cbj

What’s a csp?

<V,D,C>

- a set of variables
- each with a domain of values
- a collection of constraints (I’m going to assume binary for the present)
- assign each variable a value from its domain to satisfy the constraint

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

How will search proceed?

Demo csp5 with bt4 with system.verbose := 3

A solution is 3--3--2--1

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2
V3
V4
V5
V6
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3
V4
V5
V6
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4
V5
V6
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5
V6
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 1
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 1
V7 = 1
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 1
V7 = 2
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 1
V7 = 3
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 2
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 2
V7 = 1
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 2
V7 = 2
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 2
V7 = 3
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 3
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 3
V7 = 1
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 3
V7 = 2
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 1
V6 = 3
V7 = 3
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 1
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 1
V7 = 1
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 1
V7 = 2
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 1
V7 = 3
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 2
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 2
V7 = 1
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 2
V7 = 2
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 2
V7 = 3
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 3
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 3
V7 = 1
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 3
V7 = 2
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 2
V6 = 3
V7 = 3
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 3
V6
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 3
V6 = 1
V7
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 3
V6 = 1
V7 = 1
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
  - V[1] = V[4]
  - V[4] > V[7]
  - V[7] = V[10] + 1

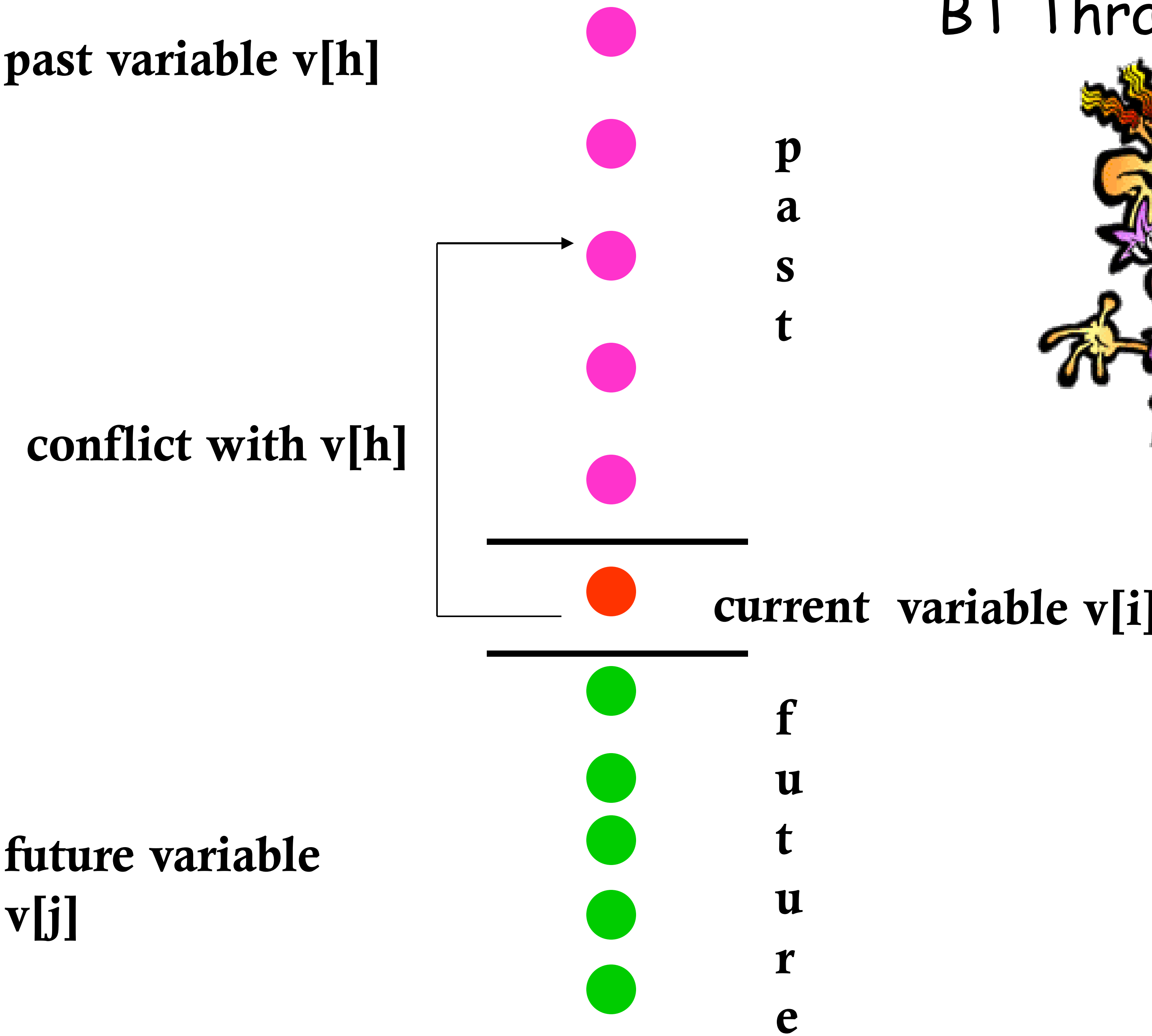
V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 3
V6 = 1
V7 = 2
V8
V9
V10

Consider the following problem (csp5)

- variables V[1] to V[10]
- uniform domains D[1] to D[10] = {1,2,3}
- constraints
    - V[1] = V[4]
    - V[4] > V[7]
    - V[7] = V[10] + 1

V1 = 1
V2 = 1
V3 = 1
V4 = 1
V5 = 3
V6 = 1
V7 = 3
V8
V9
V10

BT Thrashes!



past variable v[h]
p
a
s
t
conflict with v[h]
current variable v[i]
f
u
t
u
r
e
future variable
v[j]

Thrashing:

Slavishly repeating the same set of actions with the same set of outcomes.

