A Decomposition Technique for CSPs Using Maximal Independent Sets and Its Integration with Local Search

Joel Gompert and Berthe Y. Choueiry
Constraint Systems Laboratory
Computer Science & Engineering, University of Nebraska-Lincoln
{jgompert|choueiry}@cse.unl.edu

Abstract

We introduce INDSET, a technique for decomposing a Constraint Satisfaction Problem (CSP) by identifying a maximal independent set in the constraint graph of the CSP. We argue that this technique reduces the complexity of solving the CSP exponentially by the size of the maximal independent set, and yields compact and robust solutions. We discuss how to integrate this decomposition technique with local search, and evaluate SLS/INDSET, which combines INDSET with a stochastic local search. Finally, we discuss the benefit of identifying dangling components of the decomposed constraint graph, and evaluate SLS/INDSET+DANGLES, a strategy that exploits this structural improvement.

1 Introduction

We present a technique that exploits the structure of a Constraint Satisfaction Problem (CSP) to boost performance of solving the CSP while yielding multiple solutions. This technique is based on identifying, in the constraint graph of the CSP, a *maximal independent set* I, which is a set of variables that are pairwise not connected (see Figure 1). A constraint graph with low density is likely to have a large independent set. Our technique, INDSET, partitions the variables of the CSP into the two sets, I and its complement $\bar{I}$, and restricts the search to the variables in $\bar{I}$ in order to find a solution to the CSP induced by $\bar{I}$. We extend the solution found to the variables in I by applying directional arc-consistency between the variables in $\bar{I}$ and those in I. This can be done in linear-time in the number of variables in I and the number of constraints between I and $\bar{I}$. When arc-consistency succeeds, the process yields a family of solutions for each of the values remaining in the domain of a variable in I.

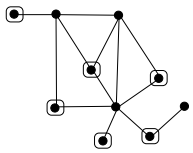

Figure 1: Circled vertices form a maximal independent set.

While *any* technique can be used to solve the variables in $\bar{I}$, we have developed and tested a method, SLS/INDSET, for using this approach in combination with stochastic local-search with steepest descent. SLS/INDSET integrates information of the constraints between $\bar{I}$ and I in order to find solutions for $\bar{I}$ that can be extended to I. We found that INDSET significantly improves the performance of SLS, and yields robust results by finding multiple solutions and returning them in a compact form.

This paper is organized as follows. Section 2 reviews background information and related work. Section 3 describes our basic decomposition technique, INDSET, and highlights its benefits. Section 4 discusses how to exploit INDSET in local search. Section 5 describes an enhancement to INDSET that further reduces the size of $\bar{I}$ and increases both the number and compactness of the solutions found. Section 6 analyzes the features of the resulting decomposition. Section 7 summarizes our contributions and identifies directions for future research.

2 Background and related work

A constraint satisfaction problem (CSP) is a tuple $\mathcal{P} = \{\mathcal{V}, \mathcal{D}, \mathcal{C}\}$, $\mathcal{V} = \{V_1, V_2, \ldots V_N\}$ is a set of N variables, $\mathcal{D} = \{D_{V_1}, D_{V_2}, \ldots, D_{V_N}\}$ is a set of domains for these variables (a domain D_{V_i} is a set of values for the variable V_i), and $\mathcal{C}$ is a set of relations on the variables resticting the allowable combination of values for variables. Solving a CSP requires assigning to each variable V_i a value chosen from D_{V_i} such that all constraints are satisfied. We denote $\mathcal{P}_X$ the CSP induced on $\mathcal{P}$ by a set $X \subseteq \mathcal{V}$ of variables. We focus here on *binary* CSPs: each constraint is a relation on at most two variables. The *tightness* t of a constraint is the ratio of the number of tuples forbidden by the constraint to the number of all possible tuples. The *constraint ratio* r, also called here constraint density, is the ratio of the number of constraints in the CSP to the number of possible constraints in the CSP. We assume that there is at most one constraint for each pair of variables. The *constraint graph* of a CSP is a graph G where each variable in the CSP is represented by a vertex in G, and each (binary) constraint in the CSP is represented by an edge in G, connecting the two corresponding vertices. A *neighbor* of a vertex (variable) V_i is any vertex that is adjacent to V_i (i.e., shares a constraint with V_i). The set of all such variables is the *neighborhood* of V_i.

An *independent set* in G is a set of vertices I such that the subgraph induced by I has no edges (i.e., a set of pairwise non-adjacent vertices). A *maximal* independent set is one that is not a subset of any larger independent set. A maximal independent set is to be distinguished from a *maximum* independent set, which is the largest an independent set in the graph. Our experiments use *random problems* generated according to Model B (Achlioptas *et al.* 1997). The parameters for the random problems are the number of variables, domain size (same for all variables), tightness (same for all constraints), and constraint ratio.

INDSET, first reported in (Gompert 2004), may be considered as one of the techniques that exploit 'strong backdoors' (Williams, Gomes, & Selman 2003). These are techniques that divide a problem into two sets of variables, search is done on one of the sets (i.e., the backdoor), and the resulting partial solution can be expanded to a full solution (or be shown to be inconsistent) in polynomial time. In our case, the complement $\bar{I} = \mathcal{V} \setminus I$ of our independent set I forms a 'strong backdoor.' Indeed, any instantiation of $\bar{I}$ leads to a linear-time solvable sub-problem $\mathcal{P}_I$, since $\mathcal{P}_I$ has no edges.

