Fast Solutions to CSP’s

Based on

PROSSER, P. *Hybrid algorithms for the constraint satisfaction problem.*
Computational Intelligence 9 (1993), 268--299

Outline

• Motivation
• Methods of advancing CSP search
  – Backmarking
  – Forward Checking
• Hybrid algorithms
• Conclusion
Problem to be solved

Given a set of \( n \) variables where the \( i^{th} \) variable, \( 1 \leq i \leq n \), has a discrete domain of values it can take, \( \text{domain}[i] \), and a set of binary relations \( C = \{C_{1,1} \ldots C_{1,n} \ C_{2,1} \ldots C_{2,n} \ldots C_{n,n}\} \), find the first consistent instantiation of these variables which satisfies all the relations.

Notation:
- **Current** variable is indexed with \( i \), \( V_i \)
- **Past** variables will be variables that have already been instantiated (those whose index is \( <i \) )
- **Future** variables will be those yet to be instantiated (those whose index is \( >i \) )

Why Binary CSP’s

Every higher order (multiple variables), finite domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced.
What is the forward move and why change it?

Forward move – the procedure that determines what actions to take (consistency checks, bookkeeping, etc) when the next variable is instantiated.

Goal: avoid unnecessary computation

• Backmarking (BM) – remembers consistency checks it already performed
• Forward Checking (FC) – doesn’t expand nodes it knows aren’t feasible

The 5 Base styles of search

Go Backwards

Go Forwards

Different styles

More informed styles

Hybrid Algorithms

Generally Faster
Outline

• Motivation
• Methods of advancing CSP search
  – Backmarking
  – Forward Checking
• Hybrid algorithms
• Conclusion

What does BM do?

• Objective: BM prevents redundant consistency checks when
  – The current variable is known to fail with its current value because of some variable in the past still has the value that made the current variable fail.
  – The current variable is known to succeed with its current value in a check against a past value that still has the same value that made the current variable succeed.
• Tradeoff: must spend time doing, and allot space for, bookkeeping.
What does BM do?

• Maximum checking level (mcl) array
  – Size = Number of variables x domain size
  – mcl[i,k] holds the index of the deepest variable
    that v[i] = k, k ∈ Di, was checked against

• Minimum backup level (mbl) array
  – Size = Number of variables x 1
  – mbl[i] the index of the shallowest past variable
    that has changed value since v[i] was the
    current variable

mcl[i,k] < mbl[i] means the previous consistency
check between v[I] = k and some variable in the
past of mcl[i,k] failed and will still fail because
mcl[i,k] hasn’t been changed

mcl[I,k] >= mbl[i] means v[i] = k passed
consistency checking for all variables in the past
of mbl[i]. V[i] only needs to be checked for
consistency with the new variables, those in the
future of mbl[i]
Suppose no element of the domain of the 5th variable is consistent with the first element of the domain of the first variable.
Backmarking example

Variable #

Remaining domain

1

2

3

4

5

Skipping a step

Backmarking example

Variable #

Remaining domain

1

2

3

4

5

X

i R G B mbl
1 0 0 0 0
2 1 0 0 0
3 2 0 0 0
4 3 0 0 0
5 0 0 0 0

i R G B mbl
1 0 0 0 0
2 1 0 0 0
3 2 0 0 0
4 3 0 0 0
5 0 0 0 0
Backmarking example

Variable #

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Remaining domain

Backmarking example

Variable #

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Remaining domain
No consistency checks at level 5 will be performed until the value of the first variable is changed.

Eventually variable 4 also encounters DWO.
The algorithm does not recognize that no instantiation of variable 4 works while variable 1 is red, it does however save on consistency checks between variable 4 and all variables previous to the shallowest change since variable 4 was last tested.
Forward Checking

• Objective: look ahead to find impossible combinations as soon as possible; “fail early”
  • Remove inconsistent values from the domains of all future variables
  • If domain of future variable is null, then backtrack
  • Trade off: perform speculative checks hoping they will reduce the total number of checks you must perform

Suppose no element of the domain of the 5th variable is consistent with the first element of the domain of the first variable

This entire sub-tree would be impossible. FC detects this in 12 checks, BT detects this in 426 checks
### Forward checking example

5 variables to be instantiated, each has domain \{\text{red, green, blue}\}

<table>
<thead>
<tr>
<th>Variable #</th>
<th>Remaining domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>![Remaining domain for variable 1]</td>
</tr>
<tr>
<td>2</td>
<td>![Remaining domain for variable 2]</td>
</tr>
<tr>
<td>3</td>
<td>![Remaining domain for variable 3]</td>
</tr>
<tr>
<td>4</td>
<td>![Remaining domain for variable 4]</td>
</tr>
<tr>
<td>5</td>
<td>![Remaining domain for variable 5]</td>
</tr>
</tbody>
</table>

