4

Programming with Logic and Constraints

Roman Barták
Charles University, Prague (CZ)

roman.bartak@mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

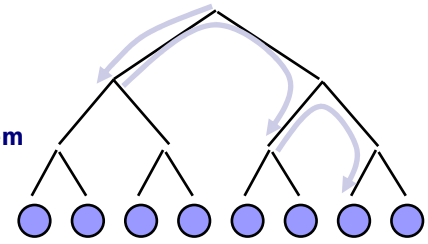
Consistency and Search

Consistency techniques are (usually) incomplete.

↯ **We need a search algorithm to resolve the rest!**

Labeling

- depth-first search
  - assign a value to the variable
  - propagate = make the problem locally consistent
  - backtrack upon failure



- X in 1..3 ≈ X=1 ∨ X=2 ∨ X=3 (enumeration)

In general, search algorithm resolves remaining disjunctions!

- X=1 ∨ X≠1 (step labeling)
- X<3 ∨ X≥3 (bisection)
- X<Y ∨ X≥Y (variable ordering)

Enumeration

- **Domain enumeration**
  - try to assign a (minimal) value to the variable
  - in case of failure try the next value

```
enum([]).
enum([H|T]):-
  indomain(H), % enumerate domain
  enum(T).
```

Choice point
`indomain` works like **`member`**
assign first value from the domain and upon backtracking try another value

Step labeling

- When a solution for a selected value is not found, the value is removed from the domain.

```
step([]).
step([H|Rest]):-
  fd_min(H,Value),
  (H#=Value ; H#\=Value),
  (var(H) ->
      step([H|Rest])
  ;   step(Rest)).
```

disjunction
can be seen as a shortcut for rules

```
try(H,Value):-H#=Value.
try(H,Value):-H#\=Value.
```

Bisection

- Split the domain into two parts until the domain becomes singleton.

```
bisection([]).
bisection([H|Rest]):-
  fd_min(H,Min), fd_max(H,Max),
  Middle is integer((Min+Max)/2),
  (H#=<Middle ; H#>Middle),
  (var(H) ->
     bisection([H|Rest])
  ;  bisection(Rest)).
```


Branching schemes

- **Which variable should be assigned first?**
  - **fail-first principle**
    - prefer the variable whose instantiation will lead to a failure with the highest probability
    - variables with the smallest domain first
    - most constrained variables first
  - defines the **shape of the search tree**
- **Which value should be tried first?**
  - **succeed-first principle**
    - prefer the values that might belong to the solution with the highest probability
    - values with more supporters in other variables
    - usually problem dependent
  - defines the **order of branches** to be explored

Generic search

```
label([]).
label(Variables):-
  select_variable(Variables,V,Rest),!,
  choice_point(V),
  (var(V) ->
     label([V|Rest])
  ;  label(Rest)).
```

Simple enumeration:

```
  select_variable([H|T],H,T).
  choice_point(V):-indomain(V).
```


Constraint optimization

- So far we have looked for feasible assignments only.
- In many cases the users require optimal assignments where optimality is defined by an objective function.

Definition:

- **Constraint Optimization Problem** (COP) consists of the standard CSP P and an objective function *f* mapping feasible solutions of P to numbers.
- **Solution to COP** is a solution of P minimizing / maximizing the value of the objective function *f*.

- To find a solution of CSOP we need in general to explore all the feasible valuations. Thus, the techniques capable to provide all the solutions of CSP are used.

Branch and bound

- **Branch and bound** is perhaps the most widely used optimisation technique based on cutting sub-trees where there is no optimal (better) solution.
- It is based on the **heuristic function** *h* that **approximates the objective function**.
  - a sound heuristic for minimisation satisfies $h(x) \leq f(x)$
  - [in case of maximisation $f(x) \leq h(x)$]
  - a function closer to the objective function is better
- During search, the **sub-tree is cut** if
  - there is no feasible solution in the sub-tree
  - there is no optimal solution in the sub-tree
    - **bound** $\leq$ h(x), where **bound** is max. value of feasible solution
- **How to get the bound?**
  - It could be an objective value of the best solution so far.