Another example of a backdoor is a cycle-cutset. In the cycle-cutset decomposition technique, CYCLE-CUTSET, one chooses a set A of variables such that removing A from the constraint graph leaves the graph acyclic (Dechter & Pearl 1987; Dechter 2003). Thus, given any assignment for A, the remaining tree can be solved in linear time with backtrack-free search (Freuder 1982). Like INDSET, CYCLE-CUTSET is also more beneficial when the constraint graph has low density, because finding a smaller cycle-cutset increases the benefit of the technique, and small cycle-cutsets are less likely to exist in dense graphs, just as large independent sets are less likely to exist in dense graphs. We compare and contrast these two decompositions in further detail in Section 6.2.

A related work, carried out in the context of SAT, partitions the set of Boolean variables in a SAT problem into 'independent' and 'dependent' variables and exploits this distinction in local search (Kautz, McAllester, & Selman 1997). The technique is heavily dependent on SAT encodings and its application to CSPs is not straightforward.

Finally, except for (Choueiry, Faltings, & Weigel 1995), none of the decomposition techniques reported in the literature discuss the 'production' of multiple solutions, a by-product of our technique. We argue that this feature of INDSET, shared to a lesser extent by CYCLE-CUTSET, sets our approach apart from the rest.

3 Basic decomposition

In this section, we show that INDSET not only improves the performance of search but enables it to return multiple solutions.

INDSET partitions the variables of a CSP into two sets I and $\bar{I}$, such that I is a maximal independent set of the constraint graph. By definition, no two variables in I are neighbors, and, because I is maximal, every variable v in $\bar{I}$ has a neighbor in I. Otherwise, we could move v from $\bar{I}$ to I to obtain a larger independent set $\bar{I}$. (Incidentally, $\bar{I}$ forms a minimal vertex cover on the constraint graph.) Let $\mathcal{P}_I$ be the subgraph of the constraint graph induced by I and $\mathcal{P}_{\bar{I}}$ be the subgraph induced by $\bar{I}$.

Now, if we choose a consistent assignment for the variables in $\bar{I}$, then the values of the variables in I that are consistent with this assignment are determined by directional arc-consistency on I, and can be found in linear time. For a given variable v in I, we know that the neighborhood of v lies entirely within $\bar{I}$, thus, by choosing an assignment for $\bar{I}$, we have fixed the values of all the neighbors of v, and we can quickly find all consistent values for v. Any of these consistent values could be used to form a solution. For a given assignment of $\bar{I}$, and for each variable v_i in I, let d_i be the set of values in the domain of v_i that are consistent with the current assignment of $\bar{I}$. For any two variables v_i, v_j in I, we have $|d_i| \cdot |d_j|$ valid combinations, since there is no constraint between v_i and v_j. In fact, every element of the Cartesian product $d_1 \times d_2 \times \cdots \times d_{|I|}$ is an expansion of the assignment of $\bar{I}$ to a full solution. Thus, for any assignment of $\bar{I}$ we can quickly obtain all $\prod_i |d_i|$ solutions possible with that assignment, and, since the result is the Cartesian product of subsets of domains, this possibly large set of solutions is represented in a compact form.

Consider the following example. Figure 2 shows a CSP decomposed using an independent set I and its complement $\bar{I}$. Figure 3 shows the result after instantiating the variables

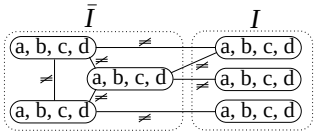

Figure 2: Example CSP.

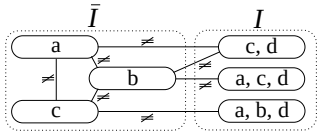

Figure 3: CSP after instantiating $\bar{I}$.

in $\bar{I}$ and then performing directional arc-consistency on I. The domains in I become $\{c, d\}$, $\{a, c, d\}$, and $\{a, b, d\}$. And the set of all possible remaining solutions consists of the Cartesian product of these three sets, along with the instantiation on $\bar{I}$. Thus we have found $2 \times 3 \times 3 = 18$ solutions.

Again, in INDSET, we restrict search to the variables in $\bar{I}$, and, for a solution of $\mathcal{P}_{\bar{I}}$, we can find all the resulting solutions using directional arc-consistency. This process reduces the search space by a factor exponential in the size of I. Consequently, we would like to choose an independent set I as large as possible. Finding the maximum independent-set of a graph is **NP**-hard (Garey & Johnson 1979). However, we do not need the maximum independent-set for this technique to be beneficial. Fortunately, many efficient approximation algorithms exist for finding independent sets (Boppana & Halldórsson 1990). Choosing an approximation algorithm depends on how much time one is willing to spend finding a large independent set.

For finding independent sets, we used the CLIQUEREMOVAL algorithm which runs in polynomial time in the number of variables (Boppana & Halldórsson 1990). For the problem instances we used in our experiments, CLIQUEREMOVAL takes negligible time to execute (less than the clock

resolution, which 10 msec).

