Semi-Independent Partitioning: A Method for Bounding the Solution to COP's

David Larkin

University of California, Irvine

Abstract. In this paper we introduce a new method for bounding the solution to constraint optimization problems called semi-independent partitioning. We show that our method is a strict generalization of the mini buckets algorithm [1]. We demonstrate empirically that another specialization of SIP, called greedy SIP, generally produces a better answer than mini buckets in much less time.

1 Introduction

In this paper we introduce a new method for approximating the solution to constraint optimization problems [5]. These problems are NP-hard in general, but have many practical applications. State of the art methods for solving them [3, 6, 8, 7] rely upon branch and bound search with a heuristic to compute a lower bound on the quality of the best solution that can be found by extending the partial assignment associated with the current node.

Our algorithm, called semi-independent partitioning, computes a lower bound on the best solution of a COP, with the solution quality and running time being controlled by a complexity parameter i. We will show that SIP is a generalization of the mini buckets algorithm [1, 4]. We will present empirical results showing that an alternative instantiation of SIP, greedy SIP, generally computes a much better lower bound than MB in less time.

This paper is divided into several parts. Following this introduction, we introduce basic concepts in Section 2. Then in Section 3 we introduce the semi-independent partitioning algorithm and compare it with mini buckets. In Section 4 we summarize the results of an experimental comparison, and in Section 5 we conclude.

2 Basic Concepts

A set of constraints C defined on finite-domain variables X is a set of functions $C = \{C_1, C_2, ..., C_m\}$, where C_i is defined on a subset of X, S_i, called its scope. The size of the scope is called the constraint arity. C_i maps allowed tuples to 0 and disallowed tuples to 1. The cost of an assignment to X is the number of constraints it does not satisfy, or $\sum_{\{c \in C\}} c$, where c is evaluated on the assignment. The cost of the optimum solution then is $\min_X \sum_{\{c \in C\}} c$. The MAX-CSP problem is to find this quantity. It is NP-hard in general.

C can be associated with a binary graph $G = (X, E)$ called the constraint graph. An edge $\{x, y\}$ is in E if and only if there exists a constraint c in C whose scope includes both x and y. The induced width w^* of C's graph is defined in reference to an ordering of the variables in X or absolutely. In reference to an ordering, it is calculated by removing the variables from the graph from last to first, connecting all neighbors of a node when it is removed. The maximum number of neighbors any node has when it is deleted is the induced width. The absolute induced width is the minimum induced width over all orderings. Finding the absolute induced width is NP-hard, but orderings with good induced width can be found with heuristics. The min-degree heuristic, for example, orders the vertices from last to first, at each point choosing the variable with minimum degree in the graph, then removing it and connecting its neighbors. More material on the induced width measure can be found in [2].

Given a variable $x \in X$, and set of constraints $C_x \subseteq C$ defined on $X' \subseteq X$ which all mention x in their scopes, the operation of projecting x out of C_x computes a new function $g = \min_x \sum_{\{c \in C_x\}} c$ which is defined on $X' - x$. It occupies $O(\exp(|X'| - 1))$ space and the time complexity of computing it is the same. Variable elimination [2] is an exact algorithm for MAX-CSP whose complexity is exponential in the induced width of the graph along an elimination ordering. It simplifies a problem C by repeatedly applying projection operations to eliminate variables from last to first. Variable x is eliminated from C by collecting all the constraints C_x that mention x and replacing them with the function g that results from projecting x out. The desired quantity $\min_X \sum_{\{c \in C\}} c$ is the result of projecting out all the variables in X one by one. Its correctness follows from the fact that $\min_X \sum_{\{c \in C\}} c = \min_{X-x} \left(\sum_{\{c \in C - C_x\}} c + \min_x \sum_{\{c' \in C_x\}} c' \right) = \min_{X-x} \sum_{\{c \in C - C_x\}} c + g$.

3 The Semi-Independent Partitioning Algorithm

In this section we introduce the semi-independent partitioning algorithm. First in subsection 3.1 we introduce the algorithm in its most general form, which allows any number of specific solution strategies. Then in subsection 3.2 we describe a specialization which uses a greedy strategy. In subsection 3.3 we show that mini buckets is another specialization.

