Making Compact-Table Compact

Linnea Ingmar

Joint work with Christian Schulte

Uppsala University

Part of the work has been carried out at
KTH Royal Institute of Technology

NordConsNet
29th May, 2018

Outline

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

1. Background

2. Dynamically Compact Sparse Bit-Sets
   Compact Bit-Sets
   Dynamically Compact Data-Structures

3. Sharing Tables

4. Evaluation

5. Conclusion

UPPSALA UNIVERSITET

Compact-Table

Background

Dynamically Compact Sparse Bit-Sets

Compact Bit-Sets

Dynamically Compact Data-Structures

Sharing Tables

Evaluation

Conclusion

References

- Propagation algorithm for table constraints [1], [2], [3]

Compact-Table

- Propagation algorithm for table constraints [1], [2], [3]
- Elegant and simple

Compact-Table

- Propagation algorithm for table constraints [1], [2], [3]
- Elegant and simple
- First described in 2016

Compact-Table

- Propagation algorithm for table constraints [1], [2], [3]
- Elegant and simple
- First described in 2016
- Outperforms previously known algorithms

Compact-Table

- Propagation algorithm for table constraints [1], [2], [3]
- Elegant and simple
- First described in 2016
- Outperforms previously known algorithms
- First implemented in OR-tools, now it exists in many solvers

Compact-Table

- Solutions defined by an **explicit table** of tuples

x_0	1	2	1	2	6	7	4	1	1	8	2	1	5	7	3	0
x_1	8	1	3	0	7	4	2	9	6	5	1	1	0	5	2	1
x_2	1	7	8	2	4	9	1	1	7	3	2	5	1	9	3	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Compact-Table

- Solutions defined by an **explicit table** of tuples
- Tuples are numbered from 0 to $n - 1$

x_0	1	2	1	2	6	7	4	1	1	8	2	1	5	7	3	0
x_1	8	1	3	0	7	4	2	9	6	5	1	1	0	5	2	1
x_2	1	7	8	2	4	9	1	1	7	3	2	5	1	9	3	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

UPPSALA UNIVERSITET

Compact-Table

- Solutions defined by an **explicit table** of tuples
- Tuples are numbered from 0 to $n - 1$
- Maintained in a bit-set (array of 64-bit words)

	w_0				w_1				w_2				w_3			
words	**1**	**1**	0	**1**	**1**	0	0	0	**1**	0	**1**	**1**	**1**	0	0	**1**

x_0	**1**	**2**	1	**2**	**6**	7	4	1	**1**	8	**2**	**1**	**5**	7	3	**0**
x_1	**8**	**1**	3	**0**	**7**	4	2	9	**6**	5	**1**	**1**	**0**	5	2	**1**
x_2	**1**	**7**	8	**2**	**4**	9	1	1	**7**	3	**2**	**5**	**1**	9	3	**0**
	0	**1**	2	**3**	**4**	5	6	7	**8**	9	**10**	**11**	**12**	13	14	**15**

Background

Dynamically Compact Sparse Bit-Sets

Compact Bit-Sets

Dynamically Compact Data-Structures

Sharing Tables

Evaluation

Conclusion

References

Compact-Table

- Solutions defined by an **explicit table** of tuples
- Tuples are numbered from 0 to $n-1$
- Maintained in a bit-set (array of 64-bit words)
- The i-th bit is set iff the i-th tuple is still valid

	w_0				w_1				w_2				w_3			
words	**1**	**1**	0	**1**	**1**	0	0	0	**1**	0	**1**	**1**	**1**	0	0	**1**

x_0	**1**	**2**	1	**2**	**6**	7	4	1	**1**	8	**2**	**1**	**5**	7	3	**0**
x_1	**8**	**1**	3	**0**	**7**	4	2	9	**6**	5	**1**	**1**	**0**	5	2	**1**
x_2	**1**	**7**	8	**2**	**4**	9	1	1	**7**	3	**2**	**5**	**1**	9	3	**0**
	0	**1**	2	**3**	**4**	5	6	7	**8**	9	**10**	**11**	**12**	13	14	**15**

Support Bit-Sets

- For each variable-value pair $\langle x, v \rangle$

Support Bit-Sets

- For each variable-value pair $\langle x, v \rangle$
- Static bit-set masks computed once