4 Using INDSET with local search

Exploiting independent sets is straightforward for systematic backtrack search, since we can obtain a performance benefit by searching over the variables of $\bar{I}$ before those of I. However, in general, it is less clear how decomposition and/or structural information can be used to improve stochastic search (Kautz, McAllester, & Selman 1997). Therefore, we focus our investigations on how to use INDSET in conjunction with local search. Our solution is to guide the local search on $\bar{I}$ with information from the constraints between I and $\bar{I}$.

4.1 Local search

Local search designates algorithms that start with a random assignment and make incremental changes to it in an attempt to converge on a solution. The basic local-search technique makes the greedy incremental changes that best improve some evaluation function. A common evaluation function is the number of broken constraints in the assignment. We examine the case of stochastic local search (SLS) with a "steepest descent" heuristic, and empirically evaluate the improvement obtained from combining it with INDSET.

We consider Algorithm 1, the SLS algorithm as described in (Dechter 2003).

Algorithm 1 SLS

```
for i = 1 to MAX_TRIES do
    make a random assignment to all variables
    repeat
        if no broken constraints then
            return current assignment as a solution
        end if
        choose variable-value pair x_i, a_i that most reduces
        number of broken constraints when x_i ← a_i
        x_i ← a_i
    until no further changes occur
end for
```

This algorithm starts with a random initial assignment to the variables, and then makes incremental changes until it finds a consistent assignment. At each step, SLS considers all variable-value pairs in the problem and evaluates, for each variable-value pair, the number of broken constraints resulting from changing the assignment of the variable to the value in the pair, and chooses the pair that yields the minimum number of broken constraints breaking ties lexicographically. This heuristic is called the "steepest descent" (Galinier & Hao 1997). This atomic step is repeated until either a solution is found or no improvement is possible, in which case the search is restarted from a new random assignment. The process is repeated for a specified number of restarts or a given time duration unless a solution is found.

4.2 SLS/INDSET

We extend SLS into SLS/INDSET, which performs SLS on $\bar{I}$, and is guided by the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. We incorporate the information about the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$ by modifying the method by which SLS counts broken constraints.

When counting broken constraints, SLS/INDSET also includes some measurement of the number of broken constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. Whether or not a particular constraint between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$ is broken is a vague concept. As a result of this imprecision, there is more than one way to consider these constraints to be broken or not. For two variables in $\bar{I}$, the situation is clear: values have been assigned to them and these values either do or do not violate a constraint. For a constraint between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$, we have an assignment of the variable v in $\bar{I}$, but not to the variable u in I. There may be multiple values for u which support the assignment to v. Even if there are multiple values in u supported by every constraint on u, it is still possible for directional arc-consistency on u to annihilate its domain. Below we discuss five ways to measure the brokenness of the constraints on u.

4.3 Counting broken constraints

When designing a criterion for determining whether the constraints are broken or not, we would like to maintain the property that the number of broken constraints is zero if and only if we have at least one consistent solution to the CSP. Once we have no broken constraints, then, given the assignment on $\bar{I}$ and the filtered domains of the variables in I, we obtain at least one solution, and usually a large number of solutions. We implemented and tested five ways to count the number of broken constraints: `None`, `Zero-domain`, `Some`, `All`, and `PrefRelax`.

`None`: The simplest approach is to simply ignore $\mathcal{P}_I$ and the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$ (see Figure 4). We perform search solely to find a solution to $\mathcal{P}_{\bar{I}}$, and then we check whether this partial solution extends to a full solution. If not, then we restart search again on $\mathcal{P}_{\bar{I}}$ with a new random initial assignment. Note that this heuristic does not maintain the property outlined above (i.e., the number of broken constraints may be zero even when we have not found a solution to the CSP). One would expect that this approach performs poorly. We include this heuristic in our experiments solely as a frame of reference.

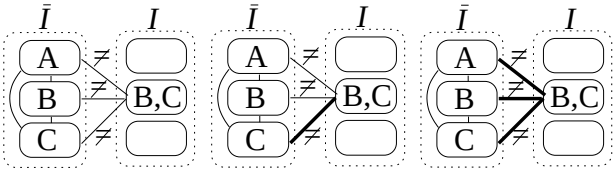

Figure 4: `None`.

Figure 5: `Some`.

Figure 6: `All`.

In the remaining methods, we filter the domains of the variables in I, given the instantiation of $\bar{I}$ and the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$. This directional arc-consistency may leave some of the variables in I with empty domains.

`Zero-domain`: In this heuristic we simply add the resulting number of variables with empty domains to the number of broken constraints in $\mathcal{P}_{\bar{I}}$. On random problems, this method performed worse than SLS alone. In special cases

SLS/INDSET(`Zero-domain`) did outperform SLS. In star graphs (West 2001), for example, it is obvious that any use of independent set information will yield an improvement, because it is easy to find an independent set containing $n-1$ of the variables. INDSET allows us to focus the search on the single, center variable that really affects the problem. SLS alone will spread its search across the entire star graph, wasting much of its effort. In trivial cases like this, even the poor-performing `zero-domain` significantly outperforms SLS. Because this heuristic performed poorly in preliminary experiments on random CSPs, we will not discuss it further.

