Making Compact-Table Compact

Linnea Ingmar[1] and Christian Schulte[2]

[1] Uppsala University, Sweden, linnea.ingmar.3244@student.uu.se
[2] KTH Royal Institute of Technology, Sweden, cschulte@kth.se

Abstract. The compact-table propagator for table constraints appears to be a strong candidate for inclusion into any constraint solver due to its efficiency and simplicity. However, successful integration into a constraint solver based on *copying* rather than trailing is not obvious: while the underlying bit-set data structure is *sparse* for efficiency it is not *compact* for memory, which is essential for a copying solver.
The paper introduces techniques to make compact-table an excellent fit for a copying solver. The key is to make sparse bit-sets *dynamically compact* (only their essential parts occupy memory and their implementation is dynamically adapted during search) and tables *shared* (their read-only parts are shared among copies). Dynamically compact bit-sets reduce peak memory by 7.2% and runtime by 13.6% on average and by up to 66.3% and 33.2%. Shared tables even further reduce runtime and memory usage. The reduction in runtime exceeds the reduction in memory and a cache analysis indicates that our techniques might also be beneficial for trailing solvers. The proposed implementation has replaced Gecode's original implementations as it runs on average almost an order of magnitude faster while using half the memory.

1 Introduction

The compact-table propagator [5] implements table constraints, where an explicit table of tuples defines the solutions to the constraint. Its basic idea is to assign a number to each tuple in the table and maintain a sparse bit-set where bit number i is set iff the tuple with number i is still considered a possible solution. The sparse bit-set is represented as an array of words (typically, words of 64 bits) and is *sparse*: operations performed on it by the propagator only consider words that have at least one bit set (that is, non-zero or non-empty words), where the emptiness information is tracked by an *index* structure. Compact-table and its extensions have already shown great potential [5,15,14] and sparse bit-sets have also been successfully used for itemset mining constraints [12].

The above-mentioned papers use constraint solvers that are based on *trailing* where changes during propagation and search are recorded and undone when backtracking occurs. Solvers based on *copying* create copies of the solver's state to which they can return during backtracking [13,10]. For a copying solver it is crucial that the state to be copied be small, so any of its propagators should require as little memory as possible to be copied. This paper contributes how this can be achieved for the compact-table propagator.

Operations on sparse bit-sets save time as they safely ignore empty words. However, empty words might still occupy memory as they are interleaved with non-empty words in memory. This does not matter for a trailing solver as the bit-set exists in one copy, however it poses a problem for a copying solver where the bit-set needs to be copied for each node of the search tree. We contribute how to make bit-sets *sparse as well as compact*: non-empty words move to the beginning of the word array and hence only its non-empty prefix needs copying. Additionally, compact bit-sets are more cache-friendly and require fewer indirections during update, which might be also beneficial for trailing solvers.

We make the propagator parametric with respect to its sparse bit-set implementation, so that we can get variants specialized for small tables. Here we take advantage by compressing the index structure or dropping it altogether. This optimization is *dynamic*: when the propagator is copied, the current size of its sparse bit-set decides which implementation is best for the copy. The rationale is that most copies are created close to the leaves of the search tree and hence many words of the bit-sets might be empty.

It is important that as much information as possible of the table that is read-only to the propagator be *shared* among its copies and among propagators using the same table for different constraints. We introduce a design where tables can be shared and only requires two mutable pointers per propagator variable.

The paper evaluates the various design decisions showing that the implementation of table constraints based on compact-table outperforms the original implementations in Gecode. We demonstrate that compactness is important, identify a promising hybrid candidate, and demonstrate that sharing is beneficial while residues (discussed below) are not beneficial in the context of Gecode.

Plan of the Paper. The next section reviews the compact-table propagator. Sect. 3 shows how tables can be shared between several propagators. Sect. 4 introduces techniques for dynamic compact sparse bit-sets which are evaluated in Sect. 5 and Sect. 6 concludes the paper.

