An Optimal Iterative Algorithm for Extracting MUCs in a Black-box Constraint Network

Philippe Laborie[1]

Abstract. We present a non-intrusive iterative algorithm for extracting Minimal Unsatisfiable Cores in black-box constraint networks. The problem can be generalized as the one of finding a minimal subset satisfying an upward-closed property $\mathcal{P}$. The proposed algorithm, ADEL, integrates several ingredients (Acceleration, Dichotomy, Estimation of distance between consecutive elements, Lazy checks) that are motivated by a probabilistic study on the position of elements in the selected subset. Considering the number of infeasibility property checks we show that the proposed approach is optimal both for small and for large MUCs and that it outperforms existing approaches.

1 Introduction

When a Constraint Satisfaction Problem is infeasible, providing a minimal explanation for infeasibility in the form of a minimal subset of constraints that are mutually contradictory (known as Minimal Unsatisfiable Core or MUC) helps identifying the causes of the infeasibility [2]. Many existing approaches for MUC extraction exploit the particular features of the problem or the resolution engine. For instance techniques such as *elastic filters* can be used for Linear Programming models [1], *explanations* recording [6] for Constraint Programming models or *no-goods* learning for Boolean Satisfiability models [7].

In this paper, we tackle the problem of MUC extraction in a *non-intrusive* way with respect to the resolution engine by considering that checking for the (in)feasibility of a subset of constraints is a *black-box* operation. Advances of the state-of-the-art in Optimization result in increasingly sophisticated and efficient engines. Engines sophistication makes it harder to implement intrusive methods to compute MUCs whereas increase of engines efficiency makes infeasibility property check faster. Both aspects tend to make the black-box approach attractive.

The problem studied in this paper is thus more general than MUC extraction and can be defined as follows: given a finite set U and a property $\mathcal{P}$ on the subsets of U that is upward-closed (that is, whenever it holds for a subset X it also holds for any superset of X), find a *minimal subset* of U that satisfies property $\mathcal{P}$. For MUC extraction the property $\mathcal{P}(X)$ is the *infeasibility* of a particular subset of constraints X, it clearly is upward-closed as the superset of any infeasible subset of constraint is infeasible too. Beside MUC extraction, this problem occurs in several other fields for instance in diagnosis (find a minimal subset of faulty components of a system that explains the current observations) or in non-monotonic logics (find a minimal subset of abnormalities in the clauses that restore admissibility). In general checking property $\mathcal{P}(X)$ for a particular subset is an expensive operation: it is NP-complete in the case of MUC extraction, for other applications it could for instance be the outcome of a complex simulation process based on a set of input events X. The complexity of the approach is measured by the number of property checks performed by the algorithm.

The work closest to ours is probably the `QuickXplain` recursive algorithm [5] developed in the context of the extraction of MUCs for constraint programming using a black-box approach and the iterative dichotomy algorithm `DC` proposed in [4].

After introducing some notations and formally defining the problem, the paper presents some probabilistic results about the relative positions of elements of the selected minimal subset. This study highlights a couple of interesting properties that are exploited in the sequel of the paper to introduce step by step the proposed algorithm `ADEL` (standing for the four ingredients of the approach: Dichotomy, Acceleration, Estimation and Lazy checks). We show in the last section that the complexity of this algorithm is optimal both for small and large minimal subsets and that it outperforms both the `QuickXplain` and the `DC` algorithm. Proofs of properties are not provided in the article for space reason.

2 Problem Definition and Notations

Let U be a finite set of cardinality n and $\mathcal{P}$ an *upward-closed* property on its powerset 2^U that is, a property such that:

$$\big(X \subseteq Y \subseteq U\big) \wedge \mathcal{P}(X) \Rightarrow \mathcal{P}(Y)$$

Definition 1 (Minimal Subset) *A subset $X \subseteq U$ is said to be* **minimal** *if and only if: $\mathcal{P}(X) \wedge \forall Y \subset X : \neg\mathcal{P}(Y)$. Let $\mathcal{M}$ denote the set of all minimal subsets of U.*

In the sequel of this paper we assume $\mathcal{P}(U)$, that is there exist at least one minimal subset. Of course, in general there is not a unique minimal subset: there can be an exponential number of such subsets, for instance the set of minimal subsets could be all subsets of cardinality $n/2$ which is lower bounded by $2^{n/2}$.

