1-11-2003

Reducing redundancy in the hypertree decomposition scheme

Peter Harvey
pah06@uow.edu.au

Aditya K. Ghose
University of Wollongong, aditya@uow.edu.au

Reducing redundancy in the hypertree decomposition scheme

Abstract

Hypertree decomposition is a powerful technique for transforming near-acyclic CSPs into acyclic CSPs. Acyclic CSPs have efficient, polynomial time solving techniques, and so these conversions are of interest to the constraints community. We present here an improvement on the opt-k-decomp algorithm for finding an optimal hypertree decomposition.

Keywords

artificial intelligence, computational complexity, constraint handling, constraint theory, optimisation, problem solving, trees (mathematics)

Disciplines

Physical Sciences and Mathematics

Publication Details

This paper originally appeared as: Harvey, P and Ghose, A, Reducing redundancy in the hypertree decomposition scheme, Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence, 3-5 November 2003, 474-481. Copyright IEEE 2003.

Reducing Redundancy in the Hypertree Decomposition Scheme*

Peter Harvey
Decision Systems Laboratory
School of IT and Computer Science
University of Wollongong, Australia
pah06@uow.edu.au

Aditya Ghose
Decision Systems Laboratory
School of IT and Computer Science
University of Wollongong, Australia
aditya_ghose@uow.edu.au

Abstract

Hypertree decomposition is a powerful technique for transforming near-acyclic CSPs into acyclic CSPs. Acyclic CSPs have efficient, polynomial time solving techniques, and so these conversions are of interest to the constraints community. We present here an improvement on the opt-k-decomp algorithm for finding an optimal hypertree decomposition.

1. Introduction

The area of constraint programming has been an AI success story, both in terms of industry applications and fundamental advances. A key problem in many applications of constraint programming has been the size of the problem to be solved. This paper makes advances in addressing this problem by improving on the hypertree decomposition approach to solving constraint networks [5, 6].

A constraint satisfaction problem is usually represented as a tuple $\langle V, C, D \rangle$ where V is a set of variables, C is a set of constraints, and D is a set of values (the domain). To solve a CSP, we attempt to find an assignment of values to variables subject to the constraints. Each constraint limits the allowable combinations of values which may be assigned to variables. A **solution** of a CSP is thus a complete assignment of values to variables which satisfies each of the constraints.

For a constraint in C we define its **scope** to be the set of variables which it constrains. If a constraint's scope consists of only two variables it is called **binary**. The **primal graph** of a CSP is defined as a graph where the nodes correspond to the variables and an edge exists between two nodes iff there exists a constraint between the corresponding variables. The **hypergraph** of a CSP is defined as a tuple $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ with the set of nodes $\mathcal{V}$ corresponds to the set of variables and the set of hyperedges $\mathcal{C}$ corresponds to the set of constraint scopes.

In general, constraint satisfaction problems (CSPs) are NP-complete and thus intractable, but certain classes of CSPs have been found to be tractable due to their structure (represented by their primal or hyper graph). Methods of identifying and solving these classes of tractable CSPs are obviously of importance to the constraints (and by extension, the artificial intelligence) community.

The simplest example of a class of tractable CSPs are those which are acyclic (it is known that binary acyclic CSPs can be solved in linear time [1, 2]). A binary CSP is said to be acyclic if there are no loops in its corresponding primal graph. A general (non-binary) CSP is acyclic iff the primal graph is chordal (cycles longer than 3 arcs have chords) and the maximal cliques of the primal graph are the edges of the hypergraph. Given these definitions of acyclicity, we are also able to recognise a CSP as being acyclic in linear time.

Another example of a scheme for recognising tractable CSPs involves identifying the biconnected components of a CSP [3]. A biconnected component of a CSP is a maximal set of variables which remains connected after the removal of any single variable. If the size of the largest biconnected component is k, then the CSP can be solved in time exponential in k, but polynomial in the size of the problem. Thus, the class of CSPs denoted by having biconnected components no larger than a fixed k are tractable.

Several more general schemes have been developed to recognise other classes of tractable CSPs, based upon varying notions of graph or hypergraph **width**. By [5], we know that a CSP with a width of k for any scheme can be solved in polynomial time, where the value of k is the index of the polynomial (naturally, for any scheme, a CSP with a width of 1 (the minimum width) is acyclic, and so can be solved in linear time). The actual technique to be used to solve the CSP in polynomial time is defined by the scheme itself, usually in the form of a **decomposition** or **transformation**.

