Efficient Algorithms and Heuristics for Strong Local Consistencies

Anastasia Paparrizou
Advisor: Kostas Stergiou

Department of Informatics and Telecommunications Engineering,
University of Western Macedonia, Greece
Contents

- Introduction to Table constraints
- GAC algorithms for table constraints
  - Simple Tabular Reduction (STR)
- Strong Local Consistencies
  - New efficient algorithms for table constraints
- Adaptive propagation
  - Heuristics
- Conclusions and future directions
Table constraints

- Table constraints are constraints given in extension by listing the tuples of values allowed or forbidden by a set of variables.
- They are widely studied in constraint programming (CP) as they are present in many real-world applications
  - design
  - configuration
  - databases
  - preferences’ modeling.
- So far, research on table constraints has mainly focused on the development of fast algorithms to enforce generalized arc consistency (GAC).
- GAC algorithms delete inconsistent values from variable domains and achieve the maximum level of filtering when constraints are treated independently.
GAC algorithms for Table constraints

- Classical algorithms iterate over lists of tuples in different ways
- Recent developments, however, suggested maintaining dynamically the list of supports in constraint tables: these are the variants of simple tabular reduction (STR)
- Alternatively, specially-constructed intermediate structures such as tries (Gent et al. 2007) or multi-valued decision diagrams (MDDs) (Cheng and Yap 2010) have been proposed.
- A more recent development of AC5-based algorithms has also been proposed in (Mairy, Van Hentenryck and Deville 2012), but its relevance has been shown on binary/ternary constraints only.
- Among this variety of algorithms, STR2 along with the MDD approach are considered to be the most efficient ones (especially, for large arity constraints).
STR algorithms

- Structures
  - initialization

- Algorithm’s steps
  - All tuples are checked until `currentLimit[c]` is reached
    - if a tuple is valid then
      - values are added to `gacValues[x]`, `gacValues[y]`, `gacValues[z]` respectively
    - else tuple is removed
  - foreach variable `x ∈ scp(c)`
    - if `gacValues[x] ⊂ dom(x)` then
      - `dom(x) ← gacValues[x]`
      - if `dom(x) = Ø` return `FALSE`
      - add any `ci` to `Q`, s.t. `ci ≠ c ∧ x ∈ scp(ci)`
STR algorithms

- Structures
- initialization
- propagation
- backtracking

STR applied after the removal of (z, 1). (y, 2) no longer has support and will therefore be deleted.

Structures obtained after backtracking
STR algorithms

- **Structures**
- **initialization**
- **propagation**
- **backtracking**

**STR** applied after the removal of \((z, 1)\).
\((y, 2)\) no longer has support and will therefore be deleted.

**Structures obtained after backtracking**
Strong Local Consistencies

- GAC algorithms process one constraint at a time and thus, they cannot exploit possible intersections that may exist between different constraints.
- On the other hand, existing algorithms for consistencies stronger than GAC that can exploit constraint intersections are generic and thus very expensive.
- A specialized algorithm for table constraints, called maxRPWC+, that achieves a consistency stronger than GAC was proposed very recently (Paparrizou and Stergiou 2012).
- This algorithm extends the GACva algorithm (Lecoutre and Szymanek 2006) and enforces a domain filtering restriction of PWC, called max Restricted PairWise Consistency (maxRPWC) (Bessiere, Stergiou, and Walsh 2008).
One objective of this research is to propose efficient algorithms for strong local consistencies that can be applied on table constraints and can be easily adopted by standard CP solvers.

Towards this, we propose a new higher-order consistency algorithm for table constraints, called $e\text{STR}^*$.

It is based on simple tabular reduction ($\text{STR}$) that is able to efficiently achieve Full PairWise Consistency ($\text{PWC+GAC}$).

Despite its high space and time requirements to construct its structures, its worst-case time complexity is quite close to that of STR algorithms.

The concept of $e\text{STR}^*$ is to extend any STR-based algorithm to achieve stronger pruning, simply by introducing a set of counters for each intersection between any two constraints $c_i$ and $c_j$. 
Extending STR algorithms

- **Structures**
- **description**

-eSTR structures for the intersection of \( C_1 \) with \( C_2 \) on variables \( Y \) and \( Z \). The highlighted values show the first occurrence of the different subtuples for \( scp(C_1) \cap scp(C_2) \).

- \( ctr[c][ci] \) holds the number of valid tuples in \( table[c] \) that include the subtuple for variables in \( scp(c) \cap scp(ci) \) that appears in at least once in \( table[c] \).

- \( ctrIndexes[c][ci] \) holds the index of the counter in \( ctr[c][ci] \) that is associated with the subtuple \( scp(c) \cap scp(ci) \).

