CSP-104: Constraint Propagation for Loose Constraint Graphs

ABSTRACT

In this paper we investigate whether we can improve propagation-based finite domain constraint solving by making use of the constraint graph to choose propagators to execute in a better order. If the constraint graph is not too densely connected we can build an underlying tree of bi-connected components, and use this to order the choice of propagator. As search progresses forward, the constraint graph becomes less and less strongly connected, so if we can determine the bi-connected components of the dynamically shrinking constraint graph we have more scope of making use of the technique. Our experiments show that there exist problems where handling bi-connected components can substantially improve the propagation performance.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classfications—*Constraint and logic langauges*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*

General Terms

Algorithms

Keywords

Constraint propagation, constraint graph

1. INTRODUCTION

Finite domain constraint propagation tackles constrained satisfaction and optimization by interleaving constraint propagation, which removes impossible values from the domains of variables, with search. The constraint propagation step is a fixpoint computation, which continually applies propagators until no further changes in domains result. In this paper we investigate how to improve the calculation of this fixpoint when the constraint graph is not highly connected.

SAC '07 Seoul, Korea

We begin with an example to motivate why it may be worthwhile taking account of the constraint graph during the fixpoint calculation.

Example 1. Consider a system of constraints $x_{i+1} = x_i + 1, 0 \leq i < n$, where the initial domain $D(x_i)$ of each variable x_i is $[0 \,..\, 2n]$. Consider a LIFO based propagation queue. Initially all propagators are on the stack. We take the top $x_1 = x_0 + 1$, we revise the domains $D(x_1) = [1 \,..\, 2n]$, $D(x_0) = [0 \,..\, 2n-1]$. Since the domain of x_1 and x_0 are changed we add all other propagators for constraints mentioning x_1 back onto the stack ($x_2 = x_1 + 1$ is the only one and its already in). Next we pop $x_2 = x_1 + 1$ and revise the domains $D(x_2) = [2 \,..\, 2n]$, $D(x_1) = [1 \,..\, 2n-1]$. This pushes $x_1 = x_0 + 1$ back on the stack which is then selected obtaining $D(x_0) = [0 \,..\, 2n-2]$ but no other constraints mention x_0. Next we pop $x_3 = x_2 + 1$ obtaining $D(x_3) = [3 \,..\, 2n]$, $D(x_2) = [2 \,..\, 2n-1]$, which pushes $x_2 = x_1 + 1$, which is then popped obtaining $D(x_1) = [1 \,..\, 2n-2]$. This pushes $x_1 = x_0 + 1$ which is popped obtaining $D(x_1) = [1 \,..\, 2n-3]$. The process continues popping $x_4 = x_3 + 1$. This pattern of execution requires $O(n^2)$ propagator executions to reach fixpoint.

Consider a FIFO based propagation queue. We evaluate each of $x_{i+1} = x_i + 1$ in order for $0 \leq i < n$, obtaining final domains $D(x_i) = [i \,..\, 2n-1], 0 \leq i < n$, and each contraint except $x_n = x_{n-1} + 1$ is added to the queue. We then evaluate each of these in turn obtaining $D(x_i) = [i \,..\, 2n-2], 0 \leq i < n-1$, ans each constraint except the last two is back on the queue. Again the pattern of execution requires $O(n^2)$ propagator executions to reach fixpoint.

Consider the same system where we propagate $x_1 = x_0+1$, then $x_2 = x_1 + 1$, ..., $x_n = x_{n-1} + 1$ and then reverse $x_{n-1} = x_{n-2} + 1$, ..., $x_2 = x_1 + 1$, $x_1 = x_0 + 1$. This gives the same fixpoint with $O(n)$ propagator executions.

What is so special about the good order of propagation in the above example. The reason is that the constraint graph is a tree and the order corresponds to an in-order traversal of the tree. In general though constraint graphs are not trees, they are highly connected, so how can we take advantage of this. The principal idea of this work is to decompose the constraint graph into bi-connected components and then apply the in-order propagation strategy on the tree of bi-connected components.

