Topic: Intelligent Backtracking Algorithms

Context: Finite domain, binary constraints

Material: Hybrid algorithms for the constraint satisfaction problem [Prosser, IC93]

Foundations of Constraint Satisfaction

CSCE421/821, Fall 2001

`www.cse.unl.edu/~choueiry/F01-421-821/`

Berthe Y. Choueiry (Shu-we-ri)

Ferguson Hall, Room 104

`choueiry@cse.unl.edu`, Tel: (402)472-5444

Required reading:

Hybrid algorithms for the constraint satisfaction problem, Prosser IC'93

Recommended reading:

Tsang, Chapter 5

Upon demand:

Notes of Fahiem Bacchus, Chapter 2

Search: two main mechanisms

1. Backtracking: when encountering dead-ends
2. Checking:
   - Back-checking: against past variables
   - Look-ahead: against future variables

$$\left\{\begin{array}{l} \text{Partial: } \left\{\begin{array}{l}\text{DAC}\\ \text{Forward checking}\end{array}\right. \\ \text{Full: AC-3 on all future variables (MAC)}\end{array}\right.$$

Outline

- Hybrid backtracking algorithms
- CSP parameters
- Parameters for comparing systematic search algorithms

Improve performance of search

1. Backtracking: Naive/Chronological backtracking (BT), backjumping (BJ), conflict-directed backtracking (CBJ)

2. Checking: Backmarking (BM), forward checking (FC)

3. Hybrids: BMJ, BM-CBJ, FC-BJ, BM-CBJ

BT	BJ	CBJ
BM	BMJ	BM-CBJ
FC	FC-BJ	**<u>FC-CBJ</u>**

Evaluation:
- Empirical, Prosser, 450 instances of Zebra
- Theoretical, Kondrak & Van Beek 1995

BT → BJ → CBJ: more informed backtracking

BT → BM → FC: tree expansion, forward move

	Go back		
Go forward	BT	BJ	CBJ
	BM	BMJ	BM-CBJ
	FC	FC-BJ	**FC-CBJ**

Notation: in Prosser's paper

- Variables: V_i, $i \in [1, n]$
- Domain: $D_i = \{v_{i1}, v_{i2}, \ldots, v_{M_i}\}$
- Constraint between V_i and V_j: $C_{i,j}$
- Constraint graph: G
- Arcs of G: $\text{Arc}(G)$
- Instantiation order (static or dynamic)
- Shallow: nodes closer root in search tree
- Deep: nodes closer to leaves/fringe in search tree
- Language primitives: list, push, pushnew, remove, set-difference, union, max-list

Data structures: in Prosser's paper

- Vector of values v: $v[i]$ is value assigned to i^{th} variable, V_i
- $v[i]$ also indicates variable V_i
- Pseudo-variable $v[0]$ (root of tree): backtracking to $v[0]$ indicates insolvability
- domain[i]: array of the domain of variable V_i, domain[0] $=$ nil
- current-domain[i], domain of V_i <u>during</u> search
  when backtracking occurs, current-domain[i] of future variables needs to be reinstantiated.
- Function check(i,j): is true if the values assigned to variable V_i and V_j are consistent

Generic search: bcssp

```
PROCEDURE bcssp (n, status):
BEGIN
consistent ← true;
status ← unknown;
i ← 1;
WHILE status = unknown
      DO BEGIN
      IF consistent
          THEN i ← label(i, consistent)
          ELSE i ← unlabel(i, consistent)
      IF i> n
      THEN status ← “solution”
      ELSE IF i = 0 THEN status ← “impossible”
      END
END
```

Prosser's notation for algorithm X:

- forward move: X-label
- backward move: X-unlabel
- Parameter of (un)label: current variable, Boolean
- Return of (un)label: new current variable

Label terminates with:

- Consistent = true and returns $i + 1$
- Consistent = false, current-domain[i] = nil, and returns i

Unlabel:

