

CSC2512
Advanced Propositional
Reasoning

CSC2512: GAC Search

```
GAC_Enforce()  
  // GAC-Queue contains all constraints one of whose variables has  
  // had its domain reduced. Before search we run GAC_Enforce  
  // with all constraints on GAC-Queue  
while GACQueue not empty  
  C = GACQueue.extract()  
  for V := each member of scope(C)  
    for d := CurDom[V]  
      Find set of assignments A for all other  
      variables in scope(C) such that  
      C(A  $\cup$  V=d) = True  
      if A not found  
        CurDom[V] = CurDom[V] - d  
        if CurDom[V] =  $\emptyset$   
          empty GACQueue  
          return DWO //return immediately  
        else  
          push all constraints C' such that  
          V  $\in$  scope(C') and C'  $\notin$  GACQueue  
          on to GACQueue  
return TRUE //while loop exited without DWO
```

CSC2512: Enforcing GAC

- However, finding a support for $V=d$ in constraint C still in the worst case requires $O(2^k)$ work, where k is $|\text{scope}(C)|$ (the **arity** of C).

CSC2512: Enforcing GAC

- Enforcing GAC more efficiently.

A Fast Arc Consistency Algorithm for n-ary Constraints. By
Olivier Lhomme and Jean-Charles Regin

CSC2512: GAC-Schema

- They consider **TABLE** constraints. These are constraints specified simply by relations: a set of satisfying tuples.
- **GAC-Schema**
- Order and number all of the satisfying tuples in the constraint. Give them their lexicographic number from the cross product of the domains. (This allows us to assign a unique number even to falsifying tuples).

CSC2512: GAC-Schema

#	X1	X2	X3	X4	X5	X6
0	0	0	0	0	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
40	1	0	1	0	0	0
48	1	1	0	0	0	0

- Lexicographically ordered.
- Build a list for each value/variable pair (x,a) of the tuples in this constraint that support it. (That is, tuples that assign $x=a$)
- $(x1,0) = \{0,5, 6\}$; $(x1, 1) = \{40,48\}$; ...; $(x6,0) = \{0, 5, 6, 40, 48\}$; $(x6,1) = \{5\}$

CSC2512: GAC-Schema

#	X1	X2	X3	X4	X5	X6
0	0	0	0	0	0	0
5	0	0	0	1	0	1
6	0	0	0	1	1	0
40	1	0	1	0	0	0
48	1	1	0	0	0	0

- For each (x,a) pair pick one tuple \mathbf{t} that assigns x the value a as a support.
- Add \mathbf{t} to the set $Sc(x_i,v)$ for each $\mathbf{t}[x_i] = v$
- Add (x,a) the set of tuples supported by \mathbf{t} , $S(\mathbf{t})$
- $(x_1, 0)$ supported by $\mathbf{0}$. Add $\mathbf{0}$ to $Sc(x_1,0)$, $Sc(x_2,0)$, $Sc(x_3,0)$, $Sc(x_4,0)$, $Sc(x_5,0)$, and $Sc(x_6,0)$
- Add (x,a) to $S(\mathbf{0})$

CSC2512: GAC-Schema

- So $Sc(x,v)$ contains all tuples currently used for supporting some value that could also support x , i.e., $\mathbf{t} \in Sc(x,v) \iff \mathbf{t}[x] = v$
- $S(\mathbf{t})$ contains all values \mathbf{t} is currently a support for.
- After set up we do the following when a value is deleted from a variable's domain, say $DEL(x,a)$
- Each support \mathbf{t} such that $\mathbf{t}[x] = a$ is now invalid, this is the set $Sc(x,a)$.
 - Remove each such tuple \mathbf{t} from $Sc(z,b)$ for each $\mathbf{t}[z] = b$.
- For each value/variable pair currently being supported by \mathbf{t} , we must find a new support, this is the set $S(\mathbf{t})$.
 - We empty $S(\mathbf{t})$ and look for a new support for each element

