Synthesis of Abstraction Hierarchies for Constraint Satisfaction by Clustering Approximately Equivalent Objects*

Thomas Ellman

Department of Computer Science
Hill Center for Mathematical Sciences
Rutgers University, New Brunswick, NJ 08903
ellman@cs.rutgers.edu
LCSR-TR-200

Abstract

Abstraction techniques are important for solving constraint satisfaction problems with global constraints and low solution density. In the presence of global constraints, backtracking search is unable to prune partial solutions. It therefore operates like pure generate-and-test. Abstraction improves on generate-and-test by enabling entire subsets of the solution space to be pruned early in a backtracking search process. Unfortunately, a suitable abstraction space may not be included in the problem description provided to a problem solving system. The benefits of abstraction will not be available unless the system can automatically construct an abstraction space. This paper describes how abstraction spaces can be generated by clustering the objects appearing in a problem into classes that contain approximately equivalent objects. It presents a program synthesis algorithm for automatically building abstraction spaces and hierarchic problem solvers that exploit approximate equivalence of objects. The fully implemented synthesis algorithm operates by analyzing declarative descriptions of classes of constraint satisfaction problems. The paper presents data from experimental tests of the synthesis algorithm and the resulting problem solvers on the NP-hard Partition and Minimum Flow Cut problems.

1 Introduction

Abstraction techniques are important for solving constraint satisfaction problems (CSPs) with global constraints and low solution density. Examples of such problems include the Partition problem and the Minimum Flow Cut problem, both of which are NP-hard [Garey and Johnson, 1979]. (See Figure 1). In the presence of global constraints, backtracking search

*Also appears in the Proceedings of the Tenth International Conference on Machine Learning, Amherst, MA, June, 1993
• Partition: Given a set $E$ and a weight $w(e)$ for each element $e$ of $E$. Find a subset $M$ of $E$ such that the total weight of $M$ equals the total weight of $E - M$.

• Minimum Flow Cut: Given a complete graph $G = (V, E)$, a function $w(v_1, v_2)$ assigning a weight to each edge, and an integer $T$. Find two equal sized subsets of $V$ such that the total weight of edges crossing between subsets is no more than $T$.

Figure 1: Testbed Problems

is unable to prune partial solutions [Nadel, 1988]. It therefore operates like pure generate-and-test. When overall solution density is low, this approach is not effective, except when applied to small problems. Abstraction can improve the performance of backtracking by enabling entire subsets of the solution space to be pruned early in the search process. Unfortunately, a suitable abstraction space may not be included in the problem description provided to a problem solving system. The benefits of abstraction will not be available unless the system can automatically construct an abstraction space. This paper presents a method for automatically constructing abstraction spaces and hierarchic problem solvers for constraint satisfaction problems. Abstraction spaces are constructed by clustering objects appearing in the problem description into classes of approximately equivalent objects. The abstraction spaces can be utilized by a hierarchic problem solver that operates in two steps: (1) Find an abstract solution by ignoring differences between approximately equivalent objects. (2) Refine the abstract solution into a true solution that takes account of all differences between objects.

The strategy of approximate equivalence can be illustrated by considering the Partition problem: The problem specification includes a set $E$ of elements and a weight $w(e)$ of each element $e$. The solution requires finding a subset $M$ of $E$ such that the weight of elements included in $M$ equals the weight of elements excluded from $M$, i.e., the total weight is $W/2$, where $W$ is the total weight of $E$. This constraint is global, since one cannot verify equality of the total weights without knowing the inclusion/exclusion status of each element. Backtracking algorithms therefore cannot easily use this constraint to prune partial solutions. The following procedure overcomes this limitation: First divide the elements of $E$ into disjoint classes such that elements of roughly equal weight are grouped together. Now pretend that each element $e$ of a given class actually has a whole interval range of weights, i.e., the range of weights spanned by the members of the class. Then select a quota for each class that represents the number of elements of the class to be included in $M$. Given a quota for each class, compute upper and lower bounds on the total weight of elements included in $M$. If the goal weight $W/2$ does not lie within the bounds, then prune away this quota assignment and try another one. Otherwise, refine the quotas into an actual selection of elements to be included in the set $M$.

