Cut-and-Traverse: A New Structural Decomposition Strategy for Finite Constraint Satisfaction Problems *

Yaling Zheng and Berthe Y. Choueiry

Constraint Systems Laboratory
University of Nebraska-Lincoln
Email: yzheng|choueiry@cse.unl.edu

Abstract. In this paper, we propose a new heuristic structural decomposition strategy for Constraint Satisfaction Problems (CSPs), called Cut-and-Traverse (CAT) decomposition. This method has three steps: cutting step, traversing step and combining step. In the first step, cutting step, we gradually decompose the structure of a CSP instance to multiple partitions by finding independent cuts in a given CSP. Every cut is a set of hyperedges in a CSP instance. We can control the size of cut by only finding independent cuts that are no greater than a given bound number h. In the second step, traversing step, we traverse each partition and get an equivalent acyclic constraint network for each partition. In the third step, combining step, we combine the acyclic constraint networks in all partitions and get an equivalent acyclic constraint network for the given CSP. Using this method, the complexity is $O(|E|^h)$ where $|E|$ is the number of constraints in given CSP instance. Our experiment shows that CAT decomposition spends far much less time than hypertree decomposition while sometimes produces same hyperwidth decomposition as hypertree decomposition. Also it is shown that CAT decomposition always produces better decomposition result than hinge decomposition.

1 Introduction

A large number of important practical problems, such as scheduling and designing problems, can be formulated as Constraint Satisfaction Problems (CSP). CSPs are **NP**-complete and mainly solved by search. While many strategies are being investigated to improve the performance of the search process itself, other research efforts are invested in identifying tractable classes of CSPs. One approach of the latter category, inspired from results on acyclic query processing in relational databases, is based on exploiting structural properties of the constraint network. For example, when the hyperwidth of this network is bounded, the CSP is guaranteed to be tractable [1]. New decomposition techniques that

* This work is supported by CAREER Award #0133568 from the National Science Foundation

decompose a cyclic CSP into an acyclic one have recently received increased attention. In this paper we explore a new heuristic decomposition technique that falls under this category.

It is known that acyclic CSPs can be solved in polynomial time [2]. A CSP is acyclic iff the primal graph is chordal (for any cycles longer than 3 arcs there is an edge joining two non-consecutive vertices along the cycle) and each maximal clique of the primal graph corresponds to an edge in the hypergraph.

A lot of structural decompositions have been proposed recently to compute an equivalent acyclic CSP for a given CSP instance. These structural decomposition methods include biconnected decomposition [3] (BICOMP), hinge decomposition [4, 5] (HINGE),tree clustering [6] (TCLUSTER), hinge decomposition combined with tree clustering [4] (HINGE$^{\mathrm{TCLUSTER}}$), cycle cutset [7] (CUTSET), hypertree decomposition [8, 9] (HYPERTREE), comparison of most of these methods is done in [10].

Among these decomposition algorithms, hinge decomposition computes an equivalent acyclic constraint network for a given hypergraph $\mathcal{H}$. The edges of the equivalent acyclic constraint network have labeled arcs. We are inspired by these labeled arcs, which are regarded as cuts in our decomposition. However, hinge decomposition is not optimal. A decomposition method is called optimal iff the hyperwidth hyperwidth of equivalent acyclic constraint network for any constraint hypergraph it computes is always equal to the hypertree width by hypertree decomposition. The hyperwidth of the equivalent acyclic network for a constraint network using hinge decomposition is usually much bigger than hypertree width of the constraint network. For the recently proposed hypertree decomposition [8, 9], although it is optimal and the hypertree decomposition scheme with redundancy reduced [9] reduces the best-case time complexity of *opt-k-decomp* to $O(|\mathcal{C}|^k|\mathcal{V}|+|\mathcal{C}|^2|\mathcal{V}|)$, our experiment shows that it is not efficient in practise, especially for a CSP instance with big hypertree width. Failing to utilise information from a 'failed' run of *opt-k-decomp* is another disadvantage of hypertree decomposition.

In this paper, we propose a new structural decomposition algorithm which is inspired by both hinge decomposition and hypertree decomposition, called Cut-and-Traverse (CAT) decomposition. CAT decomposition takes a bound number h limiting the size of cuts and a hypergraph, goes for 3 steps. The first step, cutting step, which finds a set of cuts in a hypertree, can not only find cuts with size 1, as hinge decomposition actually does, but also find cuts greater than size 1. The second step, traversing step, which produces an equivalent acyclic network with a lot of tree nodes for each partition, may produces tree nodes that are greater than the bound number h. The third step, combining step, combines all the acyclic networks for partitions, forms an equivalent acyclic network for the whole hypergraph.