`Some`: In this method, we iterate through each of the constraints between $\mathcal{P}_I$ and $\mathcal{P}_{\bar{I}}$ (see Figure 5). Consider one such constraint $C_{u,v}$ with u being the variable in I and v being the variable in $\bar{I}$. We reduce the domain of u to those values allowed by the constraint, given the value currently assigned to v. Each successive constraint on u may further reduce the domain of u. For each constraint, if this filtering annihilates the domain of u, then we consider the constraint to be broken. Any other constraint on u that we consider afterwards is also considered to be broken. Note that the value returned by this heuristic depends on the order in which we examine the constraints on u.

`All`: In this next method, we filter the domains in $\mathcal{P}_I$ and then, for each variable left with an empty domain, we consider *all* of the constraints on that variable to be broken (see Figure 6). Thus, we include in the count of broken constraints the sum of the degrees of the variables of I left with empty domains.

`PrefRelax`: We also attempted a heuristic using preferred relaxations of (Junker 2004). A relaxation is a subset of constraints that has a solution. For each variable v in I, we found the preferred relaxation R of the constraints on v, given the lexicographical order of the constraints. $|D_v|-|R|$ was used as the measurement of the number of broken constraints on v. `PrefRelax` is a more accurate measurement of the number of broken constraints. Even better would be to find the *maximum* relaxation of the constraints. However, computing the maximum relaxation requires exponential time in the worst case. For the problem sets in our experiments, `PrefRelax` did not provide a significant improvement over `Some`, likely due to the sparseness of the problems. Further investigations may reveal an improvement of `PrefRelax` on denser graphs. Also, heuristics for ordering the constraints on v may improve performance of `PrefRelax` as well as `Some`

Note that these heuristics are applicable not only for the combination of INDSET with SLS but also for that of INDSET with other iterative repair algorithms.

4.4 Empirical evaluation

We empirically compared the performance of SLS with that of SLS/INDSET with three of the heuristics described in Section 4.3. We tested each algorithm on random CSP instances (Model B), with 80 variables, 8 values, a constraint tightness of 58%, varying the number of constraints around the phase transition. For the problems used in these experiments, the CLIQUEREMOVAL algorithm found independent sets of size 36, on average, varying with density. For each point, we tested each algorithm on over 1000 instances with a cutoff time of 120 seconds per instance. We report in Figure 7 the percentage of instances each algorithm solved.

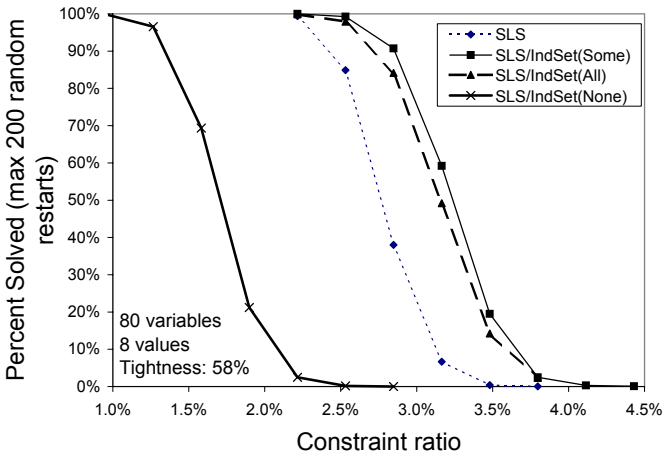

Figure 7: Percentage solved for SLS and SLS/INDSET.

Note that the curves for the different algorithms/heuristics have similar shapes but appear shifted to the left or right. The curves farther to the right correspond to solving a larger percentage of the problems. As a curve shifts to the right, it approaches the similar curve representing the percentage of problems that are actually solvable, corresponding to the phase transition. It is not feasible to compute the actual phase-transition curve for large size problems because a complete solver is needed. Consequently, we cannot easily determine how closely the algorithms tested approach the actual curve.

It is clear from the graphs that `None`, as expected, performs poorly. It is surprising however that it performs as well as it does, considering that it is merely stumbling around in the dark by ignoring a large number of variables and constraints. The best-performing algorithm is SLS/INDSET using `Some`, although `All` is not far behind.

Finally, SLS/INDSET, with the various heuristics, returns a large number of solutions on average. In general, the algorithm returns an average of about three values per variable in I. If the independent set contains 30 variables, then the number of solutions obtained is approximately 3^{30}.

The runtime for SLS and SLS/INDSET are shown in Figures 8, 9, and 10. Each graph increases density until SLS becomes unable to solve most of the problems. Note that SLS alone sometimes runs faster on problems with low tightness and density. Solving these problems is usually simple enough that the overhead of SLS/INDSET unnecessary. INDSET provides greater benefits closer to the phase transition area.

5 Identifying dangling trees

We can do some additional processing to enhance INDSET and extend its benefits. We propose to do so by detecting trees that 'dangle' off the variables in I. For a variable u in I, we find connected, acyclic, induced subgraphs that

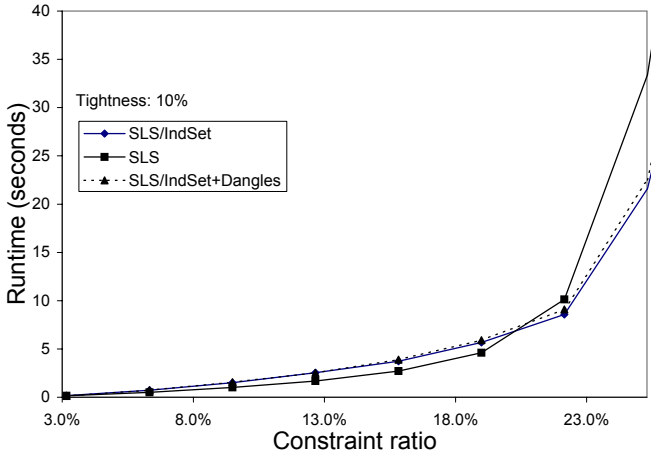

Figure 8: Runtime for $t = 10\%$.