CSC2512: GAC-Schema

- Let (x,a) be a variable/value pair in $S(\mathbf{t})$, we need a new support for (x,a)
 - First check all tuples in $Sc(x,a)$, if one such \mathbf{t} is still valid we use it as a new support for (x,a)
 - add (x,a) to $S(\mathbf{t})$
 - If any \mathbf{t} we test is not valid we remove it from $Sc(x,a)$ (but not from the other $Sc(z,\mathbf{t}[z])$ sets---we are doing this lazily.
 - If we didn't find a support in $Sc(x,a)$ we must look at the global list of tuples of the constraint supporting (x,a) looking for a valid tuple

CSC2512: GAC-Schema

- We had already built a list of supporting tuples for each variable/value (x,v) pair.
- Call these sets $T(C, x,v)$, so $T(C,x1,0) = \{0,5, 6\} \dots$
- We maintain an index into this list, initially 0, and update this index as we search it for the next **valid** tuple.
 - A **valid tuple** is one in which every variable/value pair in the tuple is still in the domain of the corresponding variable.
- **last(x,v)** is the tuple in $T(C,x,v)$ that is at the current index. This is the last tuple we checked for validity. All tuples in $T(C,x,v)$ that are lexicographically below **last(x,v)** have already been discarded as being invalid.

CSC2512: GAC-Schema

- So if we didn't find a support for (x,a) in $Sc(x,a)$ we start at **last** (x,a) in $T(C,x,a)$ and move through $T(C,x,a)$ for a valid support. If we find one we return it and update **last** (x,a)
- If we get a new support t we update the sets $Sc(z,t[z])$ and $S(t)$ (this set now contains (x,a))
- If we didn't find a support in $T(C,x,a)$ we remove a from $Dom[x]$, adding it to the deletion list. If $Dom[x]$ becomes empty we can return DWO.
- We process all elements in the deletion list until it becomes empty or we detect DWO.

CSC2512: GAC-Schema

A Fast Arc Consistency Algorithm for n-ary Constraints. By Olivier Lhomme and Jean-Charles Regin

- This paper aims to improve on GAC-Schema by using the lexicographic ordering of tuples in the constraint.
- It operates the same up to the point where we can't find a support for (x,a) in the set $Sc(x,a)$ and now have to search $T(C,x,a)$
- GAC-Schema starts searching $T(C,x,a)$ at **last(x,a)**.
- First improvement: find higher place in $T(C,x,a)$ to start searching while making sure we don't skip any valid tuples.

CSC2512: GAC-Schema

First improvement: find higher place in $T(C,x,a)$ to start searching while making sure we don't skip any valid tuples.

- A tuple \mathbf{t} is a **lower bound of the smallest valid tuple** (lbsvt) **w.r.t.** (\mathbf{x},\mathbf{a}) and \mathbf{C} , if all tuples in $T(C,x,a)$ that are less than \mathbf{t} are not valid.
- $\mathbf{last}(\mathbf{x},\mathbf{a})$ is a lbsvt
- Let $\mathbf{lb}(\mathbf{x},\mathbf{a})$ be any lbsvt
- For any variable y , let $\mathbf{minlb}(y) = \min_{\{b \in \text{Dom}[y]\}} \text{lb}(y,b)$
 - The lowest valid tuple over all domain values of variable y .
 - Note that for any variable y , $\mathbf{minlb}(y)$ is at least as small as the lexicographically minimum valid tuple in \mathbf{C} . But we might not want to search for a the new minimum valid tuple in \mathbf{C} when a the current minimal tuple is deleted.

CSC2512: GAC-Schema

- So the first improvement, instead of searching $T(C,x,a)$ at **last(x,a)**, they can start the search at
$$\max[\text{lb}(x,a), \max_{\{y \neq x\}} \text{minlb}(y)]$$
- Difficulty is that this is a tuple (or tuple index), and we can't go directly from such a tuple to an index into $T(C,y,b)$.
- Define **nextIn**((x,a), **t**). Given a tuple **t** we return the next tuple in $T(C,x,a)$ that is greater than **t**. (or an index into $T(C,x,a)$).
 - Requires some space to maintain such **nextIn**: each tuple must point to the next tuple that has $x=a$ for every value/variable pair.