The paper is organized as follows. In Section 2 we review the necessary terminology for CSPs, hypergraphs, hinge decomposition, hypertree decomposition. Then in Section 3, Cut-and-Traverse (CAT) algorithm is shown, an example is illustrated, the complexity of CAT decomposition is evaluated. After that,

experimental data is shown to compare hinge decomposition, hypertree decomposition and CAT decomposition in Section 4. Finally, in Section 5 we conclude this paper and propose future work.

2 Background

In this section, we present a set of definitions explaining basic definition of CSP, constraint hypergraph, hinge decomposition, hypertree decomposition, hyperwidth, treewidth.

Definition 1. Constraint Satisfaction Problem [2]: *A Constraint Satisfaction problem (CSP) is defined as a tuple* $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ *where* $\mathcal{V}$ *is a set of variables,* $\mathcal{C}$ *is a set of constraints and* $\mathcal{D}$ *is a set of value domains for the variables. Every constraint* $C_i \in \mathcal{C}$ *is a relation over a set* S *of variables (which determines the scope of constraint* $\text{SCOPE}(\mathcal{C})=S$*) and defines the set of allowed tuples. A solution to the CSP problem is an assignment of values to the variables satisfying all the constraints.*

The structure of a CSP constraints can be represented by a constraint hypergraph. A constraint hypergraph is defined as:

Definition 2. Constraint hypergraph [2]: *The structure of a CSP constraints can be represented by a constraint hypergraph. A hypergraph* $\mathcal{H}$ *of a CSP problem can be represented as* $\mathcal{H} = (V, S)$ *that consists of a set of vertices* $V = \{v_1, v_2, \ldots, v_n\}$ *and a set of subsets of these vertices* $S = \{S_1, S_2, \ldots, S_l\}$, $S_i \subseteq V$. S_i *is a constraint scope in a CSP, it is a hyperedge in a hypergraph.*

A hypergraph of crossword puzzle problem is shown in Figure 1.

Definition 3. Hinge decomposition [4]: *Let* (V, E) *be any hypergraph. A hinge-tree of* (V, E) *is a tree,* (N, A)*, with nodes* N *and labeled arcs* A*, such that:*

1. *The tree nodes are minimal hinges of* (V, E).
2. *Each edge in* E *is contained in at least one tree node.*
3. *Two adjacent tree nodes share precisely one edge of* E *which is also the label of their connecting tree-arc; moreover, their shared vertices are precisely the members of this edge.*
4. *The vertices of* V *shared by two tree nodes are entirely contained within each tree node on their connecting path.*

Definition 4. Hypertree decomposition [11]: *A hypertree decomposition of a hypergraph* $\mathcal{H}$ *is a hypertree* $HD = (T, \chi, \lambda)$ *for* $\mathcal{H}$ *which satisfies all the following conditions:*

1. *for each edge* $h \in edges(\mathcal{H})$*, there exists* $p \in vertices(T)$ *such that* $var(h) \subseteq \chi(p)$ *(we say* p *covers* h*).*
2. *for each variable* $Y \in var(\mathcal{H})$*, the set* $\{p \in vertices(T) \mid Y \in \chi(p)\}$ *induces a (connected) subtree of* T*;*

3. *for each* $p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$;
4. *for each* $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

The inclusion in condition (4) is actually equality, because condition (3) implies the reverse inclusion.

Definition 5. Hypertree width [11]: *the hypertree width for a given hypergraph* $\mathcal{H}$ *hw*$(\mathcal{H})$ *is the minimum width over all its possible hypertree decompositions.*

Definition 6. Treewidth [2] *The tree width of a tree decomposition* (T, χ, λ) *is* $tw = max_{v \in V} \mid \chi(V) \mid$.

3 A new heuristic structural decomposition

First, in section 3.1, we give the definition of CAT decomposition and the hyperwidth of a CAT decomposition. Then, in section 3.2, we present the CAT decomposition algorithm, which, given a hypergraph and a given bound number, returns an equivalent acyclic constraint network. After that, the complexity of CAT decomposition is evaluated in section 3.3, the process after CAT decomposition for a given CSP instance to find a solution is shown in section 3.4. Finally, a crossword example is shown in section 3.5.

3.1 Definitions

Definition 7. Cut-and-Traverse (CAT) decomposition: *A CAT decomposition of a hypergraph* $\mathcal{H}$ *is a hypertree* $HD = (T, \lambda)$ *for* $\mathcal{H}$ *which satisfies all the following conditions:*

1. *for each edge* $h \in edges(\mathcal{H})$, *there exists* $p \in vertices(T)$ *such that* $var(h) \subseteq \lambda(p)$ *(we say* p *covers* h*).*
2. *for each variable* $Y \in var(\mathcal{H})$, *the set* $\{p \in vertices(T) \mid Y \in var(\lambda(p))\}$ *induces a (connected) subtree of* T*;*

Definition 8. Hyperwidth of a CAT decomposition: *The hyperwidth of a Cut-and-Traverse decomposition* $T = (N, A)$ *for a hypergraph* $(H) = (V, E)$ *is defined as* $max_{p \in N}(min\{|p'| \mid \text{var}(p') = \text{var}(p) \wedge (p' \subseteq p)\})$.

