After defining the rule, we can query it like the facts:

```
?- doubleArc(b,W).
W=d
W=e ;
no ;
?- doubleArc(a,W).
W=c
W=d ;
W=d ;
no ;
```

only variables from the rule head are returned to user

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

How Do Deductive Rules Work?

?-doubleArc(b,W).

– find a rule whose head matches the goal and substitute variables accordingly.

```
doubleArc(b,W):-arc(b,Y),arc(Y,W).
```

– substitute query by the body of the rule

?-arc(b,Y),arc(Y,W).

– find a matching fact (**arc(b,c)**), substitute variables, and remove the fact from the query

?-arc(c,W).

– do the same with the rest (**arc(c,d)**)

W=d ;

– Try alternative facts (**arc(b,d)**,**arc(d,e)**)

W=e ;

no

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

Alternative Rules

- It is possible to define **alternative rules** (disjunction)

```
edge(X,Y):-arc(X,Y).
edge(X,Y):-arc(Y,X).

?-edge(W,b).
W=a ;
W=c ;
W=d ;
no
```

deduced using the first rule

deduced using the second rule

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

How Do Alternative Rules Work?

Just like before, but more alternative rules matches the query.

?-edge(W,b).

– find a rule whose head matches the goal, substitute variables accordingly, and substitute query by the body of the rule

```
edge(W,b):-arc(W,b).
```

?-arc(W,b).

– find all solutions to a query using facts

W=a ;

– try an alternative rule for the original query

```
edge(W,b):-arc(b,W).
```

?-arc(b,W).

– find all solutions to a query using facts

W=c ;

W=d ;

no

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

Recursive rules

- It is possible to use the rule head in its body, i.e., to use **recursion**

```
path(X,Y):-arc(X,Y).
path(X,Y):-arc(X,Z),path(Z,Y).
```

```
?-path(c,W).
W=d ;
W=e ;
no
```

deduced using the first rule and arc(c,d)

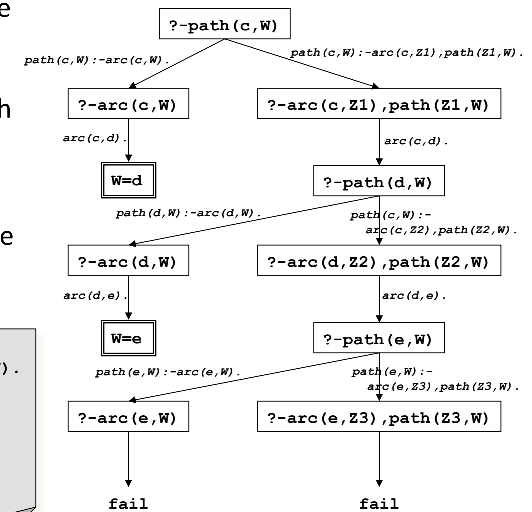
deduced using the second rule through d

```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

How Do Recursive Rules Work?

- Just like before, but the rule may be used several times.
- This is OK because each time a rule is used, its **copy with „fresh“ variables** is generated (like calling a procedure with local variables).

```
path(X,Y):-arc(X,Y).
path(X,Y):-arc(X,Z),path(Z,Y).