3.1 General Semi-Independent Partitioning

Let C be a set of constraints defined on variables X, and let i be a complexity bound. Our problem is to find a good lower bound on the cost of the optimal solution $\min_X \sum_{\{c \in C\}} c$ with $O(|C||X| \exp(i))$ time and space complexity.

The exact method variable elimination, described in Section 2, can be used if an ordering of C's graph can be found with $w^* \leq i$. However in general $w^* > i$ and this is not possible, and in any case finding an optimum ordering is an NP-hard problem.

We can partition a set of constraints C defined on X into subsets C_1 and C_2, where $C_1 \cup C_2 = C$ and $C_1 \cap C_2 = \emptyset$, and the induced width of C_1 is bounded by i. Variable elimination can be applied to completely or partially solve C_1, resulting in the value of its optimum solution or a function giving the cost of the optimum extension of any assignment to its scope variables. Formally, if Y is the set of variables we wish to eliminate from C_1, then $\min_X \sum_{\{c \in C\}} c = \min_X \sum_{\{c \in C_1\}} c + \sum_{\{c' \in C_2\}} c' \geq \min_{X-Y}(\min_Y \sum_{\{c \in C_1\}} c) +$ $(\min_Y \sum_{\{c' \in C_2\}} c') = \min_X g + \sum_{\{c' \in C_2\}} c'$, where $g = (\min_Y \sum_{\{c \in C_1\}} c)$ is the solution of C_1 that is derived by variable elimination.

Algorithm `General SIP`
Input: `Constraints` C, `complexity limit` i, `partitioning method` S.
Output: `Lower bound on the solution of` C.
`While` $w^*(C) > i$ `according to a heuristic ordering...`

1. `Select` $C_1 \subseteq C$ `s. t.` $w^*(C_1) \leq i$ `with` S, `let` $C_2 = C - C_1$.
2. `Let` S `choose a set` Y `of variables to eliminate from` C_1 `with v. e.`
3. `Set` $g = \min_Y \sum_{\{c \in C_1\}} c$, `let` $C = g \cup C_2$.

`Return the solution of` C `found with variable elimination.`

Fig. 1. The General SIP Algorithm

Pseudo-code for general SIP is given Figure 1. Each invocation of variable elimination costs $O(|X| \exp(i))$. Assuming at least two constraints are eliminated each time, and that no more than one new function is generated, then the total running time is $O(|C||X| \exp(i))$.

3.2 Greedy SIP

Up until now we have not specified how general SIP is to partition the constraints or decide what variables to eliminate. In this subsection we describe an instantiation of general SIP, called greedy SIP, which offers a practical strategy.

The basic problem is to partition a set of constraints C into C_1 and C_2, such that an ordering of C_1 can be found with induced width bounded by i. Greedy SIP's method is to greedily try to add constraints from C into C_1, maintaining at all times a heuristic ordering of C_1 with bounded induced width, and refusing to add a constraint to C_1 if that makes its heuristic ordering exceed the limit. The partitioning is completed when one attempt has been made to add every constraint in C. The set of variables Y to be eliminated is simply all but the first i variables in the final heuristic ordering.

For example, consider the problem shown in Figure 2. The i bound is 2, and the initial problem C is the clique of size 6 in the upper left corner. We will use the min-degree heuristic ordering. A greedy partitioning is shown in the upper right corner, where C_1 is to the left of the $\cup$ and C_2 is to the right. A min-degree

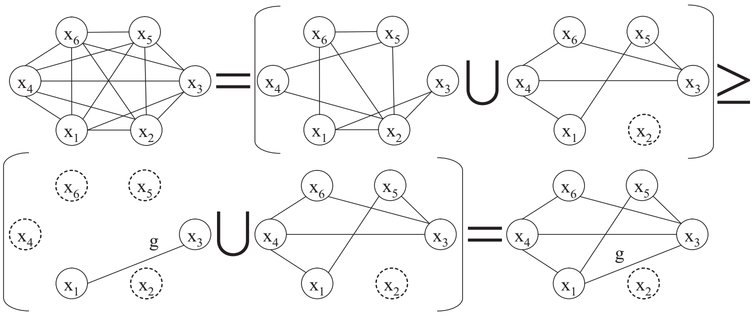

Fig. 2. Example of Greedy SIP

ordering of C_1 is x_3, x_1, x_2, x_6, x_5, x_4, which has induced width 2. Note that if we try to add any other edge from C_2 to C_1, the induced width of the min degree ordering will exceed the limit. For example, if we add (x_1, x_5) to C_1, the min degree ordering has induced width 3.