* This research was funded by an Australian Research Council Linkage Grant

Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'03)
1082-3409/03 $17.00 © 2003 **IEEE**

For each scheme there exists the decision problem of determining whether a given CSP has k width or less. This decision problem itself can at times be intractable, as was shown in [4] for query-width. In addition, for each scheme there exists the problem of finding a decomposition for a given CSP (if one exists). A decomposition describes the transformation from a cyclic CSP to an acyclic CSP. For example, the transformation of CSPs using biconnected components involves solving each biconnected component separately (which takes time exponential in the size of the component), and then combining the solutions (which is polynomial time).

Of the proposed schemes with tractable decision problems, hypertree decompositions have been shown to generate the lowest widths [4, 5]. That is, for a maximum width k, the hypertree decomposition scheme is guaranteed to recognise more CSPs as tractable than any other known scheme with a tractable decision problem. Further, for a given CSP and maximum width k, an optimal hypertree decomposition can be found in polynomial time, where the index of the polynomial is a function of k. This has been proven, with opt-k-decomp and cost-k-decomp [6] being two examples of algorithms which find an optimal decomposition in polynomial time.

One unfortunate aspect of creating hypertree decompositions using opt-k-decomp is the size of the index of the time complexity function. The time complexity of opt-k-decomp is approximately $O(|\mathcal{C}|^{2k}|\mathcal{V}|^2)$ for large $|\mathcal{C}|$. Although the index of $|\mathcal{V}|$ may be smaller for specific classes of CSPs, the index of $|\mathcal{C}|$ is unavoidable in opt- and cost-k-decomp. We present here a simple modification of hypertree decompositions which reduces the best-case time complexity of opt-k-decomp to $O(|\mathcal{C}|^k|\mathcal{V}| + |\mathcal{C}|^2|\mathcal{V}|)$. We argue that many CSPs are in the form required to exhibit near-best-case complexity.

We present our modification in three stages. In the first stage we introduce hypertree decompositions as they have been defined in previous works [4, 5], including definitions of optimality, complete form and normal form. In the second stage we introduce further restrictions on hypertree decompositions. We will prove that these restrictions do not cause any significant loss in the application of hypertree decompositions (for virtually all CSPs, an optimal decomposition can be found which satisfies our restriction). Finally, we will demonstrate that these restrictions have significant practical application by analysing the performance of a modified opt-k-decomp algorithm.

2. Hypertree Decompositions

The basis of the hypertree decomposition scheme is to join together small collections of constraints until the dual-encoding of the CSP is acyclic. Once the dual-encoding becomes acyclic, we can apply known techniques such as directed arc consistency [7] to solve the CSP in polynomial time. As the solutions to the dual-encoding of a CSP are equivalent to the solutions of the original CSP, this is an acceptable transformation.

To define hypertree decompositions (and for the rest of this paper), we need only represent a CSP by its hypergraph, ignoring the domains of constraints and variables. To easily describe hypertree decompositions, the following notation is defined for the hypergraph $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ with $v \in \mathcal{V}$ a vertex (or variable) and $c \in \mathcal{C}$ an edge (or constraint):

$var(c) = c$	variables in c
$con(v) = \{c \in \mathcal{C} : v \in var(c)\}$	constraints **covering** v
$adj(v) = \bigcup_{c \in con(v)} var(c)$	variables **adjacent** to v

Each of the single-element forms of var, con, and adj can be extended to sets by using set union in the usual way. For example, given a set $X \subseteq \mathcal{C}$, $var(X) = \bigcup_{c \in X} var(c)$.

Further, we provide parameterised definitions of components and of 'vertices' used in the definition of hypertree decompositions. Given a set $V \subseteq \mathcal{V}$, and $x, y \in \mathcal{V}$ we say that x and y are $[V]$-connected if there exists a sequence of adjacent variables (a path), none of which are contained in V, which starts at x and ends at y. A set $X \subseteq \mathcal{V}$ is said to be $[V]$-connected if every pair $x, y \in X$ are $[V]$-connected. The set $X \subseteq \mathcal{V}$ is a **[V]-component** if it is $[V]$-connected and maximal. Finally, a set of constraints $R \subseteq \mathcal{C}$ is a **k-vertex** of $\mathcal{H}$ if $|R| \leq k$, where $k \in \mathbb{N}$.

Definition 1 *A **rooted tree** is a pair $T = (N, E)$ where N are the nodes of the tree, and $E \subseteq N \times N$ are the directed edges. T_p indicates the subtree of T with root $p \in N$, $nodes(T)$ indicates the nodes of T, and $edges(T)$ indicates the directed edges of T. A node p is the parent of q if $\langle p, q \rangle \in edges(T)$.*

A hypertree decomposition consists of a rooted tree with labelling functions assigning constraints and variables to nodes of the tree. Precisely how a hypertree decomposition is used is detailed later. We will provide only shortened definitions of hypertree decompositions here. For a more complete treatment of hypertree decompositions and their relation to other decomposition techniques see [4, 5].