node(a).    arc(a,b).
node(b).    arc(a,c).
node(c).    arc(b,c).
node(d).    arc(b,d).
node(e).    arc(c,d).
            arc(d,e).
```



Prolog at Glance

Prolog „program“ consists of **rules** and **facts**.

Each **rule** has the structure **Head:-Body.**

- **Head** is a (compound) term
- **Body** is a query (a conjunction of terms)
  - typically Body contains all variables from Head
- rule semantics: **if Body is true then Head can be deduced**

Fact can be seen as a rule with an empty (true) body.

Query is a conjunction of terms: Q = **Q1,Q2,…,Qn.**

- **Find a rule** whose head matches goal Q1.
  - If there are more rules then introduce a choice point and use the first rule.
  - If no rule exists then backtrack to the last choice point and use an alternative rule there.
- **Use the rule body** to substitute Q1.
  - For facts (Body=true), the goal Q1 disappears.
- **Repeat until empty query** is obtained.

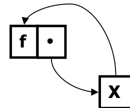
Prolog Technology

Prolog = Unification + Backtracking

- **Unification** (matching)
  - to select an appropriate rule
  - to compose an answer substitution
  - How?
    - make the terms syntactically identical by applying a substitution
- **Backtracking** (depth-first search)
  - to explore alternatives
  - How?
    - resolve the first goal (from left) in a query
    - apply the first applicable rule (from top)

Unification

- a basic mechanism for **information passing**
- syntactic equality of terms via substitution of terms to variables
- `?-X=f(a).` **-> X/f(a)**
- `?-f(X,a)=f(g(b),Y).` **-> X/g(b), Y/a**
- `?-f(X,b,g(a))=f(a,Y,g(X)).` **-> X/a, Y/b**
- `?-X=f(X).` **-> infinite term**
  - **occurs check** can forbid such structures
  - but cyclic structures might be very useful for modeling pointer structures



Selecting Rules

- **Unification is used for rule selection.**

`?-path(f(a),G).`

- rule: path(X,Y):-arc(X,Y).
- **do unification**: **`X=f(a),Y=G`**

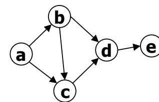
`?-arc(f(a),G).`

- rule (fact): arc(a,b).
- **do unification**: f(a)=a, G=b **-> fail**
- rule (fact): arc(a,c).
- **do unification**: f(a)=a, G=c **-> fail**
- …

Computing Results

- **Unification is used for answer composition.**

```
path(X,Y,path(X,Y)):-
  arc(X,Y).
path(X,Y,path(X,PathZY)):-
  arc(X,Z),
  path(Z,Y,PathZY).

?-path(a,d,P).
P=path(a,path(b,d)) ;
P=path(a,path(b,path(c,d))) ;
P=path(a,path(c,d)) ;
no
```




```
node(a).
node(b).
node(c).
node(d).
node(e).

arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
arc(d,e).
```

Information Passing

- How to obtain the result?
- **Accumulator**
  - Accumulate partial results in a parameter of the procedure.
  - Requires additional parameter with initialization.
- **Composition of substitutions**
  - Compute the result from partial results to be computed later.
  - Specific to Prolog and substitutions.

Accumulator

Symbolic addition of unary represented numbers
(`0, s(0), s(s(0)),` …).
Result is **accumulated** in a parameter of the procedure.

```
plus(0,X,X).
plus(s(X),Y,Z):-plus(X,s(Y),Z).
```

```
?-plus(s(s(s(0))),       s(0)    ,Sum).
?-plus(  s(s(0))  ,     s(s(0))   ,Sum).
?-plus(    s(0)   ,   s(s(s(0)))  ,Sum).
?-plus(      0    ,s(s(s(s(0)))),Sum).
```

Composition

Symbolic addition of unary represented numbers.
Result is a **composition of substitutions** that will be computed later.

```
plus2(0,X,X).
plus2(s(X),Y,s(Z)):-plus2(X,Y,Z).
```

argument for composing the result

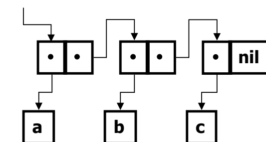
```
?-plus2(s(s(s(0))),s(0),S1). %S1=s(S2)
?-plus2(  s(s(0)) ,s(0),S2). %S2=s(S3)
?-plus2(    s(0)  ,s(0),S3). %S3=s(S4)
?-plus2(      0   ,s(0),S4). %S4=s(0)
```

Homework

- Propose a simple **genealogy database**:
  - facts
    - `man, woman, parent,` …
  - rules
    - `father, mother, son, daughter, grandparent, uncle, aunt, siblings, descendant,` …
- For example **solution** look at
  `http://ktiml.mff.cuni.cz/bartak/prolog/genealogy.html`

Lists

- How to represent a **list of elements**?
- Using terms:
  - a pointer-like structure
  - list(a,list(b,list(c,nil)))



- Prolog provides this structure directly:
  - `[Head|Tail]`
  - `[a,b,c] =[a|[b|[c|[]]]]`
  - Elements can be anything, e.g. a list again
    - `[[q,2], 12, f(a,b), [[]]]`
- This is a syntactic sugar only!

Membership

- How to check **membership in a list**?
- Explore the list from start until the element is found.

```
member(X,[X|_]).
member(X,[_|T]):- member(X,T).