Support Bit-Sets

- For each variable-value pair $\langle x, v \rangle$
- Static bit-set masks computed once
- Encode which tuples support the pair

x_0	1	**2**	1	**2**	6	7	**4**	1	1	8	**2**	1	5	7	**3**	0
x_1	8	1	3	0	7	4	2	9	6	5	1	1	0	5	2	1
x_2	1	7	8	2	4	9	1	1	7	3	2	5	1	9	3	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Support Bit-Sets

- For each variable-value pair $\langle x, v \rangle$
- Static bit-set masks computed once
- Encode which tuples support the pair
- The i-th bit set iff tuple nr. i has value v at x's position

x_0	1	**2**	1	**2**	6	7	**4**	1	1	8	**2**	1	5	7	**3**	0
x_1	8	1	3	0	7	4	2	9	6	5	1	1	0	5	2	1
x_2	1	7	8	2	4	9	1	1	7	3	2	5	1	9	3	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Support Bit-Sets

- For each variable-value pair $\langle x, v \rangle$
- Static bit-set masks computed once
- Encode which tuples support the pair
- The i-th bit set iff tuple nr. i has value v at x's position

...

supports$_{\langle x_0, 2\rangle}$

0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	0

x_0	1	**2**	1	**2**	6	7	**4**	1	1	8	**2**	1	5	7	**3**	0
x_1	8	1	3	0	7	4	2	9	6	5	1	1	0	5	2	1
x_2	1	7	8	2	4	9	1	1	7	3	2	5	1	9	3	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

UPPSALA UNIVERSITET

Support Bit-Sets

- For each variable-value pair $\langle x, v \rangle$
- Static bit-set masks computed once
- Encode which tuples support the pair
- The i-th bit set iff tuple nr. i has value v at x's position

...

$\text{supports}_{\langle x_0, 2 \rangle}$	0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	0
$\text{supports}_{\langle x_0, 3 \rangle}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0

x_0	1	**2**	1	**2**	6	7	**4**	1	1	8	**2**	1	5	7	**3**	0
x_1	8	1	3	0	7	4	2	9	6	5	1	1	0	5	2	1
x_2	1	7	8	2	4	9	1	1	7	3	2	5	1	9	3	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

UPPSALA UNIVERSITET

Background

Dynamically Compact Sparse Bit-Sets

Compact Bit-Sets

Dynamically Compact Data-Structures

Sharing Tables

Evaluation

Conclusion

References

Support Bit-Sets

- For each variable-value pair $\langle x, v \rangle$
- Static bit-set masks computed once
- Encode which tuples support the pair
- The i-th bit set iff tuple nr. i has value v at x's position

...

$\texttt{supports}_{\langle x_0, 2\rangle}$	0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	0
$\texttt{supports}_{\langle x_0, 3\rangle}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0
$\texttt{supports}_{\langle x_0, 4\rangle}$	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0

...

x_0	1	**2**	1	**2**	6	7	**4**	1	1	8	**2**	1	5	7	**3**	0
x_1	8	1	3	0	7	4	2	9	6	5	1	1	0	5	2	1
x_2	1	7	8	2	4	9	1	1	7	3	2	5	1	9	3	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

A Variable Loses Values

$$\text{dom}(x_0) = \{\not 1, 2, 3, 4, \not 5, \not 6, \not 7, \not 8\}$$

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

A Variable Loses Values

$$\mathrm{dom}(x_0) = \{\cancel{1}, 2, 3, 4, \cancel{5}, \cancel{6}, \cancel{7}, \cancel{8}\}$$

$\texttt{supports}_{\langle x_0, 2\rangle}$			
0 **1** 0 **1**	0 0 0 0	0 0 **1** 0	0 0 0 0

UPPSALA UNIVERSITET

A Variable Loses Values

$$\mathrm{dom}(x_0) = \{\not 1, 2, 3, 4, \not 5, \not 6, \not 7, \not 8\}$$

$\texttt{supports}_{\langle x_0,2\rangle}$	0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	0
$\vee$																
$\texttt{supports}_{\langle x_0,3\rangle}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0

A Variable Loses Values

$$\text{dom}(x_0) = \{\cancel{1}, 2, 3, 4, \cancel{5}, \cancel{6}, \cancel{7}, \cancel{8}\}$$