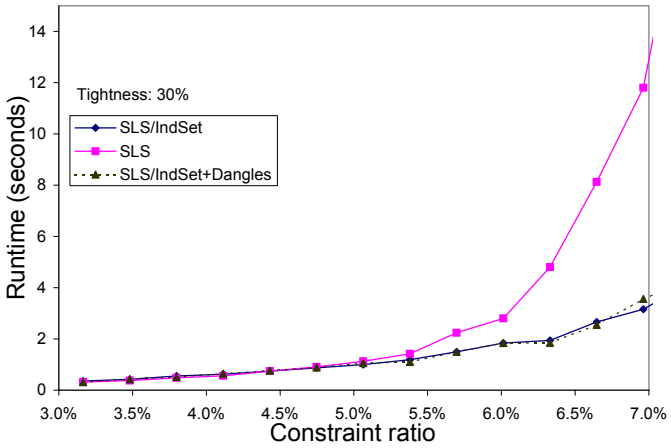

Figure 9: Runtime for $t = 30\%$.

become disconnected from the rest of the graph if u is removed. These subgraphs are trees said to 'dangle' from u. We can find these dangling trees quickly, using a linear-time breadth-first search. Algorithm 2, FINDDANGLESONVAR, finds the set of variables in trees dangling off a given variable. An example of extracting these dangling trees is shown in Figure 11. The graph on the left is a CSP decomposed using the independent set I. The graph on the right shows the result after identifying the dangling trees. Note that some of the vertices in the dangling trees were removed from I and some from $\bar{I}$, yielding $I' \subseteq I$ and $C \subseteq \bar{I}$ respectively. Let T be the set of variables in the trees dangling off I'. Algorithm 3, GETTREE, performs a breadth-first search starting with v_2, without searching past v_1, and stopping if a cycle is found. In the worst case, this algorithm requires time linear in the number of variables in the subgraph containing v_2 after v_1 is removed. Thus, in the worst case, it requires time linear in the number of variables in the graph ($O(n)$). FINDDANGLESONVAR requires time $O(n)$, because we can have mark GETTREE nodes as it searches, to ensure that we never

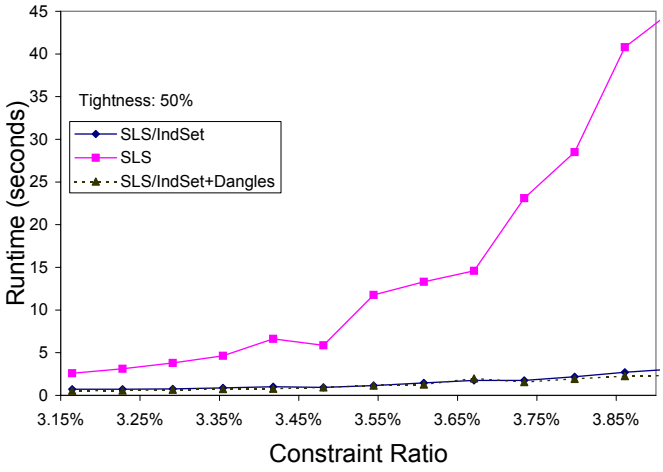

Figure 10: Runtime for $t = 50\%$.

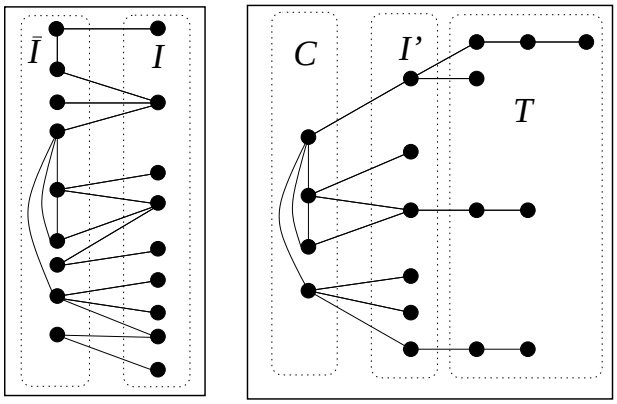

Figure 11: Dangling trees.

visit a node twice.

Algorithm 2 FINDDANGLESONVAR(v).

```
result ← ∅
for each neighbor of v do
    result ← result ⋃ GETTREE(v, neighbor)
end for
return result
```

Suppose we enforce directional arc-consistency, from the leaves of the dangling trees towards the variables in I'. If any domains are annihilated, then we know immediately, and *before any search is done*, that the problem is not solvable. Also, any selection of a value remaining in the domain of a variable in I' can be extended to a solution of the trees dangling off that variable. Now, given an assignment on $\mathcal{P}_C$, since any two variables in I' (and their respective dangling trees) are disconnected from each other, we can select their values independently of each other.