Without loss of generality we assume a total order $\prec$ over the elements in U so that the elements can be indexed: $U = (u_1, ..., u_n)$ with $u_i \prec u_{i+1}$.

The total order $\prec$ over U implies a lexicographic total order $\prec$ over the powerset 2^U. Let $X, Y \subseteq U, X \neq Y$:

$$X \prec Y \Leftrightarrow \exists j \in [1,n] : \begin{cases} \forall i < j, (u_i \in X) \Leftrightarrow (u_i \in Y) \\ u_j \in X \\ u_j \notin Y \end{cases}$$

We denote X^* the unique minimal subset $X^* \in \mathcal{M}$ that is maximal with respect to the total order $\prec$.

[1] IBM Software Group, France, email: laborie@fr.ibm.com

The problem studied in this paper is the design of efficient algorithms to compute minimal subset X^* without any knowledge about property $\mathcal{P}$ beside the assumption that it is upward-closed. We estimate the complexity of an algorithm to compute a minimal subset as the number of property checks it performs. Note that for comparing the different algorithms, we sometimes use in this paper a more fine-grain measure than the traditional *big O* comparison because we count the number of property checks which is an homogeneous measure for all the approaches. That's why we use a comparison *on the order of*: $f(n) \sim g(n)$ meaning $\lim_{n\to+\infty} \big(f(n)/g(n)\big) = 1$. This allows constant factors to be taken into account.

Note that in practice, relation $\prec$ can be used to express preferences over minimal subsets, in this case, the algorithms presented in this paper will compute a preferred minimal subset. The definition of relation $\prec$ is the same as in [5].

Let $X \subseteq U$ and $m = |X|$. Without loss of generality, we can sort the m elements of X by their index in U: $X = \{u_{\pi(X,j)}\}_{j\in[1,m]}$ with $1 \leq \pi(X,j) < \pi(X,j+1) \leq n$. Integer $\pi(X,j)$ denotes the position in U of the j^{th} element of subset X.

Figure 1 illustrates a set U, its minimal subsets for a property $\mathcal{P}$ ordered by relation $\prec$, subset X^* and the distances between consecutive elements in X^*.

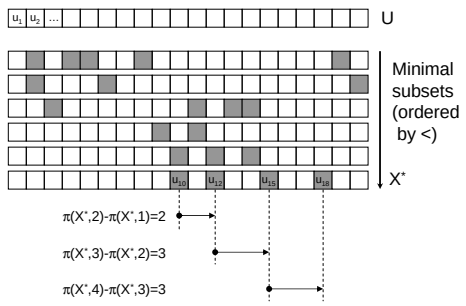

Figure 1. Minimal subsets

For $1 \leq i \leq n$, the subset $\{u_j | i \leq j \leq n\}$ of all u_j with index greater than i is denoted $U_{i\rightarrow}$.

3 Probabilistic study

In this section, we provide some analytical and experimental results about the distribution of the distances $\pi(X^*, j+1) - \pi(X^*, j)$ between consecutive elements of minimal subset X^* in U under some assumptions. The main conclusions of this study will be used to design efficient algorithms for computing X^*.

3.1 Unique minimal subset

We assume U contains a unique minimal subset of size m with $0 < m \leq n$ uniformly distributed in U.

Property 1 gives the probability distributions of the distances between consecutive elements of the unique minimal subset in U.

Property 1 *In the case of a unique minimal subset of size m, the probability that the distance $\pi(X^*, j+1) - \pi(X^*, j)$ between two consecutive elements is k does not depend on j and is equal to*

$$p(k) = \binom{n-k}{m-1} \Big/ \binom{n}{m}$$

From the formula, probability $p(k)$ can be computed recursively as follows:

- $p(1) = m/n$
- $\forall k \in [2, n-m+1], p(k+1) = \left(1 - \frac{m-1}{n-k}\right) p(k)$
- $\forall k > n-m+1, p(k) = 0$

The probability distribution $p(k)$ is illustrated on figure 2 for $n = 20$ and $m = 4$. This probability exponentially decreases with k. This result can be seen as a discrete version of the probability density of the distance x between two neighbor points from a set of points randomly and uniformly spread on a line which is $p(x) = \frac{1}{d}\, e^{-x/d}$ where d is the mean distance between two neighbor points [3].