CSC2512: GAC-Schema

- Second improvement
- Instead of stepping through from the starting point in $T(C,x,a)$ we try to skip over invalid tuples.
- Let \mathbf{s} be a lbsvt w.r.t, (x,a) and C , and $y \neq x$, define

$$n(y,\mathbf{s}) = \min_{\{b \in \text{Dom}[y]\}} [\mathbf{nextIn}((y,b), \max(\text{lb}(y,b), \mathbf{s}))]$$

- Any tuple less than \mathbf{s} is not a valid support for (x,a) . If we look at the possible supports for (y,b) that are greater than \mathbf{s} (and possibly valid, i.e., also greater than $\text{lb}(y,b)$), and take the minimum over all values for y , this is a lbsvt for (x,a) .
 - Any valid supporting tuple for (x,a) must assign y some value, and thus must be a valid support for (y,b) for some value b .
 - So any valid support for (x,a) must lie above the minimum of the possibly valid supports for (y,b) where the min is take over all values b still in $\text{Dom}[y]$.

CSC2512: GAC-Schema

- So the next valid support for (x,a) must be $\geq n(y,s)$ for all variables y and the current next possible value of $T(C,y,b)$
- With the two improvements we get the seekValidSupport function:

Algorithm 5: New Function SEEKVALIDSUPPORT

```
SEEKVALIDSUPPORT( $C$ : constraint,  $x$ : variable,  $a$ : value):
tuple
1  $\sigma \leftarrow \text{NEXTIN}((x, a), \max[lb(x, a), \max_{y \in (X(C) - \{x\})} (minlb(y))])$ 
   if  $\sigma \succ ub(x, a)$  then return nil
   while  $\neg \text{ISVALID}(\sigma)$  do
2    $\sigma \leftarrow \text{NEXT}((x, a), \sigma)$ 
3   for each  $y \in (X(C) - \{x\})$  do
      $n(y, \sigma) \leftarrow \min_{b \in D(y)} [\text{NEXTIN}((y, b), \max(lb(y, b), \sigma))]$ 
4    $t \leftarrow \text{NEXTIN}[(x, a), \max_{y \in (X(C) - \{x\})} (n(y, \sigma))]$ 
5   if  $t \succ \sigma$  then  $\sigma \leftarrow t$ 
     if  $\sigma \succ ub(x, a)$  then return nil
    $lb(x, a) \leftarrow \sigma$ 
   return  $\sigma$ 
```

CSC2512: GAC-Schema

- The results are quite good.
Random problems arity 14 constraints over Boolean variables with 2^{13} randomly chosen tuples per constraint

#ct	#bk	new time	old time	gain
8	184	1.9	26.1	13.7
10	382	5.0	59.8	12.0
12	421	6.3	73.6	11.7
14	305	5.2	59.3	11.4
16	199	3.8	41.5	10.9

- #ct is the number of constraints
 - #bk the number of choice points
 - “old” corresponds to the GAC-Scheme+allowed algorithm of (Bessière & Régin 1997)
 - “new” represents our method
 - “gain” is the cpu time ratio
- Times are expressed in seconds.

- Note they don't really describe how to backtrack their data structures.

CSC2512: GAC-Schema

- Increase the arity to 20 with fewer percentage of satisfying tuples (only 30,000).

#ct	#bk	new time	old time	gain
1	34	0.02	0.56	28
2	30	0.04	1.94	48
3	23	0.09	1.96	22
4	53	0.64	26.28	41
5	1122	21.64	> 300	> 10

CSC2512: GAC-Schema

- Small arity constraints (6) over larger domain variables 10

#ct	#bk	new time	old time	gain
1	10	0.05	0.4	8
2	8	0.01	0.6	60
3	8	0.04	1.0	25
4	59	0.3	13.5	45
5	14	0.2	7.9	40
6	155	2.9	188	65
7	133	3.3	207	63
8	313	6.3	407	65
9	2439	64	> 3,600	> 56

CSC2512: GAC-Schema

- All of these results are on random constraints. They claim that their technique works well on non-random problems but only report that on a bigger version of the example they use in the paper one reduces GAC time from 5648 ms to 10ms.
- However, this example is especially constructed to work well with their approach.
- No convincing data for non-random problems.