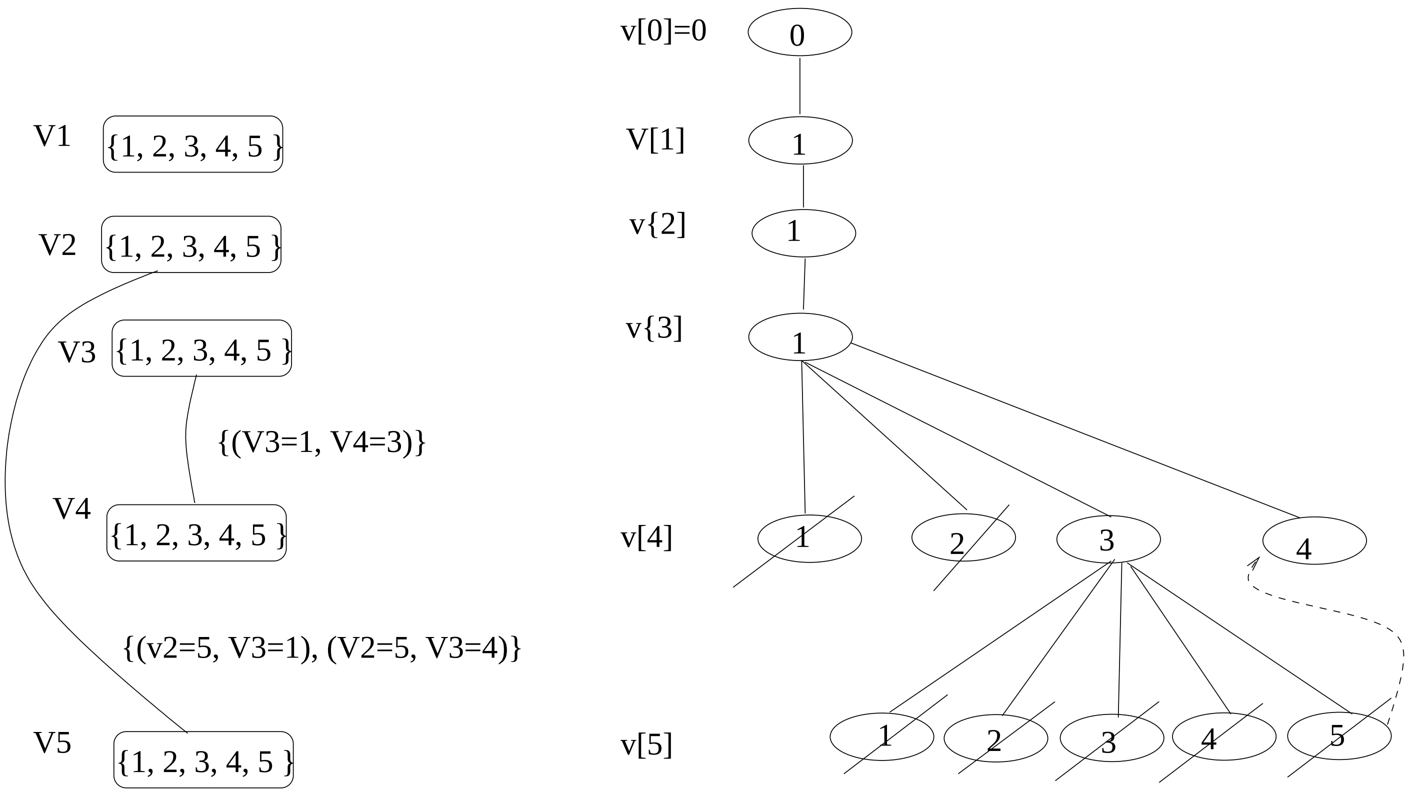
- is called when consistent = false and current-domain[i] = nil
- selects V_h to backtrack to
- Un-instantiates all variables $V_h < V_k \leq V_i$
- Removes $v[h]$ from current-domain[h]
- Sets consistent to true if current-domain[h] $\neq \emptyset$
- returns h

Chronological backtracking (BT): bt-label, bt-unlable

- When $v[i]$ is assigned a value x from current-domain[i] we perform back-checking of x against past variables (check(i,k))
- If back-checking succeeds, bt-label returns $i + 1$
- If back-checking fails, we remove x from current-domain[i], choose another value from current-domain[i], a repeat check
- If no other value exits, then current-domain[i] is refreshed (to domain[i]), $V_{h=i-1}$ is uninstantiated (assigned value is removed from current-domain[h]), and we seek an new values for V_h

All future variables j: current-domain[j] = domain[j]
All past variables g: current-domain[g] $\subseteq$ domain[g]

Example: BT


V1 {1, 2, 3, 4, 5 }
V2 {1, 2, 3, 4, 5 }
V3 {1, 2, 3, 4, 5 }
{(V3=1, V4=3)}
V4 {1, 2, 3, 4, 5 }
{(v2=5, V3=1), (V2=5, V3=4)}
V5 {1, 2, 3, 4, 5 }
v[0]=0
0
V[1]
1
v{2]
1
v{3]
1
v[4]
1
2
3
4
v[5]
1
2
3
4
5

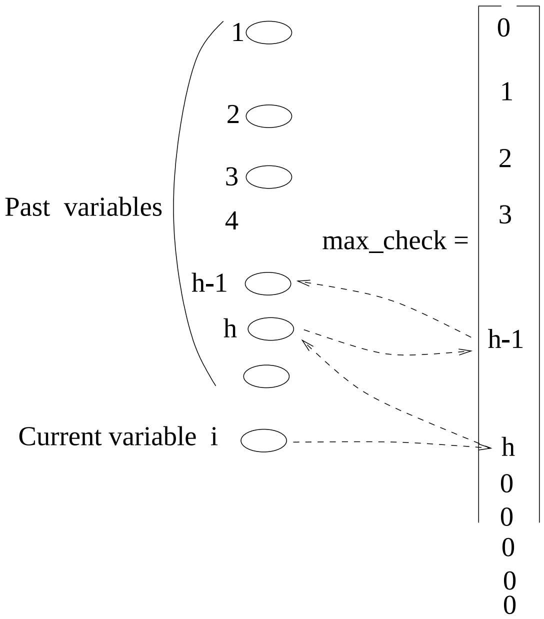
Note:
BT assumes that the instantiation of V_i was prevented by a bad choice for $h = (i-1)$. Thus it tries to change the assignment of V_h.

When this assumption proves wrong $\longrightarrow$ Thrashing

Backjumping (BJ) tries to avoid that:
$\longrightarrow$ jumps to the reason for failure then proceeds as BT