The behavior of a problem solver using a two level abstraction hierarchy is illustrated in Figure 2. The illustration shows how the search tree can be broken into two parts. The upper portion of the tree represents a space of “abstract states”, (e.g., a space of quota assignments). Backtracking tests each complete abstract solution against an “abstract goal” (e.g., testing problem constraints against quota assignments). The problem solver will
be complete provided that the abstract goal test is a necessary condition on refi
ability of abstract solutions, i.e., abstract states that fail to satisfy the abstract goal cannot be refined into concrete solutions. The lower portion of the tree represents a space of "concrete states", e.g., a space of subset membership assignments. Backtracking tests each complete concrete solution against the original "concrete goal", e.g., testing problem constraints against subset membership assignments. Search in the concrete space is also guided by "constrained variable range generators". These require each concrete state variable to assume only values that are consistent with the previously specified abstract solution, e.g., obeying previously specified quota assignments.

Figure 2: Hierarchic Problem Solver

A technique for constructing such hierarchic problem solvers is presented in this paper. It takes as input a declarative representation of a parameterized class of constraint satisfaction problems, called a PCSP. This input includes a description of a search space $S$ and a boolean-valued goal function $G$ defined on the elements of $S$. As output, it produces all of the remaining data needed to instantiate the hierarchic problem solver illustrated in Figure 2, including an abstraction space $A$; an abstract boolean-valued goal function $\bar{G}$ defined on the elements of $A$; and a set of variable range generators that constrain concrete state variables to assume values consistent with a given abstract solution. A formal characterization of the abstraction space and goal function are given in Section 2. A description of the PSCP input problem representation is given in Section 3. The algorithm for synthesizing hierarchic problem solvers is given in Section 4. Results from testing the synthesized algorithms are presented in Section 5.

2 Abstract Search Spaces

An "abstraction space" $A$ may be defined formally as a partition of the states in the underlying "concrete space" $S$. Thus each abstract state $a \epsilon A$ is a set of concrete states $s \epsilon S$. 

Abstractions based on approximate equivalence can be defined by focusing on function-finding problems, i.e., problems whose solution requires finding a function \( f \) that maps a given domain \( D \) into a given range \( R \). Thus each state \( s \in S \) is a function that accepts some \( d \in D \) and returns a value \( r \in R \). An abstraction space may constructed by clustering the domain \( D \) into a collection \( \hat{D} \) of disjoint sets. Each state \( a \in A \) will be a function \( \hat{f} \) which accepts a subset \( \hat{d} \) in the collection \( \hat{D} \) and returns a multiset of cardinality \( |\hat{d}| \) of values drawn from the range \( R \). An abstract solution thus specifies the multiset \( \hat{f}(\hat{d}) \) of values assigned to a set \( \hat{d} \) of objects; however it does not specify which value is assigned to which object. For example, in the Partition example described above, the domain \( D \) is the set \( E \) of elements and the range \( R \) is the set \{True, False\} of boolean values. While a concrete solution \( s \in S \) assigns a value of True or False to each element \( e \in E \), an abstract solution \( a \in A \) assigns to each class \( \hat{e} \in \hat{E} \) a multiset of boolean values, or equivalently, a quota of True values. Nevertheless, the abstract solution does not specify precisely which elements in \( \hat{e} \) actually receive True values. This type of abstraction is discussed more fully in [Ellman, 1993].