Can we minimise thrashing?

Recording conflicts

V1
V2
V3
V4
V5
V6
V7
V8
V9
V10

Cause for conflict in csp5

- When we hit a dead end on V[7] we should jump back to V4
    - the deepest conflicting variable for V[7] is V[4]
    - if there are no more values for V[4] jump back to V[1]
        - the deepest conflicting variable for V[4] or V[7], (excluding V[4])
- and so on

Recording conflicts

Cause for conflict in ***some other*** csp

Conflict Sets

	V1	CS1 = {0}
	V2	CS2 = {0}
	V3	CS3 = {0}
	V4	CS4 = {0,1}
	V5	CS5 = {0}
	V6	CS6 = {0}
	V7	CS7 = {0,4}
	V8	CS8 = {0}
	V9	CS9 = {0}
Current variable →	V10	CS10 = {0,6,7}

Assume search proceeded as follows
- V1, V2, and V3 were instantiated without failures
- First value tried for V4 conflicted with V1
- Second value tried for V4 was compatible with V1, V2, and V3
- V5 and V6 were instantiated without failures
- First and second value tried for V7 failed against V4
- Third value tried for V7 was compatible with all past variables V1 to V6
- V8 and V9 were instantiated without failure
- First value tried for V10 failed against V6
- Second and Third values tried for V10 failed against V7
- V10 has no more values

Recording conflicts

Conflict Sets

Cause for conflict in some other csp

V1	$CS1 = \{0\}$
V2	$CS2 = \{0\}$
V3	$CS3 = \{0\}$
V4	$CS4 = \{0,1\}$
V5	$CS5 = \{0\}$
V6	$CS6 = \{0\}$
V7	$CS7 = \{0,4\}$
V8	$CS8 = \{0\}$
V9	$CS9 = \{0\}$
Current variable → V10	$CS10 = \{0,6,7\}$

- Jump back from V10 to V7
- update $CS7$ to be $CS7 \cup CS10 - \{7\}$
    - the set of variables conflicting with V10 or V7, excluding V7

Recording conflicts

Cause for conflict in some other csp

Conflict Sets

Current variable → V7

Variables	Conflict Sets
V1	CS1 = {0}
V2	CS2 = {0}
V3	CS3 = {0}
V4	CS4 = {0,1}
V5	CS5 = {0}
V6	CS6 = {0}
V7	CS7 = {0,4,6}
V8	CS8 = {0}
V9	CS9 = {0}
V10	CS10 = {0}

- Assume V7 now has no values remaining
- jump back to V[6] and update CS6

Recording conflicts

Conflict Sets

Cause for conflict in some other csp

	V1	CS1 = {0}
	V2	CS2 = {0}
	V3	CS3 = {0}
	V4	CS4 = {0,1}
	V5	CS5 = {0}
Current variable →	V6	CS6 = {0,4}
	V7	CS7 = {0}
	V8	CS8 = {0}
	V9	CS9 = {0}
	V10	CS10 = {0}

- Assume V6 now has no values remaining
- jump back to V[4] and update CS4

Recording conflicts

Conflict Sets

Cause for conflict in some other csp

Current variable → V4

Variables	Conflict Sets
V1	CS1 = {0}
V2	CS2 = {0}
V3	CS3 = {0}
V4	CS4 = {0,1}
V5	CS5 = {0}
V6	CS6 = {0}
V7	CS7 = {0}
V8	CS8 = {0}
V9	CS9 = {0}
V10	CS10 = {0}

- Assume V4 now has no values remaining
- jump back to V[1] and update CS1

Recording conflicts

Conflict Sets

Cause for conflict in some other csp

Current variable → V1

V1	CS1 = {0}
V2	CS2 = {0}
V3	CS3 = {0}
V4	CS4 = {0}
V5	CS5 = {0}
V6	CS6 = {0}
V7	CS7 = {0}
V8	CS8 = {0}
V9	CS9 = {0}
V10	CS10 = {0}

- Assume V1 now has no values remaining
- jump back to the zeroth variable! No solution!

cbj

- Associate with each variable V[i] a conflict set CS[i]
- Initially CS[i] = {0}, for all i

- when labeling a variable V[i]
  - If a consistency check fails between V[i] and V[h]
    - add h to CS[i]

- when unlabeling a variable V[i]
  - jump back to V[h]
    - h is the largest value in CS[i]
  - update conflict set CS[h]
    - $CS[h] := CS[h] \cup CS[i] - \{h\}$
  - reset all variables V[j]
    - $h > j \geq i$

See source code
clairExamples/cbj.cl

CBJ

Remember your conflicts, and when you have used them forget them.