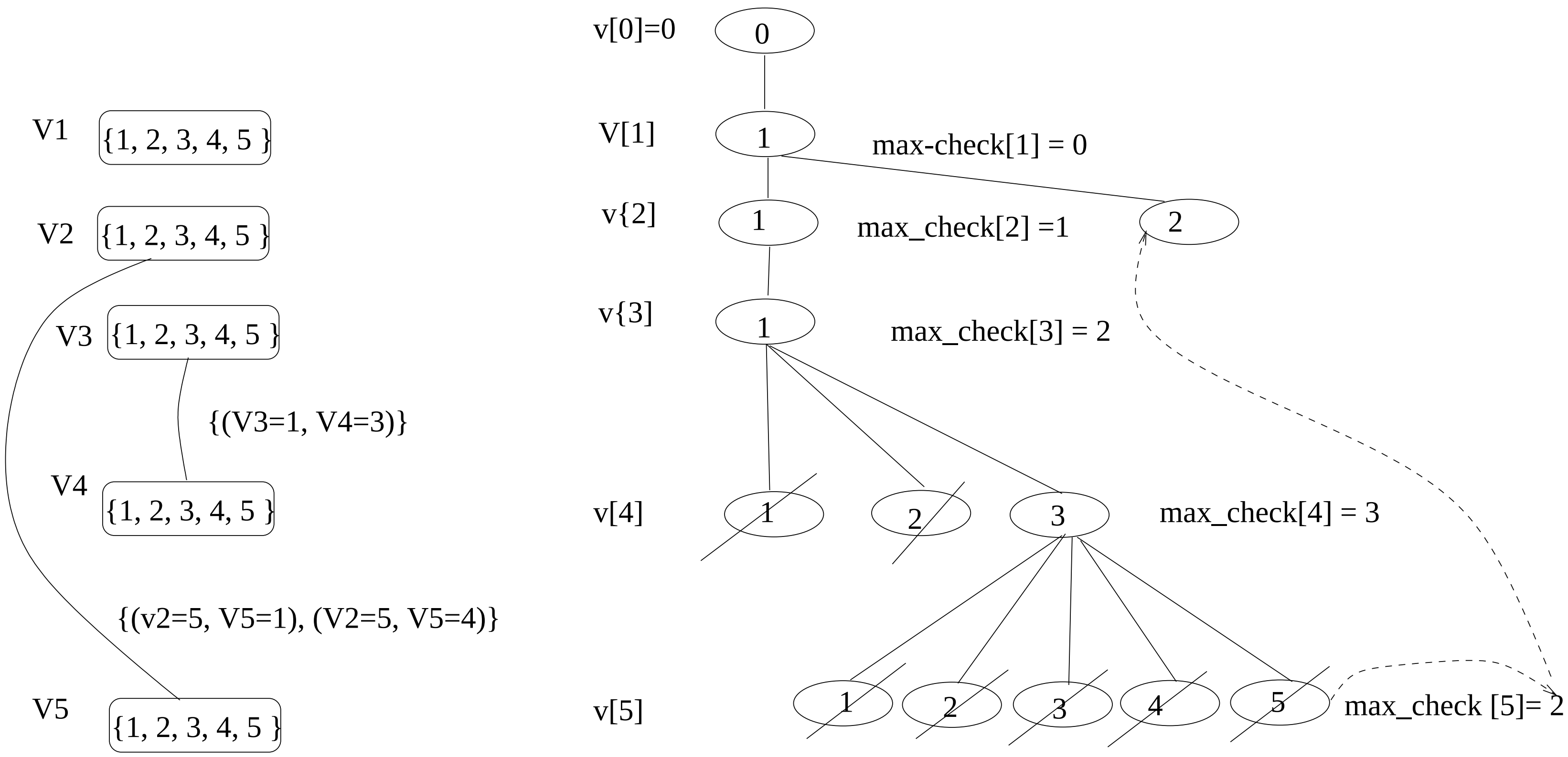
Backjumping (BJ) tries to reduce thrashing:

- When $v[i]$ is instantiated, BJ remembers V_h deepest node among past variables that $v[i]$ checked against.
- If current-domain[h] is getting empty, simple chronological backtracking from h is performed
- max-check[i]: global, initialized to 0
- at level i when check(i,h), max-check[i] $\leftarrow$ max(max-check[i], h)



$\longrightarrow$ **BJ jumps then steps**

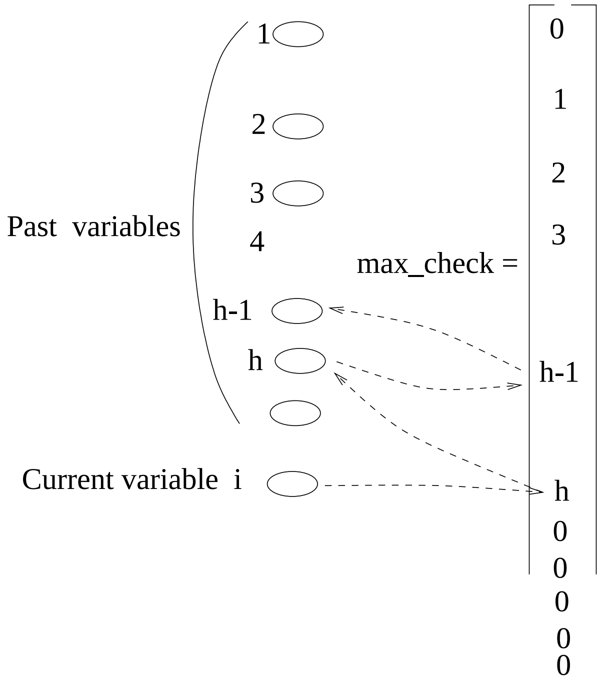
Example: BJ


V1 {1, 2, 3, 4, 5 }
V2 {1, 2, 3, 4, 5 }
V3 {1, 2, 3, 4, 5 }
{(V3=1, V4=3)}
V4 {1, 2, 3, 4, 5 }
{(v2=5, V5=1), (V2=5, V5=4)}
V5 {1, 2, 3, 4, 5 }
v[0]=0
0
V[1]
1
max-check[1] = 0
v{2]
1
max_check[2] =1
2
v{3]
1
max_check[3] = 2
v[4]
1
2
3
max_check[4] = 3
v[5]
1
2
3
4
5
max_check [5]= 2