Definition 9. var(S)*: for* S*, which can be an edge or a set of edges,* var(S) *is the vertices included in this edge or a set of edges.*

Definition 10. neighbors(S, H)*: for a set of hyperedges* S *and a hypergraph* $\mathcal{H} = (V, E)$, neighbors$(S, \mathcal{H})$ *is the set of hyperedges in* $\mathcal{H}$ *that joins at least one hyperedge in* S*. That is,* neighbors$(S, \mathcal{H}) = \{e \mid (e \in E) \wedge (e \cap \text{var}(S) \neq \emptyset)\}$.

Definition 11. remain-hg(C,H)*: for a set of hyperedges* C *in a hypergraph* $\mathcal{H} = (V, E)$, remain-hg$(C, \mathcal{H})$ *is a new hypergraph* $R = (V_r, E_r)$, *which can be defined as:*

$$V_r = V - var(S)$$
$$E_r = \bigcup_{e \in E} \{e - var(S)\}$$

For example, for $C = \{(1\ 2\ 3\ 4\ 5)(20\ 21\ 22\ 23\ 24\ 25\ 26)\}$ from the given hypergraph $\mathcal{H}_{cg}$ in Figure 1, *remain-hg*$(C, \mathcal{H}_{cg})$ is shown in Figure 2.

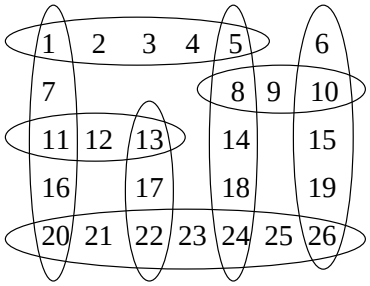

Fig. 1. *Hypergraph $\mathcal{H}_{cp}$ of the crossword puzzle.*

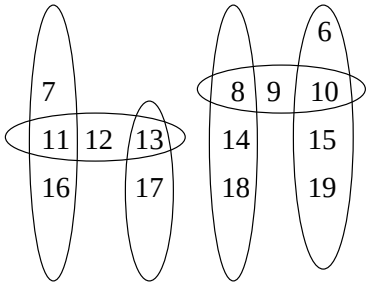

Fig. 2. *The remaining graph of the hypergraph shown in Figure 1 after removal of a set of hyperedges $C = \{(1\ 2\ 3\ 4\ 5)(20\ 21\ 22\ 23\ 24\ 25\ 26)\}$.*

Definition 12. Cut *of a hypergraph: a set of hyperedges C in a hypergraph $\mathcal{H}$ is called a cut if* remain-hg*$(C, \mathcal{H})$ has at least 2 maximal components. a set of hyperedges is called a component if these edges are connected. A component is said to be maximal if no othe hyperedges can be added into this component to be formed a larger component.*

Definition 13. *$f(C, H)$: for a cut C of a hypergraph $\mathcal{H}$, we evaluate the cut by a heursitic function f. $f(C, \mathcal{H})$ is the largest number of hyperedges among all maximal components in* remain-hg*$(C, \mathcal{H})$.*

3.2 Cut-and-Traverse (CAT) decomposition

In this section we explain describe a new decomposition method, called CAT decomposition, which, given a hypergraph $\mathcal{H}$, computes a CAT decomposition for a hypergraph $\mathcal{H}$. This algorithm is presented in Algorithm 1. It has third steps. The first step finds a set of cuts that is no greater than a given bound number, and returns a set of partitions, each containing one or 2 cuts, as shown in Algorithm 2, the related sub routines of Algorithm 2 are shown in Algorithm 3 and Algorithm 4. The second step traverses every partition found in the first step, as shown in Algorithm 5, returns a tree for each partition. The third step combines the trees found in the second step, as shown in Algorithm 6.

First we explain the following terms in CAT decomposition algorithm.

Definition 14. Cuts to be independent*: Two cuts are said to be independent with each other iff they don't include same hyperedges.*

Definition 15. Size of a set of hyperedges*: the size of a set of hyperedges S is the smallest number of hyperedges within S that covers* var*(S), formally, it is* $min\{|S'| \mid S' \in S, var(S') = var(S)\}$.