2 Compact-Table

Throughout the paper we assume: the n tuples in the table t are numbered from 0 to $n-1$; the i-th tuple is denoted as t_i; the constraint (and hence t) has arity a; the value at position k ($1 \leq k \leq a$) of tuple t_i is denoted as $t_{i,k}$; the variables are $x_1, \ldots, x_a$ where the domain of variable x_k is $\mathrm{dom}(x_k)$; a tuple t_i is a *support* for a *variable-value* pair $\langle x_k, v \rangle$ and for a *position-value* pair $\langle k, v \rangle$ if $t_{i,k} = v$.

Sparse Bit-Sets. The n tuples are maintained in a sparse bit-set per propagator where bit number i is set iff the tuple t_i is still considered a possible solution. The sparse bit-set is an array `words` of words of 64 bits and is *sparse*: its operations only consider non-empty words, where the emptiness information is tracked by an *index* structure. The index structure is an array `index` of 32-bit words that maintain a permutation of the indices of `words` and a counter `limit` for the

current number of non-empty words. The first `limit` entries of `index` store the indices of `words` that are currently non-empty. If `limit` reaches zero the entire bit-set is empty. The index structure is key to sparseness: bit-set operations only need to consider words with indices in the first `limit` entries of `index`.

Modifications to the sparse bit-set are performed by intersections (word by word bit-wise *and*, denoted as &) with a temporary mask. If the word at index `index`$[i]$ in `words` becomes empty, then the index structure records this by swapping `index`$[i]$ with `index`$[$`limit` $- 1]$ in `index` and decrementing `limit` by one. By doing so, non-empty indices move to the front of `index` while their order in `words` remains unchanged. The following invariant is maintained, where w is the number of `words`: $\forall i \in \{0, \ldots, w-1\} : i <$ `limit` $\Leftrightarrow$ `words`$[$`index`$[i]] \neq 0$.

Support Bit-Sets. For each variable x_k $(1 \leq k \leq a)$ and value $v \in \mathrm{dom}(x_k)$ a *support* bit-set is constructed when the propagator is created, denoted by $\texttt{supports}_{\langle x_k, v\rangle}$. It captures the tuples in the table t that are supports for $\langle x_k, v\rangle$: bit i in the support bit-set is set iff $t_{i,k} = v$. The support bit-sets are used to update the sparse bit-set during the filtering phase of the algorithm (discussed below). Note that the support bit-sets are created for each propagator using the same table t with respect to the initial variable domains according to [5], a design that we are going to improve on in Sect. 3.

Update and Filtering. The sparse bit-set and the support bit-sets encode the information necessary to perform the two phases of the algorithm. The *update phase* zeroes the bits in the sparse bit-set that correspond to tuples that have lost support. The *filtering phase* removes values from variable domains that are no longer supported by any tuple.

An optimization in the filtering phase of the algorithm are *residual supports*: for each variable x_k and value $v \in \mathrm{dom}(x_k)$ the word index in the sparse bit-set for which a support for $\langle x_k, v\rangle$ was found is cached.

3 Sharing Tables

Sharing tables among propagators has two aspects: copies of a propagator created during search share tables and propagators using the same table for different constraints (that is, for different variables) share it. As mentioned in Sect. 2, the latter case is not exploited in [5]. Sharing saves memory and increases spatial locality and is likely to improve cache performance. When the table is created, *admissible domains* and *supports* are computed, which are shared among all propagators and their copies using the table.

Admissible Domains. For each position k $(1 \leq k \leq a)$ the *admissible domain* is computed as the set of values $d_k = \{t_{i,k} \mid 0 \leq i < n\}$ that occur in a tuple. The admissible domains only depend on the table and hence can be shared. When a propagator with variables $x_1 \ldots, x_a$ and table t is created, the variable domains are constrained to $d_k \cap \mathrm{dom}(x_k)$.