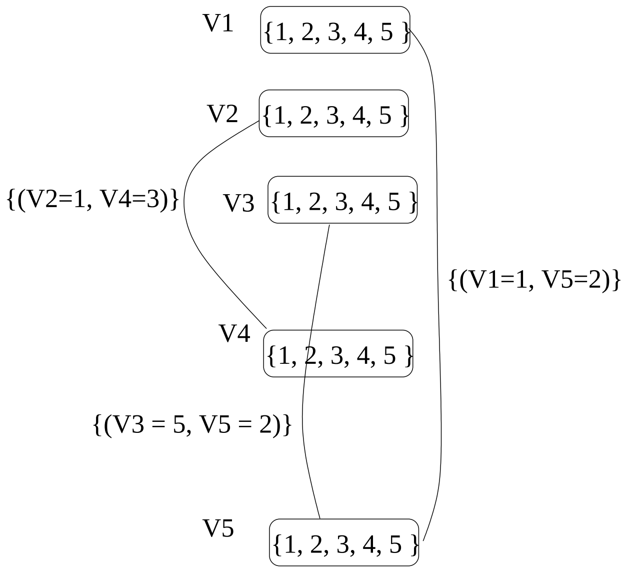
Backjumping (BJ)

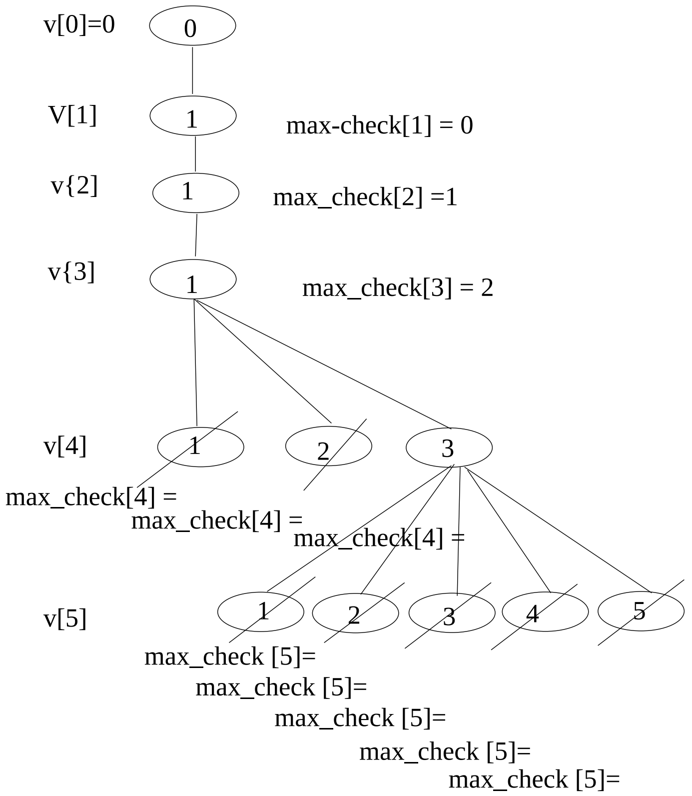
- bj-label, same as bt-label, but updates max-check[i]

- bj-unlabel, same as bt-unlabel, but
  — backtracks to h = max-check[i]
  — resets max-check[j] $\leftarrow 0$ for $j \in [h+1, i]$



Example: BJ (II) *D. Keshwani, 1999*





Note: max-check[i] is:

– the deepest value i has **checked** against (max-check)

– not the deepest value i is in **conflict** with (max-fail)

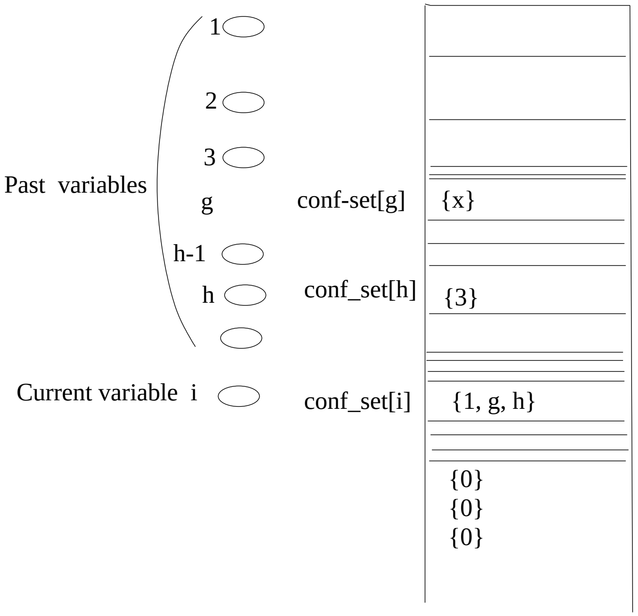
Conflict-directed backjumping (CBJ)

- BJ jumps from $v[i]$ to $v[h]$, then Steps from $v[h]$ to $v[h-1]$
- CBJ jumps from $v[i]$ to $v[h]$, and jumps again across **conflicts** involving <u>**both**</u> $v[i]$ and $v[h]$

CBJ

CBJ maintains conf-set[i] for each variable:

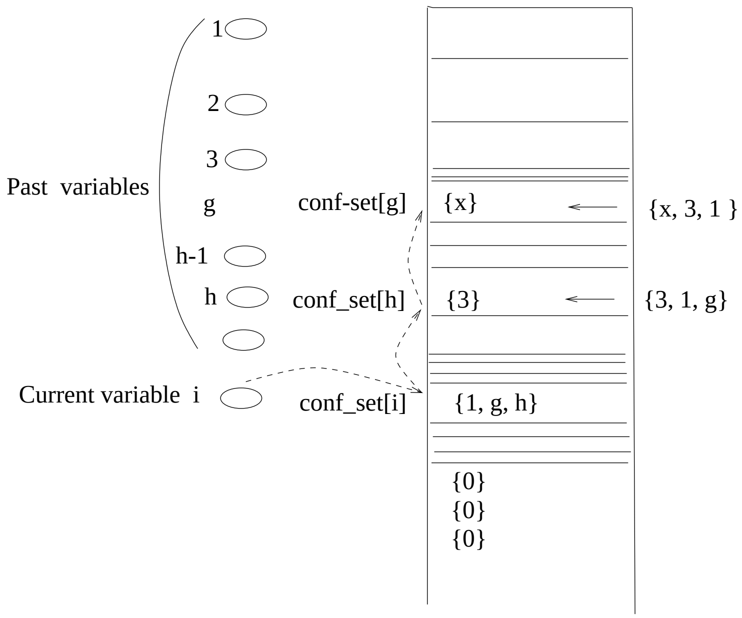
- conf-set first initialized to {0}
- conf-set[i] is subset of past variables in conflict with i



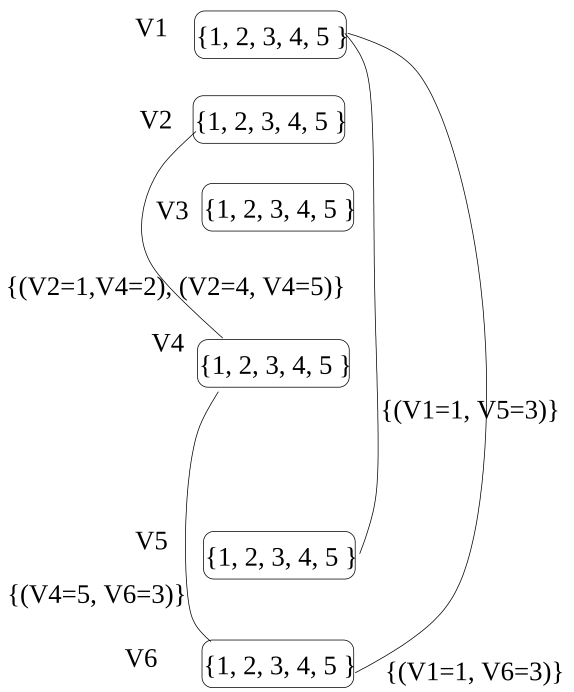
CBJ (cont.)

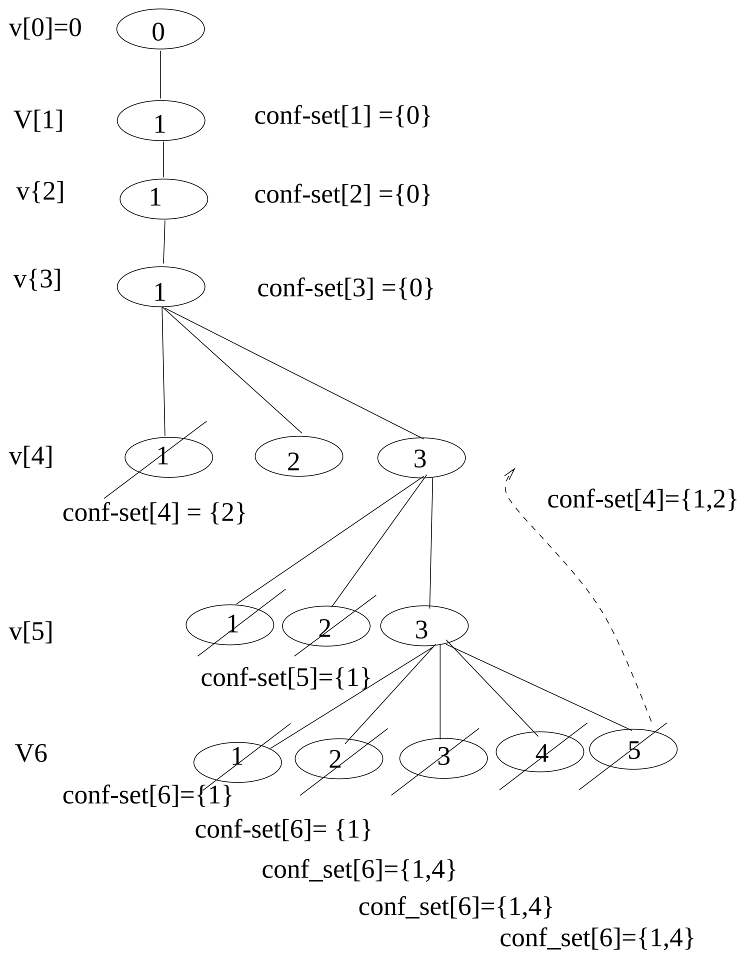
- when a check(i,h) fails, conf-set[i] $\leftarrow$ conf-set[i] $\cup\{h\}$
- when current-domain[i] $= \emptyset$, CBJ
  1. jumps to deepest past variable h in conf-set[i]

     $h \leftarrow$ max-list(conf-set[i])
  2. updates conf-set[h] $\leftarrow$ conf-set[h] $\cup$ (conf-set[i] -h)
- as conf-set updated, primitive learning (while searching)



Example CBJ *D. Keshwani, 1999*





Backtracking: summary

- Chronological backtracking:
  - steps to previous level
  - no extra data structures
- Backjumping:
  - jumps to deepest checked against variable, then steps back
  - uses array of integers: max-check[i]
- Conflict-directed backjumping:
  - jumps across deepest conflicting variables
  - uses array of sets: conf-set[i]