When we instantiate v[i] := x and check(v[i],v[h]) and it fails

- v[i] is in conflict with v[h]
- add h to the set confSet[i]

confSet[i] is then the set of past variables that conflict with values in the domain of v[i]

CBJ Conflict-directed backjumping, exploits failures within the search process

If there are no values remaining for v[i]
Jump back to v[h], where v[h] is the deepest variable in conflict with v[i]
The hope: re-instantiate v[h] will allow us to find a good value for v[i]

If there are no values remaining for v[h]
Jump back to v[g], where v[g] is the deepest variable in conflict with v[i] or v[h]
The hope: re-instantiate v[g] will allow us to find a good value for v[i] or a good value for v[h] that will be good for v[i]

If there are no values remaining for v[g]
Jump back to v[f], where v[f] is the deepest variable in conflict with v[i] or v[h] or v[g]
The hope: re-instantiate v[f] will allow us to find a good value for v[i] or a good value for v[h] that will be good for v[i] or a good value for v[g] that will be good for v[h] and v[i]

What happens if: constraint graph is dense, tight, or highly consistent?

CBJ

When jumping back from v[i] to v[h] update conflict sets

$$\text{confSet}[h] := \text{confSet}[h] \cup \text{confSet}[i] \setminus \{h\}$$
$$\text{confSet}[i] := \{0\}$$

That is, when we jump back from v[h] jump back to a variable that is in conflict with v[h] or with v[i]

Throw away everything you new on v[i]

Reset all variables from v[h+1] to v[i] (i.e. domain and confSet)

CBJ

Looks like bt?

```
PROCEDURE cbj-label (i)
BEGIN
 IF i > n
 THEN print("solution")
 ELSE BEGIN
        consistent ← false;
        FOR v[i] ← EACH ELEMENT OF current-domain[i]
        WHILE not consistent
        DO BEGIN
            consistent ← true;
            FOR h ← 1 TO i-1 WHILE consistent
            DO BEGIN
                consistent ← check(i,h)
                END;
            IF not consistent
            THEN BEGIN
                    current-domain[i]
                      ← remove(v[i],current-domain[i]);
                    pushnew(h,conf-set[i])
                    END
            END
        IF consistent
        THEN cbj-label(i+1)
        ELSE cbj-unlabel(i)
        END
END;
```

```
PROCEDURE cbj-unlabel (i)
BEGIN
 IF i = 0
 THEN print("impossible")
 ELSE BEGIN
        h ← max-list(conf-set[i]);
        conf-set[h] ← remove(h,union(conf-set[h],conf-set[i]));
        FOR j ← h+1 TO i
        DO BEGIN
            conf-set[j] ← {0};
            current-domain[j] ← domain[j]
            END;
        current-domain[h] ← remove(v[h],current-domain[h]);
        IF current-domain[h] ≠ nil;
        THEN cbj-label(h)
        ELSE cbj-unlabel(h)
        END
END;
```

CBJ

A simple modification

```
PROCEDURE cbj-label (i)
BEGIN
 IF i > n
 THEN print("solution")
 ELSE BEGIN
        consistent ← false;
        FOR v[i] ← EACH ELEMENT OF current-domain[i]
        WHILE not consistent
        DO BEGIN
             consistent ← true;
             FOR h ← 1 TO i-1 WHILE consistent
             DO BEGIN
                  consistent ← check(i,h)
                  END;
             IF not consistent
             THEN BEGIN
                    current-domain[i]
                          ← remove(v[i],current-domain[i]);
                    pushnew(h,conf-set[i])
                    END
             END
        IF consistent
        THEN cbj-label(i+1)
        ELSE cbj-unlabel(i)
        END
END;
```

record a conflict

```
PROCEDURE cbj-unlabel (i)
BEGIN
 IF i = 0
 THEN print("impossible")
 ELSE BEGIN
        h ← max-list(conf-set[i]);
        conf-set[h] ← remove(h,union(conf-set[h],conf-set[i]));
        FOR j ← h+1 TO i
        DO BEGIN
             conf-set[j] ← {0};
             current-domain[j] ← domain[j]
             END;
        current-domain[h] ← remove(v[h],current-domain[h]);
        IF current-domain[h] ≠ nil;
        THEN cbj-label(h)
        ELSE cbj-unlabel(h)
        END
END;
```

CBJ

```
PROCEDURE cbj-label (i)
BEGIN
 IF i > n
 THEN print("solution")
 ELSE BEGIN
         consistent ← false;
         FOR v[i] ← EACH ELEMENT OF current-domain[i]
         WHILE not consistent
         DO BEGIN
              consistent ← true;
              FOR h ← 1 TO i−1 WHILE consistent
              DO BEGIN
                   consistent ← check(i,h)
                   END;
              IF not consistent
              THEN BEGIN
                        current-domain[i]
                             ← remove(v[i],current-domain[i]);
                        pushnew(h,conf-set[i])
                        END
              END
         IF consistent
         THEN cbj-label(i+1)
         ELSE cbj-unlabel(i)
         END
END;
```