Supports. The table data structure provides shared access to the support bit-sets $\texttt{supports}_{\langle k,v\rangle}$ for $v \in d_k$. Note the difference from the notation $\texttt{supports}_{\langle x_k,v\rangle}$ used in Sect. 2, as support bit-sets are based on domains of variables x_k in [5]. All support bit-sets for a position k are stored contiguously in a bit-set array such that $\texttt{supports}_{\langle k,v_2\rangle}$ is stored directly after $\texttt{supports}_{\langle k,v_1\rangle}$ if value v_2 is the next larger value than v_1 in the admissible domain d_k.

The table should provide constant-time operations to find $\texttt{supports}_{\langle k,v\rangle}$ for $v \in \mathrm{dom}(x_k)$ or $v \in \Delta_{x_k}$. Here Δ_{x_k} is the *delta* of x_k as the set of values that are removed from $\mathrm{dom}(x_k)$. This is important if deltas are accurate, as for [5], even though it is not discussed there. In [5] the domain implementation relies on sparse bit-sets, which provide cheap access to deltas [11]. Gecode provides only accurate delta information in case the lower or upper bound of a variable changes [7]. Hence, the operations required skip entire ranges of values. Our propagator maintains per variable x_k pointers to $\texttt{supports}_{\langle k,\min \mathrm{dom}(x_k)\rangle}$ and $\texttt{supports}_{\langle k,\max \mathrm{dom}(x_k)\rangle}$ which are adjusted by binary search when $\min \mathrm{dom}(x_k)$ or $\max \mathrm{dom}(x_k)$ change. In case no delta information is available, the corresponding support information is computed by simultaneously iterating over variable domains and the supports between the two pointers.

4 Dynamically Compact Sparse Bit-Sets

This section introduces techniques to make sparse bit-sets compact and index structures small that can be adapted dynamically during copying. As in [5], our implementation is a data structure that could also be used in other contexts.

Compact Bit-Sets. Our implementation makes sparse bit-sets compact such that their non-empty words form a contiguous block in memory.

Let us consider part of the algorithm's update phase. The sparse bit-set with `limit=4` is about to be updated with the shown mask (computed from the support information). For simplicity, we use 4-bit rather than 64-bit words. The update with the mask is performed by word-wise in-place intersection. After the update, `limit` is 2 and the words w_1 and w_3 are empty.

mask	1010	0010	0111	0010
	w_0	w_1	w_2	w_3
words	1101	1000	1011	1001
index	0	1	2	3

The original implementation, called ORIGINAL, executes the following instructions (simplified), where $x \leftarrow_\& y$ abbreviates $x \leftarrow x \,\&\, y$:

```
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
```

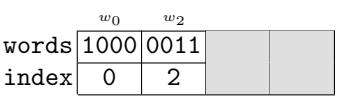
	w_0	w_1	w_2	
words	1000	0000	0011	
index	0	2		

The result is shown to the right, where dead entries (not to be copied) are

marked gray. When copying the resulting bit-set the word w_1 would be copied even though it is empty, as it is interleaved with w_0 and w_2.

Our compact implementation, called COMPACT, updates the data structures by executing the following instructions (simplified) leading to the result shown to the right of the instructions. Note that only w_0 and w_2 need copying.

```
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
```



The key benefit of COMPACT is that non-empty words are contiguous in memory as captured by the following invariant:

$$\forall i \in \{0, \ldots, \texttt{limit} - 1\} : \texttt{words}[i] \neq 0$$

As only non-empty words need copying, both memory usage and time for copying is reduced. The data structure is also more cache-friendly as it is contiguous. COMPACT uses less indirection and hence might be more efficient than ORIGINAL as the words are accessed directly and not through `index`. Even though the removal of an empty word now also requires an update to `words`, these updates are infrequent. Additional changes to the implementation are required, they are analogous. In particular, masks are constructed to be compact to match `words` directly without indirection.

Note that for a trailing solver, rather than *overwriting* $\texttt{index}[i]$ and $\texttt{words}[i]$ with $\texttt{index}[\texttt{limit} - 1]$ and $\texttt{words}[\texttt{limit} - 1]$, these entries would be swapped. Hence the idea of COMPACT is also compatible with a trailing solver.