Definition 16. Partition*: a partition P in a hypergraph (H) includes the following attributes:*

1. content*, which is a set of hyperedges representing a hypergraph;*
2. cut1*, which is a cut of $\mathcal{H}$ or is* nil *and satisfies* P.cut1 $\subseteq$ P.content*;*
3. cut2*, which is another cut for $\mathcal{H}$ or is* nil *and satisfies* P.cut2 $\subseteq$ P.content*;*
4. min-new-cut-size*, which is the minimum size of a new cut independent with* P.cut1 *and* P.cut2.

Once a paritition can not be further divided into more smaller partitions by cuts within a given number h, it will be traversed from $P.cut1$ to $P.cut2$ or $P.cut2$ to $P.cut1$ if P.cut1 $\neq \emptyset$ and $P.cut2 \neq \emptyset$, or will be traversed from $P.cut1$ until all hyperedges are visited if $P.cut2$.

Definition 17. candidate cuts*: A candidate cut C for a partition P is the set of cuts that satisfying the following conditions:*

1. *the size of cut* $C = P.min\text{-}new\text{-}cut\text{-}size$*;*
2. *C doesn't include any edge in* cut1 *and* cut2 *of P. That is,* $C \cap (P.cut1 \cup P.cut2) = \emptyset$*;*
3. remain-hg*$(C, \mathcal{P})$ has at least 2 maximal components and* P.cut1, P.cut2 *are in 2 different maximal components;*

Candidate cuts are all cuts satisfying the above conditions.

In CAT decomposition, a number is given to restrict the size of cuts in the cutting step. This number doesn't restrict the hyperwidth of a CAT decomposition. Thus, using CAT decomposition, the hyperwidth of the CAT decomposition for a hypergraph $\mathcal{H}$ is not bound by given number h. While the hypertree decomposition [11] computes an optimal hypertree of $\mathcal{H}$ with hyperwidth within given bound number k; the algorithm returns *failure* if no such decomposition exists. In our algorithm, the constant h is only restricted to the size of cuts. Therefore, CAT decomposition is much more flexible than hypertree decomposition at the expense of losing its optimality. Also, while hinge decomposition only finds cuts with size 1 as labelled edge, CAT decomposition finds cuts with given bound number h. When $h = 1$, CAT decomposition is in fact same as hinge decomposition except that in CAT decomposition, labeled edges are regarded as cuts. When $h > 1$, the CAT decomposition is better or as good as hinge decompositon as far as hyperwidth and treewidth are concerned.

3.3 Complexity of CAT decomposition

The complexity of CAT decomposition is only related to the given bound h. Since the number of possible combination of h hyperedges in a hypergraph $\mathcal{H} =$

Input: A hypergraph $\mathcal{H} = (V, E)$ and a bound number $h \leq \lfloor \frac{|E|}{2} \rfloor$
Output: A CAT decomposition of $\mathcal{H}$
initial-partition.content $\leftarrow \mathcal{H}$;
initial-partition.cut1 $\leftarrow \emptyset$;
initial-partition.cut2 $\leftarrow \emptyset$;
initial-partition.min-new-cut-size $\leftarrow 1$;
partitions $\leftarrow$ FindCuts(*initial-partition*, h);
if *partitions = "failure"* **then**
  /* separate E into two groups and return the result */
  $N_1 \leftarrow$ a set of hyperedges in E s.t. $|N_1| = \lfloor \frac{|E|}{2} \rfloor$;
  $N_2 \leftarrow E - N_1$;
  $N \leftarrow \{N_1, N_2\}$;
  $A \leftarrow (N_1, N_2)$;
  $T \leftarrow (N, A)$;
end
else
  $S_{trees} = \emptyset$;
  foreach *partition* P *in* partitions **do**
    add Traverse(P) to S_{trees};
  end
  $T \leftarrow$ Combine(S_{trees});
end
return T;

Algorithm 1: *CAT($\mathcal{H}$, h).*

Input: a partition P and a bound number $h \leq \lfloor \frac{n}{2} \rfloor$
Output: a set of partitions
partitions $\leftarrow \emptyset$;
newpartitions $\leftarrow$ FindANewCut(P, h);
if *newpartitions* $\neq$ *"Failure"* **then**
  foreach $P' \in$ *newpartitions* **do**
    FindCuts(P', h)
  end
  else
    push P into *parititions*
  end
end
return *partitions*;

Algorithm 2: FindCuts*(P, h).*

Input: C and P, C is a found cut for partition P
Output: a set of new partitions
new-partitions $\leftarrow \emptyset$
foreach *maximal component* G *of* P **do**
    /* F is a mapping function that maps original hyperedge */
    /* to a remain hyperedge in *remain-hg*(C, P.content). */
    P'.content $\leftarrow \{x \mid F(x) \in G\} \cup C$
    if *(*P'.content $\cap$ *(*P.cut1 $\cup$ P.cut2*))* $\neq \emptyset$ **then**
        p'.*cut1*:= P.*cut1* $\cup$ P.*cut2*;
        p'.*cut2*:= C;
    end
    else
        p'.*cut1*:= C;
    end
    add P' to *new-partitions*;
end
return *new-partitions*;

Algorithm 3: BuildNewPartitions*(C, P)*.