A simple modification

get back jumping point

```
PROCEDURE cbj-unlabel (i)
BEGIN
 IF i = 0
 THEN print("impossible")
 ELSE BEGIN
         h ← max-list(conf-set[i]);
         conf-set[h] ← remove(h,union(conf-set[h],conf-set[i]));
         FOR j ← h+1 TO i
         DO BEGIN
              conf-set[j] ← {0};
              current-domain[j] ← domain[j]
              END;
         current-domain[h] ← remove(v[h],current-domain[h]);
         IF current-domain[h] ≠ nil;
         THEN cbj-label(h)
         ELSE cbj-unlabel(h)
         END
END;
```

CBJ

```
PROCEDURE cbj-label (i)
BEGIN
 IF i > n
 THEN print("solution")
 ELSE BEGIN
        consistent ← false;
        FOR v[i] ← EACH ELEMENT OF current-domain[i]
        WHILE not consistent
        DO BEGIN
            consistent ← true;
            FOR h ← 1 TO i−1 WHILE consistent
            DO BEGIN
                consistent ← check(i,h)
                END;
            IF not consistent
            THEN BEGIN
                    current-domain[i]
                        ← remove(v[i],current-domain[i]);
                    pushnew(h,conf-set[i])
                    END
            END
        IF consistent
        THEN cbj-label(i+1)
        ELSE cbj-unlabel(i)
        END
END;
```

A simple modification

```
PROCEDURE cbj-unlabel (i)
BEGIN
 IF i = 0
 THEN print("impossible")
 ELSE BEGIN
        h ← max-list(conf-set[i]);
        conf-set[h] ← remove(h,union(conf-set[h],conf-set[i]));
        FOR j ← h+1 TO i
        DO BEGIN
            conf-set[j] ← {0};
            current-domain[j] ← domain[j]
            END;
        current-domain[h] ← remove(v[h],current-domain[h]);
        IF current-domain[h] ≠ nil;
        THEN cbj-label(h)
        ELSE cbj-unlabel(h)
        END
END;
```

update conflict set of backjumping point (aka "culprit")

CBJ

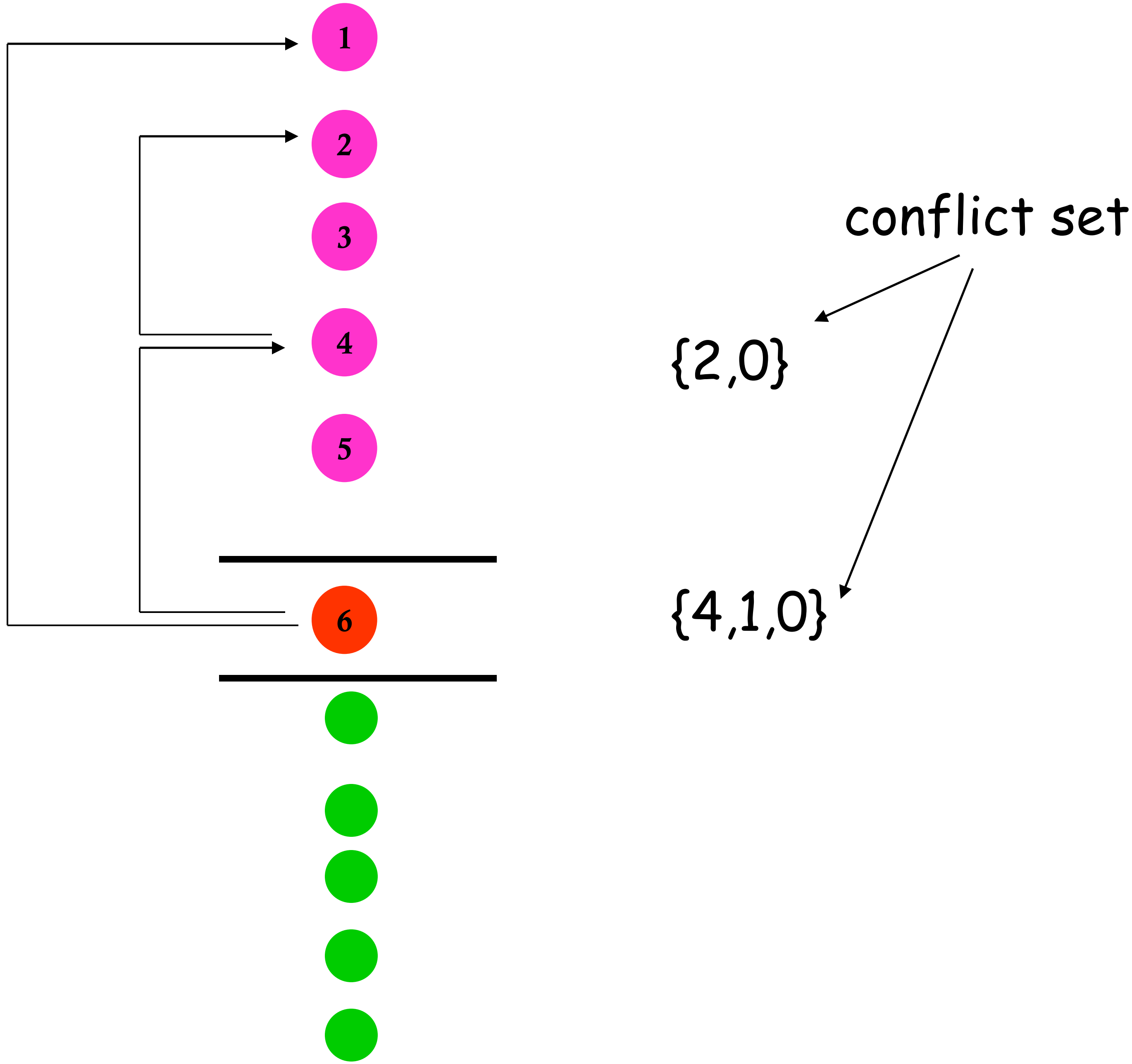
A simple modification

```
PROCEDURE cbj-label (i)
BEGIN
 IF i > n
 THEN print("solution")
 ELSE BEGIN
        consistent ← false;
        FOR v[i] ← EACH ELEMENT OF current-domain[i]
        WHILE not consistent
        DO BEGIN
            consistent ← true;
            FOR h ← 1 TO i−1 WHILE consistent
            DO BEGIN
                consistent ← check(i,h)
                END;
            IF not consistent
            THEN BEGIN
                    current-domain[i]
                        ← remove(v[i],current-domain[i]);
                    pushnew(h,conf-set[i])
                    END
            END
        IF consistent
        THEN cbj-label(i+1)
        ELSE cbj-unlabel(i)
        END
END;
```