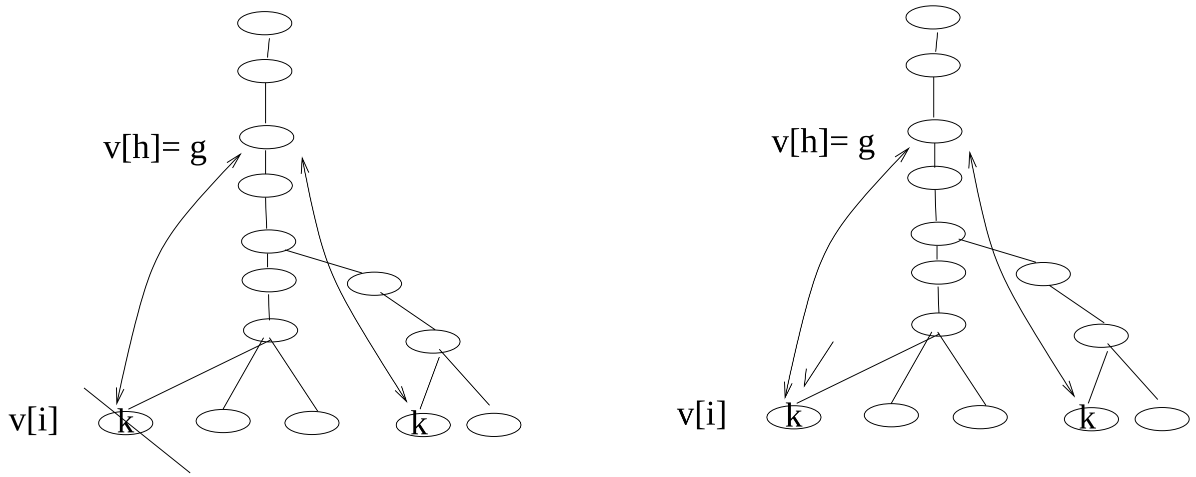
Backmarking tries to reduce amount of consistency checking

Consider:

Current variable V_i is about to be reinstantiated value k.

$V_i \leftarrow k$ was checked against $V_h \leftarrow g$

V_h has not been modified



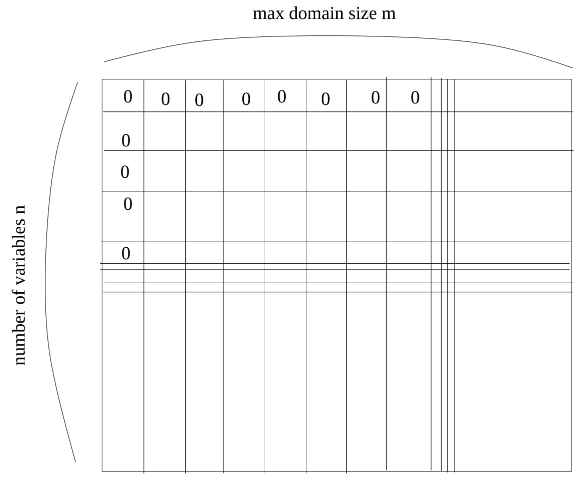
Two situations:

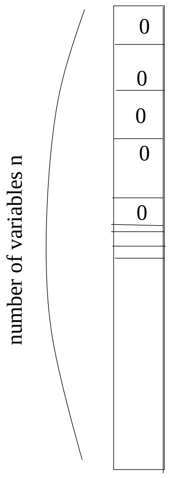
1. Either, $(V_i = k, V_h = g)$ has failed $\Rightarrow$ it will fail again, or
2. $(V_i = k, V_h = g)$ was consistent $\Rightarrow$ it will remain consistent

Backmarking: data structures, two arrays:

maximum checking level: *mcl* (dim. nxm)

minimum backup level: *mbl* (dim. n)

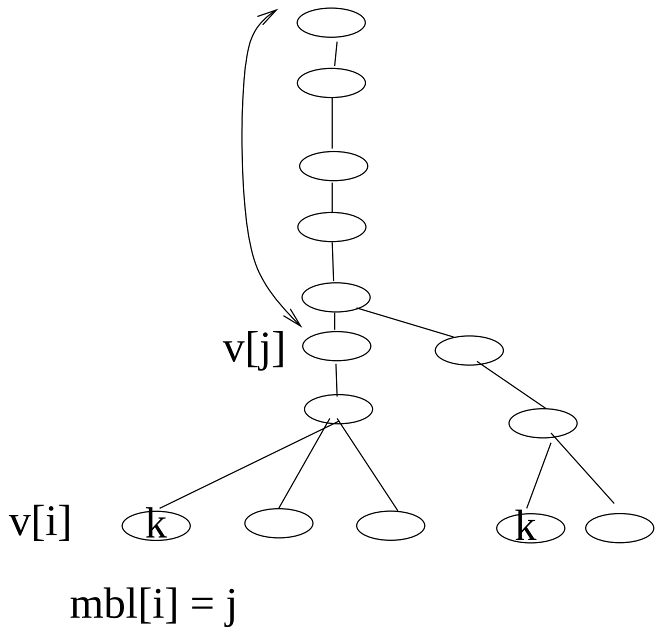




- mcl[i,k]: deepest variable v[i] ← k checked against
  → mcl[i,k] is a finer version of max-check[i]
- mbl[i]: shallowest past variable whose value has changed since v[i] was the current variable
- BM (and all its hybrids) do not allow dynamic variable ordering