Now greedy SIP will eliminate all but the first 2 variables x_3 and x_1 from C_1. The result is shown in the lower left corner. C is then set to the function g defined on x_1 and x_3 joined with C_2, as shown on the lower right. Since a min degree ordering of C now has induced width 2, variable elimination can be applied to finish the problem.

3.3 Mini Buckets

In this subsection we describe mini buckets [1, 4] as another specialization of general SIP.

Mini buckets always maintains a variable to eliminate, x. When it partitions a set of constraints C with complexity bound i, it selects a subset B_x of C called x's *bucket*, which is the set of all constraints in C mentioning x. Then it selects a maximal subset of M_x of B_x called a *mini bucket*, such that the total number of variables appearing in M_x is not more than $i + 1$, and no other member of B_x can be added without destroying this property. M_x is chosen to be C_1 and C_2 becomes $C - M_x$. M_x is ordered arbitrarily, except that the bucket variable x is placed at the end of the ordering and it is selected as the only variable to eliminate. Doing this creates a function g of arity i which does not mention x. C is then set to $g \cup C_2$ and the process continues. If B_x is empty, then a new variable x' is selected to be eliminated next. The algorithm halts when all variables have been eliminated.

4 Empirical Results

To compare MB and greedy SIP, we tested them on random binary MAX-CSP problems with 55 variables and domain size 4. Every constraint had a 40 percent chance of being present. All constraints randomly disallowed half of the possible value pairs and allowed the other half. We averaged the results of 25 experiments for each value of i from 6 to 9. The results are summarized in Figure 3. For all settings of i, greedy SIP achieved a significantly better lower bound in less time than MB. For example, at $i = 6$ SIP computed an average lower bound of 67.5 in 12 seconds. Even at $i = 9$ MB was not quite as accurate, computing a lower bound of 64.1 in 1020 seconds.

	$i = 6$		$i = 7$		$i = 8$		$i = 9$	
	MB	SIP	MB	SIP	MB	SIP	MB	SIP
Lower Bound	41.9	67.5	49.7	77.4	57.1	84	64.1	90.5
Time	19s	12s	72s	39s	272s	136s	1020s	485s
Max. Memory	1.1M	0.2M	3.7M	0.6M	12M	2.2M	42M	9M

Fig. 3. Empirical results (average $w^* = 39$)

5 Conclusion

In this paper, we introduced a new algorithm for computing lower bounds on the quality of the best solution of a MAX-CSP problem. We compared it empirically with the mini buckets method, showing that it performed significantly better.

For future work, of course it would be of interest to directly evaluate the efficiency of our method as a heuristic for branch and bound search to find an exact optimum. Since our method, unlike mini buckets, does not follow a natural static variable ordering, it would have to be called dynamically at every node.

References

[1] Rina Dechter. Mini-buckets: A general scheme for generating approximations in automated reasoning. In *IJCAI*, 1997.

[2] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, October 1999.

[3] Rina Dechter, Kalev Kask, and Javier Larrosa. A general scheme for multiple lower bound computation in constraint optimization. In *Proc. of the Conf. on Principles and Practice of Constraint Programming*, 2001.

[4] Rina Dechter and Irina Rish. A scheme for approximating probabilistic inference. In *Proc. of the Conf. on Uncertainty in Artificial Intelligence*, 1997.

[5] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992. Unobtained.

[6] Kalev Kask and Rina Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 129:91–131, 2001.

[7] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107:149–163, 1999.

[8] Javier Larrosa and Pedro Meseguer. Partition-based lower bound for Max-CSP. In *Principles and Practice of Constraint Programming*, pages 305–315, 1999.