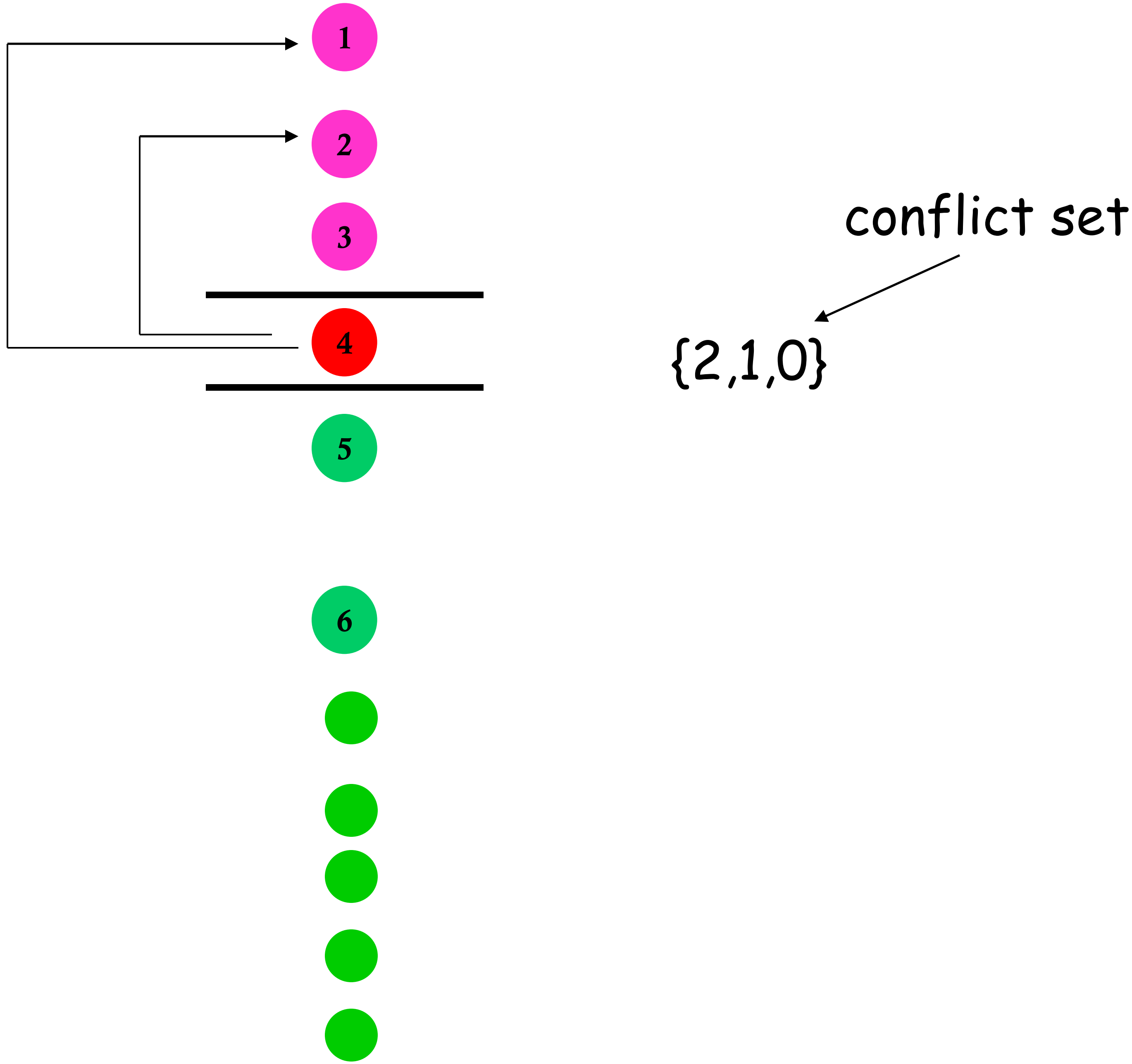
reset variables we jump over

```
PROCEDURE cbj-unlabel (i)
BEGIN
 IF i = 0
 THEN print("impossible")
 ELSE BEGIN
        h ← max-list(conf-set[i]);
        conf-set[h] ← remove(h,union(conf-set[h],conf-set[i]));
        FOR j ← h+1 TO i
        DO BEGIN
            conf-set[j] ← {0};
            current-domain[j] ← domain[j]
            END;
        current-domain[h] ← remove(v[h],current-domain[h]);
        IF current-domain[h] ≠ nil;
        THEN cbj-label(h)
        ELSE cbj-unlabel(h)
        END
END;
```