$\texttt{supports}_{\langle x_0, 2\rangle}$	0 **1** 0 **1**	0 0 0 0	0 0 **1** 0	0 0 0 0
		$\vee$		
$\texttt{supports}_{\langle x_0, 3\rangle}$	0 0 0 0	0 0 0 0	0 0 0 0	0 0 **1** 0
		$\vee$		
$\texttt{supports}_{\langle x_0, 4\rangle}$	0 0 0 0	0 0 **1** 0	0 0 0 0	0 0 0 0
		=		

A Variable Loses Values

$$\mathrm{dom}(x_0) = \{\not{1}, 2, 3, 4, \not{5}, \not{6}, \not{7}, \not{8}\}$$

$\texttt{supports}_{\langle x_0,2\rangle}$	0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	0
$\vee$																
$\texttt{supports}_{\langle x_0,3\rangle}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0
$\vee$																
$\texttt{supports}_{\langle x_0,4\rangle}$	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0
=																
mask	0	**1**	0	**1**	0	0	**1**	0	0	0	**1**	0	0	0	**1**	0

A Variable Loses Values

$$\text{dom}(x_0) = \{\not 1, 2, 3, 4, \not 5, \not 6, \not 7, \not 8\}$$

$\texttt{supports}_{\langle x_0, 2\rangle}$	0 1 0 1	0 0 0 0	0 0 1 0	0 0 0 0
∨				
$\texttt{supports}_{\langle x_0, 3\rangle}$	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0
∨				
$\texttt{supports}_{\langle x_0, 4\rangle}$	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0
=				
mask	0 1 0 1	0 0 1 0	0 0 1 0	0 0 1 0
&				
words (old)	1 1 0 1	1 0 0 0	1 0 1 1	1 0 0 1
=				

A Variable Loses Values

$$\mathrm{dom}(x_0) = \{\not{1}, 2, 3, 4, \not{5}, \not{6}, \not{7}, \not{8}\}$$

$\mathtt{supports}_{\langle x_0, 2\rangle}$	0 1 0 1	0 0 0 0	0 0 1 0	0 0 0 0
			∨	
$\mathtt{supports}_{\langle x_0, 3\rangle}$	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 0
			∨	
$\mathtt{supports}_{\langle x_0, 4\rangle}$	0 0 0 0	0 0 1 0	0 0 0 0	0 0 0 0
			=	
mask	0 1 0 1	0 0 1 0	0 0 1 0	0 0 1 0
			&	
words (old)	1 1 0 1	1 0 0 0	1 0 1 1	1 0 0 1
			=	
words (new)	0 1 0 1	0 0 0 0	0 0 1 0	0 0 0 0

A Variable Loses Values

$$\mathrm{dom}(x_0) = \{\not{1}, 2, 3, 4, \not{5}, \not{6}, \not{7}, \not{8}\}$$

$\mathtt{supports}_{\langle x_0, 2\rangle}$	0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	0
∨																
$\mathtt{supports}_{\langle x_0, 3\rangle}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0
∨																
$\mathtt{supports}_{\langle x_0, 4\rangle}$	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0
=																
mask	0	**1**	0	**1**	0	0	**1**	0	0	0	**1**	0	0	0	**1**	0
&																
words (old)	**1**	**1**	0	**1**	**1**	0	0	0	**1**	0	**1**	**1**	**1**	0	0	**1**
=																
words (new)	0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	0

	0	**1**	2	**3**	4	5	6	7	8	9	**10**	11	12	13	14	15
x_0	1	**2**	1	**2**	6	7	4	1	1	8	**2**	1	5	7	3	0
x_1	8	**1**	3	**0**	7	4	2	9	6	5	**1**	1	0	5	2	1
x_2	1	**7**	8	**2**	4	9	1	1	7	3	**2**	5	1	9	3	0

Filtering Variable Domains

- Filter out values where `words` & $\texttt{supports}_{\langle x,v \rangle} = 0$

Filtering Variable Domains

- Filter out values where $\texttt{words} \ \& \ \texttt{supports}_{\langle x,v \rangle} = 0$

$$\text{dom}(\texttt{x}_1) = \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$$

Filtering Variable Domains

- Filter out values where `words` & $\texttt{supports}_{\langle x,v \rangle} = 0$

$$\text{dom}(\mathtt{x}_1) = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots\}$$

`words`	0	**1**	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
								&								
$\texttt{supports}_{\langle x_1,0 \rangle}$	0	0	0	**1**	0	0	0	0	0	0	0	0	**1**	0	0	0
								=								
	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0

Value `0` is kept.