?-member(a,[a,b,a]). -> yes
?-member(X,[a,b,a]). -> X=a; X=b; X=a
?-member(a,L).       -> L=[a|_]; L=[_,a|_], …
```

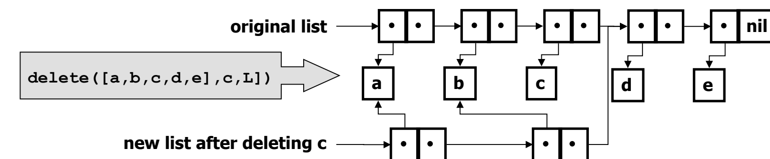
Deleting element

- Delete the first occurrence of X from the list.
  `delete(List,X,ListWithoutX)`

```
delete([],_,[]).
delete([X|T],X,T).
delete([Y|T],X,[Y|NewT]):-
    X\=Y, delete(T,X,NewT).
```

X and Y cannot be unified

- The part of the list before X is duplicated!



Inserting

- Insert X before the list `insert(L,X,LStartWithX)`:

```
insert(L,X,[X|L]).
```

- Add X to the end of the list `add(L,X,LEndWithX)` :

```
add([],X,[X]).
add([Y|T],X,[Y|NewT]):-
  add(T,X,NewT).
```

  - Again, the list is completely duplicated!
  - The procedure can also remove the last element from the list!

```
?-add(NewList,X,[a,b,c,d]).
NewList=[a,b,c]
X=d
```

Concatenating

- concatenate two lists
  - `concat(L1,L2,L)`
  - L1=[a,b,c], L2=[d,e] -> L=[a,b,c,d,e]

```
concat([],L,L).
concat([H|T],L2,[H|NewT]):-
  concat(T,L2,NewT).
```

- Time and space complexity depends on the size of the first list!
- The procedure can also be used to split the list.

```
?-concat(List1,List2,[a,b,c,d]).
List1=[], List2=[a,b,c,d] ;
List1=[a], List2=[b,c,d] ;
…
```

Reverting

- Revert the list
  - **revert(L,Rev)**
  - L=[a,b,c] -> Rev=[c,b,a]

```
revert([],[]).
revert([H|T],Rev):-
  revert(T,RT),
  add(RT,H,Rev).
```

Much better solution is using **accumulator**!

```
revert1(List,Rev):-
  rev(List,[],Rev).
rev([],L,L).
rev([H|T],Acc,Rev):-
  rev(T,[H|Acc],Rev).
```

Slow and memory consuming!

Try to omit **add** (**concat**) in your code.

list length	revert	revert1
50000	**39 s.**	**0 s.**

Operators

- writing everything as a term is not always comfortable
  - compare '='(X,'+'(2,3)) and X=2+3
- a more human readable form of terms would be appropriate
  - e.g. **infix notation of "standard" operations** (provided by Prolog)
- moreover, user may define **own operators** via

```
:- op(precedence, type, name).
```

- this is only a **"syntactic sugar"**

Arithmetic expressions

```
?-X=1+2.   -> X=1+2
?-3=1+2.   -> no
```

Number is a special type of atom.
It has a semantics (it is a number)!

- Term 1+2 is different from the term 3.
  - No semantics is associated with terms!
- We need a special procedure to evaluate the numerical expression: "is"

```
?-X is 1+2.
X=3
```

- **X is Expr** works as **arithmetic evaluator**:
  - evaluate Expr and compare (unify) the result with X
- Be careful: **"is" is not an assignment** command!

```
?-X is 1+2, X is 7.
```

Arithmetic comparison

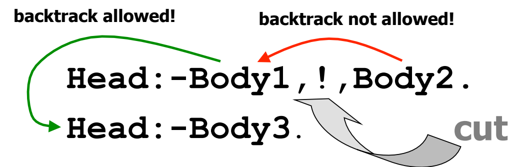
- If we have numbers, can we compare them?
- Prolog provides standard comparison of numbers:
  - **X < Y**
    - The numeric value of X is less than the numeric value of Y

```
?-1<2.     -> yes
?-1+1<3.   -> yes
?-3<1+2.   -> no
```

  - **X > Y, X =< Y, X >= Y**

Cut

- Prolog uses **depth-first search** to cover **non-determinism** of alternative rules.
  - use choice point when there is an alternative
- Can we **prune alternatives explicitly**?
  - **Cut** removes the choice point so no alternative rules will be tried.



Practicing cuts

```
test1(X,Y):-
  member(Y,[[1,2],[3,4]]),member(X,Y).
test1(0,[]).

test2(X,Y):-
  !,member(Y,[[1,2],[3,4]]),member(X,Y).
test2(0,[]).