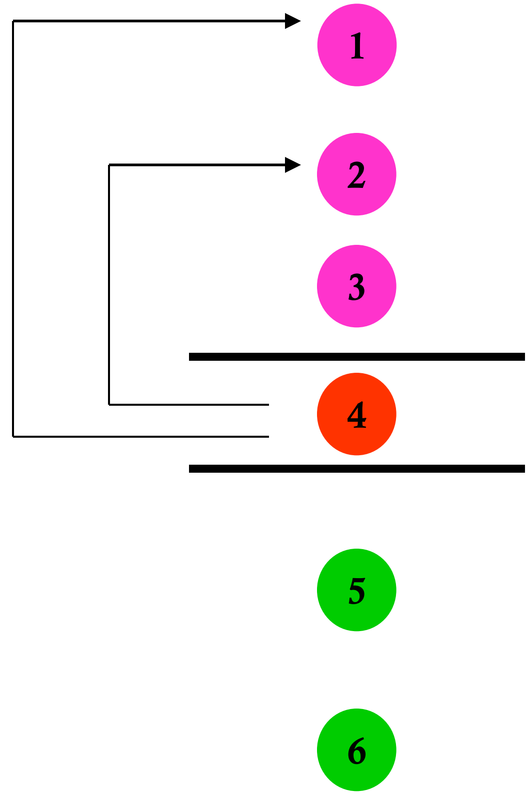
CBJ



CBJ



CBJ (reduce thrashing)



{2,1,0}

Jump back to deepest past variable in confSet (call it h) and then combine confSet[i] with confSet[h]

- History:
  - Konkrat and V Beek,
  - Gent and Underwood

The algorithms were applied to 450 instances of the zebra problem, described in [Dechter 1990 and Smith 1992]. That is, 450 different instantiation orders of the zebra were created, and each algorithm was applied to those problems in turn. Table 1 shows the average number of consistency checks performed by an algorithm, the standard deviation, the minimum number of consistency checks performed, and the maximum number performed over the 450 problems. Table 2 shows the same information but with respect to nodes visited.

Algorithm	μ	σ	min	max
BT	6,357,703	15,024,056	8,755	172,074,472
BJ	940,248	2,321,478	1,393	24,393,906
GBJ	1,120,336	2,745,411	1,563	27,475,100
CBJ	132,492	319,107	538	3,991,581
BM	681,802	1,750,022	1,144	18,439,620
FC	66,386	95,855	432	903,400
BJ-D2C	473,437	1,203,653	1,219	18,156,213
CBJ-DkC	61,681	120,009	538	988,049
FC-D2C	47,492	74,414	396	885,786

Table 1. Consistency Checks

Algorithm	μ	σ	min	max
BT	1,249,087	2,845,631	1,893	29,942,330
BJ	173,620	397,502	274	4,056,985
GBJ	207,848	481,350	364	4,413,676)
CBJ	24,178	55,954	111	632,847
BM	1,249,087	2,845,631	1,893	29,942,330
FC	7,092	9,922	33	76,405
BJ-D2C	87,858	215,807	229	3,077,572
CBJ-DkC	11,317	21,790	111	205,774
FC-D2C	5,422	7,793	33	75,541

FC-D2C	3	7,588

Table 3. CPU Time

Although BT performed on average 8 times as many consistency checks as BM (Table 1) BT took only 20% longer to run than BM (Table 3). This is due to the poor "checking rate" of BM (and this is explained more fully in [Prosser 1991 and 1993]). CBJ has a higher checking rate than BJ. Therefore, not only does CBJ perform less checks than BJ, it performs these checks with less overheads (these tests used the more efficient version of BJ described in [Prosser 1991], rather than the derived version here). This is because CBJ updates conf-set[i] conditionally, and BJ updates max-check[i] unconditionally. Generally, there is an insignificant overhead associated with the modifications performed to BJ (to give us BJ-D2C), CBJ (giving CBJ-DkC), and FC (to FC-D2C). These modifications resulted in a reduction in consistency checks performed, nodes visited, and a reduction in run time. Therefore, with respect to run time the algorithms may be ranked: FC-D2C, CBJ-DkC, FC, CBJ, BJ-D2C, BJ, BM. With the exception of BM, this ranking agrees with those above, and in fact there is little to choose between CBJ-DkC and FC-D2C.