Definition 2 *A **hypertree decomposition** of a hypergraph $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ is a triple $\langle T, \chi, \lambda \rangle$ where $T = (N, E)$ is a rooted tree, and χ and λ are two functions labelling each node of T. The functions χ and λ map each node $p \in N$ to two sets $\chi(p) \subseteq \mathcal{V}$ and $\lambda(p) \subseteq \mathcal{C}$, subject to certain restrictions:*

1. *Each constraint must be covered by at least one node of the hypertree.*
   $\forall c \in \mathcal{C}, \exists p \in N$ *st.* $var(c) \subseteq \chi(p)$

2. *Each variable must induce a connected subtree of the hypertree.*
   $\forall v \in \mathcal{V}, (\{p \in T : v \in \chi(p)\}, E)$ *is a connected subtree*
3. *For each node,* $\chi(p)$ *is covered by the combined scopes of the constraints.*
   $\forall p \in N, \chi(p) \subseteq var(\lambda(p))$
4. *For each node,* $\chi(p)$ *includes variables referenced by* $\lambda(p)$ *and any child node.*
   $\forall p \in N, var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$*

The width of a hypertree decomposition $\langle T, \chi, \lambda \rangle$ is defined as the maximum of $|\lambda(p)|$ for all $p \in nodes(T)$. The **hypertree-width** of a CSP is the minimum width for all hypertree decompositions of that CSP. If the width of a hypertree decomposition is equal to the hypertree-width of the CSP, we say the decomposition is **optimal**.

Definition 3 *A* ***complete hypertree decomposition*** *of* $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ *is any hypertree decomposition* $\langle T, \chi, \lambda \rangle$ *where for every constraint* $c \in \mathcal{C}$ *there exists a node* p *of the hypertree such that* $var(c) \subseteq \chi(p)$ *and* $c \in \lambda(p)$.

A complete hypertree decomposition $\langle T, \chi, \lambda \rangle$ describes the transformation from the original cyclic CSP to a new acyclic CSP. Each node p of the hypertree represents a single variable in the new CSP, with its domain defined as the solutions to the subproblem $\langle \lambda(p), \chi(p), D \rangle$. In other words, the labels $\chi(p)$ and $\lambda(p)$ of each hypertree node p are used as follows:

1. The set of constraints $\lambda(p)$ from the original CSP are solved as a subproblem.
2. The set of solutions of the subproblem are projected over the variables $\chi(p)$.
3. The set of projected solutions are used as the domain for a variable in the new acyclic CSP.

An arc between nodes of the hypertree represents a constraint between their corresponding variables in the new CSP. The form of the constraint is the same as that used in the dual-encoding of a CSP. The new CSP is obviously acyclic as its primal graph corresponds to the tree T.

The definitions of hypertree decompositions given so far are very general. In [6] a more restrictive definition is given which simplifies the search process for a hypertree decomposition, but does not prevent us from finding an optimal hypertree decomposition.

Definition 4 *A hypertree decomposition is in* ***normal form*** *if, for any pair of nodes* $\langle p, q \rangle \in edges(T)$*, there is exactly one* $[\chi(p)]$*-component* C *satisfying all of:*

1. $\chi(T_q) = C \cup (\chi(p) \cap \chi(q))$
2. $\chi(q) \cap C \neq \emptyset$
3. $\forall c \in \lambda(q), var(c) \cap adj(C) \neq \emptyset$
4. $\chi(q) = adj(C) \cap var(\lambda(q))$

It should be noted that it is trivial to convert any hypertree decomposition to a complete hypertree decomposition without increasing its width. Given the definition of normal form for hypertree decompositions we can derive χ from λ, which makes the task of generating a hypertree decomposition simpler. This particular result is used advantageously in opt-k-decomp.

3. Reduced Normal Form

The original definition of hypertree decompositions allowed for very poor decompositions. Using normal form it was ensured that certain very poor decompositions were never considered, which led to the development of opt-k-decomp. We now introduce further restrictions on the form of hypertree decompositions, removing even more possible decompositions. The aim of these restrictions is to allow certain assumptions to be made early in opt-k-decomp, reducing the size of data structures and thus time complexity.