Often a constraint graph will consist of only one bi-connected component since a non-bi-connected component means fairly independent parts of the constraint graph, so the question

is how often is this useful. We claim there are a number of circumstances when the approach is useful:

- In some applications such as test data generation we have to solve very simple constraint problems very many times. In these cases the constraint graph is often loosely connected.
- Even when the constraint graph is totally bi-connected, it may be only made so by propagators at low priority. We can apply the technique to the high priority propagators since they need to reach fixpoint before the low priority propagators are executed.
- As propagation continues variables are fixed and there nodes are removed from the constraint graph, hence the graph will become more an more loosely connected as search progresses. If we can efficiently dynamically calculate bi-connected components we can make use of the approach.

In this paper we define an efficient queuing mechanism for selecting propagators in an in-order traversal of the bi-connected components of the constraint graph. We show that for certain classes of problem this is can improve propagation speed substantially. The remainder of the paper is organized as follows. In the next section we introduce our notation for constraints and propagation, as well as constraint graphs, bi-connectedness and tree traversal. Then in Section 3 after briefly introducing the fixpoint algorithm for constraint propagation we define a priority queue to select the propagators in the appropriate order, without undue overhead. In Section 4 we give some preliminary results of using the method. In Section 5 we talk about related work and conclude.

2. PRELIMINARIES

This section defines terminology and the basic components of a constraint propagation engine. In this paper we restrict ourselves to integer constraint solving.

2.1 Valuations and Constraints

An *integer valuation* θ is a mapping of variables to integer values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. We extend the valuation θ to map expressions and constraints involving the variables in the natural way.

Let *vars* be the function that returns the set of variables appearing in a valuation. A *constraint* c over variables $x_1, \ldots, x_n$ is a set of valuations θ such that $vars(\theta) = \{x_1, \ldots, x_n\}$. We also define $vars(c) = \{x_1, \ldots, x_n\}$.

The *constraint graph* $G = (N, E)$ for a conjunction of constraints $c_1 \wedge \cdots \wedge c_m$ is a bipartite (undirected) graph with nodes $N = N_C \cup N_V$ where the *constraint nodes* $N_C = \{c_1, \ldots, c_m\}$ and the *variable nodes* $N_V = \cup \cup_{i=1}^{m} vars(c_i)$, and the edges are $E = \{(c_i, x_j) \mid x_j \in vars(c_i)\}$.

2.2 Domains

We shall use range notation $[l \,..\, u]$ to define the set of integers $\{d \mid l \leq d \leq u\}$. A *domain* D is a complete mapping from a fixed (countable) set of variables $\mathcal{V}$ to finite sets of integers. A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all $x \in \mathcal{V}$.

A *(constraint satisfaction) problem* is a conjunction of constraints C where $vars(C) \subseteq \mathcal{V}$ with an initial domain D.

Example 2. The SEND+MORE=MONEY problem encoded using carry variables is expressed as $D(S) = S(M) = [1 \,..\, 9]$, $D(E) = D(N) = D(D) = D(O) = D(R) = D(Y) = [0 \,..\, 9]$ $D(C_1) = D(C_2) = D(C_3) = D(C_4) = [0 \,..\, 1]$ $e_1 \equiv D + E = Y + 10 \times C_1$, $e_2 \equiv C_1 + N + R = E + 10 \times C_2$, $e_3 \equiv C_2 + E + O = N + 10 \times C_3$, $e_4 \equiv C_3 + S + M = O + 10 \times C_4$, $e_5 \equiv C_4 = M$, and `alldifferent`([S,E,N,D,M,O,R,Y]). The constraint graph for the linear constraints is shown in Figure 1.

2.3 Propagators

We will *implement* a constraint c by a set of propagators $prop(c)$ which map domains to domains. We extend the *vars* function so that $vars(f) = vars(c)$ for $f \in prop(c)$. A *propagator* f is a monotonically decreasing function from domains to domains: $f(D) \sqsubseteq D$, and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$. A propagator f is *correct* for a constraint c iff for all domains D $\{\theta \in c \mid \forall x \in vars(c).\theta(x) \in D(x)\} = \{\theta \in c \mid \forall x \in vars(c).\theta(x) \in f(D)(x)\}$, that is it does not remove solutions of c. This is a very weak restriction, for example the identity propagator is correct for all constraints c.

A propagator f is *idempotent* if $f(D) = f(f(D))$ for all domains D. That is, applying f to any domain D yields a fixpoint of f. We will assume all propagators are idempotent for simplicity.