3 Parameterized CSPs

Classes of constraint satisfaction problems are represented using a notation called “parameterized constraint satisfaction problems” (PCSPs). A PCSP includes a signature \( S \) and a goal-function \( G \). The signature is a declaration of the types of sets and functions that appear in problem instances, including: a list of names and data types (symbol or integer) of finite sets; a list of names of functions along with a declared domain and range for each; a list of names and data types of constants. The signature also declares each function to be either “known” or “unknown”. The known functions are supplied as part of each problem instance specification. The unknown functions must be found by the problem solver in order to solve the problem.\(^1\)\(^2\) The goal function \( G \) serves to determine how the unknown functions may be specified in order to solve the problem. \( G \) is conceptually a map from the constants, known functions and unknown functions into the booleans. We use the notation \( G(p, s) \) to indicate that \( G \) depends on the problem instance \( p \) (i.e., the known functions) and the solution \( s \), (i.e., the unknown functions). A particular goal function \( G \) is represented by a boolean formula that may reference the known and unknown functions; primitive arithmetic \((+, -, *, /)\), relational \((<, \leq, >, \geq)\) and boolean \((-, \lor, \land)\) operations; as well as conditionals \( If(c, x, y) \) and absolute values \( Abs(x) \). \( G \) may also include universal \( \forall \) or existential \( \exists \) quantification and sums \( \Sigma \) or products \( \Pi \) of functions over sets declared in the signature. The signature and goal function for the Partition problem class are found in Figure 3.

\(^3\)Unknown functions are required to have symbolic domains. This requirement is necessary for the correctness of the algorithm for synthesizing abstract goal functions. No generality is lost since it could be enforced by mechanical translation.

\(^2\)We use the notation \( f(e, p) \) to indicate that the known function \( f \) depends on the problem instance \( p \) as well as its argument \( e \). We use the notation \( f(e, s) \) to indicate that the unknown function \( f \) depends on the solution \( s \), as well as its argument \( e \). The problem instance \( p \) or solution \( s \) parameter will occasionally be omitted from \( f(e, p) \) or \( f(e, s) \) when its role is clear from the context.
• Signature:
  - Sets:
    * Elements $E$ of type symbol.
    * Weights $W$ of type integer.
  - Known: Weight $w : E \rightarrow W$
  - Unknown: Member $m : E \rightarrow \{True, False\}$

• Goal Function:
  \[
  G(p, s) = S_1(p, s) = S_2(p, s) \\
  S_1(p, s) = \sum_{e \in E(p)} if(m(e, s), w(e, p), 0) \\
  S_2(p, s) = \sum_{e \in E(p)} if(m(e, s), 0, w(e, p))
  \]

Figure 3: Partition Problem Class Specification

4 Synthesis of Problem Solvers

An algorithm for synthesizing a hierarchic problem solver is shown in Figure 4. The algorithm begins by clustering the domain $D$ of the unknown function $f$ into a collection $\hat{D}$ of equivalence classes. The clustering process divides $D$ into classes based on similarity of values of some known function whose domain is $D$, and which is referenced by the goal function $G(p, s)$. Next, the original goal function $G(p, s)$ is transformed into an abstract goal $\hat{G}(p, a)$ in order that it may operate on abstract states $a$. This transformation is achieved, in part, by a process of replacing each function $f$ appearing in $G(p, s)$ with the corresponding “set operation” $\hat{f}$ or the corresponding “symmetric operation” $\hat{f}$, as defined in Figure 4. Finally, the revised goal is surrounded with a test for the appearance of $True$ in the returned set of boolean values. These transformations weaken the original goal so that it fails to distinguish between distinct members $d$ and $d'$ that occur the same class $\hat{d}$ of the collection $\hat{D}$ of equivalence classes. The resulting abstract goal is by construction a necessary condition on solvability of the abstract state $a$. The algorithm also constructs a refinement function $R_d$ for each element $d$ of the domain $D$. The function $R_d(p, a, s)$ takes the problem specification $p$, an abstract solution $a$, and a partially specified concrete solution $s$, and returns the set of possible values for $f(d)$ that are consistent with $a$ and $s$.