Filtering Variable Domains

- Filter out values where $\texttt{words} \ \& \ \texttt{supports}_{\langle x,v \rangle} = 0$

$$\text{dom}(\texttt{x}_1) = \{\texttt{0}, \texttt{1}, 2, \ldots\}$$

Same for 1.

Filtering Variable Domains

- Filter out values where $\texttt{words} \; \& \; \texttt{supports}_{\langle x,v \rangle} = 0$

$\text{dom}(\texttt{x}_1) = \{\texttt{0}, \texttt{1}, \not{\texttt{2}}, \ldots\}$

words	0	**1**	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
&																
$\texttt{supports}_{\langle x_1,2 \rangle}$	0	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	**1**	0
=																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Value $\texttt{2}$ is removed.

Filtering Variable Domains

- Filter out values where $\texttt{words} \,\&\, \texttt{supports}_{\langle x,v\rangle} = 0$

$\text{dom}(\texttt{x}_1) = \{\texttt{0}, \texttt{1}, \not{\texttt{2}}, \ldots\}$

words	0	**1**	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
&																
$\texttt{supports}_{\langle x_1,2\rangle}$	0	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	**1**	0
=																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Value 2 is removed.

And so on. . .

Sparse Bit-Sets

- Indexing structure tracks emptiness
- Operates on non-empty words only
- Performs well even when non-empty words are **sparse**

words	**1 1** 0 **1**	**1** 0 0 0	**1** 0 **1 1**	**1** 0 0 **1**
index	0	1	2	3

limit $= 4$

Sparse Bit-Sets

Intersection with `mask`:

`mask`	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
`words`	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	**1** 0 0 **1**
`index`	0	1	2	3
`limit`	= 4			

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
				&
words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	**1** 0 0 **1**
index	0	1	2	3

limit $= 4$

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	0 0 0 0
index	0	1	2	3

limit $= 3$

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0

words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	
index	0	1	2	

limit = 3

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
			&	
words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	
index	0	1	2	

limit $= 3$

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0

words	**1** **1** 0 **1**	**1** 0 0 0	0 0 **1** **1**	
index	0	1	2	

limit $= 3$

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
		&		
words	**1** **1** 0 **1**	**1** 0 0 0	0 0 **1** **1**	
index	0	1	2	
limit	= 3			

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0

words	**1** **1** 0 **1**	0 0 0 0	0 0 **1** **1**	
index	0	1	2	

limit $= 3$

Sparse Bit-Sets

`mask`	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0

`words`	**1** **1** 0 **1**	0 0 0 0	0 0 **1** **1**	
`index`	0	1	2	

`limit` $= 3$

`index`[2] overwrites (or is swapped with) `index`[1]

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
words	**1** **1** 0 **1**	0 0 0 0	0 0 **1** **1**	
index	0	2		

limit = 2

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
	&			
words	**1** **1** 0 **1**	0 0 0 0	0 0 **1** **1**	
index	0	2		
limit	= 2			

Sparse Bit-Sets

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
words	**1** 0 0 0	0 0 0 0	0 0 **1** **1**	
index	0	2		

limit $= 2$

Sparse Bit-Sets

	w_0	w_1	w_2	
words	**1** 0 0 0	0 0 0 0	0 0 **1** **1**	
index	0	2		
limit	$= 2$			

- Further operations only consider w_0 and w_2

Sparse Bit-Sets

	w_0	w_1	w_2	
words	**1** 0 0 0	0 0 0 0	0 0 **1** **1**	
index	0	2		
limit	= 2			

- Further operations only consider w_0 and w_2
- Trailing solver: undo operations upon backtrack

Sparse Bit-Sets

	w_0	w_1	w_2	
words	**1** 0 0 0	0 0 0 0	0 0 **1** **1**	
index	0	2		
limit	= 2			

- Further operations only consider w_0 and w_2
- Trailing solver: undo operations upon backtrack
- Copying solver: make copies of the state

Sparse Bit-Sets

	w_0	w_1	w_2	
`words`	**1** 0 0 0	0 0 0 0	0 0 **1** **1**	
`index`	0	2		
`limit`	= 2			