2.4 Bi-connectedness and in-order traversal

Given a graph $G = (N, E)$, a subset S of the nodes N is a *bi-connected component* (BCC) if S is a maximal set of nodes that are connected in each graph $G(e) = (N, E - \{e\})$ where $e \in E$. A node n that occurs in two bi-connected components are called *cut nodes*. Algorithms for determining bi-connected components [7] are based on a depth-first traversal of the graph G and require $O(|N| + |E|)$ time.

A graph G is a *tree* iff each node forms a unique bi-connected component, or equivalently there are no cycles. The bi-connected component tree $B(G)$ of a graph $G = (N_G, E_G)$ is defined as follows. Let $N_S = \{S \mid S$ is a bi-connected component of $G\}$ the BCCs of G and $N_C = \{n \mid n \in S_1 \wedge n \in S_2 \wedge \{S_1, S_2\} \subseteq N_S\}$ the cut nodes. Then the nodes of $B(G)$ are $N_S \cup N_C$, and the edges are $E = \{(S_1, S_2) \mid \exists (n_1, n_2) \in E_G, n_1 \in S_1 - N_C, n_2 \in S_2 - N_C\} \cup \{(n, S) \mid \exists (n, n_1) \in E_G, n_1 \in S - N_C\} \cup \{(n, S) \mid n \in S\}$.

Example 3. Consider the constraint graph G shown in Figure 1(a). The bi-connected components of the graph are $b_1 \equiv \{D\}$, $b_2 \equiv \{Y\}$, $b_3 \equiv \{e_1, E, C_1, e_2, C_2, e_3\}$, $b_4 \equiv \{R\}$ $b_5 \equiv \{e_3, O, C_3, e_4\}$, $b_6 \equiv \{S\}$, $b_7 \equiv \{e_4, M, C_4, e_5\}$. Note how (cut) nodes, such as e_3 and e_4, can be in more than one bi-connected component. The bi-connected components form a tree $B(G)$ as illustrated in Figure 1(b). The *cut nodes* that are part of multiple bi-connected components are shown as boxed nodes. A triple edge indicates a cut node is part of the adjacent bi-connected component (the third class of edge above).

An *in-order traversal* of a tree $G = (N, E)$ starting from node $n_0 \in N$ is defined as a cycle P of ordered edges $e_0 = (n_0, \_), \ldots, e_{2|E|-1} = (\_, n_0)$ in E where $\exists n.e_i = (\_, n) \wedge e_{i+1} = (n, \_), 0 \leq i < 2|E|-1$ and whenever $e_i = (n, n'), e_j = (n', n), 0 \leq i < j \leq 2|E|-1$ then all other edges e adjacent to n' appear in $\{e_{i+1}, \ldots, e_{j-1}\}$. We can assign visitation numbers $P(n)$ to each node $n \in N$ as follows: $P(n) = \{i \mid e_i =$

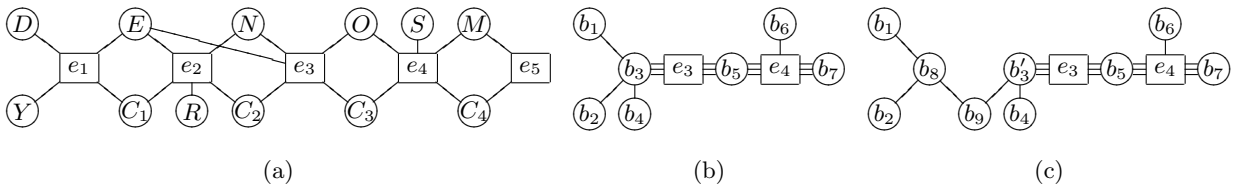

Figure 1: (a) Constraint graph for linear constraints of SEND+MORE=MONEY, (b) tree decomposition of bi-connected components, and (c) tree decomposition after removing E.

$(n, \_)\}$. Note that $|P(n)|$ is the degree of n. Note that the traversal P is cyclic so the sequence $[m \mod 2|E| \mid m \geq 0]$ corresponds to an infinite cylic traversal of the tree.