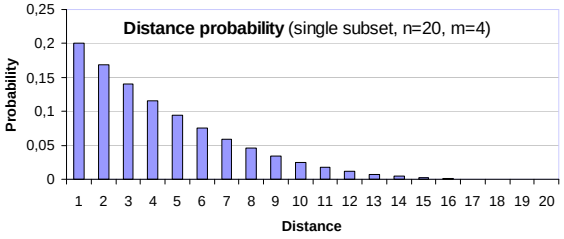

Figure 2. Distance probability between two consecutive elements

3.2 Disjoint minimal subsets of random size

In this section we assume U contains c uniformly distributed disjoint minimal subsets of random sizes m_i, $i \in [1, c]$ with $0 < \sum_{i\in[1,c]} m_i \leq n$. The following property holds, independently from the probability distribution of the minimal subset sizes m_i:

Property 2 *In the case of disjoint minimal subsets, the probability that the distance $\pi(X^*, j+1) - \pi(X^*, j)$ between two consecutive elements is k does not depend on j and is a non-increasing function of k.*

3.3 Random minimal subsets of random size

In this section, we experimentally study the distributions of random minimal subsets of random size. For each sample, a set of c subsets of size uniformly selected in $[1, n]$ is generated and the non-minimal subsets are eliminated.

For $n = 20$ and $c = 15$, Figure 3 shows the probability distribution of (1) the size of all generated minimal subsets and (2) the size of the selected minimal subset X^*. As we could expect, selected subsets X^* are usually smaller than the average size of all minimal subsets. This is because large minimal subsets are in general lower than small minimal subsets in the sense of total order $\prec$.

For the same configuration and only considering the samples for which the selected minimal subset X^* is of size 4, Figure 4 displays the probability distributions of the distances between consecutive elements in X^*. Here, unlike for disjoint subsets, we see that these distributions slightly depend on which consecutive elements of X^*

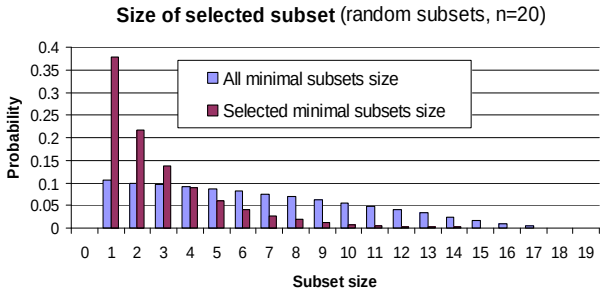

Figure 3. Minimal subset sizes

are considered. Elements of X^* tend to be slightly more concentrated in the end. It is to be noted that, here again, all those probability distributions are non-increasing functions of the distance.

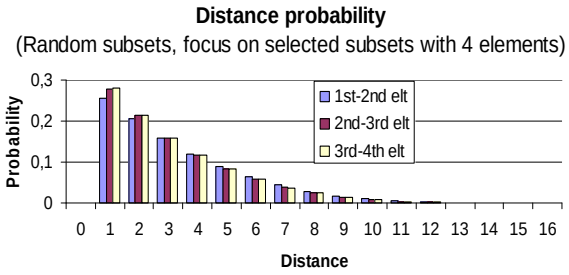

Figure 4. Distance probability between two consecutive elements

3.4 Conclusion of the probabilistic study

We can draw two conclusions from this short study on uniformly distributed minimal subsets:

1. The probability distribution of the distance between consecutive elements of the selected minimal subset X^* is in general a non-increasing function. This was formally shown in the case of disjoint minimal sets. Stated otherwise, the $j+1^{th}$ element of the selected minimal subset is likely to be close to the j^{th} element.
2. The probability distributions of the distance between consecutive elements of the selected minimal subset X^* do not heavily depend on which element is considered. In case of disjoint minimal subsets it can even be shown that these distributions are the same.