When the synthesis algorithm is applied to the Partition problem description in Figure 3, it generates the abstract goal function shown in Figure 5. This goal function $G(p, a)$ begins by selecting an arbitrary candidate solution $s$ consistent with the quotas described in $a$. It then applies to $s$ an approximation $\hat{G}(p, s)$ of the original goal function that is symmetric with respect to the collection $\hat{E}$ of equivalence classes. This symmetric goal function $\hat{G}(p, s)$ treats each $e \in E$ as if it has an interval $\bar{w}(e)$ of weights, rather than an actual weight $w(e)$. It performs interval arithmetic, interval comparisons, and boolean-set algebra to compute two intervals: $\hat{S}_1(p, a)$ represents a range of values for the sum of weights of included elements. $\hat{S}_2(p, a)$ represents a range of values for the sum of weights of excluded elements. Finally the abstract goal function checks whether these two intervals overlap. It thereby computes
Definitions:

- **Set Functions**: Given an arbitrary function \( h : A \rightarrow B \), define the corresponding set function \( \hat{h} : A \cup 2^A \rightarrow 2^B \), such that \( \hat{h}(x) = \{h(x)\} \), if \( x \in A \), and \( \hat{h}(x) = \{h(y) \mid y \in x\} \) if \( x \subseteq A \).

- **Symmetric Functions**: Given an arbitrary function \( h : A \rightarrow B \), and a partition \( \hat{A} \) of \( A \), define the corresponding symmetric function \( \tilde{h} : A \cup 2^A \rightarrow 2^B \) such that \( \tilde{h}(x) = \{h(y) \mid y \in \text{Class}(x)\} \), if \( x \in \hat{A} \), and \( \tilde{h}(x) = \bigcup_{y \in x} \{h(z) \mid z \in \text{Class}(y)\} \), if \( x \subseteq A \).

Algorithm: Given a search space \( S \) representing all functions \( f \) from \( D \) into \( R \), and a goal function \( G(p, s) \) defined on all \( s \in S \) and problem instances \( p \):

1. Define an abstraction space \( A \):
   
   (a) Define a partition \( \hat{D} \) of \( D \) by clustering \( D \) based on similarity of values of \( D \) itself or some known function defined on \( D \) that is referenced by the goal function \( G(p, s) \).
   
   (b) Let the abstraction space \( A \) represent all functions \( \tilde{f} \) with domain \( \hat{D} \) such that for all \( d \in \hat{D} \), \( \tilde{f}(d) \) returns a multiset over \( R \) of cardinality \( |d| \).

2. Synthesize an abstract goal \( \tilde{G}(p, a) \) defined on all \( a \in A \) and problem instances \( p \):
   
   (a) Construct an approximate goal function \( \hat{G}(p, s) \) that is exactly symmetric with respect to \( \hat{D} \):
     
     i. Transform the original goal function \( G(p, s) \) into a boolean set valued function \( \hat{G}(p, s) \):
        Replace each reference to a known function \( h \) defined on domain \( D \), with the corresponding symmetric function \( \tilde{h} \). Replace each primitive or known function \( h \) defined on \( D' \neq D \), with the corresponding set function \( \hat{h} \).
     
     ii. Let \( \hat{G}(p, s) = \text{True} \in \hat{G}(p, s) \).
   
   (b) Define the abstract goal \( \hat{G}(p, a) \) to select an arbitrary \( s \in A \) and return \( \hat{G}(p, s) \).

3. Synthesize functions \( R_d(p, a, s) \) that incorporate constraints between the abstract space \( A \) and the concrete space \( S \). For each \( d \in D \), \( R_d(p, a, s) \) is the set of values of \( f(d) \) that are consistent with the abstract solution \( a \) and the partially specified concrete solution \( s \). \( R_d(p, a, s) \) is computed by taking the multiset difference between \( \tilde{f}(\hat{d}, a) \), and the multiset of all \( f(d, s) \) (\( ded \)) such that \( f(d) \) is assigned a value in state \( s \).

Figure 4: Synthesis Algorithm and Definitions
\[
\begin{align*}
\hat{G}(p, a) &= True \epsilon (\hat{S}_1(p, a) \Rightarrow \hat{S}_2(p, a)) \\
\hat{S}_1(p, a) &= \sum_{\hat{e} \in E(p)} \sum_{(v, e) \in \hat{e}, \tilde{w}(\hat{e}, a)} f(v, \tilde{w}(e, p), \{0\}) \\
\hat{S}_2(p, a) &= \sum_{\hat{e} \in E(p)} \sum_{(v, e) \in \hat{e}, \tilde{w}(\hat{e}, a)} f(v, \{0\}, \tilde{w}(e, p))
\end{align*}
\]

Figure 5: Abstract Partition Goal

a necessary condition on the original goal.