Definition 5 *A hypertree decomposition* $\langle T, \chi, \lambda \rangle$ *is in* ***reduced normal form*** *(RNF) if it is in normal form and for every* $p \in nodes(T)$*:*

1. *The constraints in* $\lambda(p)$ *do not completely cover the constraints of* p*'s parent node.*
   $\forall \langle q, p \rangle \in edges(T), var(\lambda(p)) \not\supseteq var(\lambda(q))$
2. *Every constraint* c *in* $\lambda(p)$ *contains a variable which is (a) covered by no other constraint in* $\lambda(p)$ *and (b) adjacent to a variable not covered by* c*.*
   $\forall c \in \lambda(p), \exists v \in var(c), con(v) \cap \lambda(p) = \{c\} \wedge adj(v) \neq var(c)$
3. *If there is only one* $[var(\lambda(p))]$*-component then* p *is a leaf node.*

The definition of RNF places limitations on possible k-vertices which can be used in the decomposition, and where they may be placed. The guiding intuition of RNF is that only nodes which split the problem into multiple components may appear in the body and root of the hypertree decomposition. As a result, nodes which do not split the problem into multiple components may appear only in the leaves of the hypertree.

The following three lemmas (and their proofs) indicate how to transform a hypertree decomposition in normal form to a hypertree decomposition in reduced normal form. The associated figures provide a good intuition of reduced normal form by describing why it removes many decompositions from consideration.

* $\chi(T_p) = \bigcup_{q \in nodes(T_p)} \chi(q)$

Lemma 1 *Let $\langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form. It is possible to transform $\langle T, \chi, \lambda \rangle$ to also satisfy condition 1 of RNF, without increasing the width.*

Proof: Assume there exists a pair $\langle p, q \rangle \in edges(T)$ such that $var(\lambda(p)) \subseteq var(\lambda(q))$. The following transformation will delete the child q and modify the parent p to satisfy condition 1 for p:

1: let $\lambda(p) := \lambda(q)$.
2: let $edges(T) := edges(T) + \{\langle p, r \rangle : \langle q, r \rangle \in E\} - \{\langle q, r \rangle \in E\} - \{\langle p, q \rangle\}$
3: let $nodes(T) := nodes(T) - \{q\}$

After the transformation, we can recompute $\chi(p)$ by the definition of normal form. Note that the width of the decomposition does not increase, as the size of $\lambda(p)$ did not increase beyond that of $\lambda(q)$ (which was already in the decomposition).

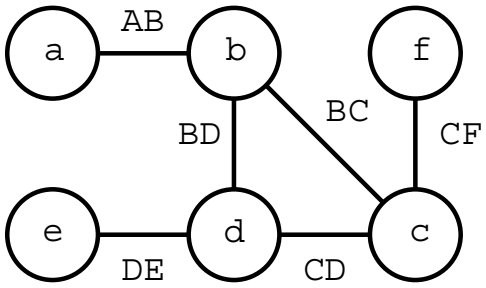


Figure 1. Simple example binary CSP

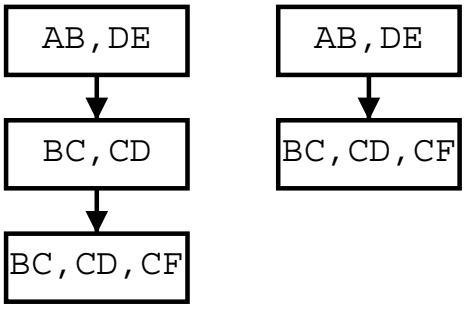


Figure 2. Two hypertree decompositions showing the transformation described in Lemma 1.

Lemma 2 *Let $\langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form and satisfying condition 1 of RNF. It is possible to transform $\langle T, \chi, \lambda \rangle$ to also satisfy condition 2 of RNF, without increasing the width.*

Proof: Assume $p \in nodes(T)$ is a node which does not satisfy condition 2 of RNF. Let $c \in \lambda(p)$ be the constraint with smallest $|var(c)|$ and no variable $v \in var(c)$ which satisfies both $con(v) \cap \lambda(p) = \{c\}$ and $adj(v) \neq var(c)$. The following transformation will remove the constraint from p (effectively satisfying condition 2 for p):

1: let $\lambda(p) := \lambda(p) - \{c\}$
2: **if** $var(c) \not\subseteq var(\lambda(p))$ **then**
3: let create new node $q \in nodes(T)$
4: let $edges(T) := edges(T) + \{\langle p, q \rangle\}$
5: let $\lambda(q) := \{c\}$

If $var(c) \subseteq var(\lambda(p))$, then a new node need not be created (it may be added later when completing the hypertree decomposition). Alternatively, if $var(c) \not\subseteq var(\lambda(p))$, the newly created node q will satisfy condition 1 of RNF (c is the smallest constraint which could be chosen, and so $var(\lambda(q)) = var(c) \not\supseteq var(\lambda(p))$. Again, after the transformation we let $\chi(p)$ and $\chi(q)$ be computed by the definition of normal form. Note that the width of the decomposition can only have decreased as a result of this transformation.