Example 4. An in-order traversal P of the tree shown in Figure 1(b) starting from b_1 is (b_1, b_3), (b_3, e_3), (e_3, b_5), (b_5, e_4), (e_4, b_6), (b_6, e_4), (e_4, b_7), (b_7, e_4), (e_4, b_5), (b_5, e_3), (e_3, b_3), (b_3, b_2), (b_2, b_3), (b_3, b_4), (b_4, b_3), (b_3, b_1). The visitation numbers are $P(b_1) = \{0\}$, $P(b_2) = \{12\}$, $P(b_3) = \{1, 11, 13, 15\}$, $P(b_4) = \{14\}$, $P(e_3) = \{2, 10\}$, $P(b_5) = \{3, 9\}$, $P(b_6) = \{5\}$, $P(e_4) = \{4, 6, 8\}$, and $P(b_7) = \{7\}$.

3. CONSTRAINT PROPAGATION

A constraint propagation system determines the mutual fixpoints of propagators F during search. For that reason we often know that we are fixpoint for some propagators in F, since nothing has changed with respect to them since the last fixpoint calculation. This motivates the need for an *incremental* propagation solver $\mathsf{isolv}(F_o, F_n, D)$ which assumes that D is a fixpoint for the propagators in F_o (the "old" propagators) but not necessarily for F_n (the "new" propagators). The basic incremental propagation solver algorithm is as follows:

$\mathsf{isolv}(F_o, F_n, D)$
  $F := F_o \cup F_n$; $Q := \mathsf{enque}(F_n, \mathsf{initq})$
  while $(\neg\mathsf{empty}(Q))$
    $f := \mathsf{top}(Q)$; $D' := f(D)$
    $Q := \mathsf{enque}(\{f' \in F \mid \exists x \in vars(f).D(x) \neq D'(x)\}, Q)$
    $Q := \mathsf{deque}(f, Q)$; $D := D'$
  return D

The algorithm uses a priority queue Q of propagators to apply. Q is initialised to contain the "new" propagators. Each time the while loop is executed, the top propagator f in the priority queue is applied, and then all propagators that may no longer be at a fixpoint at the new domain D' are added to the priority queue. The propagator f is then removed from the priority queue (since we assume it is idempotent). An invariant of the algorithm is that at the **while** statement $f(D) = D$ for all $f \in F - Q$. In this paper we will examine the implementation of Q.

3.1 Tree based propagator selection

Clearly propagation only occurs through connectedness in the constraint graph. A propagator f for constraint c will only update the domains of variables in $vars(f) = vars(c)$. Hence the constraint graph gives us a basis for selecting propagators.

The idea of our approach is to schedule the propagators according to an in-order traversal of the tree of bi-connected components of the constraint graph. We will compute a fixpoint of the propagators in each BCC before continuing with the next BCC in the traversal. Note that on the motivating example (Example 1) this gives exactly the desired behaviour.

In order to so we need to assign visitation numbers to individual propagators. We extend the notion of visitation numbers of a traversal P of $B(G)$ to give visitation numbers for the propagators of the problem as follows. For $f \in prop(c)$ we have $P_f(f) = \cup\{P(b) \mid c \in b\}$. That is a propagator has visitation numbers given by the union of the visition numbers of the BCCs in which it appears. Note that variable BCC and cut node visitation numbers are not used.

Example 5. Consider the traversal P of Example 4. The propagator visitation numbers are: $P_f(e_1) = \{1, 11, 13, 15\}$, $P_f(e_2) = \{1, 11, 13, 15\}$, $P_f(e_3) = \{1, 3, 9, 11, 13, 15\}$, $P_f(e_4) = \{3, 7, 9\}$, and $P_f(e_5) = \{7\}$.

We reach a fixpoint for each bi-connected component of $B(G)$ before considering another bi-connected component. This requires a two level priority queue Q: made up of a heap of simple queues of propagators. Each queue $q \in Q$ is associated with a traversal number $tn[q]$, and the heap is ordered by traversal numbers (smallest at the top). Traversal numbers represent the repeated in-order traversal of the $B(G)$ using some in-order traversal P. Let M be one than the largest visitation number, i.e., $M = 1 + \max(\cup_{n \in B(G)} P(n))$. Then $k = tn[q] \mod M$ means that the queue q stores propagators for visitation number k in P.

We choose the first propagator of the topmost queue in the heap. So $\mathsf{top}(Q) = \mathsf{q\_top}(\mathsf{h\_top}(Q))$.[1] The interesting part is the enqueing and dequeing operations.