Furthermore, we can completely ignore the nodes in T during search, because we know that any value that remains

Algorithm 3 GETTREE(v_1, v_2).

Require: v_1 and v_2 are adjacent variables.
Ensure: Return set of variables of the tree rooted at v_2 dangling off v_1; otherwise return $\emptyset$

```
result ← v1
stack ← v2
while NOTEMPTY(stack) do
    nextvar ← POP(stack)
    N ← NEIGHBORSOF(nextvar)
    I ← result ⋂ N
    if SIZEOF(I) ≠ 1 then
        return ∅ {We found a cycle}
    end if
    N ← N \ I
    result ← result ⋃ nextvar
    stack ← stack ⋃ N
end while
return result \{v1}
```

in the domain of a variable in I has a support. Thus, if we find a partial solution for the variables in C and if directional arc-consistency can successfully extend this partial solution to the variables I', then we know that this partial solution can necessarily be extended to at least one full solution in a backtrack-free manner (Freuder 1982). We can also determine a lower bound on the number of solutions as described in Section 6.1. In summary, we can perform search using an algorithm like SLS/INDSET on C and I' just as we did before on $\bar{I}$ and I, respectively, and ignore the vertices in T. Thus, identifying dangling trees reduces our search space because $|C| \leq |\bar{I}|$. It also reduces the cost of the filtering step at each iteration because $|I'| \leq |I|$ and the number of constraints between C and I' is smaller than the number of constraints between I and $\bar{I}$. Figure 12 shows the effect of dangle identification on reducing the number of variables that search needs consider.

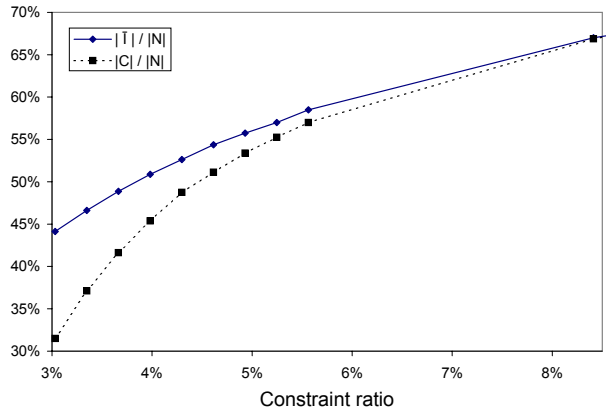

Figure 12: Size of C and $\bar{I}$ relative to the number of variables.

We implemented SLS/INDSET+DANGLES based on SLS/INDSET. Because using the dangling trees requires a preprocessing step of at least directed arc-consistency from the leaves of the trees to the roots, we decided to apply (full) arc-consistency as a preprocessing step to our experiments. For a fair comparison, we perform arc-consistency before each algorithm is executed. Because arc-consistency itself can find a problem instance to be unsolvable, we kept only randomly generated problems that can be made arc-consistent and considered 1000 such instances per point.

Figures 13 and 14 show the results of comparing SLS, SLS/INDSET, and SLS/INDSET+DANGLES using `Some` and at two different values of tightness.

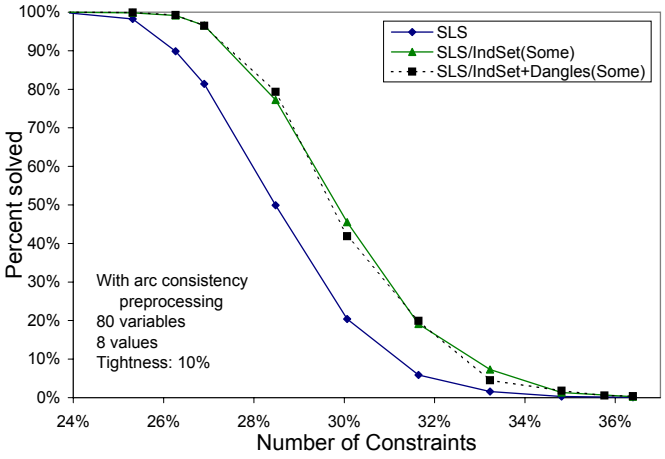

Figure 13: Percentage solved for SLS/IndSet+Dangles.

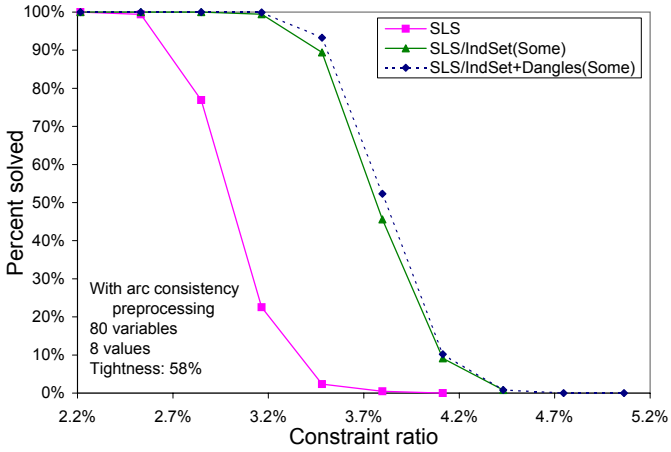

Figure 14: Percentage solved for SLS/IndSet+Dangles.