Also note that the node p may still break condition 2 of RNF, requiring repeated applications of this transformation. As there are a finite number of constraints in $\lambda(p)$, these repetitions will terminate. See Figure 3 for an example of this transformation.

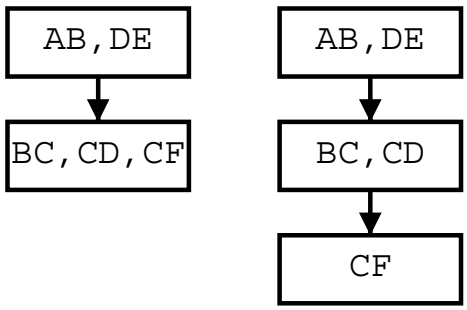


Figure 3. Two hypertree decompositions showing the transformation described in Lemma 2.

Lemma 3 *Let $\langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form, satisfying conditions 1 and 2 of RNF, with $|nodes(T)| \geq 3$. It is possible to transform $\langle T, \chi, \lambda \rangle$ to also satisfy condition 3 of RNF, without increasing the width.*

Proof: For a hypertree decomposition in normal form and satisfying condition 1 of RNF, any node p where only one $[var(\lambda(p))]$-component exists must be either a leaf or

root node (proof omitted). If p is a leaf node then no transformation of the hypertree is necessary.

If, however, p is the root node, then it must have a single child node q, and there must exist more than one $[var(\lambda(q))]$-component. A transformation of the hypertree can then be performed as follows.

1: let q be the (only) child of the root node p.
2: **for all** $R \subseteq \lambda(p)$ **where** $var(R) - var(\lambda(q))$ is connected **and** R is maximal **do**
3: create new node $r \in nodes(T)$
4: let $edges(T) := edges(T) + \{\langle q, r\rangle\}$
5: let $\lambda(r) := R$
6: let $edges(T) := edges(T) - \{\langle p, q\rangle\}$
7: let $nodes(T) := nodes(T) - \{p\}$

After the transformation we let $\chi(q)$ and $\chi(r)$ (for all new nodes r) be computed by the definition of normal form. This transformation ensures any node p with only one $[var(\lambda(p))]$-component is a leaf node. See Figure 4 for an example of this transformation.

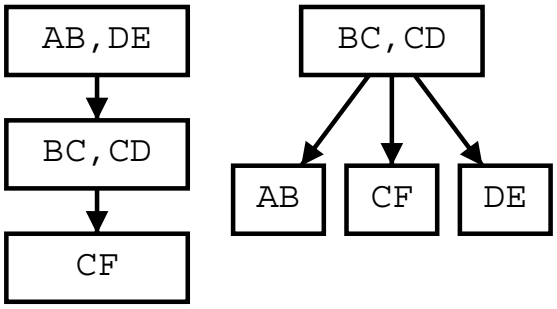


Figure 4. Two hypertree decompositions showing the transformation described in Lemma 3.

From Lemmas 1, 2, and 3 we can show that most CSPs with a hypertree decomposition in normal form have an equivalent or better hypertree decomposition in reduced normal form. The best intuition of CSPs which may not have a reduced normal form hypertree decomposition is those whose optimal hypertree decompositions have only two nodes.

Theorem 1 *If all optimal normal form hypertree decompositions of $\mathcal{H} = \langle \mathcal{V}, \mathcal{C}\rangle$ have more than three nodes, then there exists an optimal reduced normal form hypertree decomposition.*

Proof: By Lemmas 1 and 2 we know that for any optimal normal form hypertree decomposition $\langle T, \chi, \lambda\rangle$ we can derive a new optimal normal form hypertree decomposition $\langle T', \chi', \lambda'\rangle$ which obeys conditions 1 and 2 of RNF. By our assumptions, $|\, nodes(T')| \geq 3$, and so by Lemma 3 we can derive an optimal reduced normal form hypertree decomposition.

It is possible to construct CSPs which do not have a hypertree decomposition in reduced normal form. However, by Theorem 1 we know they have very small numbers of nodes in their optimal hypertree decompositions, and so are unlikely to benefit from hypertree decompositions.

Finally, we present a theorem which will form an important part of our optimisation. Using this theorem and condition 2 of RNF we will be able to reduce the number of k-vertices without regard for where they may have possibly appeared in any eventual hypertree decomposition.

Theorem 2 *If a hypertree decomposition $\langle T, \chi, \lambda\rangle$ is in reduced normal form, and for a vertex $q \in nodes(T)$ there exists only one $[var(\lambda(q))]$-component, then $var(\lambda(q))$ is connected.*

Proof: By condition 3 of RNF, q must be a leaf node. Let p be the parent of q, so there exists a single $[var(\lambda(p))]$-component C satisfying the conditions of normal form for q. By condition 3 of normal form, every constraint in $\lambda(q)$ must intersect C. Additionally, as q is a leaf node, $C \subseteq var(\lambda(q))$. As C is connected, we can conclude that $var(\lambda(q))$ is connected.