test3(X,Y):-
  member(Y,[[1,2],[3,4]]),!,member(X,Y).
test3(0,[]).

test4(X,Y):-
  member(Y,[[1,2],[3,4]]),member(X,Y),!.
test4(0,[]).
```

Examples of red cuts

Their usage is discouraged because they change computation!

X	1	2	3	4	0
Y	[1,2]	[1,2]	[3,4]	[3,4]	[]

1
2
3
4

Cut for determinism

- Prune branches that will not be visited (green cut).

Example:

split the list into a list with elements smaller than X and a list with elements not smaller than X

```
split([],_,[],[]):-!.
split([H|T],X,[H|T1],T2):-
  H<X,!,
  split(T,X,T1,T2).
split([H|T],X,T1,[H|T2]):-
  split(T,X,T1,T2).
```

Cut for determinism

- Prune branches that will not be visited (green cut).

Example:

split the list into a list with elements smaller than X and a list with elements not smaller than X

```
split([],_,[],[]).
split([H|T],X,[H|T1],T2):-
  H<X,
  split(T,X,T1,T2).
split([H|T],X,T1,[H|T2]):-
  H>=X,
  split(T,X,T1,T2).
```

Cut for determinism

- Prune branches that will not be visited (**green** cut).

Example:

split the list into a list with elements smaller than X and a list with elements not smaller than X

Cut can be sometimes substituted by **if-then-else**

```
split([],_,[],[]).
split([H|T],X,L1,L2):-
  (H<X ->
     L1=[H|T1], L2=T2
  ;
     L1=T1, L2=[H|T2]
  ),
  split(T,X,T1,T2).
```

Negation

- How to prove non-existence of the solution?
- Useful for complex tests like non-member.

```
\+ :Goal
```

- no variable binding!
- **Inside negation:**

```
not(Query):-
  call(Query),!,fail.
not(_Query):-
  true.
```

META-PREDICATE
Prolog goal is a term so any term can be used as a query

If Query succeeds then fail (cut forbids using the alternative rule), otherwise succeed using the alternative rule.

Practicing negation

- Negation in Prolog is **negation-as-failure**
  - It is not a full logical negation!

```
p(a).
p(b).
q(a).
?- \+ (p(X),q(X)), X=b.     -> fail
?- X=b, \+ (p(X),q(X)).     -> X=b
```

- **Be especially careful when negation is applied to non-ground goal (contains variables)!**

All solutions

- How to find all answers to a Query?

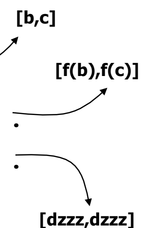
```
findall(?Template,:Query,?List)
```

Collects all answers to Query in the form of Template in a List.

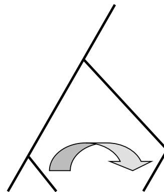
Example:

Find all neighboring nodes of "a".

```
?-findall(X,edge(a,X),Neigborhood).
?-findall(f(X),edge(a,X),Neigborhood).
?-findall(dzzz,edge(a,X),Neigborhood).
```



- How to pass information back when backtracking?
- How to pass information between search branches?
- We can use the Prolog database!
  - `assert` the information in one branch
  - access it in the other branch
- It is better to use **blackboard**!
  - clear and efficient



- Each information stored in the blackboard is identified by a unique atom called a **key** (an atom defined by the user).

- `bb_put(:Key, +Term)`
- `bb_get(:Key, ?Term)`
- `bb_delete(:Key, ?Term)`
- `bb_update(:Key, ?OldTerm, ?NewTerm)`

SICStus 4

- Test satisfiability of Query without binding variables.

```
sat(Query, _Answer):-
  bb_put(sat,no),
  once(Query), % finds one solution (if any)
  bb_put(sat,yes),
  fail.
sat(_Query,Answer):-
  bb_delete(sat,Answer).
```

Another solution using negation and if-then-else:

```
sat2(Query,Answer):-
  (\+ call(Query) -> Answer=no ; Answer=yes).
```


- Count the number of answers to Query
  `sat_num(:Query,-NumAnswers)`

```
sat_num(Query,_NumAnswers):-
  bb_put(counter,0),
  call(Query),
  bb_get(counter,N),
  N1 is N+1,
  bb_put(counter,N1),
  fail.
sat_num(_Query,NumAnswers):-
  bb_delete(counter,NumAnswers).
```

```
arc(a,b).
arc(a,c).
arc(a,d).