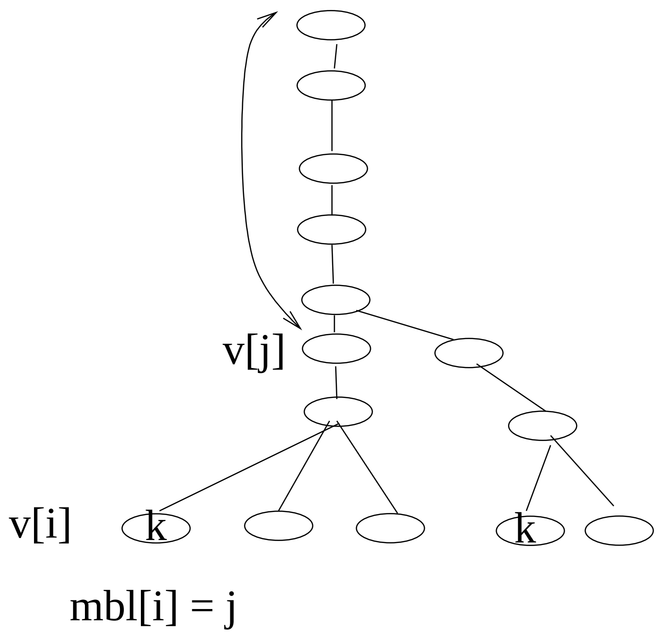
Maximum checking level: mcl[i,k] = j

BM is aware the deepest variable (v[i] $\leftarrow$ k) checked against is v[j]



Minimum backup level: mbl[i] = j

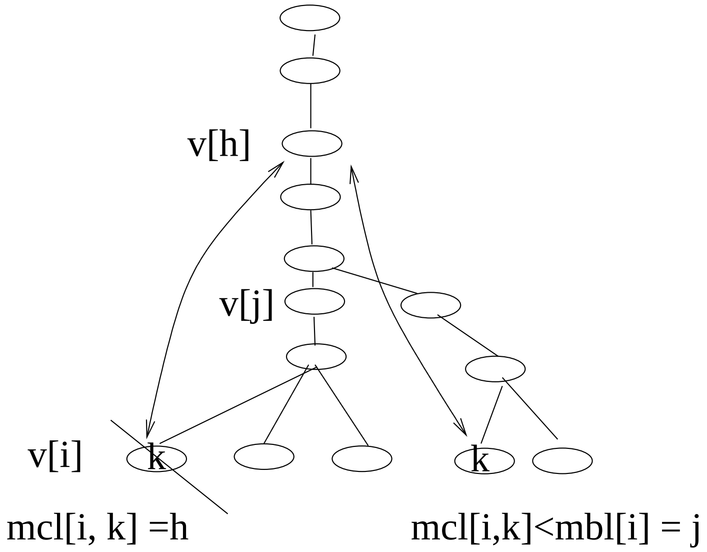
BM is aware that all variables in the past of v[j] have not changed values



Type a savings:

When mcl[i,k]$<$ mbl[i],

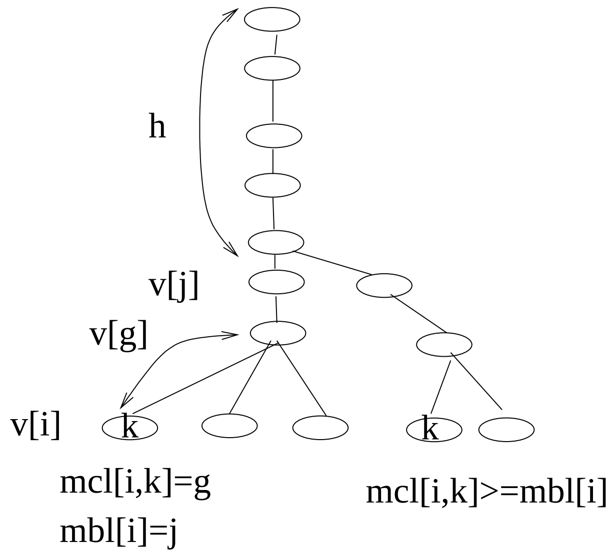
do not check v[i]$\leftarrow$ k, because it will fail



Type b savings:

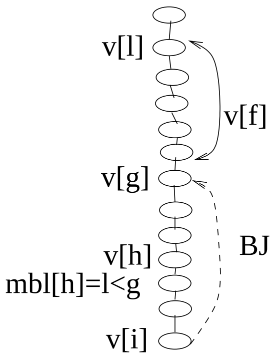
When mcl[i,k]$\geq$ mbl[i],

do not ckeck(i, h$<$j) because they'll succeed



Hybrids of BM

- mcl can be used to allow BJ within BM
- BJ and BM can be mixed, yielding BMJ: BMJ avoids redundant consistency checking (Type a + b savings) and reduces number of nodes visited during search (by jumping)
- Caveat: BMJ can perform worst than BM
  v[i] BJs to v[g], when v[h] is reconsidered, it will be checked against all f, v[l] $\leq$ f <g :–(



  BMJ enjoys <u>some</u> of the advantages of BM
- Phenomenon will worsen with CBJ

Forward checking (FC)

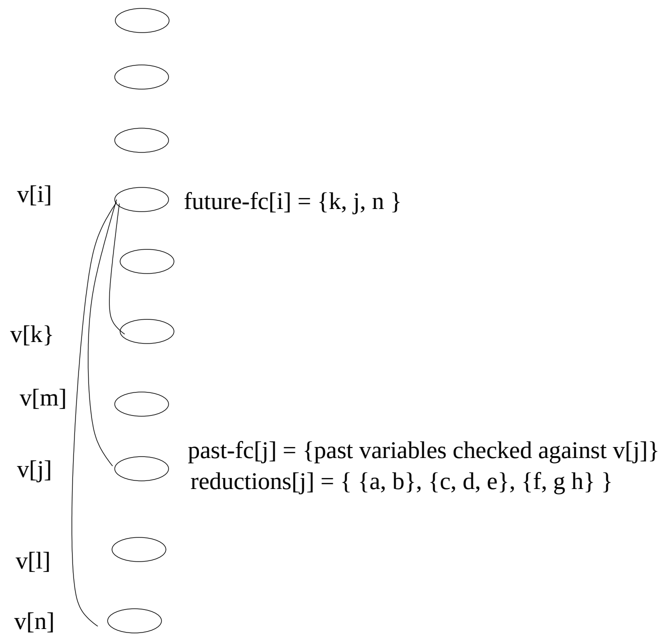
- Looking ahead: from current variable, consider all future variables and clear from their domains the values that are not consistent with current partial solution.
- FC makes more work at every instantiation, but will expand less nodes
- When FC moves forward, the values in current-domains of future variables are all compatible with past instantiations, thus saving back-checking
- FC may "wipe out" the domain of a future variable (a.k.a. domain annihilation) and thus discover conflicts early on. FC then backtracks chronologically
- Goal of FC is to fail early (avoid expanding fruitless subtrees)