First step does 3 x 4 consistency checks

In general, nth step can require up to (N-n) x max domain size consistency checks, N = total number of variables
Forward checking example

<table>
<thead>
<tr>
<th>Variable #</th>
<th>Remaining domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Domain wipe out (DWO) occurs, must backtrack</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example when FC is beat by BT

A problem with 5 variables, each with domain of size = 3

BT would pick the first element in each variable’s domain and find the solution while performing the minimum possible number of consistency checks ($4 + 3 + 2 + 1 = 10$ consistency checks)

FC would propagate the domain reductions downward which would be fruitless effort ($4 \times 3 + 3 \times 3 + 2 \times 3 + 1 \times 3 = 30$ consistency checks)
Outline

• Motivation
• Methods of advancing CSP search
  – Backmarking
  – Forward Checking
• Hybrid algorithms
• Conclusion

Why create a hybrid algorithm?

• Objective: create better CSP algorithms by mixing forward and backward algorithms
  • Requires that the backward and forward algorithms be compatible: information stored by the forward algorithm won’t be corrupted by the actions of the backward algorithm and vice versa
How do you create a hybrid algorithm?

• Implement each algorithm iteratively instead of recursively
  • Two functions: move-forward() and move-backward()
  • WHILE (!solution and !impossible) {
    IF consistent then move-forward()
    ELSE move-backward()
  }
  • Maintain the information required by each algorithm in the forward and backward move

Hybrid Algorithms Implemented by Prosser

Backmark Jumping (BMJ)
Backmarking and Conflict Directed Backjumping (BM-CBJ)
Forward Checking and Backjumping (FC-BJ)
Forward Checking and Conflict Directed Backjumping (FC-CBJ)

He compared these with the base algorithms BT, BJ, CBJ, BM, FC on the ZEBRA problem
## Results: Consistency checks (from Prosser)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>3,858,989</td>
<td>9,616,407</td>
<td>1773</td>
<td>102,267,383</td>
</tr>
<tr>
<td>BJ</td>
<td>503,324</td>
<td>1,524,193</td>
<td>358</td>
<td>19,324,081</td>
</tr>
<tr>
<td>CBJ</td>
<td>63,212</td>
<td>193,846</td>
<td>339</td>
<td>19,324,081</td>
</tr>
<tr>
<td>BM</td>
<td>396,945</td>
<td>1,276,415</td>
<td>401</td>
<td>18,405,514</td>
</tr>
<tr>
<td>BMJ</td>
<td>125,474</td>
<td>361,595</td>
<td>300</td>
<td>5,214,608</td>
</tr>
<tr>
<td>BM-CBJ</td>
<td>25,470</td>
<td>72,004</td>
<td>297</td>
<td>1,237,283</td>
</tr>
<tr>
<td>FC</td>
<td>35,582</td>
<td>71,012</td>
<td>262</td>
<td>802,069</td>
</tr>
<tr>
<td>FC-BJ</td>
<td>16,839</td>
<td>29,977</td>
<td>262</td>
<td>280,302</td>
</tr>
<tr>
<td>FC-CBJ</td>
<td>10,361</td>
<td>16,383</td>
<td>262</td>
<td>119,767</td>
</tr>
</tbody>
</table>

Consistency checks and Nodes visited are two possible, implementation independent metrics for algorithm efficiency. FC’s performance is unfairly enhanced by the Nodes Visited metric since it prunes the tree. Also, consistency checks could possibly be computationally intense for more difficult problems and therefore is the preferable metric.

## How Often One Algorithm (Row) Bettered Another (Column) (from Prosser)

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>BJ</th>
<th>CBJ</th>
<th>BM</th>
<th>BMJ</th>
<th>BM-CBJ</th>
<th>FC</th>
<th>FC-BJ</th>
<th>FC-CBJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BJ</td>
<td>450</td>
<td>0</td>
<td>132</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CBJ</td>
<td>450</td>
<td>450</td>
<td>370</td>
<td>280</td>
<td>0</td>
<td>130</td>
<td>35</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>BM</td>
<td>450</td>
<td>318</td>
<td>80</td>
<td>--------</td>
<td>31</td>
<td>8</td>
<td>13</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>BMJ</td>
<td>450</td>
<td>450</td>
<td>170</td>
<td>419</td>
<td>--------</td>
<td>17</td>
<td>29</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>BM-CBJ</td>
<td>450</td>
<td>450</td>
<td>450</td>
<td>442</td>
<td>433</td>
<td>--------</td>
<td>286</td>
<td>117</td>
<td>35</td>
</tr>
<tr>
<td>FC</td>
<td>450</td>
<td>450</td>
<td>320</td>
<td>437</td>
<td>421</td>
<td>163</td>
<td>--------</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FC-BJ</td>
<td>450</td>
<td>450</td>
<td>415</td>
<td>445</td>
<td>443</td>
<td>333</td>
<td>438</td>
<td>--------</td>
<td>0</td>
</tr>
<tr>
<td>FC-CBJ</td>
<td>450</td>
<td>450</td>
<td>445</td>
<td>448</td>
<td>447</td>
<td>415</td>
<td>440</td>
<td>388</td>
<td>--------</td>
</tr>
</tbody>
</table>