4 Algorithms

The input problem is given by:

- The finite set $U = (u_1, ..., u_n)$
- The upward-closed property $\mathcal{P}$

All the iterative algorithms studied in this paper share the same framework presented on Algorithm 1.

The input set U is stored as an array of size n where $U[i] = u_i, i \in \{1, ..., n\}$.

At line 1, the array U is shuffled, this will rule out any particular structure in the set U and, informally speaking, will ensure that

Algorithm 1 FindMinimalSubset(U, $\mathcal{P}$)

Require: $\mathcal{P}(U)$
1: $\text{Shuffle}(U)$ ▷ Called once: $O(n)$
2: $X \leftarrow \emptyset$ ▷ X: minimal subset under construction
3: $i \leftarrow 0$ ▷ i: index of last element added to X
4: **repeat**
5: $\quad i \leftarrow \text{FindNext}(X, U, i+1, \mathcal{P})$
6: $\quad X \leftarrow X \cup \{U[i]\}$
7: **until** $(i = n) \vee \mathcal{P}(X)$
8: **return** X

minimal subsets are uniformly distributed in U so that the main conclusions of section 3 can be exploited. This shuffling defines a total order $\prec$ among the elements of U. Procedure $\text{FindNext}(X, U, i, \mathcal{P})$ is in charge of finding and returning the largest index j ($i \leq j \leq n$) such that $\mathcal{P}(X \cup U_{j\rightarrow})$. The algorithms described in this paper differ according to their implementation of this procedure.

Figure 5 illustrates a set U, its minimal subsets for property $\mathcal{P}$ ordered by relation $\prec$ and the subsets X and $U_{i\rightarrow}$ at the iteration of algorithm 1 when $U[i]$ is added to the current minimal subset X (line 6 of Algorithm 1).

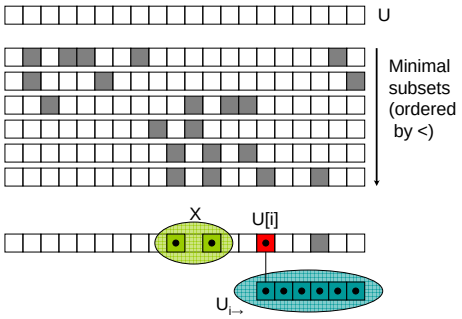

Figure 5. Minimal subsets

Property 3 *Algorithm 1 is correct and returns the unique minimal subset $X^* \in \mathcal{M}$ that is maximal with respect to the total order $\prec$.*

4.1 Naive Algorithm

The naive algorithm simply iterates over all elements $u_j, i < j$ until the first j such that $\neg\mathcal{P}(X \cup U_{j\rightarrow})$ in order to return index $j-1$ (See Algorithm 2). This algorithm is clearly a correct implementation for procedure FindNext.

Algorithm 2 FindNext-Naive(X, U, i, $\mathcal{P}$)

Require: $\mathcal{P}(X \cup U_{i\rightarrow})$
1: $j \leftarrow i$
2: **repeat**
3: $\quad j \leftarrow j+1$
4: **until** $(j = n+1) \vee \neg\mathcal{P}(X \cup U_{j\rightarrow})$
5: **return** $j-1$

Property 4 *If $\pi(X^*, |X^*|) < n$, the naive algorithm performs $\pi(X^*, |X^*|) + |X^*|$ property checks. Otherwise, if $\pi(X^*, |X^*|) = n$, the naive algorithm performs $n + |X^*| - 2$ property checks.*

4.2 Dichotomy Algorithm

The dichotomy algorithm corresponds to the DC algorithm proposed in [4]. It performs a dichotomic search, see Algorithm 3.

Algorithm 3 FindNext-D(X, U, i, $\mathcal{P}$)

Require: $\mathcal{P}(X \cup U_{i\rightarrow})$
1: $l \leftarrow i, r \leftarrow n$
2: **while** $l \neq r$ **do**
3: $m \leftarrow \lceil (l+r)/2 \rceil$
4: **if** $\mathcal{P}(X \cup U_{m\rightarrow})$ **then**
5: $l \leftarrow m$
6: **else**
7: $r \leftarrow m-1$
8: **end if**
9: **end while**
10: **return** l

The dichotomy algorithm is clearly a correct implementation for procedure FindNext.