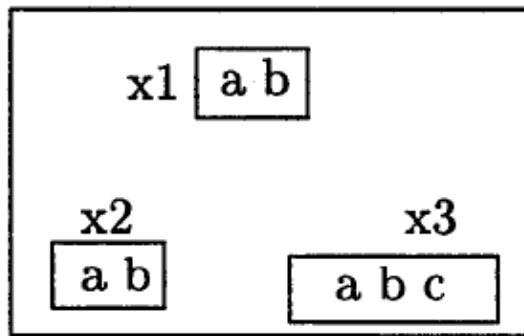
CSC2512: GAC Propagators

CSC2512: GAC-Propagators

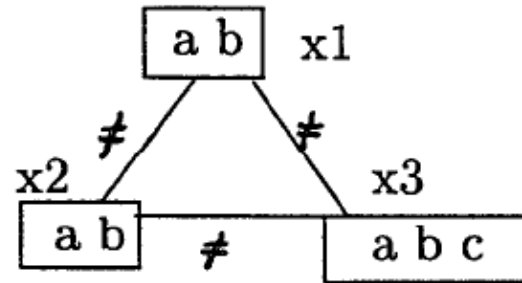
- Some constraints appear frequently in real problems, and also have special structure.
- We don't need to represent them as TABLE constraints, and we don't need to employ the generic GAC algorithm.
- The generic GAC algorithm is exponential in the arity of the constraint, we can enforce GAC on some constraints in time polynomial in their arity.
- The first such propagator developed in the literature was the all-different propagator due to Regin.
- All-diff(X_1, \dots, X_n) satisfied by assignments to these variables if and only if every variable is assigned a unique value.

CSC2512: GAC-Propagators

- All-diff can be represented by a clique of binary not equal constraints:
 - All-diff(x_1, x_2, x_3) can be represented by $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_3$



Representation by
3-ary constraint



Representation by
binary constraints
of difference

Figure 1.

CSC2512: GAC-Propagators

- However, enforcing GAC on the collection of binary not-equals is not as powerful as enforcing GAC on the 3-ary constraint!
- $\text{Dom}[X1] = \{a, b\}$
- $\text{Dom}[X2] = \{a, b\}$
- $\text{Dom}[X3] = \{a, b, c\}$
- Each value of each variable is supported in each not-equals constraint.
- However, the valid tuples in the 3-ary constraint are
 $X1=a, X2 = b, X3 = c$
 $X1=b, X2 = a, X3 = c$

so $X3=a$, and $X3=b$ can be pruned by GAC on the 3-ary constraint

CSC2512: GAC-Propagators

- We use graph theory to find GAC-inconsistent values.
- Build a value graph for the all-diff constraint

$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$Dx_1 = \{1, 2\}$

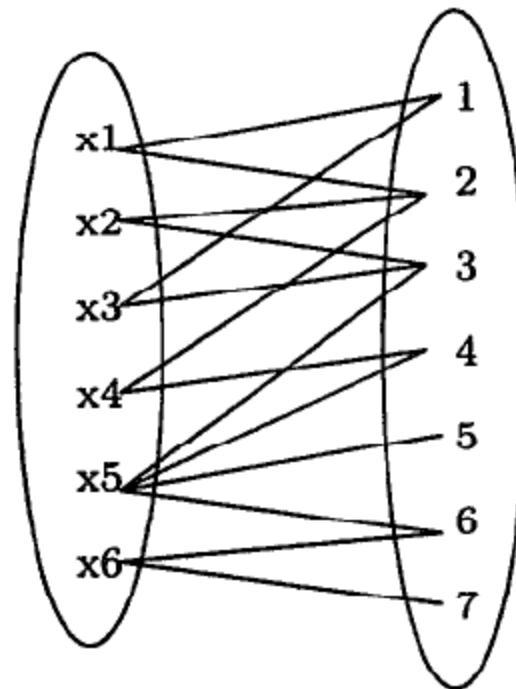
$Dx_2 = \{2, 3\}$

$Dx_3 = \{1, 3\}$

$Dx_4 = \{2, 4\}$

$Dx_5 = \{3, 4, 5, 6\}$

$Dx_6 = \{6, 7\}$



CSC2512: GAC-Propagators

- Vertex for each value and each variable
- Edge between (x,v) if and only if v is in $\text{Dom}[x]$
- Results in a bipartite graph. (Two sets, A, B, each edge goes from A to B).