For the lower tightness (i.e., Figure 13), the fall-off curve appears with a larger number of constraints, and the distinction between SLS/INDSET, and SLS/INDSET+DANGLES disappears. The improvement of SLS/INDSET+DANGLES over SLS/INDSET is more visible at tightness 58% (Figure 14). For this tightness, the average CPU time of SLS/INDSET was up to twice that of SLS/INDSET+DANGLES, for these experiments. In conclusion, identifying trees dangling off the independent-set variables of sparse graphs further focuses the efforts

of search, and reduces the amount of time required per iteration of search. Further, by removing some variables from $\bar{I}$, DANGLES may increase the number of solutions found by search.

Like SLS/INDSET, SLS/INDSET+DANGLES has a greater benefit closer to the phase transition area. Intuitively, it also has a greater benefit for lower-density graphs, which have a higher probability of having larger dangles. Indeed, we find that SLS/INDSET+DANGLES provides the greatest benefit when the phase transition is found in lower-density graphs, which occurs as tightness increases. An example is shown in Figure 15, where the runtime cumulative distribution shows SLS/INDSET+DANGLES dominating SLS/INDSET for these tight, sparse problems.

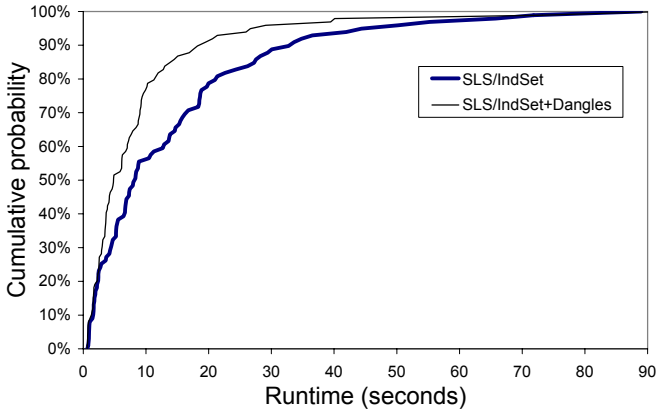

Figure 15: Runtime Cumulative Distribution for $t = 60\%$ and $r = 3.5\%$.

6 Analysis

Below, we report some qualitative analysis of INDSET. First, we explain how our approach allows us to compute lower bound on the solutions extendible from a given partial assignment. Then we re-discuss, this time in greater detail, the relationship of our technique with CYCLE-CUTSET.

6.1 Computing lower bounds of the number of solutions

After instantiating the variables C, the nodes in I' become the roots of a forest (see Figure 11). Each of the trees in this forest can be solved independently of the others, which increases the robustness of our solutions because modifying the solution to one tree does not affect the others. Note that CYCLE-CUTSET does not necessarily yield such independent trees, and when it does, recognizing them requires additional effort.

In addition to improving the performance of problem solving and returning multiple solutions, INDSET allow us to compute a lower bound of the number of solutions that can be extended from the partial assignment.

After instantiating the variables in C, when performing directed arc-consistency (DAC) on I' does not annihilate the domain of any of the variables in I', then the number of solutions to the CSP is bounded from below by the product of the size of the largest domain of the trees rooted in I'. Indeed, in each tree, we know that any value of each domain is part of some solution. Thus, we can choose any value for any of the variables in a tree and extend it to a full solution of the tree using backtrack-free search. Consequently, the number of solutions must be at least the size of the largest domain in the tree. Furthermore, since each of the trees can be solved independently, the product of the maximum domains of all the trees gives us a lower bound on the number of solutions we have obtained for the entire CSP.

Another possible method for obtaining perhaps an even better lower bound is to perform backtrack-free search on each tree, using bundling (Haselböck 1993)[1]. At each step, we choose the largest bundle. Each element of the cross product of the domain bundles gives a solution to the tree. Thus, the product of the sizes of the bundles chosen gives us a lower bound on the number of solutions in the tree. The product of these lower bounds of each tree again gives us a lower bound on the number of solutions for the original CSP.

6.2 Comparison to cycle-cutset decomposition

INDSET and INDSET+DANGLES are both special cases of CYCLE-CUTSET, because they both identify a subset of vertices that, when instantiated, leave the rest of the constraint graph acyclic. However, both of these decompositions go further than the general CYCLE-CUTSET of (Dechter & Pearl 1987). INDSET goes further than leaving the graph acyclic, it leaves the graph with no edges at all. The particularity of INDSET+DANGLES with respect to CYCLE-CUTSET is a little less obvious.

In CYCLE-CUTSET, removing the cutset nodes leaves the constraint graph acyclic. The remaining graph may have multiple components, each a tree. These trees may be connected to the cycle-cutset in multiple places. In contrast, INDSET+DANGLES leaves trees, each of which has *at most* one vertex adjacent to the cutset. This fact results in the advantage of allowing us to ignore all but one vertex of the tree when performing search. On the other hand, in CYCLE-CUTSET, a tree that has even two vertices adjacent to the cutset greatly complicates matters. We can no longer afford to only look at those vertices adjacent to the cutset, because there may be a cycle in the graph going through the tree. In this case, *arc-consistency is not enough* to guarantee our independent choices for values of the neighborhood of the cutset.