Property 5 *In average, the dichotomy algorithm performs* $O(|X^*| \times \log_2(n))$ *property checks.*

If $|X^*| = 1$, the dichotomy algorithm performs $O(\log_2(n))$ checks which is clearly better than the $O(n)$ checks of the naive algorithm. At the other side of the spectrum if $|X^*| = n$, the dichotomy algorithm performs worse ($O(n \log_2(n))$) than the naive one ($O(n)$). The purpose of the algorithms described in the following sections is to take the best of both worlds.

4.3 AD (Accelerate & Dichotomize) Algorithm

The idea of the basic *accelerate & dichotomize* algorithm is to exploit the second conclusion of the probabilistic study that shows that the index j returned by procedure $\mathrm{FindNext}(X, U, i, \mathcal{P})$ is likely to be quite close to index i. This is of course especially true if the size of the selected minimal subset is large. So it may pay off to search for such an index j starting from index i with an acceleration phase (trying $i+1$, $i+2$, $i+4$, ..., $i+2^k$) until the first index such that $\neg\mathcal{P}(X \cup U_{i+2^k\rightarrow})$ and then applying a dichotomic search on the index segment $[i+2^{k-1}, i+2^k)$.

4.4 ADE (Accelerate, Dichotomize & Estimate) Algorithm

We can elaborate further on the previous algorithm by learning the initial step s of the acceleration phase from the previous calls to the FindNext procedure. This is supported by the second conclusion of the probabilistic study that shows that the probability distribution of the distances between consecutive elements of X^* does not depend much on which element is considered, thus this information can be estimated from the previous elements. In this context, the initial value of s represents the expected average distance between two successive elements of the minimal set. For the first call to FindNext, we take $s = n$ so, as $s > s_{max}$, this first call boils down to the pure dichotomy algorithm. For the later calls, the initial s is computed as the average of the distance between successive elements already added to the minimal set X under construction.

4.5 ADEL (Accelerate, Dichotomize, Estimate & Lazy checks) Algorithm

We finally propose a last improvement to the previous algorithm. Property checks on the subset X under construction in Algorithm 1, line 7 will be called m times for a minimal set of size m. If m is large and typically, getting close to n, this may represent a large proportion of the property checks of the algorithm. On the other side, it is not necessary to stop the algorithm as soon as one has proved the current subset X satisfies the property: one can let the function FindNext show that the current subset does not have to be extended. It will show it with approximately $\log_2(n)$ property checks in the acceleration step. So the idea is to consider that once the size of the current subset X is larger than $\log_2(n)$, as the effort to be spent for proving the property for X^* with function FindNext won't exceed the effort already spent checking the property for each elements added to X so far, we can stop checking the property in Algorithm 1, line 7 . This idea results in a *lazy* version of the algorithm denoted ADEL[2] and shown in Algorithms 4 and 5.

Algorithm 4 FindMinimalSubset(U, $\mathcal{P}$)

Require: $\mathcal{P}(U)$
1: Shuffle(U) ▷ Called once: $O(n)$
2: $X \leftarrow \emptyset$ ▷ X: minimal subset under construction
3: $i \leftarrow 0$ ▷ i: index of last element added to X
4: $s \leftarrow n, d_1 \leftarrow 0, d_0 \leftarrow 0$
5: **repeat**
6: $j \leftarrow$ FindNext($X, U, i+1, \mathcal{P}, s$)
7: $d_0 \leftarrow d_0 + 1$
8: $d_1 \leftarrow d_1 + j - i, s = \lfloor d_1/d_0 \rfloor$ ▷ Distance estimation
9: $i \leftarrow j$
10: $X \leftarrow X \cup \{U[i]\}$
11: **until** $(i = n) \vee (i \leq \log_2(n) \wedge \mathcal{P}(X))$ ▷ Lazy checks
12: **return** X

Algorithm 5 FindNext(X, U, i, $\mathcal{P}$,s)