$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$\text{Dom}x_1 = \{1, 2\}$

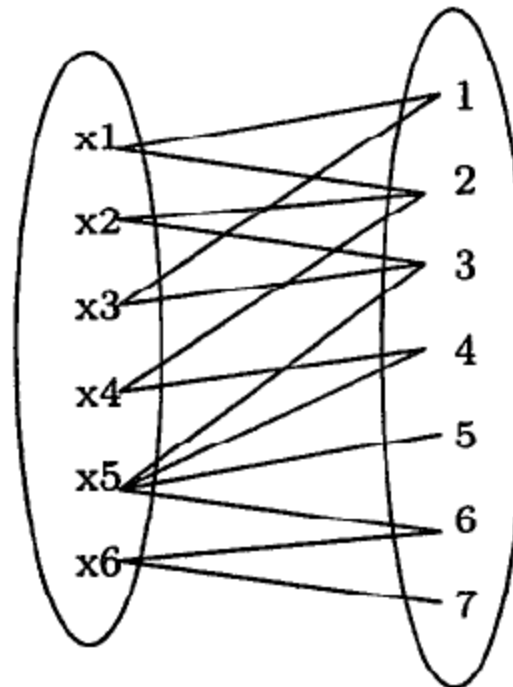
$\text{Dom}x_2 = \{2, 3\}$

$\text{Dom}x_3 = \{1, 3\}$

$\text{Dom}x_4 = \{2, 4\}$

$\text{Dom}x_5 = \{3, 4, 5, 6\}$

$\text{Dom}x_6 = \{6, 7\}$



CSC2512: GAC-Propagators

- A matching is a subset of edges if no two edges have a vertex in common. A maximum cardinality matching is a **maximum matching**. And a matching **M** covers a set X if every vertex of X is an end point of some edge in **M**
- For the All-Diff value graph a valid tuple in the constraint corresponds to a maximal matching that covers all of the variable nodes.
- So we have a correspondence between matchings and valid tuples.
- We can find a matching covering X in time $O(X^{1/2} m)$, where m is the number of edges = sum of the domain sizes of the variables.
- Note that if the variable domains are very different and overlapping in complex ways, there isn't an obviously better way to determine if the constraint can be satisfied.

CSC2512: GAC-Propagators

- Finding that there is at least one satisfying tuple isn't enough to do GAC.

Let \mathbf{M} be a matching, and edge in \mathbf{M} is a **matching** edge, every edge not in \mathbf{M} is **free**. A vertex is matched if it is incident to a matching edge and **free** otherwise. An **alternating** path or **cycle** is a simple path or cycle whose edges alternate between **free** and **matching**. An **even** path is one with an **even** number of edges. And an edge that appears in every matching is **vital**.

Berge 1970: An edge belongs to some of, but not all maximum matchings iff, for an arbitrary matching \mathbf{M} it belongs to either an even alternating path that starts at a free vertex, or an even alternating cycle

Basically we can alter \mathbf{M} by switching the edges in the cycle/path from free to matching.

CSC2512: GAC-Propagators

- Now the idea is immediate:
 - Find a maximal matching that covers the variable vertices
 - If none found, we have a DWO
 - Else, every variable value pair (x,a) is supported iff it is contained in a maximal matching.
 - So search for alternating paths starting from free vertices (these must be value vertices) or even alternating cycles (cycles detected by computing strongly connected components). Any edge not in the original matching (and thus not vital), and not in an alternating path or alternating cycle is pruned.
- More recent ways of propagating all-diff use Hall Sets: a set of variables such that union of their domains is the same size as the set of variables → this set of variables **uses up** these values.

CSC2512: Next Week

- Modeling, different ways a problem can be configured as a CSP and the influence of this on our ability to solve it.
- [Modelling](#)
Barbara M. Smith. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Chapter 11, pages 377-406. Elsevier, 2006. ISBN 0-444-52726-5
 - Long but easy to read.
- Sat and CSP. **To be Presented:**
 - **GAC Via Unit Propagation.** [Bacchus, F.](#) 2007. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP-2007)*, 133-147.