Input: a partition P and a bound number $h \leq \lfloor \frac{n}{2} \rfloor$
Output: return a set of partitions if a new cut is found or *nil* if no new cut is found
candidate-cuts $\leftarrow \emptyset$;
while *(candidatecuts* $\neq \emptyset$*) and (*P.min-new-cut-size $\leq h$*)* **do**
    /* possible cuts in a partition P is all the combinations of */
    /* $k = P$.*min-new-cut-size* hyperedges */
    /* except the hyperedges in P.*cut1* and P.*cut2* */
    possible-cuts $\leftarrow$ possible cuts of P;
    /* refer definition of candidate cuts in Definition 17 */
    candidate-cuts $\leftarrow$ candidate cuts of P in *possible-cuts*;
    if *candidate-cuts* $= \emptyset$ **then**
        P.*min-new-cut-size* $\leftarrow$ P.*min-new-cut-size* + 1;
    end
end
if $k > h$ **then**
    return "failure"
end
else
    /* refer definition of f in Definition 13 */
    chosencut $\leftarrow$ a candiate cut with minimum $f(C, P)$ among *candidate-cuts*;
    return BuildNewPartitions(C, P);
end

Algorithm 4: FindANewCut*(P, h)*.

Input: a partition $P = (V, E)$
Output: a traverse tree $T = (N, A)$ for P
$C_1 = P.cut1$;
$C_2 = P.cut2$;
foreach $e \in E - C_2$ **do**
    mark e as *unvisited*;
end
add C_1 into N;
mark $e \in C_1$ as *visited*;
unvisited-edges $\leftarrow E - C_2$;
justvisited-edges $\leftarrow C_1$;
visiting-edges $\leftarrow \emptyset$;
while *(justvisited-edges* $\neq C_2$*)* $\wedge$ *(unvisited-edges* $\neq \emptyset$*)* **do**
    visiting-edges $\leftarrow$ *neighbors*(*justvisited-edges*) $\cap$ *unvisited-edges*;
    /* add those edges included in *visiting-edges* */
    visiting-edges $\leftarrow \{e \mid e \subseteq var($*visiting-edges*$) \cap e \in$ *unvisited-edges*$\}$;
    if *visiting-edges* $\neq \emptyset$ **then**
        add *visiting-edges* to N;
        add an arc $(visiting - edges, justvisited - edges)$ to A;
        mark $e \in visiting - edges$ as *visited*;
        remove *visiting-edges* from *unvisited-edges*;
        $justvisited - edges \leftarrow visiting - edges$;
    end
end
$T \leftarrow (N, A)$;
return T;

Algorithm 5: TRAVERSE*(P)*.

Input: A set of trees S_{trees}
Output: A tree $T = (N, A)$ that is the combination of the input trees
$N \leftarrow \emptyset$;
$A \leftarrow \emptyset$;
foreach $S_i = (N_i, A_i) \in S_{trees}$ **do**
    add $N_i - N$ into N;
    add A_i into A;
end
$T \leftarrow (N, A)$;
return T;

Algorithm 6: COMBINE*(S_{trees})*.

(V, E) is no greater than $O(|E|^h)$ where $|E|$ is the number of constraints in the hypergraph and only cuts with size no greater than h is considered, so the complexity of our algorithm in the worst case is bound by $O(|E|^h)$. Thus, the complexity of CAT decomposition is polynomial. Although the complexity of hypertree decomposition with redundancy reduced is bound by $|E|^k|V|+|E|^2|V|$ where $|E|$ is the number of hyperedges and $|V|$ is the number of vertices in $\mathcal{H}$, experiment in Section 4 shows that CAT decomposition takes far less time than hypertree decomposition.

3.4 Solving a CSP by Cut-and-Traverse decomposition

The procedure to solve the CSP after finding an equivalent acyclic constraint network using CAT decomposition is shown in Algorithm 7. The complexity of the Algorithm 7, is $O(|V||E|^2) + O(|E|l^d \log l)$, where l is the maximal size of a constraint in C, and d is the hyperwidth of the hypergraph (V, E).

Input: A constraint satisfaction problem $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$
Output: A solution to $\mathcal{P}$
Using Algorithm 1, compute a CAT decomposition T for hypergraph $(\mathcal{V}, \mathcal{S})$ where S is the constraint scope of $\mathcal{C}$;
Using T, compute the equivalent constraint satisfaction problem, $\mathcal{P}'$ by solving the subproblems corresponding to the nodes of T;
Find a solution to the tree-structured problem $\mathcal{P}'$ in a backtrack-free manner, as described in [3].