During the filtering phase of the algorithm, the bit-set is intersected with support bit-sets which are not compact. The original implementation executes instructions of the following form:

$$\texttt{words}[\texttt{index}[i]] \mathbin{\&} \texttt{supports}_{\langle k,v \rangle}[\texttt{index}[i]]$$

while our compact implementation uses less indirection:

$$\texttt{words}[i] \mathbin{\&} \texttt{supports}_{\langle k,v \rangle}[\texttt{index}[i]]$$

for position-value pairs $\langle k, v \rangle$ and $0 \leq i < \texttt{limit}$.

Compressing the Index Structure. We save additional memory for the index structure. When possible, we use 16- or 8-bit data types for the entries in `index` instead of 32-bit; this optimization we refer to as COMPACT++. Consider a table with 16 384 tuples, which results in `words` and `index` with 256 entries each. Assuming two pointers for `words` and `index` with 64 bits each, COMPACT++ can use 8-bit entries and reduce memory usage from 3 092 to 2 321 bytes, a reduction by $\approx 25\%$ (ignoring memory layout requirements).

Two specialized implementations for sufficiently small tables with w words are as follows. $\textsc{Small}_w$ uses 4-bit index entries, that is up to 15 entries and the `limit` field are packed into a single 64-bit word. $\textsc{Dense}_w$ drops the entire index structure and considers all words in the bit-set.

Dynamic Data Structures. Making the propagator parametric with respect to its sparse bit-set implementation gives opportunity for further optimization. The decision which implementation to use is made *statically* when the propagator is created or *dynamically* when the propagator is copied. For implementations where only static decisions are made, we use "S" as subscript; for implementations also making dynamic decisions we use "D". For example, $\textsc{Compact}^{++}_{\mathrm{S}}$ may decide to use 16-bit integers initially and all of its copies also use 16-bit integers, while $\textsc{Compact}^{++}_{\mathrm{D}}$ might create copies using 8-bit integers if possible.

5 Evaluation

We evaluate our implementation of compact-table and the various optimizations on top of Gecode (Version 6.0.0), on the same benchmark set as in [5], involving 1 621 table instances. Being originally in format XCSP 2.1, which is not supported by Gecode, the benchmarks are translated into *MiniZinc* [9] using the tool `xcsp2mzn`[3]. Time measurements are run on a Windows 7 (64 bit) computer with two four-core Intel Xeon E5462 of 2.80 GHz and 8 GB RAM. Measurements of memory usage are run on a server cluster. A time out of 1 000 seconds is used on each instance. We skip instances that i) cannot be translated to *MiniZinc* due to parse errors (117 instances); ii) require more than 8 GB of RAM (43 instances); iii) cannot be solved within the time out for the Original configuration (170 instances); or iv) are solved in less than 1 second for the Original configuration (1014 instances). In total, 277 instances are evaluated.

Table 1 shows the relative performance of various implementations, using Original as baseline. No implementation except Residues uses residual supports. We report the minimum and maximum relative performance, the geometric mean of the relative performance, as well as the geometric standard deviation of the relative performance. Solvetime means wall time excluding the time for parsing *FlatZinc* and peak heap memory usage also excludes the peak memory for parsing. For presenting the relation of implementation a to a baseline implementation b (Original in Table 1), we use the relative measures $100 \cdot \frac{a_i^{\mathrm{T}}}{b_i^{\mathrm{T}}}$ and $100 \cdot \frac{a_i^{\mathrm{M}}}{b_i^{\mathrm{M}}}$, where c_i^{T} and c_i^{M} are solvetime respectively peak memory usage for implementation c on instance i. Detailed results are available from the authors.

As shown in Table 1, compressing the bit-set (Compact) improves runtime by 14.4% and peak memory usage by 4.5% on average compared to the original implementation (Original). The decrease in runtime is most likely achieved by a combination of i) fewer operations for copying, ii) less indirection as argued in Sect. 4, and iii) better cache performance due to the more compact

[3] Available at `https://github.com/CP-Unibo/mzn2feat`, last accessed April 17, 2018.