Simple optimization

- Minimize/maximize a value of a selected variable
  - typically, there is a constraint X#=ObjectiveFunction
  - propagation from ObjectiveFunction to X corresponds to the heuristics function h
- **Straightforward method** for minimization of X:
  - try to find a solution with a minimal value of X
  - in case of failure increase the minimal value of X by one

```
minimizeSimple(Vars,X):-
  fd_min(X,X),
  label(Vars),!.
minimizeSimple(Vars,X):-
  fd_min(X,Min),
  X#>Min,
  minimizeSimple(Vars,X).
```

note
finds single solution such that the value of the objective function is minimal

B&B in Prolog

Enumeration can be modified into **branch and bound.**
The bound is stored on blackboard and checked after every assignment.

```
minimizeBB(Vars,X,InitialBound):-
  bb_put(bound,InitialBound),     % save upper bound
  minBB(Vars,Vars,X).
minimizeBB(Vars,_,_):-
  bb_get(best,Vars).              % restore best solution

minBB([],AllVars,X):-             % all variables known
  bb_put(bound,X),                % save new upper bound
  bb_put(best,AllVars),           % save best solution
  fail.                           % explore alternatives
minBB([H|Rest],AllVars,X):-
  indomain(H),                    % assign a value
  bb_get(bound,Bound),
  fd_min(X,MinX),
  MinX<Bound,                     % check bound
  minBB(Rest,AllVars,X).
```


B&B with splitting

- If the domain of the minimized variable is bounded then we can use domain splitting over this variable.

```
minimizeBBsplit(Vars,X):-
  (var(X) ->
    fd_min(X,MinX),fd_max(X,MaxX),
    Middle is integer((MinX+MaxX)/2),
    ((X#=<Middle, \+ \+ label(Vars)) ->
       true
    ;  X#>Middle
    ),!,
    minimizeBBsplit(Vars,X)
  ;
    label(Vars)
  ).
```

double negation
find out whether it is possible to label variables but do not bound them

note
finds all optimal solutions, i.e., solutions with the same value of objective function

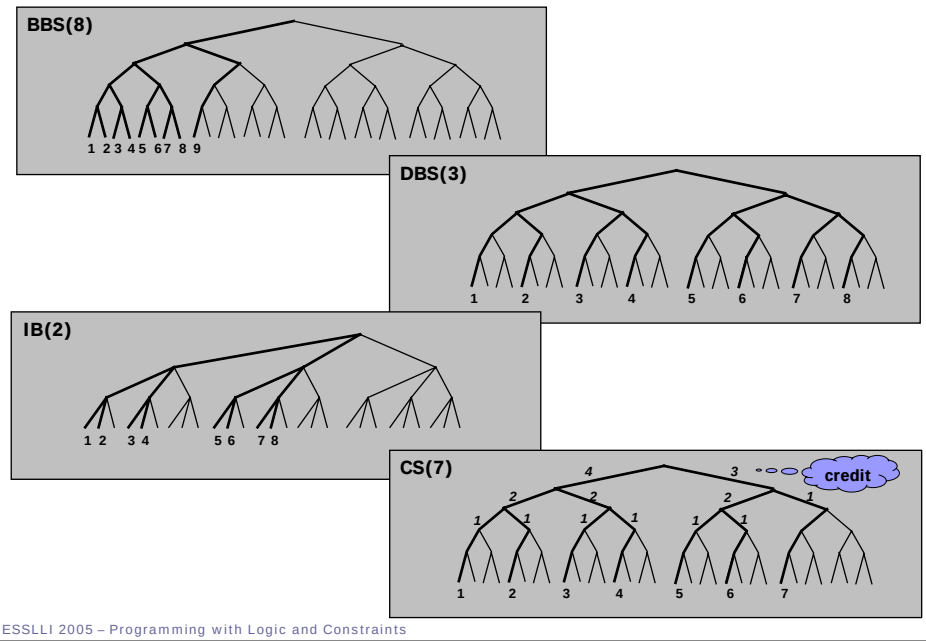
Incomplete search

A **cutoff limit** to stop exploring a (sub-)tree
- some branches are skipped → incomplete search

When no solution found, **restart** with enlarged cutoff limit.