- \( ctrLink[c][ci] \) is an array of size \( ctr[c][ci].length \) that links \( ctr[c][ci] \) with \( ctr[ci][c] \). It holds the index of the counter in \( ctr[ci][c] \) that is associated with that subtuple. If the subtuple is not included in any tuple of \( table[ci] \) then \( ctrLink[c][ci][j] \) is set to NULL.
**eSTR algorithm**

- **Structures**
- **initialization**

- **Algorithm’s steps**
  - All tuples are checked until `currentLimit[c]` is reached
    - **if** a tuple is valid AND PW-consistent
      - values are added to `pwValues[x]`, `pwValues[y]`, `pwValues[z]` respectively
    - **else** tuple is removed
      - counter is updated
  - **foreach** variable `x ∈ scp(c)`
    - **if** `pwValues[x] ⊂ dom(x)`
      - `dom(x) ← pwValues[x]`
    - **if** `dom(x) = Ø` return **FALSE**
    - add any `ci` to `Q`, s.t. `ci ≠ c` ∧ `x ∈ scp(ci)`

AAAI 2013
eSTR algorithm

- Structures
- propagation

ESTR checks if the tuple (1, 0, 0) of C₁ is PW-consistent

Function 2 isPWconsistent(c, index)
1: for each c₄ ≠ c₁ s.t. |scp(c₄) ∩ scp(c)⟩ > 1 do
2: j ← ctrIndexes[c][c₁][index]
3: k ← ctrLink[c][c₁][j]
4: if k = NULL OR ctr[c][c][k] = 0 then
5: return FALSE
6: return TRUE

Function 3 updateCtr(c, index)
1: for each c₄ ≠ c₁ s.t. |var(c₄) ∩ var(c)| > 1 do
2: j ← ctrIndexes[c][c₁][index]
3: ctr[c][c₁][j] ← ctr[c][c₁][j] - 1
4: if ctr[c][c₁][j] = 0 then
5: add c₄ to Q
The eSTR algorithm

- **Structures**
- propagation

The eSTR algorithm operates on databases to ensure consistency. It involves the following steps:

1. **Function 2** `isPWconsistent(c, index)`
   - for each `c_i` ≠ `c` s.t. `|scp(c_i) ∩ scp(c)| > 1`
   - `j ← ctrIndexes[c][c_i][index]`
   - `k ← ctrLink[c][c_i][j]`
   - if `k = NULL OR ctr[c_i][c][k] = 0`
     - return `FALSE`
   - return `TRUE`

2. **Function 3** `updateCtrl(c, index)`
   - for each `c_i` ≠ `c` s.t. `|var(c_i) ∩ var(c)| > 1`
   - `j ← ctrIndexes[c][c_i][index]`
   - `ctr[c_i][c][j] ← ctr[c][c_i][j] - 1`
   - if `ctr[c][c_i][j] = 0`
     - add `c_i` to `Q`

A weak version of eSTR, denoted by eSTRw, can be obtained by discarding lines 4–5 of Function 3 (i.e., the update of Q is ignored when a PW-support is lost).

The algorithm involves creating and managing databases with different attributes and ensuring consistency through these functions. The diagram illustrates the process with tables and counters.
Theoretical results

- Algorithm eSTR applied to a CN P enforces Full PairWise Consistency on P.
- PWC+GAC and PWC+maxRPWC are equivalent.
- The consistency level achieved by Algorithm eSTRw is incomparable to maxRPWC and PWC.
- The worst-case time complexity of one call to eSTR is $O(rd + \max(r, g)t)$ where $r$ denotes the arity of the constraint, $t$ the size of its table and $g$ the number of intersecting constraints.
  - The worst-case time complexity of STR is $O(rd + rt)$ (Lecoutre 2011).
- The worst-case space complexity of eSTR for handling one constraint is $O(n + \max(r, g)t)$.
  - The worst-case space complexity of STR is $O(n + rt)$ per constraint (Lecoutre 2011). Each additional eSTR structure is $O(t)$ per intersecting constraint, giving $O(gt)$. 

AAAI 2013
**eSTR2w vs. STR2**

- Points above the diagonal are solved faster by eSTR2w. The majority of the instances are above and belong to Random, Random-forced and Dubois.

- On Aim classes eSTR2w can outperform STR2 by several orders of magnitude on some instances.

- They are particularly expensive on classes of problems which include intersections on large sets of variables, as is the case with the Positive-table and BDD instances.
Adaptive Propagation

- Since GAC may still be superior in many problems we also suggest ways to interleave GAC with stronger consistency algorithms.
- One such way is to apply heuristics that can dynamically select between GAC and a stronger propagator during search.
- We describe and evaluate simple, fully automated heuristics that monitor the effects of propagation and are applicable on constraints of any arity.
- Experimental results demonstrate that the proposed heuristics for adaptive propagation result in a more robust solver.
Fully Automated Heuristics

- **Objective**: The exploitation of the filtering power offered by strong propagation methods without incurring severe CPU time penalties or requiring user involvement.

- **Concept**: Switching between a weak (W) and a strong (S) propagator for individual constraints during search when a propagation event occurs.