The preceding discussion blurs the distinction between operations that happen at compile time (when the problem class is specified), and operations that happen at run time (when the problem instance is specified). In the current implementation, the abstract goal function and constrained variable range generators are synthesized at compile time. These operations are done only once for an entire class of constraint satisfaction problems. The costs of this compilation can therefore be amortized over many problem solving episodes. For this reason, the costs of constructing abstract goals and constrained variable range generators are not included in the CPU time measurements reported in Section 5. In contrast to this, the domain \(D\) is clustered into equivalence classes at run time, when the problem instance is known. Clustering must occur at run time whenever the domain or the features used to cluster the domain are expected to change from one problem instance to the next. Since clustering occurs at run time, and must be redone for each problem instance, the costs of clustering cannot be amortized over problem solving episodes. For this reason, clustering costs are included in the CPU time measurements reported in Section 5.

Several different algorithms are used for clustering the domain \(D\) into equivalence classes.

These algorithms are described in Figure 6. Random clustering ignores all features of the objects to be clustered. It is expected to be useful when the details of the resulting clusters are not as important as minimizing the cost of the clustering process itself. In contrast, bottom-up clustering does consider features of the objects to be clustered. It is expected to be useful when the value of good clusters outweighs the cost of finding them. Bottom-up clustering requires a user to specify at compile time which known function should be used to guide the clustering process. (In the Partition and Minimum Flow Cut problem classes, the user has only one choice.) Bottom-up clustering operates in three different versions depending on the data type of the feature \(f\) that guides the clustering process. The feature may be a scalar integer function (e.g., the weight function in the Partition problem); an integer vector function, (e.g., the rows of the weight matrix in the the Minimum Flow Cut problem); or a boolean vector function, (e.g., a bit vector representation of adjacency lists in an unweighted graph version of the Minimum Flow Cut problem). In each case, the bottom-up method attempts to minimize the variation in the function \(f\) over elements that are placed in common clusters. Since the final clusters are used to induce the symmetric abstraction \(\tilde{f}\) of \(f\) that appears in the abstract goal \(\tilde{G}\), pruning at the abstract level will be most successful when the value of \(f\) varies little over elements of a single cluster. Both clustering algorithms terminate when the induced abstraction space has a user-specified

---

3 The details of these clustering algorithms are not central to the claims of this paper. More sophisticated clustering algorithms are presented in [Kaufman and Rousseeuw, 1990].
**Random Clustering:** Form a list of singleton sets. Put the sets in random order. Repeatedly merge the first two sets on the list and put the merged set at the end of the list. Terminate when the abstract space induced by the clusters has the desired size.

**Bottom-Up Clustering:** Let $f$ be the feature or feature vector selected to guide the clustering. First compute $f(d)$ for each $d$ in the domain $D$. Let the initial clusters be a collection of singleton sets of the elements of $D$. Repeatedly select two sets $X$ and $Y$ to be merged. At each step, choose the two sets $X$ and $Y$ such that the function $f$ varies as little as possible over the set $X \cup Y$. Terminate when the abstract space induced by the clusters has the desired size:

- **Integer scalar features:** If $f$ is an integer scalar valued function, then at each step we choose $X$ and $Y$ to minimize the difference between the largest and smallest values of $f$ over $X \cup Y$.
- **Integer vector features:** If $f$ is a integer vector valued function, then at each step we choose $X$ and $Y$ to minimize the average over all components $f_i$ of the difference between the largest and smallest values of $f_i(z)$ for $z \in (X \cup Y)$.
- **Boolean vector features:** If $f$ is boolean vector valued function then at each step we choose $X$ and $Y$ to minimize the number of components of $f$ in which members of $X \cup Y$ disagree.