- Further operations only consider w_0 and w_2
- Trailing solver: undo operations upon backtrack
- Copying solver: make copies of the state
- `words` is not compact in memory

Sparse Bit-Sets

	w_0	w_1	w_2	
`words`	**1** 0 0 0	0 0 0 0	0 0 **1** **1**	
`index`	0	2		
`limit`	$= 2$			

- Further operations only consider w_0 and w_2
- Trailing solver: undo operations upon backtrack
- Copying solver: make copies of the state
- `words` is not compact in memory
- **Non-compactness problem for a copying solver**

Outline

1. Background

2. **Dynamically Compact Sparse Bit-Sets**
   - Compact Bit-Sets
   - Dynamically Compact Data-Structures

3. Sharing Tables

4. Evaluation

5. Conclusion

Outline

1. Background

2. **Dynamically Compact Sparse Bit-Sets**
   - Compact Bit-Sets
   - Dynamically Compact Data-Structures

3. Sharing Tables

4. Evaluation

5. Conclusion

UPPSALA UNIVERSITET

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Compact Bit-Sets

The operations we just watched:

Compact Bit-Sets

The operations we just watched:

$$
\begin{array}{l}
\textbf{for } i \leftarrow \texttt{limit} - 1 \textbf{ downto } 0 \textbf{ do} \\
\quad \texttt{words}[\texttt{index}[i]] \leftarrow_{\&} \texttt{mask}[\texttt{index}[i]] \\
\quad \textbf{if } \texttt{words}[\texttt{index}[i]] = 0 \textbf{ then} \\
\quad\quad \texttt{index}[i] \leftarrow \texttt{index}[\texttt{limit} - 1] \\
\quad\quad \texttt{limit} \leftarrow \texttt{limit} - 1 \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

Compact Bit-Sets

The operations we just watched:

```
for i ← limit − 1 downto 0 do
    words[index[i]] ←& mask[index[i]]
    if words[index[i]] = 0 then
        index[i] ← index[limit − 1]
        limit ← limit − 1
    end
end
```

Compact implementation:

```
for i ← limit − 1 downto 0 do
    words[i] ←& mask[i]
    if words[i] = 0 then
        index[i] ← index[limit − 1]
        words[i] ← words[limit − 1]
        limit ← limit − 1
    end
end
```

Example

Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0

words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	**1** 0 0 **1**
index	0	1	2	3

limit $= 4$

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Example
Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
				&
words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	**1** 0 0 **1**
index	0	1	2	3

limit $= 4$

Example
Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	0 0 0 0
index	0	1	2	3

limit = 3

Example
Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0

words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	
index	0	1	2	

limit $= 3$

Example

Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
			&	
words	**1** **1** 0 **1**	**1** 0 0 0	**1** 0 **1** **1**	
index	0	1	2	
limit	= 3			

Example

Compact Implementation

mask	1 0 1 0	0 0 1 0	0 1 1 1	0 0 1 0

words	1 1 0 1	1 0 0 0	0 0 1 1	
index	0	1	2	

limit $= 3$

UPPSALA UNIVERSITET

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Example
Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
		&		
words	**1** **1** 0 **1**	**1** 0 0 0	0 0 **1** **1**	
index	0	1	2	

limit $= 3$

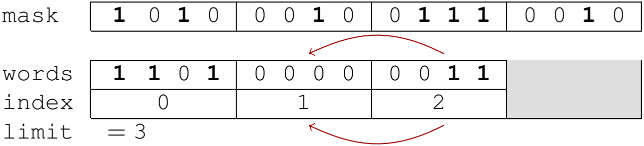
Example

Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
words	**1** **1** 0 **1**	0 0 0 0	0 0 **1** **1**	
index	0	1	2	

limit = 3

Example
Compact Implementation



- index[2] overwrites index[1], and
- words[2] overwrites words[1].