- **Bounded Backtrack Search (Harvey, 1995)**
  - restricted number of backtracks
- **Depth-bounded Backtrack Search (Cheadle et al., 2003)**
  - restricted depth where alternatives are explored
- **Iterative Broadening (Ginsberg and Harvey, 1990)**
  - restricted breadth in each node
  - still exponential!
- **Credit Search (Beldiceanu et al., 1997)**
  - limited credit for exploring alternatives
  - credit is split among the alternatives

Incomplete search



BBS

Bounded Backtrack Search

- restricted number of backtracks

```
bbs_search(Variables,Limit):-
  bb_put(limit,Limit),
  bb_put(stage,fw),
  bbs(Variables).

bbs([]).
bbs([X|RestVariables]):-
  (bbs_assign_value(X) ; bb_put(stage,bw),fail),
  bbs(RestVariables).

bbs_assign_value(X):-
  assign_value(X),
  bb_update(stage,Stage,fw),
  (Stage=fw -> true
  ;   bb_get(limit,L), NL is L-1, bb_put(limit,NL),
      (NL>0 -> true ; !,fail)
  ).
```

```
assign_value(X):-
  indomain(X).
```

trick
indicate on the blackboard that we started to backtrack

DBS

Depth-bounded Backtrack Search

- restricted depth where alternatives are explored

```
dbs_search([],_).
dbs_search([X|RestVariables],Depth):-
  (Depth>0 ->
    NewDepth is Depth-1,
    assign_value(X)
  ;
    NewDepth = 0,
    once(assign_value(X))
  ),
  dbs_search(RestVariables,NewDepth).
```

forbidden alternatives
only the first solution is returned (if any), no alternatives are allowed

IB

Iterative Broadening

- restricted breadth in each node

```
ib_search(Variables,Width):-
  bb_put(width,Width),
  ib(Variables,Width).

ib([],_).
ib([X|RestVariables],Width):-
  bb_update(width,TW,Width),
  (ib_assign_value(X) ; bb_put(width,TW),!,fail),
  ib(RestVariables,Width).

ib_assign_value(X):-
  assign_value(X),
  bb_get(width,RestWidth),
  (RestWidth=0 -> !,fail
  ;  NewW is RestWidth-1, bb_put(width,NewW)
  ).
```

trick
the number of remaining values for the previous variable is kept by the next variable

Heuristics in search

- **Observation 1:**
  The **search space** for real-life problems is so **huge** that it cannot be fully explored.
- **Heuristics** - a guide of search
  - they recommend a value for assignment
  - quite often lead to a solution
- What to do upon a **failure of the heuristic**?
  - BT cares about the end of search (a bottom part of the search tree) so it rather repairs later assignments than the earliest ones thus BT assumes that the heuristic guides it well in the top part
- **Observation 2:**
  The **heuristics** are **less reliable in the earlier parts** of the search tree (as search proceeds, more information is available).
- **Observation 3:**
  The number of **heuristic violations** is usually **small**.

Discrepancies

Discrepancy
= the heuristic is not followed

Basic principles of discrepancy search:
change the order of branches to be explored
- prefer branches with **less discrepancies**

- prefer branches with **earlier discrepancies**

Discrepancy search

- **Limited Discrepancy Search (Harvey & Ginsberg, 1995)**
  - restricts a maximal number of discrepancies in the iteration

- **Improved LDS (Korf, 1996)**
  - restricts a given number of discrepancies in the iteration

- **Depth-bounded Discrepancy Search (Walsh, 1997)**
  - restricts discrepancies till a given depth in the iteration

- **…**

*** heuristic = go left**

Homework

- Try different search strategies (enum, split, bisect) for N queens problem. Which one is the best?
- Write a procedure **`select_variable`** for generic search modeling fail-first strategy (a variable with the smallest domain is selected first).
  - **`fd_size(Variable,DomainSize)`** gives actual size of the variable domain
- Write procedures **`choice_point`** for generic search in such a way that split and bisection strategies are obtained.