**Figure 6:** Clustering Procedures

Random clustering takes $O(n)$ time, where $n$ is the number of objects to be clustered. Bottom-up scalar clustering can be done in $O(n \log(n))$ time, by sorting the objects at the beginning of the process. Bottom-up vector clustering takes $O(n^2 l)$ time, where $l$ is the length of the feature vector.

## 5 Experimental Results

A series of experiments was run to evaluate the performance of automatically synthesized problem solvers based on the idea of approximate equivalence. The experiments were performed on the NP-hard constraint satisfaction problems defined in Figure 1. Each experiment compared the performance of a “flat” problem solver, which uses no abstraction, to the performance of “hierarchic” problem solvers, which use abstraction based on approximate equivalence. The hierarchic and flat problem solvers use the same underlying backtracking code, but with different input specifications. In the first set of these experiments, the hierarchic solver used the bottom-up clustering strategy described in Figure 6. Clustering of elements $eeE$ in the Partition problem was guided by the integer scalar function giving the weight of each element. Clustering of vertices $veV$ in the Minimum Flow Cut problem was guided by the integer vector function giving the weights of the edges from a given vertex to every other vertex. Two different methods were used to choose the size of the abstraction spaces. The “default” abstraction space has size $|S|^{1/2}$, where $|S|$ is the size of the
The "optimized" abstraction space has size $|S|^k$. The exponent $k$ was chosen manually, and separately for each problem class and clustering strategy, in order to minimize overall CPU time. The optimal values of $k$ were experimentally found to be 0.74 for the Partition problem, and 0.83 for the Minimum Flow Cut problem. Thus the default abstraction spaces ($k = 0.5$) are smaller than the optimal size.

Results of these experiments are shown in Figures 7 and 8. Each graph plots problem size against performance measured in terms of CPU time. The Partition graph examines performance on randomly generated problems with the number of elements in the set $E$ running from 4 to 16. The Minimum Flow Cut graph compares performance on randomly generated problems with the number of vertices in the graph $G$ running from 4 to 16. In both graphs, each data point represents an average of 10 randomly generated problems. Notice that for large enough problems, both the default and optimized hierarchic problem solvers run more quickly than the flat problem solver, on both the Partition and Minimum Flow Cut problem classes. Furthermore, the difference becomes more dramatic as the problem size increases. A series of single-tailed paired t-tests was run to test the significance of these results. For both the Partition and Minimum Flow Cut problem classes, the default hierarchic solver is faster than the flat solver with significance above 99% on the largest problem size. Thus the default abstraction space improves performance in comparison to the ordinary flat problem solver. Furthermore, for both problem classes, the optimized hierarchic solver is faster than the default hierarchic solver with significance above 99% on the largest problem size. Thus a careful choice of the size of the abstraction space can significantly improve performance over the default size.

A series of experiments was also run to compare the "random" and "bottom-up" clustering methods shown in Figure 6. Comparisons based on the default size abstraction space are shown in Figure 9, for the Partition problem class, and Figure 10 for the Minimum Flow Cut problem class. Notice that bottom-up clustering leads to a faster problem solver than random clustering for the Partition problem class. (The single-tailed paired t-test significance level is above 99% on the largest problem size.) Notice also that bottom-up clustering does not lead to a faster problem solver than random clustering on the Minimum Flow Cut problem class. (The single-tailed paired t-test significance level is only 29% on the largest problem size.) Comparisons based on optimal size abstraction spaces are shown in Figure 11, for the Partition problem class, and Figure 12 for the Minimum Flow Cut problem class. These graphs show that for optimally sized spaces, bottom-up clustering leads to a faster problem solver than random clustering, when applied to both the Partition problem class and the Minimum Flow Cut problem class. (Single-tailed paired t-test significance levels for each problem class are above 99% on the largest problem size.)