Algorithm 7: *Solving a CSP by Cut-and-Traverse decomposition*

3.5 An example

In this section, we illustrate CAT decomposition for a given hypergraph of a crossword puzzle problem as shown in Figure 1 [10]. We bound the cutting size as $h = 2$, the three steps to find an equivalent acyclic constraint network are as follows:

1. *Cutting step.*
   For the hypergraph, it has 7 candiate cuts with size 2, they are

```
{(1 2 3 4 5)(20 21 22 23 24 25 26)} (f = 3)
{(11 12 13)(20 21 22 23 24 25 26)}  (f = 5)
{(8 9 10)(20 21 22 23 24 25 26)}  (f = 5)
{(1 7 11 16 20)(13 17 22)}   (f = 5)
{(1 7 11 16 20)(20 21 22 23 24 25 26)} (f = 4)
{(5 8 14 18 24)(6 10 15 19 26)} (f = 5)
{(5 8 14 18 24)(20 21 22 23 24 25 26)} (f = 4)
```

f is the value of a cut in a hypergraph is the largest component size of the remaining graph after removing the cut in a hypergraph, as defined in Definition 13.

we choose the cut { 1 2 3 4 5)(20 21 22 23 24 25 26)} whose f value is smallest. The partitions separted by the cut is as follows:

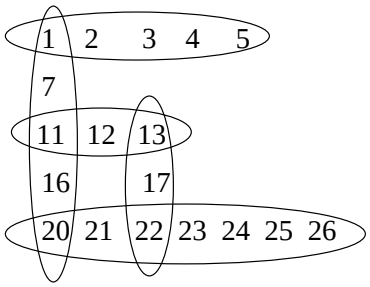

Fig. 3. *Partition 1.*

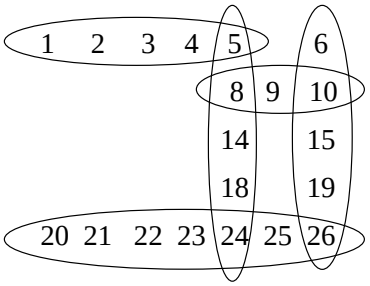

Fig. 4. *partition 2*

For partition P_1, there are no candiate cuts.

For partition P_2, there are no candiate cuts.

Cutting step finishes.

2. *Traversing step*
   Traverse P_1, we get a tree as shown in Figure 5. Traverse P_2, we get a tree as shown in Figure 6.

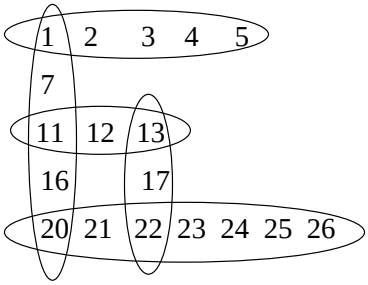

Fig. 5. *Traverse partition 1*

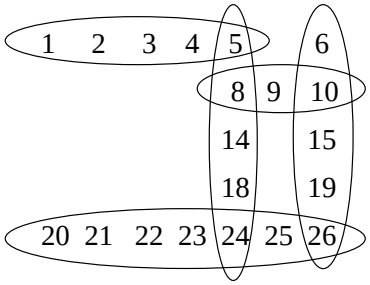

Fig. 6. *Traverse partition 2*

3. *Combining step.*
   Combine the tree in Figure 5 and Figure 6, we get the result as shown in Figure 7

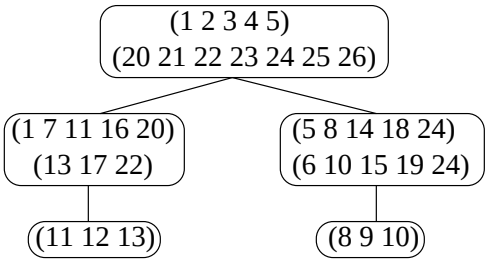

Fig. 7. *CAT decomposition*

4 Experimental data

In the experiment, we randomly generate 100 hypergraphs with 20 variables and 10 constraints. We compare three decomposition methods: hinge decomposition (HINGE), hypertree decomposition with redundancy reduce (HTD), and Cut-and-Traverse decomposition (CAT). We compare three items: CPU time, treewidth and hyperwidth. Here, we regard the treewidth of hinge decomposition as the largest number of variables covered in a vertex of hinge tree. the treewidth of CAT decomposition as the largest number of variables covered in a vertex of a CAT decomposition. Formally, the treewidth of a CAT or hinge decomposition $T = (N, A)$ for a hypergraph $\mathcal{H}$ is $min\{|var(p)| \mid p \in N\}$. We regard the hyperwidth of hinge decomposition and CAT decomposition as the largest size of tree nodes. The size of a set of hyperedges S is the smallest number of hyperedges within S that covers $var(S)$. Formally, the hyperwidth of a CAT or hinge decomposition $T = (N, A)$ for a hypergraph $\mathcal{H}$ is $max_{p \in N}(min\{|p'| \mid (var(p') = var(p)) \wedge (p' \subseteq p)\})$.