If (i,j) does not equal (j,i) then there were ties.
**Beware**

- If one part of the hybrid performs poorly on a problem, then it's possible that the hybrid will perform poorly as well.
  - CBJ is usually better than BM, but when BM beats CBJ, it's possible that BM might outperform BM-CBJ
    - In jumping back, you change some past variable further up the tree than you would with BM alone. This causes you to forget about the previous consistency checks that have been performed since $\text{mbl}[i] < \text{mcl}[i,k]$
Why do BM schemes sometimes outperform FC-CBJ?

- FC performs speculative consistency checks, these may pare down the tree (detect DWO) or they may just be a waste of time
- Can you detect DWO without performing excess consistency checks?
  - No, but Minimal Forward Checking (MFC) will reduce the number of excess consistency checks

MFC

- To detect DWO you only need to check for existence of a single consistent instantiation for a variable
  - Use lazy evaluation for all consistency checks beyond those necessary to find the first consistent instantiation
- MFC = adding DWO detection to BM with lazy evaluation of consistency checks (hybrid of FC and BM)
- Results: MFC usually performs better than BM, FC, and FC-CBJ
Outline

- Motivation
- Methods of advancing CSP search
  - Backmarking
  - Forward Checking
- Hybrid algorithms
- Conclusion

Conclusion

- BM and FC are powerful in their ability to reduce consistency checks
- Hybrids generally perform better than their constituent parts
  - Poor performance of one algorithm in the hybrid might mean poor hybrid performance
- Can even make hybrids within the forward move (MFC)

Information on MFC from
Framework for the iterative Constraint Satisfaction Search Problem

1  PROCEDURE bcssp(n, status)
2    BEGIN
3      consistent = true
4      status = “unknown”
5      i = 1
6      WHILE status == “unknown”
7      DO BEGIN
8      IF consistent
9      THEN i = label(i, consistent)
10     ELSE i = unlabel(i, consistent)
11     IF i > n
12     THEN status = “solution”
13     ELSE IF i = 0
14     THEN status = “impossible”
15     END
16    END

FUNCTION bm-label(I, consistent) return INTEGER
BEGIN
    consistent = false
    FOR k = EACH ELEMENT OF current-domain[I] WHILE not consistent
    DO BEGIN
        consistent = mcl[I,k] >= mbl[I]
        FOR h = mbl[I] TO I-1 WHILE consistent
        DO BEGIN
            v[I] = k
            consistent = check(I, h)
            mcl[I,k] = h
        END
        IF not consistent
        THEN current-domain[I] = remove(v[I], current-domain[I])
    END
    IF consistent THEN return(I+1) ELSE return(I)
END
FUNCTION bm-unlabel(I, consistent) return INTEGER
BEGIN
    h = I-1
    current-domain[I] = domain[I]
    mbl[I] = h
    FOR j = h+1 TO n DO mbl[j] = min(mbl[j], h)
    current-domain[h] = remove(v[h], current-domain[h])
    consistent = current-domain[h] != null
    return(h)
END

FUNCTION check-forward(I,j) returns BOOLEAN
BEGIN
    reduction = null
    FOR v[j] = EACH ELEMENT OF current-domain[j]
        DO IF not check(I,j)
            THEN push(v[j], reduction)
        IF reduction != null
            THEN BEGIN
            current-domain[j] = set-difference(current-domain[j], reduction)
            push(reduction, reductions[j])
            push(j, future-fc[I])
            push(I, past-fc[j])
            END
        END
    return (current-domain[j] != null)
END
PROCEDURE update-current-domain(I)
BEGIN
    current-domain[I] = domain[I]
    FOR reduction = EACH ELEMENT OF reductions[I] DO
        current-domain[I] = set-difference(current-domain[I], reduction)
    END
END

FUNCTION fc-label(I, consistent) returns INTEGER
BEGIN
    consistent = false
    FOR k = EACH ELEMENT OF current-domain[I] WHILE not consistent DO
        consistent = true
        FOR j = I+1 TO n WHILE consistent DO
            consistent = check-forward(I,j)
        IF not consistent THEN
            current-domain[I] = remove(v[I], current-domain[I])
            undo-reductions(I)
        END
    END
    IF consistent THEN return(I + 1) ELSE return(I)
END
FUNCTION fc-unlabel(I,consistent) returns INTEGER
BEGIN
    h = I-1
    Undo-reductions(h)
    Update-current-domain(I)
    current-domain[h] = remove(v[h],current-domain[h])
    consistent = current-domain[h] != null
    return(h)
END