?-sat_num(arc(a,X),N).
N=3;
no
```

- Another solution using `findall`:

```
sat_num(Query,NumAnswers):-
  findall(x,Query,List),
  length(List,NumAnswers).
```

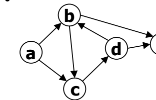
- Blackboard works as a global „variable“.
- **Be careful of nesting!**
  - If Query in the previous examples calls **`sat`** then the blackboard data are mishandled.
- Structure of the term is preserved but a connection to the „local“ variables is lost!!

```
?-A=term(X,f(X)), bb_put(test,A), X=a,
  bb_get(test,B).
A = term(a,f(a)),
B = term(_A,f(_A)),
X = a ? ;
no
```


Compute (one of) the shortest path between two nodes (avoid cycling).

- **Database (graph):**

```
arc(a,b).
arc(a,c).
arc(b,c).
arc(b,e).
arc(c,d).
arc(d,b).
arc(d,e).
```

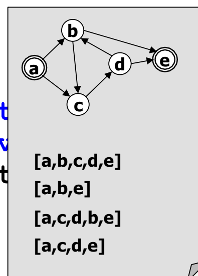


- **Expected answers:**

```
?-shortest_path(a,a,P).
  P = [a]

?- shortest_path(a,e,P).
  P = [a,b,e]

?- shortest_path(e,b,P).
  no
```


- Find **all paths in a DFS manner** and then select the shortest.

```
shortest_path(From,To, ShortestPath):-
  findall(Path,path(From,To,[],Path),AllPaths),
  shortest_list(AllPaths,ShortestPath).

path(From,From,Visited,Path):-!,
  revert([From|Visited],Path).
path(From,To,Visited,Path):-
  arc(From,Through),            % next
  \+ member(Through,Visited),   % prev
  path(Through,To,[From|Visited],Pat
```

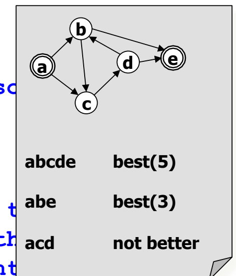


- **Branch&Bound** exploring all paths in a DFS manner

```
shortest_pathBB(From,To,_Path):-
  bb_put(best,no_path),
  spathBB(From,To,[],0).
shortest_pathBB(_From,_To,Path):-
  bb_get(best,path(_,Path)).
```

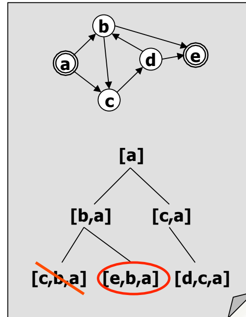
```
can_be_shorter(_):-
    bb_get(best,no_path).
can_be_shorter(Length):-
    bb_get(best,path(BestLength,_)),
    Length<BestLength.
```

```
spathBB(From,From,Visited,Length):-!,
  revert([From|Visited],Path),
  bb_put(best,path(Length,Path)),% save so
  fail.
spathBB(From,To,Visited,OldLength):-
  NewLength is OldLength+1,
  can_be_shorter(NewLength),      % check t
  arc(From,Through),              % find th
  \+ member(Through,Visited),     % prevent
  spathBB(Through,To,[From|Visited],NewLength).
```



- **Breadth-first search** with concatenation

```
shortest_pathBFS(From,To,Path):-
  spathBFS([[From]],To,Path).

spathBFS([Visited|Rest],To,Path):-
  Visited = [N|_],
  (N=To ->        % we found the path
      revert(Visited,Path)
  ;               % expand the node N
      findall([N1|Visited],
       (arc(N,N1),
      \+ member(N1,Visited),
      \+ member([N1|_],Rest)),
       NewNodes),
      concat(Rest,NewNodes, Nodes),
      spathBFS(Nodes,To,Path)
  ).
```



© 2013 Roman Barták
Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz

- Write procedures (rules) defining:
  - `length(List,Length)`
  - `shortest_list(ListOfLists,ShortestList)`
- Write a Prolog program solving the **water pouring problem**.
  - We have three (N) **cups**, each with a given **capacity** and a given **level** of water. It is possible to pour completely a cup into another cup (if capacity is not exceeded) or pour part of a cup to fill another cup. Find a **shortest plan** for reaching a given level of water in each cup.
  - Tip: use the shortest path algorithms!