From above experimental data, we know that, CAT decomposition produces better decomposition result than hinge decomposition. In some cases, CAT decomposition produces same hyperwidth result as HTD decomposition, as shown in Figure 10. Also, CAT decomposition spends much less time than HTD decomposition. In these 100 instances, CAT decomposition spend no more than 10 millseconds while HTD decomposition spends much more time, as shown in Figure 8. In fact, under all cases, the treewidth of CAT decomposition never exceeds the treewidth of HINGE decomposition, as shown in Figure 9.

5 Conclusion and future work

In this paper, we propose a new decomposition algorithm, which was inspired by both hinge-tree decomposition and hypertree decomposition. This decomposition algorithm has three basic steps. In the first step, cutting step, a set of independent cuts are found and a set of partitions cut by these cuts are computed. Then in the second step, traversing step, each partition that was found in the first step is traversed and an equivalent tree is computed. Finally, in the

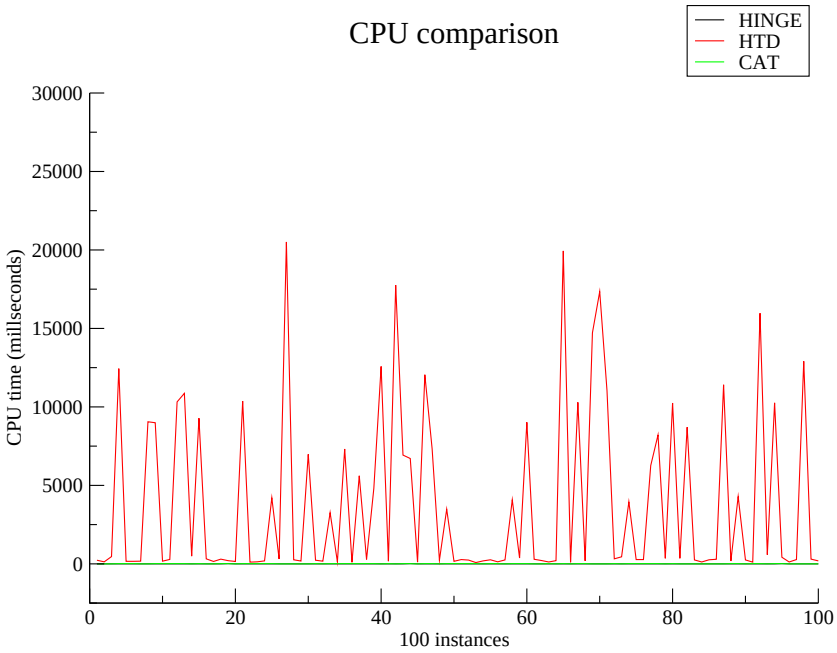

Fig. 8. *CPU comparison of HINGE, HTD and CAT.*

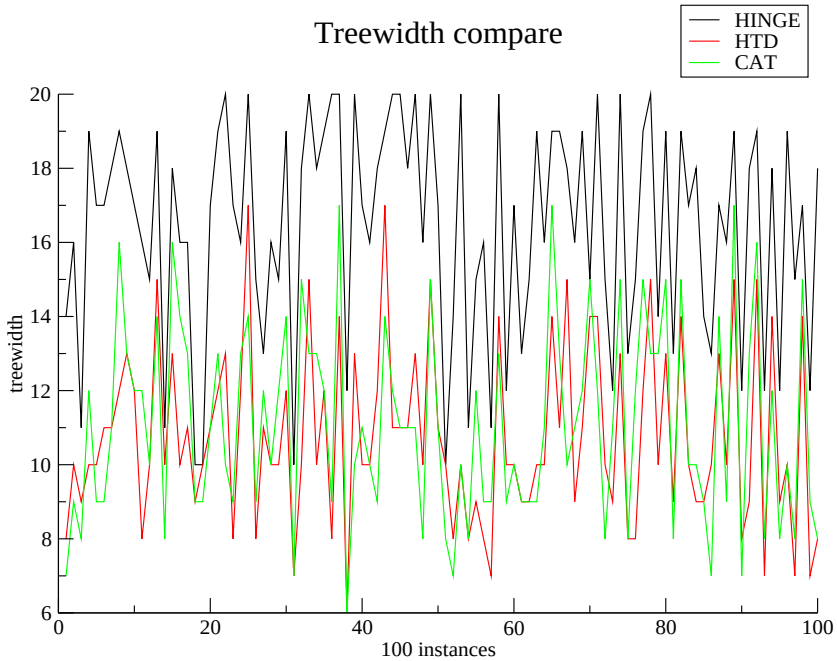

Fig. 9. *Treewidth Comparison of HINGE, HTD and CAT.*

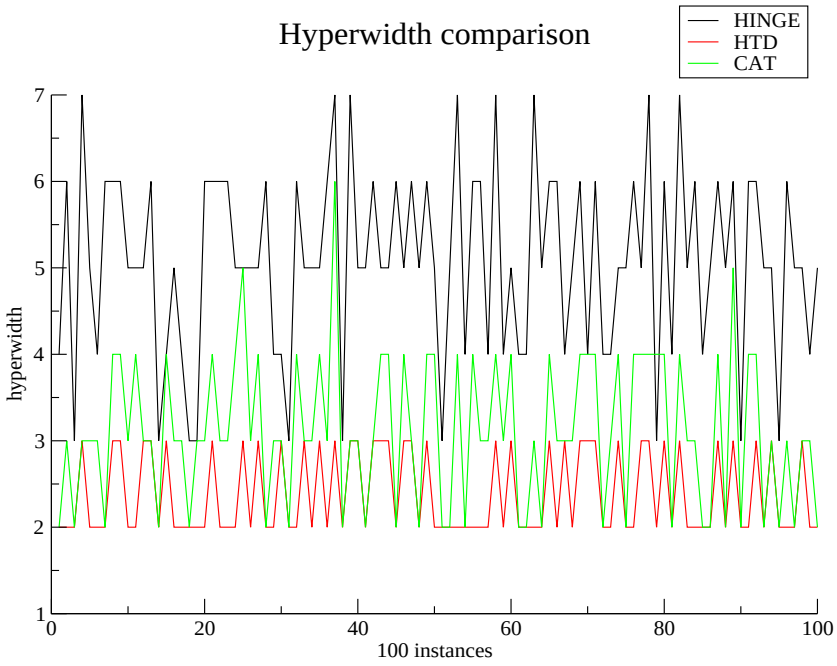

Fig. 10. *Hyperwidth Comparison of HINGE, HTD and CAT.*

third step, combining step, the trees computed in the second step is combined and a CAT tree is achieved. In conclusion, CAT is a heuristic structural decomposition. It choose a best cut among multiple candiate cuts which are evaluated by a heuristic function.

We implement CAT decomposition and compare CAT decomposition with hinge decomposition and hypertree decomposition on random hypergraphs, which shows that CAT decomposition spends much less time than HTD decomposition while sometime produces same hyperwidth decomposition result as HTD. Also, CAT decomposition produces much better result than HINGE decomposition for almost all cases.