Table 1. Solvetime and peak memory relative to ORIGINAL.

Solvetime	COMPACT	COMPACT$^{++}_{S}$	COMPACT$^{++}_{D}$	BEST$_{S}$	BEST$_{D}$	RESIDUES
min	−67.1%	−66.8%	−66.4%	−66.7%	−66.3%	−8.5%
mean	−14.4%	−14.4%	−13.7%	−14.6%	−13.6%	13.1%
max	0.4%	1.1%	0.7%	2.2%	0.9%	32.4%
deviation	±30.8%	±31.1%	±29.7%	±31.0%	±29.6%	±8.3%

Peak memory	COMPACT	COMPACT$^{++}_{S}$	COMPACT$^{++}_{D}$	BEST$_{S}$	BEST$_{D}$	RESIDUES
min	−27.2%	−33.4%	−33.4%	−33.4%	−33.2%	−0.0%
mean	−4.5%	−6.8%	−6.8%	−7.2%	−7.2%	10.0%
max	0.2%	0.0%	0.0%	−0.3%	−0.3%	71.2%
deviation	±8.5%	±11.4%	±11.4%	±11.2%	±11.2%	±13.1%

representation. Additional analysis with the cache profiling tool *Cachegrind* [8] on instances with solvetime of more than 10 seconds (using a time out of one hour), indicates that COMPACT reduces the miss rate of the first-level data cache by $\approx 3\%$ on average compared to ORIGINAL.

Compressing the bit-set as well as the index structure (COMPACT^{++}), further reduces peak memory usage to -6.8% on average. Comparing COMPACT$^{++}_{S}$ and COMPACT$^{++}_{D}$, there is no visible difference in peak memory usage, while the runtime is slightly higher for COMPACT$^{++}_{D}$ due to extra overhead during copying.

Our proposed winning strategy, BEST, is the combination of COMPACT^{++} and DENSE$_4$. This strategy has the best memory usage among all evaluated implementations by a very modest $\approx 0.5\%$ in average while being only marginally slower in average. The winning strategy is chosen from evaluating the specialized implementations SMALL$_w$ for $w \in \{1, 2, 4, 8, 15\}$ and DENSE$_w$ for $w \in \{1, 2, 4, 8, 16\}$ respectively, whose results we omit for lack of space. Overall, these variants perform similarly to each other for all w, though DENSE tends to be slightly faster than SMALL. For simplicity, we use only one of DENSE and SMALL, with DENSE being the winner of the two. The threshold value $w = 4$ is chosen as larger values for w yield slightly higher maximum memory usage.

In Table 1, RESIDUES denotes an implementation that is like ORIGINAL except that it uses residues, discussed in Sect. 2. Clearly, computation saved during propagation by residues do not compensate for the memory and copying overhead they incur, as both runtime and memory usage is increased.

To evaluate the impact of sharing tables between propagators, as discussed in Sect. 3, we extend Gecode's *FlatZinc* interpreter so that propagators that use the same set of tuples can share the corresponding tables. Measurements are made with and without sharing for BEST$_{D}$, and on average runtime is reduced by 4.6% and memory usage by 56.5% when using sharing.

Note that the comparison is slightly approximate in the sense that it does not only reflect sharing of the admissible domains and the support bit-sets, but

also the actual tuples, as the entire tables are duplicated in the version without sharing (even though the actual tuples are never used during propagation). That being said, as the runtime measurements do not include *FlatZinc* overhead, during which the tables are created, and as tables are not copied, the runtime measurements are likely to reflect actual difference from sharing admissible domains and support bit-sets. Analysis with *Cachegrind* indicates that sharing tables reduces the miss rate of the first-level data cache by $\approx 18\%$ on average on sufficiently time-consuming instances.

We compare the performance of BEST_D with the two original implementations of table constraints in Gecode, based on [3] and [4]. The comparison is without sharing tables in *FlatZinc* as this is not available for the two implementations. Our implementation reduces runtime on average by 85.7% and up to 99.6% and reduces peak memory on average by 45.4% and up to 67.7% compared to the best of Gecode's two implementations. Note that these numbers are based on fewer instances than the other studied configurations, as the best old implementation timed out on 43 additional instances.