5. The Bridge (and the Long Jump)

It was expected that CBJ-DkC would always perform at least as well as CBJ. However, on analysing the experimental results it was discovered that out of the 450 problem instances there were 2 cases where CBJ performed better than CBJ-DkC. This was a surprise. One of these problems was then examined in detail. This was the problem with the instantiation order: <Water, Tea, Coffee, Japanese, Kools, Blue, Ukranian, Chesterfield, Old-Gold,

instantiated[j] ← false;

ally, since FC and BJ only reason over failures between pairs of variables we can only detect c inconsistencies (1st order learning). On the since CBJ reasons over failures within a set of it can detect directed k-inconsistencies (nth ing).

imental Evaluation

ving algorithms were compared against each T (naive/chronological backtracking), BJ s backjumping routine), GBJ (Dechter's graph-kjumping routine), CBJ (described here), BM s backmarking routine), FC (Haralick and orward checking routine), BJ-D2C, CBJ-DkC, C (again, described here).

algorithms were applied to 450 instances of the blem, described in [Dechter 1990 and Smith at is, 450 different instantiation orders of the e created, and each algorithm was applied to lems in turn. Table 1 shows the average number ency checks performed by an algorithm, the eviation, the minimum number of consistency rformed, and the maximum number performed 50 problems. Table 2 shows the same informa-

The algorithms were then applied again to 100 instances of the zebra problem, and the cpu time was measured. Table 3 below shows the average cpu time used (on a SPARCstation IPC, with 24 mega-bytes of memory, using Sun Common Lisp 4.0) by the algorithms for solving an instance of the problem, and the average number of consistency checks performed in a second.

Algorithm	seconds	checks/sec
BT	123	12,221
BJ	32	8,771
GBJ	29	10,311
CBJ	6	8,953
BM	102	1,659
FC	5	7,707
BJ-D2C	13	7,682
CBJ-DkC	3	8,769
FC-D2C	3	7,588

Table 3. CPU Time

Although BT performed on average 8 times as many consistency checks as BM (Table 1) BT took only 20% longer to run than BM (Table 3). This is due to the poor "checking rate" of BM (and this is explained more fully in [Prosser 1991 and 1993]). CBJ has a higher checking rate than BJ. Therefore, not only does CBJ perform less checks than BJ it performs these checks with less over-

CBJ Variants

BM-CBJ, FC-CBJ, MAC-CBJ

CBJ DkC

If we jump from v[i] to v[h] and confSet[i] = {0,h}
then remove value(v[h]) from domain(h)
value(v[h]) is 2-inconsistent wrt v[i]

If we jump from v[h] to v[g] and confSet[h] = {0,g}
then remove value(v[g]) from domain(g)
value(v[g]) is 3-inconsistent wrt v[i] and v[h]

If we jump from v[g] to v[f] and confSet[g] = {0,f}
then remove value(v[f]) from domain(f)
value(v[f]) is 4-inconsistent wrt v[i] and v[h] and v[g]

What happens if the problem is highly consistent?
See JAIR 14 2001, Xinguang Chen & Peter van Beek

CBJ DkC

gla.ac.uk/publications/PAPERS/8349/hs005.pdf - Windows Internet Explorer

http://www.dcs.gla.ac.uk/publications/PAPERS/8349/hs005.pdf

6.2 IF length(conf-set[i]) = 2
6.3 THEN domain[h] ← remove(v[h],domain[h]);

The above modification gives us CBJ-DkC, where DkC stands for "directed k-consistency" [Dechter and Pearl 1988] (and is similar to nth order learning [Dechter 1990]). The effect of this modification can be described as follows. Let us assume that CBJ has successfully

264 **Constraint Satisfaction Problems**

D2C.

(a) In cbj-label move line 17 to line 12.1

(b) In cbj-label replace line 21 with the fo ment

```
21    THEN BEGIN
21.1       instantiated[i] ← true;
21.2       cbj-label(i+1)
21.3       END
```

(c) In cbj-unlabel add the following lines

```
6.1  IF not(instantiated[i]) and length(conf-set[i]) = 2
6.2  THEN domain[h] ← remove(v[h],domain[h]);
10.1           instantiated[j] ← false;
```

More generally, since FC and BJ only reason over failures ... between pairs of variables we can only detect

If we take consistency checks performed as a search effort we may rank the algorithms as fo D2C, CBJ-DkC, FC, CBJ, BJ-D2C, BM, BJ, With respect to nodes visited the algorithms FC-D2C, FC, CBJ-DkC, CBJ, BJ-D2C, BJ, GB BT).

The algorithms were then applied ag instances of the zebra problem, and the cpu measured. Table 3 below shows the average cp

CBJ ATMS

If we jump from v[i], over v[h], to v[g] and confSet[h] $\subseteq$ {0 .. g-1}
then do NOT reset domain(h) and
do NOT reset confSet(h)

- v[h] is in conflict only with variables "above" v[g]
- none of those conflicting variables have been re-instantiated
- consequently confSet[h] and currentDomain[h] remains valid!

Consider the past variables as assumptions and confSet[i] as an explanation

Down side, we have more work to do.
This is a kind of learning (what kind?)