4. Application to opt-k-decomp

The algorithm opt-k-decomp [6] achieves two things: it determines if a hypergraph has a hypertree decomposition with width less than k and, if the answer is yes, finds a hypertree decomposition with the smallest width possible. To explain the optimisations we have made to opt-k-decomp we first need an intuitive understanding of how it achieves its task.

Consider that a k-vertex in a hypertree decomposition is used to split (or decompose) a CSP into independent problem components. The algorithm opt-k-decomp represents the relationship between a 'decomposition' and its child 'problems' explicitly by way of a directed, bipartite, weakly acyclic graph.

This graph contains all possible problem components (we will call these **problem nodes**) and all ways to decompose them (we will call these **decomposition nodes**). For each problem node, there must exist at least one child decomposition node by which it can be decomposed. For each decomposition node, a child problem node (if any) represents parts of the CSP which require further decomposition. The algorithm fragment for constructing the graph can be briefly described as:

1: initialise the list of k-vertices
2: initialise the set of problem nodes, including a node representing the entire CSP
3: **for** each problem node **do**

```
4:  for each k-vertex do
5:    if the current k-vertex decomposes the problem node then
6:      create a new decomposition node using the current k-vertex
7:      add an arc from the current problem node to the new decomposition node
8:      for each component of the new decomposition node do
9:        find the problem node with the current component and k-vertex
10:       add an arc from the current decomposition node to the found problem node
```

The algorithm opt-k-decomp consists of two parts: constructing its graph, and extracting a hypertree decomposition. We will note that the complexity of extracting a hypertree decomposition is far less than that of constructing the graph.

While it is possible to execute opt-k-decomp with a large value of k to decompose and solve a wider selection of CSPs, it is not practically feasible. The worst-case complexity of opt-k-decomp as it was originally given is $O(|\mathcal{C}|^{2k}|\mathcal{V}|^2)$ (for large values of $|\mathcal{C}|$). The $|\mathcal{C}|^{2k}$ term provides the greatest difficulty for even relatively small values of k (like 3 or 4). To understand where this term originates, note that the two outer loops iterate independently over data structures whose length is linear in the number of k-vertices. Also note that the total number of k-vertices is linear with respect to $|\mathcal{C}|^k$. Blindly accepting any k-vertex as a candidate for a decomposition node can thus be blamed for the poor worst-case complexity of opt-k-decomp.

The best-case complexity of opt-k-decomp is not noticeably better than the worst-case complexity. Figure 5 presents CPU time data for executions of opt-k-decomp (with $k = 3$) on random CSPs with 20 variables and increasing numbers of constraints. Figure 5(a) presents the execution times divided by the worst-case complexity function for opt-k-decomp. The ratio between CPU times and worst-case complexity values converges to a finite value as the number of constraints increases, suggesting that average-case and worst-case complexity are the same.

The definition of reduced normal form provides us with the ability to remove redundant k-vertices with the aim of reducing the length of the outer loops of opt-k-decomp. Utilising Theorem 2 and condition 2 of RNF we can remove many k-vertices from consideration before encountering the outer loops. Also by identifying k-vertices with multiple components (and assuming the first node in a hypertree decomposition must always have more than one child) we can limit the number of problem nodes created.

Using these filtering techniques, best-case complexity of opt-k-decomp can be reduced to $O(|\mathcal{C}|^k|\mathcal{V}| + |\mathcal{C}|^2|\mathcal{V}|)$ (a class of CSPs has been identified which achieves best-case complexity by ensuring the number of k-vertices remaining for consideration increases linearly with the number of constraints). Unfortunately the worst-case complexity of opt-k-decomp remains at $O(|\mathcal{C}|^{2k}|\mathcal{V}|^2)$.

For more general CSPs, we can show (but not prove) that an implementation of opt-k-decomp which removes redundant k-vertices tends to have best-case complexity rather than worst-case. Figure 6 shows recorded CPU times of this implementation executed on random binary CSPs with 20 variables, with $k = 3$. Figure 6(a) shows that the ratio between worst-case complexity and recorded CPU time tends to zero as the number of constraints increases. This indicates that average-case complexity is less than the theoretical worst-case. It is possible that the actual worst-case complexity is less than $O(|\mathcal{C}|^{2k}|\mathcal{V}|^2)$.