Our future work is to do more experiments on random CSPs to compare the efficiency of solving CSPs using different decomposition methods.

References

1. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions: A Survey. In: 26^{th} International Symposium on Mathematical Foundations of Computer Science (MCFS 2001), Marianske Lazne, Czech Republic (2001) 37–57
2. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
3. Freuder, E.C.: A Sufficient Condition for Backtrack-Bounded Search. JACM **32 (4)** (1985) 755–761

4. Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing Constraint Satisfaction Problems Using Database Techniques. Artificial Intelligence **66** (1994) 57–89
5. Jeavons, P.G., Cohen, D.A., Gyssens, M.: A Structural Decomposition for Hypergraphs. Contemporary Mathematics **178** (1994) 161–177
6. Dechter, R., Pearl, J.: Tree Clustering for Constraint Networks. Artificial Intelligence **38** (1989) 353–366
7. Dechter, R., Pearl, J.: The Cycle-Cutset Method for improving Search Performance in AI Applications. In: Third IEEE Conference on AI Applications, Orlando, FL (1987) 224–230
8. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions and Tractable Queries. Journal of Computer and System Sciences **64** (2002) 579–627
9. Harvey, P., Ghose, A.: Reducing Redundancy in the Hypertree Decomposition Scheme. In: The 15^{th} IEEE International Conference on Tools with Artificial Intelligence (ICTAI 03). (2003) 474–481
10. Gottlob, G., Leone, N., Scarcello, F.: A Comparison of Structural CSP Decomposition Methods. Artificial Intelligence **124** (2000) 243–282
11. Gottlob, G., Leone, N., Scarcello, F.: On Tractable Queries and Constraints. In: 10^{th} International Conference and Workshop on Database and Expert System Applications (DEXA 1999). (1999) 1–15