Forward checking data-structures

When v[i] $\leftarrow$ k,

current-domain[j] is filtered ($i < j \leq n$)

- reduction $\in$ reductions[j] is a set of values removed from current-domain[j] by some variable before v[j]
- future-fc[i]: subset of future variables v[i] checks against
- past-fc[j]: past variables that checked against v[j]
- all these sets are treated as stacks



Forward checking: functions

- check-forward
- undo-reductions
- update-current-domain
- fc-label
- fc-unlable

Forward checking

- Function: check-forward(i,j) is called when v[i] is instantiated, it performs REVISE (j, i) (ref. AC) returns false if current-domain[j] is empty, true otherwise
- values removed from current-domain[j] are pushed into reductions[j]
- these values will be popped if we have to backtrack over v[i] (undo-reductions)
- Function update-current-domain: current-domain[i] $\leftarrow$ domain[i] - reductions[i] (actually, we have to iterate over reductions=set of sets)
- fc-label: attempts to instantiate current-variable, then filters domains of all future variables (push into reductions) whenever current-domain of a future variable wiped out, v[i] uninstantiated, and domain filtering undone (pop reductions)

Hybrids for FC

FC suffers from thrashing, since it is based on BT

- FC-BJ: max-check is integrated in fc-bj-label, fc-bj-unlabel
- FC-BJ: enjoys advantages of FC and BJ.. but suffers malady of BJ (jump then step)
- FC-CBJ: (best algorithm so far)
  fc-cbj-label and fc-cbj-unlable

Consistency checking: summary

- Chronological backtracking:
  - steps to previous level
  - no extra data structures
  - simple back-checking
- Backmarking:
  - uses mcl, and mbl
  - two types of consistency-checking savings
- Forward checking:
  - works more at every instantiations, but expands fewer subtrees
  - reductions[i], future-fc[i], past-fc[i], current-domain[i]

Experiments: Empirical evaluation on Zebra problem

- representative of design/scheduling problems
- 25 variables, 122 constraints
- binary constraints
- permutation of variable ordering yields new search spaces
- variable ordering: different (band/induced) 'width' of graph

450 problems instances were generated

Each algorithm was applied to each instance

Analysis of experiments

Algorithms compared with respect to:

1. Number of consistency checks performed: (on average)
   FC-CBJ < FC-BJ < BM-CBJ < FC < CBJ < BMJ < BM < BJ < BT

2. Number of nodes visited: (on average)
   FC-CBJ < FC-BJ < FC< BM-CBJ < BMJ = BJ < BM = BT

3. CPU time: (on average)
   FC-CBJ< FC-BJ < FC< BM-CBJ < CBJ<BMJ< BJ< BT< BM

$\longrightarrow$ FC-CBJ is (apparently) the champion

Additional developments (I)

- Other backtracking algorithms
  (e.g., Graph-based backjumping, GBJ)
- Theoretical evaluations: Kondrak and Van Beek IJCAI'95
- Other look-ahead techniques: MAC, etc.
- Empirical evaluations

CSP parameters $\langle n, a, t, d \rangle$

- n is number of variables
- a is maximum domain size
- t is constraint tightness

$$t = \frac{|forbidden\ tuples|}{|all\ tuples|}$$

- d is constraint density

$$d = \frac{e - e_{min}}{e_{max} - e_{min}},$$

where e is the number of constraints,
$e_{min} = (n-1)$, and
$e_{max} = \frac{n(n-1)}{2}$

$\longrightarrow$ Constraints in random problem often generated uniform

Alternatively, $\langle n, m, p_1, p_2 \rangle$

- n is number of variables
- m is the number of values per variable (assuming all variables have the same domain size)
- p_1 is the probability of existence of a constraint between two variables
- p_2 is the probability that two values for two variables are in conflict (assuming that a constraint exists between the two variables).

We show that $p_1 \geq d$ for all graphs and
$p_1 \approx d$ for graphs of sizable n.

Comparision criteria

1. Number of consistency checks (CC)
2. Number of nodes in search tree (NV)
3. CPU time
4. (Some specific parameter to assess the quality of the improvment proposed)

How?

- Theoretical evaluations: Values of above criteria for finding all solutions and under static variable ordering
- Empirical evaluations: Average, median of above criteria

Empirical comparisions of MAC vs. FC

Pointers: [Sabin & Freuder, ECAI'94], [Bessière and Régin, CP'97], [Sabin & Freuder, CP'97], [Gent & Prosser, APES-20-2000], [Experiments by Lin Xu, 2001]

Results: (very sketchy)

Constraint		tightness	
		Low	High
density	Low	FC	MAC
	High	FC	FC

Note: Results depend on variable ordering (static vs. dynamic) and difficulty of problems (position relative to crossover point)

Performance comparison *Courtesy of Lin XU*