$\mathsf{enque}(F, Q)$
  if $(\mathsf{h\_empty}(Q))$ **then** $t := 0$ **else** $t := tn[\mathsf{h\_top}(Q)]$
  foreach $f \in F$
    let $t' = \min\{k \mid m \in P_f(f) \wedge k \mod M = m \wedge k \geq t\}$
    if exists $q \in Q$ where $tn[q] = t'$ **then**
      $q := \mathsf{q\_put}(f, q)$
    else
      $q := \mathsf{q\_put}(f, \mathsf{q\_init})$; $tn[q] := t'$
      $Q := \mathsf{h\_put}(q, Q)$
  return Q

Enqueing proceeds by finding the traversal number t of the queue on top of the heap. Each propagator is added to the queue with traversal number t' given by the least number $\geq t$ which is equal mod M to one of the visitation numbers of the BCC of the propagator. In other words we place the

[1]We use prefix $\mathsf{q\_}$ for standard heap operations and $\mathsf{h\_}$ for standard heap operations.

propagator in the queue with traversal number corresponding to the next visit of the BCC of that propagator in an in-order traversal P of $B(G)$. If such a queue doesnt exists we create it an add it to the heap. Clearly in the implementation we have data structures supporting direct access to q where $tn[q] = t'$.

```
deque(f,Q)
    q := h_top(Q); q := q_delete(f,q)
    if q_empty(q) then Q := h_remove_top(Q)
    return Q
```

Dequeing is only from the top queue in the heap. The only complication is that if the top queue in the heap becomes empty we need to remove it and find a new heap top.

Example 6. Consider the initial propagation for the SEND+MORE=MONEY problem. We add each constraint to an initially empty Q, so $t = 0$, obtaining a heap $\{e_1, e_2, e_3\}[1]$, $\{e_4\}[3]$, $\{e_5\}[7]$ (showing in braces the traversal number of each queue). Note how e_3 is treated as part of b_3, and e_4 as part of b_5 initially. Now $t = 1$, the top propagator e_1 is executed it makes no changes, similarly for e_2 and e_3. The top queue is removed, we in effect move to traversal number $t = 3$. We execute e_4 and again nothing changes, we remove the top queue and $t = 7$. We execute e_5 and the domain of M and $C4$ change so e_4 is enqueued, this time with a traversal number of 7 (as part of b_7). It is executed modifying the domain of S and O. We enque e_3 once more, now with a traversal number of 9. e_3 causes no new propagation and we are done. Now the lower priority `alldifferent` constraint would be executed.

This example above does not illustrate the cycling behaviour. When executing say e_3 with traversal number 15 (part of b_3) then if e_4 were enqueued it would use traversal number 19 ($19 \mod 16 = 3$) as part of b_5.

In a constraint propagation engine the overhead of queuing is a key factor. Any complexity of queuing operations if important since each propagator typically has low complexity. The priority queue we describe has worst case complexity $O(\log n)$ for enqueing and dequeing a propagator, where n is the number of propagators in the queue. This arises from the heap operations h_remove_min and h_put. This may seem too expensive, but the worst case costs are not paid too often since they only occur on waking a propagator in another BCC, and finishing the propagation in the current BCC. The calculation of the traversal number t' can be made constant time by recording information with edges in $B(G)$.

3.2 Dynamic BCC calculation

After variables are fixed or constraints become redundant they can be removed from the constraint graph, hence existing BCCs can break into pieces. There are algorithms [8] for incrementally calculating BCCs under changes to a graph, but they do not necessarily make it easy to compute the new BCC component tree. At present we have not implemented an efficient incremental BCC calculation, we naively apply the BCC algorithm to BCCs where nodes have been removed. This is too expensive in practice but lets us see the possibly benefits of dynamically maintaining BCCs in terms of the resulting number of propagations.

New problems arise: we wish to only split a BCC into parts leaving the remaining BCCs untouched, but (a) we need space for new visitation numbers, and (b) the in-order traversal may have to change! To fix these problems we need to (a) keep space for new visitation numbers and (b) evaluate visitation numbers by deltas rather than absolute values to allow reordering of the traversal.