6 Conclusions and Future Work

This paper shows that compact-table is an excellent fit for a copying solver; the algorithm runs on average almost an order of magnitude faster than Gecode's implementations while using only half the memory. Our proposed implementation of compact-table is more compact in memory than the original implementation, reducing peak memory usage by 7.2% and solvetime by 13.6% on average on our benchmark set. We introduce how to share tables among propagators and demonstrate that sharing moderately reduces solvetime by 4.6% and considerably reduces memory usage by 56.5%. A trailing solver can most likely benefit from our optimizations as well: support bit-sets can be shared; and it might be beneficial to make sparse bit-sets compact as it is better for cache performance and uses less indirection.

Future Work. Our implementation only uses approximate information about variable deltas; more accurate information maintained by the propagator might speed up propagation. Heuristics for re-ordering the tuples in the table have not been explored. Re-ordering can have an impact on how much the sparse bit-sets can be compressed, as the order in which the tuples appear in the table decides which bits become zero first during propagation.

Acknowledgments. We are grateful for numerous comments and assistance from Mats Carlsson and Roberto Castañeda Lozano. Part of the work has been carried out in the first author's bachelor thesis [6] and during her student internship at KTH. We are grateful for the helpful comments from the anonymous reviewers.

References

1. Beck, J.C. (ed.): International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 10416. Springer-Verlag, Melbourne, Australia (Aug 2017)
2. Bessière, C. (ed.): International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, vol. 4741. Springer-Verlag, Providence, RI, USA (Sep 2007)
3. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: Preliminary results. In: International Joint Conference on Artificial Intelligence (IJCAI). vol. 1, pp. 398–404. Nagoya, Japan (Aug 1997)
4. Bessière, C., Régin, J.C., Yap, R.H.C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. Artificial Intelligence **165**(2), 165–185 (2005)
5. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J., Schaus, P.: Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In: Rueher, M. (ed.) International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 9892, pp. 207–223. Springer-Verlag, Toulouse, France (Sep 2016)
6. Ingmar, L.: Implementation and Evaluation of a Compact-Table Propagator in Gecode. Bachelor thesis, Department of Information Technology, Uppsala University, Sweden (August 2017), available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-328679`
7. Lagerkvist, M.Z., Schulte, C.: Advisors for incremental propagation. In: Bessière [2], pp. 409–422
8. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Ferrante, J., McKinley, K.S. (eds.) Conference on Programming Language Design and Implementation (PLDI). pp. 89–100. ACM, San Diego, CA, USA (2007)
9. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessière [2], pp. 529–543
10. Reischuk, R.M., Schulte, C., Stuckey, P.J., Tack, G.: Maintaining state in propagation solvers. In: Gent, I. (ed.) International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 5732, pp. 692–706. Springer-Verlag, Lisbon, Portugal (Sep 2009)
11. de Saint-Marcq, V.l.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS). pp. 1–10. Uppsala, Sweden (Sep 2013)
12. Schaus, P., Aoga, J.O.R., Guns, T.: Coversize: A global constraint for frequency-based itemset mining. In: Beck [1], pp. 529–546
13. Schulte, C.: Comparing trailing and copying for constraint programming. In: De Schreye, D. (ed.) International Conference on Logic Programming. pp. 275–289. The MIT Press, Las Cruces, NM, USA (Nov 1999)
14. Verhaeghe, H., Lecoutre, C., Deville, Y., Schaus, P.: Extending compact-table to basic smart tables. In: Beck [1], pp. 297–307
15. Verhaeghe, H., Lecoutre, C., Schaus, P.: Extending compact-table to negative and short tables. In: Singh, S.P., Markovitch, S. (eds.) AAAI Conference on Artificial Intelligence. pp. 3951–3957. San Francisco, CA, USA (Feb 2017)