Figure 6(b) also shows that the ratio between best-case complexity and recorded CPU time changes only slightly as the number of constraint increases, appearing to converge to a fixed value. While this is not conclusive evidence, it does indicate that the average-case complexity of the modified opt-k-decomp algorithm may be very close to the best-case complexity. We expect that many classes of CSPs will be decomposable with near-best-case complexity.

Finally, as a comparison between the original and modified opt-k-decomp, we present a graph of the ratios of CPU execution times. Each line represents a class of CSP (described by using a fixed random seed, and progressively adding more constraints). The main observations with respect to this graph is that:

1. At no time does the modified opt-k-decomp take more time to decompose a CSP than the original opt-k-decomp.
2. As the number of constraints increases beyond a certain threshold, the modified opt-k-decomp can become significantly faster. For one sample point, the modifications provide a 1000-fold increase in speed.

5. Conclusion

We have shown that opt-k-decomp can be significantly optimised by removing redundant k-vertices. Although the modified algorithm offers improvements over opt-k-decomp, it is still (in it's current form) unsuitable for decomposing very large CSPs with high hypertree-width. In such cases, the cost of finding all k-vertices will significantly outweigh the cost of any other part of algorithm. It appears that this is almost unavoidable as there are no obvious, efficient methods for generating all k-vertices suitable for use in 'branch' nodes of a decomposition. In the event that domain knowledge is already available on the structure of the CSP to be decomposed (eg. a pro-

duction planning problem), it may be possible to devise heuristics. However, the heuristics may influence the completeness of the algorithm.

We foresee two paths for future work:

1. Take advantage of the new distinction between k-vertices which may appear in leaf and branch nodes. It may be possible to iteratively generate k-vertices, restricting the search for larger k-vertices to those not already covered by smaller 'branch' k-vertices. This idea will form the basis of future work into a general algorithm for hypertree decompositions.
2. Utilise information from a 'failed' run of opt-k-decomp. It may be possible to generate a new cyclic CSP with lower width than the original CSP. We also expect that the accumulated data could be used to highlight areas where higher levels of local consistency (such as path consistency) would provide greatest benefit.

References

[1] Rina Dechter. Constraint Networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992.

[2] Eugene C. Freuder. A Sufficient Condition for Backtrack-free Search. *JACM*, 29(1):24–32, January 1982.

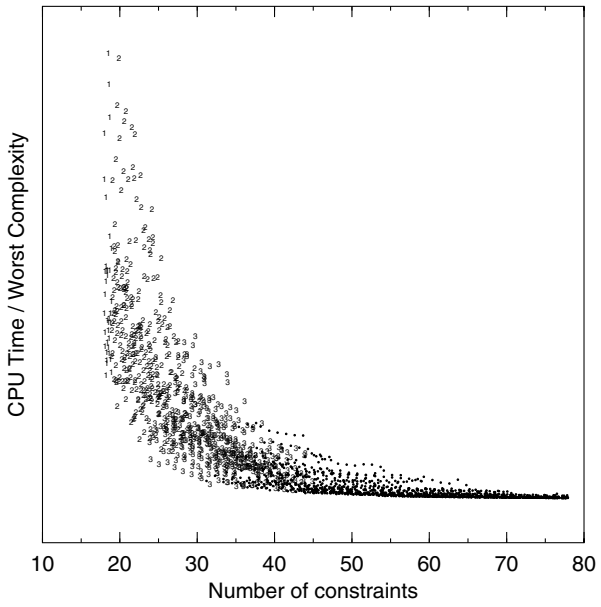
[3] Eugene C. Freuder. A Sufficient Condition for Backtrack-bounded Search. *JACM*, 32(4):755–761, October 1985.

[4] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 21–32. ACM Press, May 31 - June 2 1999.

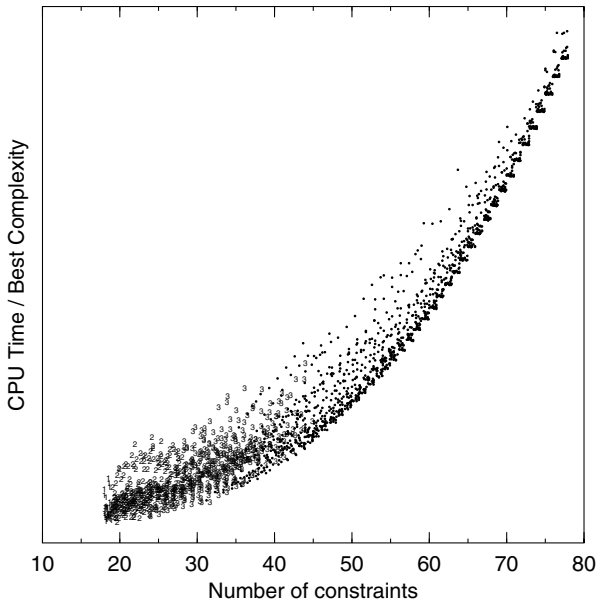
[5] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, December 2000.