Require: $\mathcal{P}(X \cup U_{i\rightarrow})$
1: $l \leftarrow i, r \leftarrow n$
2: $s_{max} \leftarrow \lfloor (n-i)/2 \rfloor$ ▷ Maximal step
3: **while** $(s \leq s_{max}) \wedge \mathcal{P}(X \cup U_{i+s\rightarrow})$ **do** ▷ Accelerate
4: $l \leftarrow i + s, s \leftarrow s * 2$
5: **end while**
6: **if** $s \leq s_{max}$ **then**
7: $r \leftarrow i + s - 1$
8: **end if**
9: **while** $l \neq r$ **do** ▷ Dichotomize
10: $m \leftarrow \lceil (l+r)/2 \rceil$
11: **if** $\mathcal{P}(X \cup U_{m\rightarrow})$ **then**
12: $l \leftarrow m$
13: **else**
14: $r \leftarrow m-1$
15: **end if**
16: **end while**
17: **return** l

[2] Note that the same idea can apply to algorithm DC.

5 Performance Study

In this section, we compare the performances of the algorithms described in previous section together with the QuickXPlain algorithm.

5.1 Unique minimal subset

We assume U, with $n = |U|$, contains a unique minimal subset of size m uniformly distributed in U.

The curves on Figures 6 and 8 show the performance of the different algorithms (number of property checks) as a function of the size of the minimal subset m and the size n of set U. Each point is computed as the average over 100 samples for the position of the minimal set in U. Note that both axis use a logarithmic scale. For the x-axis this is to give a special importance to small values of m.

Figure 6 shows for a fixed value of n ($n = 2048$) the average number of property checks performed by the different algorithms as a function of the size m of the unique minimal subset of U.

For the naive algorithm, for minimal subsets of unit size ($m = 1$), the average number of checks is roughly $n/2$ which is the average position of the element of the minimal set in U. For a minimal subset of size n, the naive algorithm has to perform $2\,n$ checks: n checks to verify that removing any element of U leads to a subset that do not check the property and n checks to verify whether the current subset under construction satisfies the property.

The dichotomy algorithm D roughly performs $m\,(1 + \log_2(n))$ checks which is linear with m for a fixed n. For $m = 1$ it performs $1+\log_2(n)$ checks and for $m = n$, it requires about $n\,(1+\log_2(n))$ checks.

Algorithm AD performs worse than D for small values of m. This is because when the elements of the selected minimal subset are sparse in U, the acceleration phase that starts with a unit step need to perform about $\log_2(n)$ iterations before to give the hand to the dichotomy phase that also will run in about $\log_2(n)$. For instance, for $m = 1$, it requires about $1 + 2\ \log_2(n)$ steps and AD is about twice worse as D. The situation is the opposite when m is getting closer to n: starting the acceleration phase with small values pays off and when $m = n$, the number of check of AD is $2\,n$ (same as the naive algorithm) which is a $n/\log_2(n)$ improvement factor over D.

As expected, algorithm ADE takes the best of both algorithms D and AD: by learning the initial acceleration step, it can achieve the same performance as D for small m and as AD for large m values.

The last improvement in algorithm ADEL pays off for large values of m by replacing the m checks in the FindNext functions by $\log_2(n)$ checks. For $m = n$, the number of checks is about $n+\log_2(n)$ where $\log_2(n)$ is the initial dichotomy to find the first element of the subset. So the improvement factor compared to ADE is about 2.

In fact, ADEL has *optimal complexity*[3] with n both for *small* minimal sets ($m = 1$) and *large* ones ($m = n$). When $m = 1$ it boils down to a binary search in $\log_2(n)$. When $m = n$, complexity is in $n + 2\ \log_2(n)$ and any algorithm need indeed to at least check that each of the n elements is member of the minimal subset by performing a check without it. The number of useless checks, due to the fact we do not know *a priori* that $m = n$, is only $2\ \log_2(n)$ and is negligible compared to n: $\log_2(n)$checks for identifying the first element with the initial binary search and the $\log_2(n)$ initial checks in Algorithm 4, line 11.

Figure 7 compares the number of property checks performed by algorithms ADEL and QuickXplain [5]. We see that ADEL consistently performs less checks than QuickXplain (between 10% and 50% less checks).