Example 7. Fixing E breaks b_3 into $b_8 \equiv \{e_1\}$, $b_9 \equiv \{C_1\}$, $b_3' \equiv \{e_2, N, C_2, e_3\}$ as shown in Figure 1(c). No in-order traversal can be obtained from P by replacing edges involving only b_3 since it visits b_1 then e_3 then b_4 then b_2 traversing b_8 and b_9 four times! To overcome this we use differences in visitation numbers rather than absolute values. We first increase the deltas on arcs from $+1$, to $+|b|$ for arc (b, b') so we have enough space to split b. For the original graph we get (absolute) visitation numbers: $P(b_1) = \{0\}$, $P(b_2) = \{31\}$, $P(b_3) = \{1, 25, 32, 39\}$, $P(b_4) = \{38\}$, $P(e_3) = \{7, 24\}$, $P(b_5) = \{8, 20\}$, $P(b_6) = \{13\}$, $P(e_4) = \{12, 14, 19\}$, and $P(b_7) = \{15\}$. We record for each leaving arc the delta in numbers: e.g. visiting the tree rooted by e_3 from b_3 requires delta $25 - 7 = +18$, while visiting each other neighbour is $+1$. When we reorder the visitations we can quickly calculate the deltas from the current BCC to its neighbours. When we break b_3 into parts the deltas of new BCCs need to be calculated but not for the other parts of the tree.

Note that the density of the visitation numbers is irrelevant to the algorithm, only their order, gaps in visitation numbers makes no difference.

4. RESULTS

The experiments were performed on a 2400MHz Pentium IV running Debian 3.1 "Sarge", using the Mercury [6] finite domain propagation library. We show time and total number of propagation to find first solutions, including all BCC calculations.

The $star_{n,l}$ benchmarks have constraint graphs in the shape of a star. The center of the star consists of two linear constraints between three variables. Each benchmark has n arms leading out from this center, where each arm consists of a chain of l equals constraints ending with another BCC of three variables constrained by two linear constraints. This arrangement results in information flow in both directions along the arms. The *sendmoney* benchmark is the SEND+MORE=MONEY problem from Example 2. For this problem the low priority `alldifferent` constraint is ignored for the purpose of building the tree of BCCs. The $nplus1_l$ benchmarks are as described in Example 1, where l indicates the number of variables. The $chain_l$ benchmarks consist of l links where each link i involves three variables with a constraint between them $A_i = B_i + C_i$. These links are then joined in a chain using constraints $B_i + C_i = A_{i+1} + B_{i+1}$. Note that this results in a chain of BCCs connected through propagator cut points.

Clearly the inorder BCC traversal is almost always beneficial in terms of number of propagations on these benchmarks. As the size of the problems grow the cost of building the BCC tree, and the overheads in using the heap are repaid in time. The *sendmoney* program does not benefit from using inorder BCC traversal since the problem is just too small. The *chain* examples show the benefit of allowing propagators to exist in multiple BCCs. If we do not allow this, then the whole benchmark is forced to be a single BCC, and results are as for FIFO.

Test	FIFO		Inorder BCC	
	time (ms)	props.	time (ms)	props.
$star_{1,1}$	**10.8**	10050	19.4	**4405**
$star_{1,5}$	**13.9**	17882	19.8	**4513**
$star_{1,10}$	**18.2**	27672	20.1	**4648**
$star_{2,1}$	**15.4**	17887	29.3	**4293**
$star_{2,5}$	**26.3**	38451	29.7	**4413**
$star_{2,10}$	38.0	64156	**30.5**	**4563**
$star_{3,1}$	**21.0**	23292	33.0	**4347**
$star_{3,5}$	**33.4**	51312	**33.4**	**4507**
$star_{3,10}$	49.3	86337	**34.9**	**4677**
$star_{4,1}$	**26.1**	31533	41.2	**4366**
$star_{4,5}$	44.8	72653	**42.2**	**4538**
$star_{4,10}$	69.7	124053	**42.9**	**4723**
$star_{5,1}$	**29.9**	36958	44.7	**4443**
$star_{5,5}$	52.0	85538	**45.3**	**4631**
$star_{5,10}$	81.2	146263	**46.5**	**4836**
sendmoney	**1.6**	**53**	1.9	55
$nplus1_{20}$	**1.9**	247	2.0	**112**
$nplus1_{50}$	2.9	1372	**2.6**	**292**
$nplus1_{100}$	6.1	5247	**3.7**	**592**
$nplus1_{150}$	11.5	11622	**4.9**	**892**
$chain_6$	**8.9**	6038	10.8	**4424**
$chain_8$	**56.5**	38405	63.0	**24830**
$chain_{10}$	339.8	226892	**335.5**	**132888**
$chain_{12}$	1928.9	1278137	**1750.5**	**688725**

Table 1: Comparative results of BCC tree based scheduling versus the usual FIFO scheduling.