Another issue of importance is that by going further than CYCLE-CUTSET (i.e., cutsets that leave the graph with even less connectivity), our cutsets are going to be larger than the ones CYCLE-CUTSET. This tradeoff is worth exploring. In fact, work has been done exploring the other direction, finding and using cutsets that leave the graph with more connectivity with the benefit of having smaller cutsets (e.g., w-cutsets (Bidyuk & Dechter 2004)).

[1]Note that static (Haselböck 1993) and dynamic (Beckwith, Choueiry, & Zou 2001) bundling on trees yield the same result.

6.3 Heuristics and DANGLES

We compared the effect of using the different heuristics (of applying INDSET to local search) to the effect of using DANGLES. The runtime comparison is shown in Figure 16. SLS/INDSET+DANGLES(`All`) performs worse on average

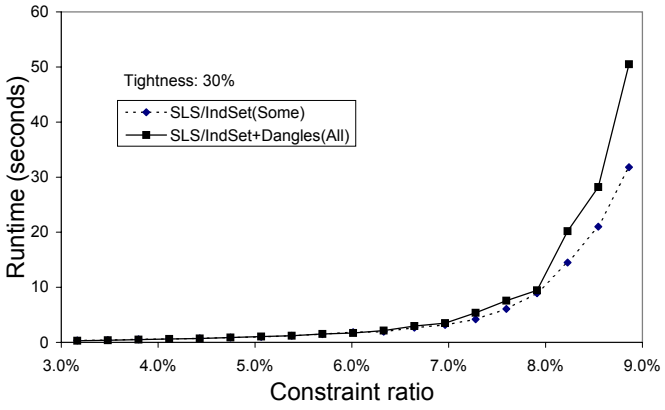

Figure 16: Comparison of heuristic to use of DANGLES.

than SLS/INDSET(`Some`). From Section 4.3, we know that `Some` performs better than `All`, and we know that using DANGLES performs better than not using DANGLES, thus, from Figure 16 it appears that the benefit of `Some` over `All` is greater than the benefit of DANGLES. Consequently, the choice of heuristic has a greater influence than whether or not DANGLES is used.

7 Conclusions

Our results demonstrate that finding a large independent set in the constraint graph and using it to decompose the CSP improves the performance of solving the CSP. An additional benefit of our approach over other decomposition techniques is that it inherently provides us with many solutions, at no extra cost beyond that incurred by the search process, and the multiple solutions are represented in a compact form. We can gain further improvement by identifying and extracting trees that dangle off the independent set. Additional information can be reaped from these trees, such as estimating a lower bound on the number of solutions obtained. Future investigations include the following:

1. Investigate the effect of the size of the independent set on the performance of the technique.
2. Perform experiments on structured problems such as clustered graphs (Hogg 1996).

Acknowledgments

We express gratitude to Stephen D. Kachman in the Department of Biometry and Bradford Danner in the Department of Statistics for their guidance. This research is supported by NSF CAREER Award #0133568. Experiments were conducted on the Research Computing Facility of the University of Nebraska-Lincoln.

References

Achlioptas, D.; Kirousis, L. M.; Kranakis, E.; Krizanc, D.; Molloy, M. S.; and Stamatiou, Y. C. 1997. Random Constraint Satisfaction: A More Accurate Picture. In *Proc. of CP*, volume 1330 of *LNCS*, 107–120. Springer.

Beckwith, A. M.; Choueiry, B. Y.; and Zou, H. 2001. How the Level of Interchangeability Embedded in a Finite Constraint Satisfaction Problem Affects the Performance of Search. In *Advances in Artificial Intelligence, LNAI 2256*, 50–61. Springer.

Bidyuk, B., and Dechter, R. 2004. On Finding Minimal w-Cutset Problem. In *Proceedings of the Conference on Uncertainty in AI (UAI 04)*.

Boppana, R., and Halldórsson, M. M. 1990. Approximating maximum independent sets by excluding subgraphs. In *2nd Scandinavian Workshop on Algorithm Theory*, volume 447, 13–25.

Choueiry, B. Y.; Faltings, B.; and Weigel, R. 1995. Abstraction by Interchangeability in Resource Allocation. In *Proceedings of IJCAI*, 1694–1701.

Dechter, R., and Pearl, J. 1987. The Cycle-Cutset Method for Improving Search Performance in AI Applications. In *Third IEEE Conference on AI Applications*, 224–230.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Freuder, E. C. 1982. A Sufficient Condition for Backtrack-Free Search. *JACM* 29 (1):24–32.

Galinier, P., and Hao, J.-K. 1997. Tabu search for maximal constraint satisfaction problems. In *CP 97*, 196–208.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W J Freeman and Co.

Gompert, J. M. 2004. Local Search With Maximal Independent Sets. In *Proceedings of CP*, volume 3258 of *LNCS*, 795. Springer.

Haselböck, A. 1993. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proceedings of IJCAI*, 282–287.

Hogg, T. 1996. Refining the phase transition in combinatorial search. *Artificial Intelligence* 81:127–154.

Junker, U. 2004. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. *AAAI* 167–172.

Kautz, H.; McAllester, D.; and Selman, B. 1997. Exploiting variable dependency in local search. *Poster session abstract, IJCAI*.

West, D. B. 2001. *Introduction to Graph Theory*. Prentice Hall, second edition.

Williams, R.; Gomes, C. P.; and Selman, B. 2003. Backdoors to typical case complexity. In *IJCAI 03*.