Constraint Propagation for Loose Constraint Graphs

Kathryn Francis,
Department of Computer Science and Software Engineering
University of Melbourne, 3010, Australia
francis@students.csse.unimelb.edu.au

Peter J. Stuckey
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, 3010, Australia
pjs@cs.mu.oz.au

ABSTRACT

In this paper we investigate how to 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. 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*

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. A full version of the paper is available [1].

Example 1. Consider a system of constraints $e_i \equiv x_{i+1} = x_i + 1, 0 \leq i < n$, where the initial domain of each variable x_i is $\{0, \ldots, 2n\}$. Using a FIFO based propagation queue this requires $O(n^2)$ executions to reach a fixpoint. Similarly for a LIFO based propagation queue. But if we propagate in order $e_1, e_2, \ldots, e_{n-1}, e_{n-1}, \ldots, e_2, e_1$ we require only $O(n)$ steps to reach the same fixpoint.

SAC'07 March 11-15, 2007, Seoul, Korea.

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.

2. OUR APPROACH

A *constraint satisfaction problem* (CSP) consists of a set of variables $v \in V$, each with a domain of possible values $D(v)$, and a set of constraints $c \in C$, where $vars(c) \subseteq V$. Constraint propagation solves constraint satisfaction problems by repeatedly executing propagators for each constraint to remove values from the domains of its variables that cannot take part in a solution (see e.g. [2]). In this paper we investigate how to order the choice of which propagator to execute next by taking into account the constraint graph.

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

Clearly propagation only occurs through connectedness in the constraint graph. A propagator for constraint c can only update the domains of variables in $vars(c)$, which can only wake propagators for constraints involving these changed variables. 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.

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 $G(n) = (N - \{n\}, E - \{(n, n') \mid n' \in N\})$ for each $n \in N$. A node n that occurs in two bi-connected components is called a *cut node*. Algorithms for determining bi-connected components [3] are

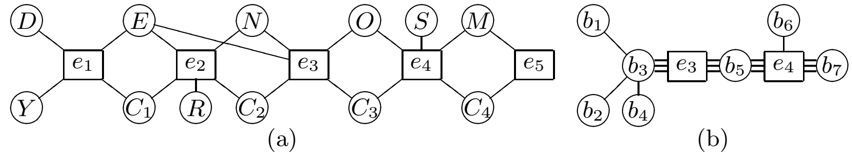

Figure 1: (a) Constraint graph for linear constraints of SEND+MORE=MONEY, (b) tree decomposition of bi-connected components.

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.

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: $P(n) = \{i \mid e_i = (n, \_)\}$. 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 and traversing clockwise (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_4), (b_4, b_3), (b_3, b_2), (b_2, b_3), (b_3, b_1). The visitation numbers are $P(b_1) = \{0\}$, $P(b_2) = \{14\}$, $P(b_3) = \{1, 11, 13, 15\}$, $P(b_4) = \{12\}$, $P(e_3) = \{2, 10\}$, $P(b_5) = \{3, 9\}$, $P(b_6) = \{5\}$, $P(e_4) = \{4, 6, 8\}$, and $P(b_7) = \{7\}$.

To support our approach we need to queue propagators so they are executed in an in-order traversal. To do so we use a min-heap of propagation queues, one for each BCC. Let t be the traversal number assigned to the top queue in the heap. When a propagator for a constraint is awoken it is placed in the queue for its BCC b. If there is currently no queue for the BCC b on the heap, an entry is added with heap weight given by $\min\{m \mid m > t \wedge (m \mod 2|E|) \in P(b)\}$, that is with the next traversal number for that BCC in the infinite cycle traversal. If the constraint occurs in more than one BCC, it is placed in the queue for the BCC b which returns the least value using the above calculation.

Example 5. 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 (as part of b_5). 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.

We compared performance of our approach to the usual FIFO queuing approach on chain examples like Example 1, and examples with star shaped constraint graphs with multiple arms from a single centre joining small BCCs at the end and the center. The number of propagations required to solve these problems reduces by 1 or two orders of magnitude. For sufficiently large examples the time savings caused by less propagations easily pay for the costs of calculating BCCs and traversal numbers (see [1] for details). Note our approach uses $O(\log |N_S|)$ time in the worst case to enque and deque rather than $O(1)$.

Obviously the benchmarks we use are quite artificial, but there do exist classes of constraint problems with loosely connected constraints graphs, for example in test generation. We need to investigate more practical classes of problem for our approach. The approach can be extended to dynamically recalculate BCCs as the search progresses since ground variables and redundant constraints effectively disappear from the graph. We need to balance the overhead of recalculation against the advantages of less propagations. The dynamic version is potentially applicable to all CSPs.

3. REFERENCES

[1] K. Francis and P. Stuckey. Constraint propagation for loose constraint graphs. Technical report, 2006. `www.cs.mu.oz.au/~/pjs/papers/loose.pdf`.

[2] C. Schulte and P. Stuckey. Speeding up constraint propagation. In *Procs. of CP2004*, volume 3258 of *LNCS*, pages 619–633, 2004.

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