CBJ ~ DB

confSet[x,i] gives the past variable in conflict with v[i] := x

Finer grained:
on jumping back we can deduce better what values to return to domains

Down side, we have more work to do.
This is an algorithm between CBJ and DB

The bridge and the long jump

Maybe too subtle for part of a lecture

cess v[21] again becomes the current variable and CBJ-DkC considers the instantiation v[21] ← 2. At the same point in the search space CBJ considers the instantiation v[21] ← 1. The two search trees now differ significantly, and in CBJ's search tree it is possible to jump back to a conflicting variable higher up in the search tree than CBJ-DkC.

More generally, CBJ-DkC may remove an infeasible value k from the domain of a variable v[i]. At some later stage in the search process CBJ may move forwards from v[i-1] to v[i], and be unable to re-instantiate v[i] with the value k. CBJ-DkC may then jump back to v[h]. At the same point in the search tree CBJ is allowed to make the instantiation v[i] ← k, and move forwards to v[j]. CBJ may then jump back from v[j] to v[g], where g < h. Therefore, the value k has acted as a bridge that allows the search process to move from one area of the search space to another, where it can then make a "long jump" back to a conflicting variable.

To confirm this analysis, the value 1 was removed from domain[21], the problem was reset, and CBJ and CBJ-DkC were re-run. It was expected that CBJ would be unable to "cross the bridge" and unable to make "a long jump". With the bridge in place CBJ performed 10,746 checks, and visited 1,974 nodes (and CBJ-DkC performed 13,097 checks, and visited 2,390 nodes). With the bridge

bridge has been masked by a reduction in th
should now assume that increased consiste
removal of redundancies, can only guarantee
in search effort if that search is unintelliger
chronological backtracker). Conversely, we s
that we can improve the performance of a
backjumping algorithm by adding an infeasi
the domain of a variable.

References

[Benson and Freuder 1992] B.W. Benson
Freuder, Interchangeability preprocessing can
ward checking search. *Proceedings ECA*
(1992)

[Bitner and Reingold 1975] J.R. Bitner and
Backtrack programming techniques, *Commu*
(1975) 651-656

[Dechter and Pearl 1988] R. Dechter and
Network-based heuristics for constraint-satisf
lems, *Artif. Intell.* 34(1) (1988) 1-38

[Dechter 1990] R. Dechter, Enhancement
constraint processing: backjumping, learning
decomposition, *Artif. Intell.* 41 (3) (1990) 27

[Dechter 1992] R. Dechter, Constraint Ne
Encyclopedia of Artificial Intelligence, Sec

Funny things about cbj (part 2)

Value ordering on insoluble problems can have an effect

But never with BT!

Funny things about cbj

Value ordering on insoluble problems can have an effect

Problem: V1 to V7, each with domain {A,B}
nogoods {(1A,7A),(3A,7B),(5A,7B),(6A,7A),(6A,7B),(6B,7A),(6B,7B)}

Var	Val	confSet
V1	A	
V2	A	
V3	A	
V4	A	
V5	A	
V6	A	
V7	A/B	{1,3}

Var	Val	confSet
V1	A	
V2	A	
V3	B	
V4	A	
V5	A	
V6	A	
V7	A/B	{1,5}

Var	Val	confSet
V1	A	
V2	A	
V3	B	
V4	A	
V5	B	
V6	A	
V7	A/B	{1,6}

Var	Val	confSet
V1	A	
V2	A	
V3	B	
V4	A	
V5	B	
V6	B	
V7	A/B	{1,6}

Finally V6 has no values and cbj jumps to V1

Insoluble because nogoods {(6A,7A),(6A,7B),(6B,7A),(6B,7B)}

Funny things about cbj

Value ordering on insoluble problems can have an effect

Problem: V1 to V7, each with domain {A,B}
nogoods {(1A,7A),(3A,7B),(5A,7B),(6A,7A),(6A,7B),(6B,7A),(6B,7B)}

We now order domains and choose B then A!

Var	Val	confSet
V1	B	
V2	B	
V3	B	
V4	B	
V5	B	
V6	B	
V7	A/B	{6}

Var	Val	confSet
V1	B	
V2	B	
V3	B	
V4	B	
V5	B	
V6	A	
V7	A/B	{6}

Finally V6 has no values and cbj jumps to V0

Value ordering made a difference to an insoluble problem!

Conflicting claims

Smith & Grant IJCAI95:
CBJ helps minimise occurrence of EHP's
random problems as evidence

Bessier & Regin CP96:
CBJ is nothing but an overhead
random problems as evidence

Chen & van Beek JAIR 2001:
CBJ is a tiny overhead
When it makes a difference it is a HUGE difference
random & real problems as evidence

New CBJ

I believe all state of the art sat solvers are using cbj
(or have rediscovered cbj but don't know it)

CBJ for QSAT: see recent AIJ
conflict and solution directed!

Who is not using cbj?

Constraint programming!

We don't jump and we don't learn

Is speed everything?

No

How about explanations and retraction?

Why is cbj not in CP?

Need to propagate laterally (see MAC-CBJ tech report)
but this is no big deal

Need to *get* explanations out of constraints!

Not just writing a good constraint propagator
but a good constraint explainer!

Maybe there is not yet the demand for retraction and explanation
(but I don't believe that)