[3] In terms of the *order of* comparison: $f(n) \sim g(n)$.

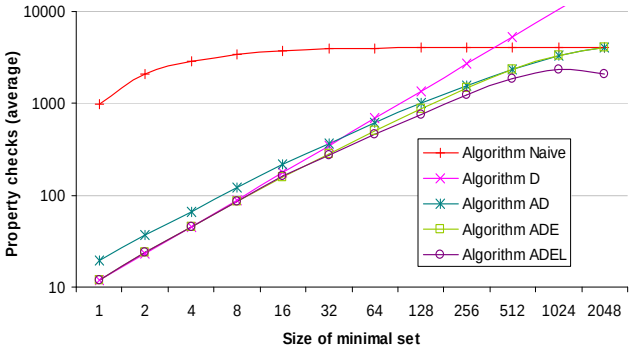

Figure 6. Number of property checks for $n = 2048$

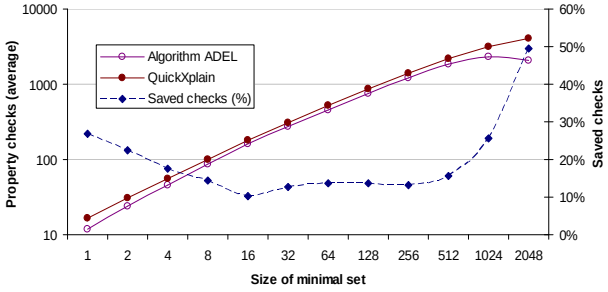

Figure 7. Number of property checks for $n = 2048$

Property 6 shows the average complexity for QuickXplain in case of a unique minimal subset of unit size. Note that this average complexity turns out to be the average value of the best and worst case complexities (resp. $\log_2(n)$ and $2\ \log_2(n)$) shown in [5]. Compared to the $\log_2(n)$ complexity of algorithm ADEL, this gives a 3/2 factor in favor of ADEL.

Property 6 *In case of a unique minimal set of size* 1*, the average complexity of the* QuickXplain *algorithm is* $3/2\ \log_2(n)$.

In case of a minimal subset of size n, QuickXplain need to perform $2\ n - 2$ property checks that is, the number of nodes of a binary tree with n leaves minus one because the top level node is not checked. Compared with the $n + 2\ \log_2(n)$ complexity of ADEL in this case this gives, for large values of n a factor 2 in favor of ADEL.

Figure 8 shows, for a fixed value of m ($m = 8$) the average number of property checks performed by the different algorithms as a function of the size n of U.

5.2 Random minimal subsets of random size

For each sample, a set of c subsets of size uniformly selected in $[1, n]$ is generated and the non-minimal subset are eliminated. Curve on Figure 9 shows the performance of the different algorithms (number of property checks) for $n = 2048$ as a function of the number of subsets c generated. When c grows, the size of the selected minimal

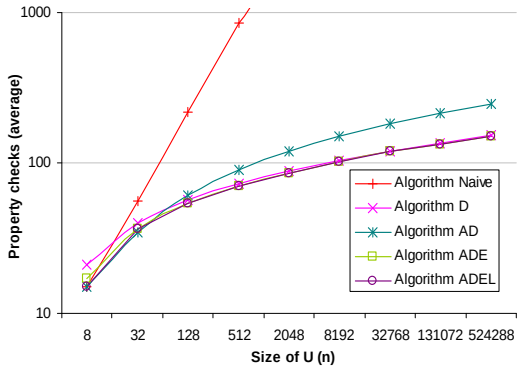

Figure 8. Number of property checks for $m = 8$

subset X^* tends to decrease. The average size of the selected subset is also shown on the figure as an indicator. We can make similar observations as in the previous section: algorithm `ADEL` outperforms all other algorithms.