Example
Compact Implementation

mask	1 0 1 0	0 0 1 0	0 1 1 1	0 0 1 0
words	1 1 0 1	0 0 1 1		
index	0	2		

limit $= 2$

Example
Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0
	&			
words	**1** **1** 0 **1**	0 0 **1** **1**		
index	0	2		
limit	= 2			

Example

Compact Implementation

mask	**1** 0 **1** 0	0 0 **1** 0	0 **1** **1** **1**	0 0 **1** 0

words	**1** 0 0 0	0 0 **1** **1**		
index	0	2		

limit $= 2$

Example

Compact Implementation

	w_0	w_2		
words	**1** 0 0 0	0 0 **1** **1**		
index	0	2		
limit	= 2			

- Non-empty words are **contiguous** in memory

Example

Compact Implementation

	w_0	w_2		
`words`	**1** 0 0 0	0 0 **1** **1**		
`index`	0	2		

`limit` $= 2$

- Non-empty words are **contiguous** in memory
- Uses **less indirection** and has better **spatial locality**

Example
Compact Implementation

	w_0	w_2		
`words`	**1** 0 0 0	0 0 **1** **1**		
`index`	0	2		

`limit` = 2

- Non-empty words are **contiguous** in memory
- Uses **less indirection** and has better **spatial locality**
  - `words`[i] $\leftarrow_{\&}$ `mask`[i]
    instead of
    `words`[`index`[i]] $\leftarrow_{\&}$ `mask`[`index`[i]]

Example
Compact Implementation

	w_0	w_2		
words	**1** 0 0 0	0 0 **1** **1**		
index	0	2		
limit	= 2			

- Non-empty words are **contiguous** in memory
- Uses **less indirection** and has better **spatial locality**
  - `words`[i] $\leftarrow_{\&}$ `mask`[i]
    instead of
    `words`[`index`[i]] $\leftarrow_{\&}$ `mask`[`index`[i]]
- Trailing solvers can use the implementation (if elements are swapped)

Outline

1. Background

2. **Dynamically Compact Sparse Bit-Sets**
   Compact Bit-Sets
   Dynamically Compact Data-Structures

3. Sharing Tables

4. Evaluation

5. Conclusion

Dynamically Compact Data-Structures

- Small tables can be further compacted

Dynamically Compact Data-Structures

- Small tables can be further compacted
- **Specialised bit-sets** used when possible:
  - 16- or 8-bit integers instead of 32 for indexing
  - No indexing for sufficiently small tables

Dynamically Compact Data-Structures

- Small tables can be further compacted
- **Specialised bit-sets** used when possible:
  - 16- or 8-bit integers instead of 32 for indexing
  - No indexing for sufficiently small tables
- Best representation chosen **dynamically** during copying

UPPSALA UNIVERSITET

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Dynamically Compact Data-Structures

- Small tables can be further compacted
- **Specialised bit-sets** used when possible:
  - 16- or 8-bit integers instead of 32 for indexing
  - No indexing for sufficiently small tables
- Best representation chosen **dynamically** during copying
- Most copies created close to the leaves of the search tree, where many words are empty

Outline

1. Background

2. Dynamically Compact Sparse Bit-Sets
   Compact Bit-Sets
   Dynamically Compact Data-Structures

3. Sharing Tables

4. Evaluation

5. Conclusion

UPPSALA UNIVERSITET

Background

Dynamically Compact Sparse Bit-Sets

Compact Bit-Sets

Dynamically Compact Data-Structures

Sharing Tables

Evaluation

Conclusion

References

Sharing Tables

- Sharing is caring

Sharing Tables

- Sharing is caring
  - ...for memory usage

UPPSALA UNIVERSITET

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Sharing Tables

- Sharing is caring
  - ...for memory usage
  - ...for copying time

Sharing Tables

- Sharing is caring
  - ...for memory usage
  - ...for copying time
  - ...for cache performance

Sharing Tables

- Sharing is caring
  - ...for memory usage
  - ...for copying time
  - ...for cache performance
- Tuples and `supports` bit-sets are **shared:**

Sharing Tables

- Sharing is caring
  - ...for memory usage
  - ...for copying time
  - ...for cache performance
- Tuples and `supports` bit-sets are **shared:**
  1. Between a propagator and its **copies**

Sharing Tables

- Sharing is caring
  - ...for memory usage
  - ...for copying time
  - ...for cache performance
- Tuples and `supports` bit-sets are **shared:**
  1. Between a propagator and its **copies**
  2. Between **different propagators** reasoning on the same set of tuples

Sharing Tables

- Sharing is caring
  - ...for memory usage
  - ...for copying time
  - ...for cache performance
- Tuples and `supports` bit-sets are **shared:**
  1. Between a propagator and its **copies**
  2. Between **different propagators** reasoning on the same set of tuples
- `supports` are computed based on the tuples (domain-independent)