  - The $H_{dwo}$ (resp. $H_{del}$) heuristic applies a standard propagator on a constraint (e.g. domain consistency) until the constraint causes a domain wipeout - DWO (resp. at least one value deletion). Then, in the immediately following revision of the constraint, a stronger local consistency (e.g. SAC) is applied. (Stergiou 2008)

- **Refinements** of $H_{dwo}$ and $H_{del}$
  - $H^v_{dwo}$ (resp. $H^v_{del}$) restricts the application of the strong propagator on variables that suffered a propagation event (DWO or value deletion) in the immediately preceding constraint revision as opposed to all variables in the constraint’s scope.
AC3 schema with $H_{dwo}$

1: Q←C
2: while Q ≠ Ø do
3:   pick and delete c from Q
4:   rev[c]++
5:   if rev[c]−dwo[c]=1 then
6:     apply $S$
7:   else apply $W$
8:   if dom(x)=Ø {∀ x ∈ scp(c)} then
9:     dwo[c]=rev[c]
10:    return FAIL
11: return SUCCESS
Experiments

- We have considered \textit{GAC} as the standard propagator \textit{W}, given that it is the most commonly used local consistency.
- As the \textit{S} propagator we have considered two strong local consistencies, \textit{maxRPWC} and \textit{SAC}, since we are interested in non-binary problems.
- This figure clearly demonstrates the performance gap between \textit{GAC} and \textit{maxRPWC}.
  - \textit{GAC} is faster on the majority of the instances, often by large margins.
  - Since it is a weaker consistency level, it sometimes thrashes, while the stronger \textit{maxRPWC} does not.
  - These results justify the need for a robust method that can achieve a balance between the two.
GAC vs. $H^v_{dwo}$

- This figure clearly demonstrates the benefits of the adaptive heuristics.
- Although the majority of the instances is still below the diagonal they are much closer to it, indicating small differences between the two methods.
- These are instances where the application of $\text{maxRPWC}^+$ does not offer any notable reductions in search tree size.
- On the other hand, there are still instances where GAC$\text{va}$ thrashes while $H^v_{dwo}$, following $\text{maxRPWC}^+$, does not.
Conclusions

- We have introduced a new higher-order consistency algorithm for table constraints that enforces FPWC.
- It is based on an original combination of two techniques that have proved their worth: simple tabular reduction and tuple counting.
- Moreover, we have shown that adaptive propagation schemes can exploit efficiently the advantages offered by strong propagators in a fully automated way.
- The presented work can pave the way for the design and implementation of even more efficient higher-order methods for table constraints.
- Also, it can perhaps help initiate a wider study on specialized higher-order consistency algorithms for global constraints.
- We believe that strong local consistencies can pay off, provided that we have efficient methods to apply them.
Publications

Extending STR algorithms

• The central idea of eSTR* is to store the number of times that each subtuple appears in the intersection of any two constraints.
  • For each constraint \( c \), we introduce a set of counters for each (non trivial) intersection between \( c \) and another constraint \( c_i \).

• Assuming that \( S \) is the set of variables that are common to both \( c \) and \( c_i \), at any time each counter in this set holds the number of valid tuples in \( c \)'s table that include a specific combination of values for \( S \).

• In this way, once a tuple \( \tau \in \text{table}(c) \) has been verified as valid, we can check if it has a PW-support in \( \text{table}(c_i) \) simply by observing the value of the corresponding counter (i.e., the counter for subtuple \( [scp(c) \cap scp(c_i)] \)).

• If this counter is greater than 0 then \( \tau \) has a PW-support in \( \text{table}(c_i) \).
• Importantly, this check is done in constant time.

AAAI 2013
Extending STR algorithms

- **Structures**
  - **description**

  - The eSTR structures for the intersection of $C_1$ with $C_2$ on variables Y and Z. The highlighted values show the first occurrence of the different subtuples for $\text{scp}(C_1) \cap \text{scp}(C_2)$.

  - $\text{ctr}[c][ci]$, holds the number of valid tuples in $\text{table}[c]$ that include the subtuple for variables in $\text{scp}(c) \cap \text{scp}(ci)$ that appears in at least once in $\text{table}[c]$.

  - $\text{ctrIndexes}[c][ci]$ holds the index of the counter in $\text{ctr}[c][ci]$ that is associated with the subtuple $[\text{scp}(c) \cap \text{scp}(ci)]$.

  - $\text{ctrLink}[c][ci]$ is an array of size $\text{ctr}[c][ci].\text{length}$ that links $\text{ctr}[c][ci]$ with $\text{ctr}[ci][c]$. It holds the index of the counter in $\text{ctr}[ci][c]$ that is associated with that subtuple. If the subtuple is not included in any tuple of $\text{table}[ci]$ then $\text{ctrLink}[c][ci][j]$ is set to NULL.
Indicative instances...

- Comparing eSTR2 to STR2 it seems that there are problem classes where it can be considerably more efficient (Random, Random-forced and Dubois).
- eSTR2 can outperform STR2 by several orders of magnitude on some instances of Aim classes.
- The new algorithms are over one order of magnitude faster than STR2 on Positive table-10 instances which are proven unsatisfiable without search.
- The extra filtering of eSTR2 does pay off on some classes as node counts are significantly reduced (Aim) while on other classes it does not (Random).
- On the other hand, STR2 is better than the proposed algorithm on Positive table problems and of course BDD, where eSTR2 and eSTR2w exhausted the available memory.
- Finally, comparing our algorithms to maxRPWC+ it is clear that they are superior as they are faster on all the tested classes (except BDD).