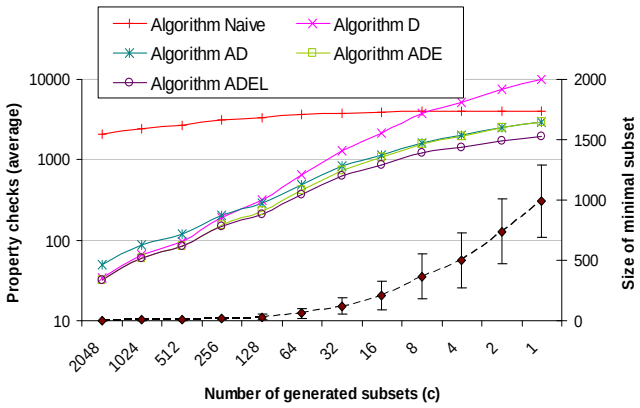

Figure 9. Number of property checks for $n = 2048$

The comparison with algorithm `QuickXplain` is shown on Figure 10. We see that `ADEL` consistently performs less checks than `QuickXplain` (between 10% and 30% less checks).

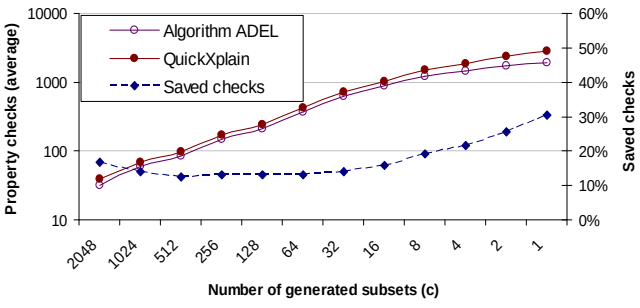

Figure 10. Number of property checks for $n = 2048$

6 Conclusion

This paper studies the problem of finding a minimal subset satisfying an upward-closed property. No assumption is made on the property being checked and the objective is to minimize the number of property checks. The approach can be applied to the extraction of MUCs in Constraint Networks. The proposed `ADEL` algorithm initially shuffles the set of elements and then exploits the probabilistic properties of the position of consecutive elements in the selected subset. We show that:

- it is *optimal* for small subsets, with a complexity in the order of $\log_2(n)$ for a unique minimal subset of size 1.
- it is *optimal* for large subsets, with a complexity in the order of n for a minimal subset of size n.
- the average number of checks behaves continuously in between those two extremal cases and outperforms all variants studied in this paper.
- in the case of a unique minimal set of size one, it performs in average ½ times less checks than the `QuickXplain` algorithm (33% less checks).
- in the case of a unique minimal set of size n, for large values of n, it performs about twice less checks than the `QuickXplain` algorithm (50% less checks).
- in the case of a unique minimal set of size n, for large values of n, it performs about $\log_2(n)$ less checks than the `DC` algorithm.
- in between those two extremal cases it consistently performs less property checks than the `QuickXplain` algorithm (between 10% and 50% less checks in the instances generated for our experiments).
- the algorithm is by nature iterative and can thus be easier to implement than a recursive algorithm like `QuickXplain`.

Algorithm `ADEL` is used in the implementation of the MUC extraction functionality (Conflict Refiner) in CP Optimizer since version 12.5.

REFERENCES

[1] John W. Chinneck, 'Finding a useful subset of constraints for analysis in an infeasible linear program', *INFORMS Journal on Computing*, **9**, 164–174, (1997).
[2] John W. Chinneck, *Feasibility and Infeasibility in Optimization: Algorithms and Computational Methods*, Springer, 2007.
[3] J.C. Demaret and A Gareet, 'Sum of exponential random variables', *Electronics and Communication (AE)*, **31**, 445–448, (1977).
[4] Fred Hemery, Christophe Lecoutre, Lakhdar Sais, and Frédéric Boussemart, 'Extracting mucs from constraint networks', in *Proc. 17th European Conference on Artificial Intelligence (ECAI'06)*, (2006).
[5] Ulrich Junker, 'QuickXplain: Preferred explanations and relaxations for over-constrained problems', in *Proc. AAAI-04*, (2004).
[6] Narendra Jussien and Olivier Lhomme, 'Local search with constraint propagation and conflict-based heuristics', *Artificial Intelligence*, **139**(1), 21–45, (July 2002).
[7] Joao P. Marques Silva and Karem A. Sakallah, 'Conflict analysis in search algorithms for satisfiability', in *Proceedings of the 8th International Conference on Tools with Artificial Intelligence (ICTAI'96)*, (1996).