Sharing Tables

- Sharing is caring
  - ...for memory usage
  - ...for copying time
  - ...for cache performance
- Tuples and `supports` bit-sets are **shared:**
  1. Between a propagator and its **copies**
  2. Between **different propagators** reasoning on the same set of tuples
- `supports` are computed based on the tuples (domain-independent)
- Sharing `supports` not exploited in the original implementation [1]

Outline

1. Background

2. Dynamically Compact Sparse Bit-Sets
   Compact Bit-Sets
   Dynamically Compact Data-Structures

3. Sharing Tables

4. Evaluation

5. Conclusion

Evaluation Setup

- Standard benchmark set available at http://becool.info.ucl.ac.be/resources/positive-table-constraints-benchmarks
- 1 621 CSP instances (table constraints only), *min-domain* + *min-value* branching strategy
- Solvetime and peak memory usage on top of Gecode

(More detailed description provided on extra slide 22)

Results I

COMPACT	Compact bit-set
COMPACT++	Compact bit-set and compact indexing structure
HYBRID	COMPACT++ and drops indexing for #words ≤ 4

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Results I

COMPACT	Compact bit-set
COMPACT++	Compact bit-set and compact indexing structure
HYBRID	COMPACT++ and drops indexing for #words ≤ 4

Solvetime	COMPACT	COMPACT++	HYBRID
min			
mean			
max			
deviation			

Peak memory	COMPACT	COMPACT++	HYBRID
min			
mean			
max			
deviation			

Results I

COMPACT	Compact bit-set
COMPACT++	Compact bit-set and compact indexing structure
HYBRID	COMPACT++ and drops indexing for #words $\leq$ 4

Solvetime	COMPACT	COMPACT++	HYBRID
min	-67.1%		
mean	-14.4%		
max	0.4%		
deviation	$\pm30.8\%$		

Peak memory	COMPACT	COMPACT++	HYBRID
min	-27.2%		
mean	-4.5%		
max	0.2%		
deviation	$\pm8.5\%$		

Results I

COMPACT Compact bit-set

COMPACT++ Compact bit-set and compact indexing structure

HYBRID COMPACT++ and drops indexing for #words ≤ 4

Solvetime	COMPACT	COMPACT++	HYBRID
min	-67.1%	-66.4%	
mean	-14.4%	-13.7%	
max	0.4%	0.7%	
deviation	$\pm 30.8\%$	$\pm 29.7\%$	

Peak memory	COMPACT	COMPACT++	HYBRID
min	-27.2%	-33.4%	
mean	-4.5%	-6.8%	
max	0.2%	0.0%	
deviation	$\pm 8.5\%$	$\pm 11.4\%$	

Results I

COMPACT	Compact bit-set
COMPACT++	Compact bit-set and compact indexing structure
HYBRID	COMPACT++ and drops indexing for #words ≤ 4

Solvetime	COMPACT	COMPACT++	HYBRID
min	−67.1%	−66.4%	−66.3%
mean	−14.4%	−13.7%	−13.6%
max	0.4%	0.7%	0.9%
deviation	±30.8%	±29.7%	±29.6%

Peak memory	COMPACT	COMPACT++	HYBRID
min	−27.2%	−33.4%	−33.2%
mean	−4.5%	−6.8%	−7.5%
max	0.2%	0.0%	−0.3%
deviation	±8.5%	±11.4%	±11.4%

Results I

COMPACT Compact bit-set

COMPACT++ Compact bit-set and compact indexing structure

HYBRID COMPACT++ and drops indexing for #words ≤ 4

Solvetime	COMPACT	COMPACT++	HYBRID
min	−67.1%	−66.4%	−66.3%
mean	−14.4%	−13.7%	−13.6%
max	0.4%	0.7%	0.9%
deviation	±30.8%	±29.7%	±29.6%

Peak memory	COMPACT	COMPACT++	HYBRID
min	−27.2%	−33.4%	−33.2%
mean	−4.5%	−6.8%	−7.5%
max	0.2%	0.0%	−0.3%
deviation	±8.5%	±11.4%	±11.4%

- Miss rate of D1 cache decreases by $\approx 3\%$ on average

Results II

- **Sharing tables**:
  - Solvetime decreases by 4.6% and memory usage by 58.3% on average
  - Miss rate of D1 cache decreases by $\approx$ 18% on average