Another set of benchmarks, $wheel_{n,l}$, show the benefit of updating the BCC tree dynamically in terms of number of propagations. These benchmarks are the same as the $star_{n,l}$ benchmarks, except that the end of each arm is connected (through two not equal constraints and an extra variable) to those on either side. This results in a constraint graph which is initially a single BCC, but gradually becomes a star as the extra variables are labelled first.

Clearly a more efficient method of maintaining the BCC tree is required to show a benefit in terms of time, but the improved propagation behaviour is clear.

5. CONCLUSION AND RELATED WORK

There has been considerable work in making use of tree decomposition to solve constraint satisfaction problems. Since tree constraint graphs can be solved efficiently [2], it makes sense to break the problem into tree parts. This has been used to give bounds on backtrack search [3]. The drive of the work on tree decomposition has been to define tractable (polynomial) classes of constraint satisfaction problem. Most of this work concentrates on the dynamic case where the order of variable labelling is used to break the constraint graph into components effectively; approaches like psuedo tree-search [4] and join-tree clustering [1] can then be used.

In this work we simply concentrate on using the constraint graph to efficiently calculate a fixpoint of propagators, independent of what particular search is taking place. We are unaware of earlier work examining how to effectively schedule propagation so that the biconnected tree decomposition of the constraint graph is effectively used. Previous work also does not consider separating BCCs with constraint cut nodes. Since we are in a propagation framework we can benefit from the priority system, the full constraint graph for the SEND+MORE=MONEY problem is one biconnected component because of the `alldifferent` constraint, but since it is at a lower priority we can use the biconnected approach to order the higher priority propagators.

Test	Static		Dynamic	
	time (ms)	props.	time (ms)	props.
$wheel_{2,1}$	**49.2**	16894	92.3	**4088**
$wheel_{2,5}$	**83.4**	36358	116.0	**4224**
$wheel_{2,10}$	**125.4**	60688	148.0	**4394**
$wheel_{2,15}$	**168.4**	85018	180.9	**4564**
$wheel_{3,1}$	**61.8**	21985	121.0	**4134**
$wheel_{3,5}$	**107.4**	48585	157.0	**4308**
$wheel_{3,10}$	**164.6**	81835	209.3	**4498**
$wheel_{3,15}$	**220.1**	115085	261.3	**4688**
$wheel_{4,1}$	**82.5**	29699	153.4	**4164**
$wheel_{4,5}$	**151.1**	68603	205.7	**4362**
$wheel_{4,10}$	**236.8**	117233	276.3	**4582**
$wheel_{4,15}$	**323.3**	165863	352.6	**4802**
$wheel_{5,1}$	**94.6**	34843	183.9	**4243**
$wheel_{5,5}$	**174.8**	80895	250.0	**4461**
$wheel_{5,10}$	**274.0**	138460	344.2	**4706**
$wheel_{5,15}$	**375.1**	196025	451.2	**4951**

Table 2: Comparative results of dynamic BCC calculation versus the static method.

Our approach has some similarity of spirit with the approach of [5], which dynamically determines which propagators are causing a tight cycle of propagation, and schedules them to reach fixpoint before scheduling other constraints. We calculate statically using the constraint graph, rather than using the evolving propagation behaviour.

Obviously the benchmarks we use here are quite artificial, but there do exist classes of constraint problems with loosely connected constraints graphs, for example in test generation. There remains substantial future work in determining when it is worth applying this scheduling approach, and how to most efficiently make use of dynamic BCCs which arise in all applications.

6. REFERENCES

[1] R. Dechter. *Constraint Processing*. Morgan-Kaufmann, 2003.

[2] E. Freuder. A sufficient condition for backtrack-free search. *JACM*, 29(1):24–32, 1982.

[3] E. Freuder. A sufficient condition for backtrack-bounded search. *JACM*, 32(4):755–761, 1985.

[4] E. Freuder and M. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of IJCAI-85*, 1076–1078, 1985.

[5] O. Lhomme, A. Gotlieb, M. Rueher, and P. Taillibert. Boosting the interval narrowing algorithm. In *Proceedings of JICSLP*, 378–392. MIT Press, 1996.

[6] Mercury. `www.cs.mu.oz.au/mercury`.

[7] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[8] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.