[6] Nicola Leone, Alfredo Mazzitelli, and Francesco Scarcello. Cost-Based Query Decompositions. In *Proceedings of SEBD 2002*, 2002.

[7] Richard Wallace. Directed Arc Consistency Processing. In Manfred Meyer, editor, *Constraint Processing, Selected Papers*, volume 923 of *Lecture Notes in Computer Science*. Springer, 1995.

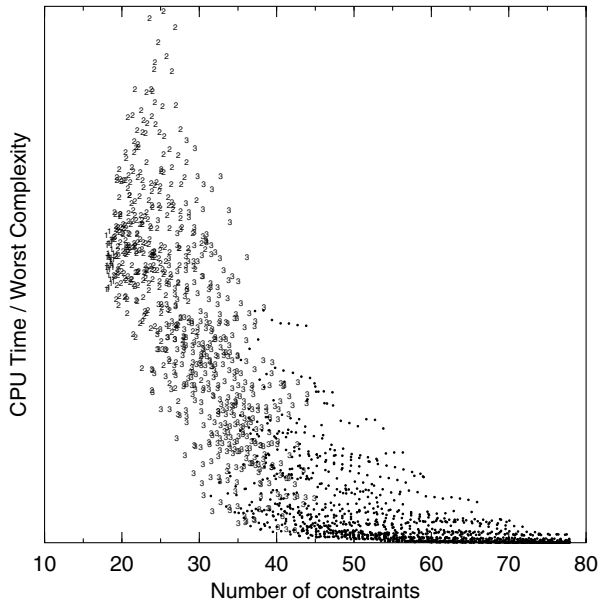


(a)

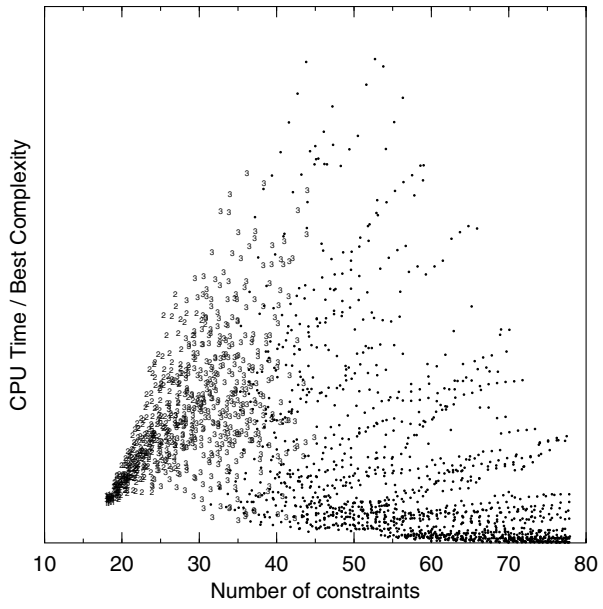


(b)

Figure 5. Actual runs of the unmodified opt-k-decomp on random binary CSP instances with 20 variables and progressively increasing numbers of constraints.



(a)



(b)

Figure 6. Actual runs of the modified opt-k-decomp on random binary CSP instances with 20 variables and progressively increasing numbers of constraints.

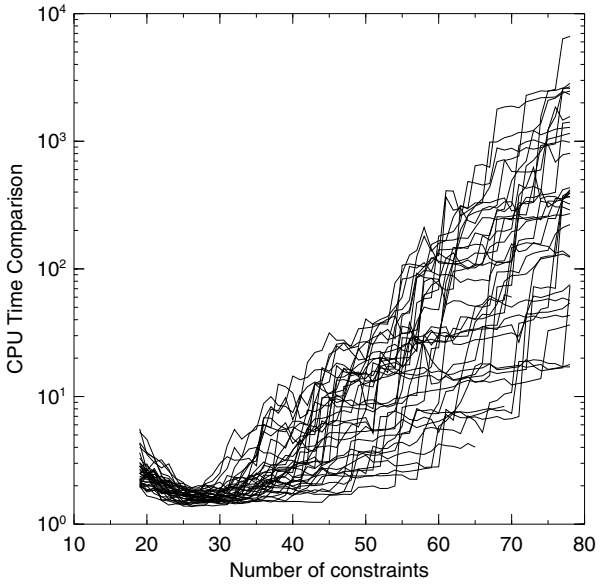


Figure 7. A comparison of CPU times for random CSP instances. Values are computed as the CPU time for the original opt-k-decomp divided by the CPU time for the modified opt-k-decomp.