Background
Dynamically Compact Sparse Bit-Sets
Compact Bit-Sets
Dynamically Compact Data-Structures
Sharing Tables
Evaluation
Conclusion
References

Results II

- **Sharing tables**:
  - Solvetime decreases by 4.6% and memory usage by 58.3% on average
  - Miss rate of D1 cache decreases by $\approx$ 18% on average
- **Previous propagators in Gecode**:
  - Solvetime decreases by 85.7% and memory usage by 45.4% on average
  - Timed out on 85 additional instances

Results II

- **Sharing tables**:
  - Solvetime decreases by 4.6% and memory usage by 58.3% on average
  - Miss rate of D1 cache decreases by $\approx$ 18% on average

- **Previous propagators in Gecode**:
  - Solvetime decreases by 85.7% and memory usage by 45.4% on average
  - Timed out on 85 additional instances

- So called residual supports shown not to be beneficial

Outline

1. Background

2. Dynamically Compact Sparse Bit-Sets
   Compact Bit-Sets
   Dynamically Compact Data-Structures

3. Sharing Tables

4. Evaluation

5. Conclusion

Contributions:

- A **compact implementation** of sparse bit-sets
- Tables are **shared**
- Table constraints in Gecode are about a **magnitude faster** than before and use **half the memory**
- Potential benefit for trailing solvers

Future work:

- Exact variable deltas might speed up propagation
- Re-ordering tuples
- Extensions of compact-table

UPPSALA UNIVERSITET

Background

Dynamically Compact Sparse Bit-Sets

Compact Bit-Sets

Dynamically Compact Data-Structures

Sharing Tables

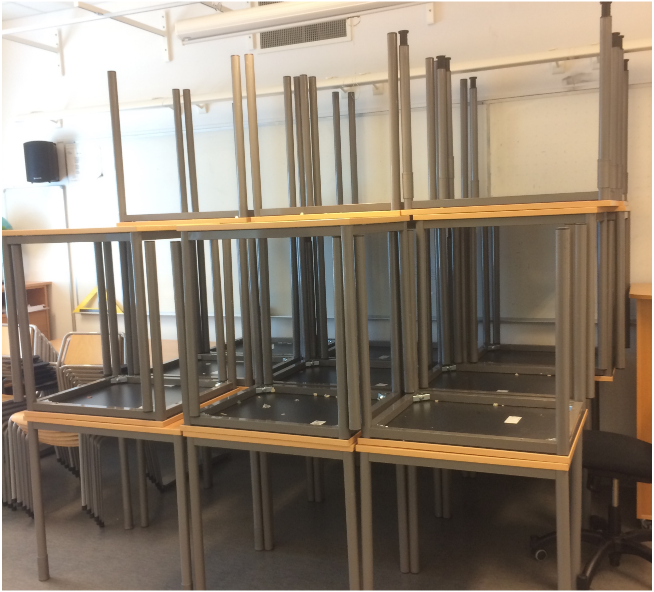
Evaluation

Conclusion

References

- J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J. Régin and P. Schaus, ‘Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets’, in *Proceedings of CP 2016*, pp. 207–223.
- H. Verhaeghe, C. Lecoutre and P. Schaus, ‘Extending compact-table to negative and short tables’, in *AAAI*, 2017, pp. 3951–3957.
- H. Verhaeghe, C. Lecoutre, Y. Deville and P. Schaus, ‘Extending compact-table to basic smart tables’, in *International conference on principles and practice of constraint programming*, Springer, 2017, pp. 297–307.
- N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck and G. Tack, ‘Minizinc: Towards a standard CP modelling language’, in *Proceedings of CP 2007*, 2007, pp. 529–543.

Compact Tables



Detailed Evaluation Setup

- Translated into *MiniZinc* [4] using the tool `xcsp2mzn`, available at https://github.com/CP-Unibo/mzn2feat.
- We skip instances that
  1. cannot be translated to *MiniZinc* due to non-trivial parse errors (117 instances);
  2. require more than 8 GB of RAM (43 instances);
  3. cannot be solved within the time out for the ORIGINAL configuration (170 instances); or
  4. are solved in less than 1 second for the ORIGINAL configuration (1014 instances).

  In total, 277 instances are evaluated.
- Solvetime does not include parsing *FlatZinc*
- Cache analysis uses Cachegrind