Data Structures for Generalised Arc Consistency for Extensional Constraints

Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale

( AAAI – 2007 )

Presented by:
Zissis Poulos (MASc, Dept. of ECE)

CSC2512S: Advanced Propositional Reasoning
Feb 10th 2012
OUTLINE

Introduction
- Background
- Contributions
- Proposed Data Structures
- Experimental Results
- Conclusions
INTRODUCTION

- Constraint programming can solve a wide range of combinatorial problems (scheduling, industrial design etc.)
- Based on a backtracking search process
  - Decide a value – Propagate the constraints
- Extensional (table) constraints are an important tool
- Recent interest in fast propagation algorithms for extensional constraints
OUTLINE

- Introduction
- Background
- Contributions
- Proposed Data Structures
- Experimental Results
- Conclusions
Extensional constraints express relations straightforwardly:

- Explicitly list all the allowed combinations of values to a specific subset of the variables

Example: \( x \) “likes” \( y \)

Domains of \( x \) and \( y \): \{bill, bert, tom\}

Set of satisfying assignments:

\{[bill, bert], [bill, tom], [bert, tom]\}

Avoid the complexity of using algorithms to compute the satisfying assignments (intensional constraints)
A constraint \( c \) is GAC iff for every variable \( x_i \) in \( \text{scope}(c) \) and every value \( v \) in \( D_i \), there is at least one assignment in \( \text{scope}(c) \) that assigns \( v \) to \( x_i \) and satisfies \( c \).

The support: values for variables other than \( x_i \) participating in the assignment \( (x_i, v) \) (or \( x_i = v \)).
OUTLINE

- Introduction
- Background
- Contributions
- Proposed Data Structures
- Experimental Results
- Conclusions
CONTRIBUTIONS

- Improve the search of “support” for a given assignment
- Two efficient data structures to improve the propagation of extensional constraints
  - Next Difference Lists (novel)
  - Tries
- Explore effectiveness on both random and structured problems
- Evaluation in two different implementations:
  - GAC-schema
  - Integration into the solver Minion
OUTLINE

- Introduction
- Background
- Contributions
- Proposed Data Structures
- Experimental Results
- Conclusions
**NEXT DIFFERENCE LISTS**

- **Naive way:**
  - Divide the set of tuples into lists of supporting tuples for each variable $x$ and value $a$
  - Linearly search for a valid tuple (expensive)

- **Next Difference List:**
  - Improvement to simple approach
  - Can skip some tuples while searching for support
NEXT DIFFERENCE LISTS

Each item in the list is a record:
- a tuple \( t \)
- a pre-computed array of list indices \( ND \)

\( ND(x) \) is the index of the next tuple containing a different value for \( x \)

Constraint scope: \{x,y,z\}
Allowed tuples:
\{[0,0,0], [0,0,1], [1,1,0], [1,1,1]\}

Jump from tuple 1 to tuple 3 when searching for a support for \( (z,0) \)
- Can use one or multiple lists
A trie (/ˈtriː/ or /ˈtraiː/) is a tree efficient at storing/retrieving strings (Fredkin, 1960)

- Searching takes $O(\text{length of string})$

Key-idea: view tuples as a string and insert them into a trie

- Branches of uniform length
- Each level corresponds to a variable in the scope
- Testing if a tuple satisfies the constraint is cheap
- Testing if $(x, v)$ has a support is cheap when $x$ is first in the scope of the constraint
TRIES

Constraint scope: \{x,y,z\}

Allowed tuples:
\{[0,0,0], [0,0,1], [1,1,0], [1,1,1]\}

0 has been pruned from \text{Dom}(z)

Search for support for \((x,0)\): cheap!

However: if variable is last in the scope, then search becomes expensive

Use one trie per element \(e\)

(Space vs. Time trade-off)

- Tries can also be used to re-establish support
OUTLINE

- Introduction
- Background
- Contributions
- Proposed Data Structures
- Experimental Results
- Conclusions
EXPERIMENTAL RESULTS – RELATED WORK

- **Hologram** (Lhomme and Regin, 2005)
  - can find next tuple with any value in fixed # of steps
- **Lecoutre and Szymanski** (2006)
  - binary search, sort lists in lexicographic order
- **Carlsson** (2006)
  - DAG
- **Cheng and Yap** (2006)
  - ROBDDs, only binary variables
- **Zhang and Stickel** (2000)
  - tries for storing clause sets
EXPERIMENTAL RESULTS – GAC SCHEMA

Tries vs. Simple
Can improve overall time more than 5 times

Next Difference Lists vs. Simple
Similar improvements compared to Tries
Use of multiple lists often causes memory explosion
EXPERIMENTAL RESULTS – GAC SCHEMA

Tries vs. Hologram

Tries always faster (1.5x to 2.5x)
EXPERIMENTAL RESULTS – GAC WITH WATCHED LITERALS

Minion with Tries vs. Simple

Significant improvement (up to 500-fold)

Next Difference Lists performed well with one list, but worse than Tries in difficult instances

Minion with Tries vs. Hologram

Minion with Tries outperforms Hologram in difficult instances. Hologram does better (up to 2x) for simple instances (<5 sec)
EXPERIMENTAL RESULTS – GAC WITH WATCHED LITERALS

Minion with Tries vs. Lecoutre & Szymanek

Almost always slower than Minion with tries
OUTLINE

- Introduction
- Background
- Contributions
- Proposed Data Structures
- Experimental Results
- Conclusions
CONCLUSIONS

- Data structures for tuple search
- Tries perform well both on watched-literal GAC and GAC-Schema
- The novel Next Difference Lists are simple but not as effective compared to tries
- Next Difference with one list has lower space complexity
- Tries > Lecroute & Szymanek > Next Difference Lists
- Minion benefits from integrating tries
- Extensive experiments to support the pros/cons of each implementation