The comparisons between "random" and "bottom-up" clustering can be explained by considering two factors that impact on the performance of a hierarchic problem solver: (1) The cost of the clustering that is needed to construct the abstract search space; (2) The effectiveness of pruning of abstract states. On the Partition problem, bottom-up (scalar) clustering is only marginally slower $O(n \log(n))$ than random clustering $O(n)$. In addition, bottom-up clustering leads to more effective pruning of abstract states than does random

---

4The choice of the default abstraction space size is based on a rough analogy with Korf's result on the optimal size of abstraction spaces for state space search [Korf, 1987].
clustering. Therefore bottom-up clustering dominates random clustering for both
the default and optimal abstraction spaces. In contrast to this, on the Minimum Flow Cut
problem, bottom-up (vector) clustering is considerably slower $O(n^3)$ than random
clustering $O(n)$. Thus bottom-up clustering will only be useful when the cost of
clustering vertices is justified by an increase in the effectiveness of pruning of abstract
states. This cost appears not to be justified when the abstract space is small, i.e., the
default size. In this case, the clusters of vertices are so large that most information about
the graph structure is lost, and pruning is relatively ineffective, even when clusters are
chosen carefully. In contrast to this, the cost of clustering vertices is justified when
the abstract space is larger, i.e., the optimal size. The smaller clusters of vertices
preserve more of the graph structure and lead to more effective pruning of abstract
states.

6 Future Work

Future work is planned to determine whether more sophisticated clustering algorithms,
such as those discussed in [Kaufman and Rousseeuw, 1990], might yield more compact
clusters and more effective pruning at the abstract level. More sophisticated clustering
algorithms may also be necessary to handle more complex constraint satisfaction
problems. In the test problems studied so far, only one function is the “unknown function”.
Thus only one domain is available to serve as the basis of abstraction. Likewise, in the test problems studied so far,
only one feature function was available to guide clustering. More complex problems would likely include many unknown functions, defined on multiple domains, and many possible feature functions that can be used to guide clustering. For such complex problems, methods of automatically selecting the best domain(s) for abstraction, and the best features to guide clustering would likely be useful. Future work will investigate methods of automatically determining how much abstraction is suitable for attacking a given constraint satisfaction problem. For example, in the current implementation, the size of the abstraction space is chosen arbitrarily. Our experiments suggest that the size of the abstraction space is a crucial parameter that must be chosen with care. A method of automatically determining the right size for a given problem would be useful. Finally, a natural extension of this work would synthesize a multi-level abstraction hierarchy. The sequence of abstraction spaces, from most concrete to most abstract, would correspond to the sequence of partitions generated by the bottom-up clustering algorithm. Use of such a multi-level hierarchy might eliminate the need for a means of automatically choosing an abstraction space of the best possible size, since many sizes would be available at once.

7 Related Work

Abstraction techniques have been studied previously in the context of planning [Knoblock et al., 1991] and theorem proving [Giunchiglia and Walsh, 1992]. The research presented here
is similar in spirit; however, it differs by focusing on constraint satisfaction problems, rather than planning or theorem proving. A program called “HiT” for automatically constructing abstraction spaces for CSPs is presented in [Mohan, 1991]. HiT differs from the method developed here by partitioning the range, rather than the domain, in function finding problems. HiT may therefore be seen as a complementary method of building abstraction spaces. Methods of attacking hierarchical CSPs are discussed in [Mackworth et al., 1985]; however, the methods are aimed at exploiting an existing hierarchy rather than automatically constructing a new one. Abstractions based on quotas have been studied in the context of resource allocation problems [Lowry and Linden, 1992]. Approximate equivalence provides a rational reconstruction of the quota concept. Furthermore, approximate equivalence is more general, because it depends only on the algebraic form of the problem and not on semantic notions such as “resources”. Approximations and abstractions have been used to construct heuristic evaluation functions in the context of constraint satisfaction [Dechter and Pearl, 1987] and state space search [Prieditis, 1991].

8 Summary

The research reported in this paper contributes to the field of Machine Learning in two distinct ways. To begin with, it has developed an analytic learning technique for improving the performance of problem solvers on constraint satisfaction problems. The technique is called
Figure 10: Min Flow Cut CPU Time: Hierarchic Bottom-Up Clustering Default Size (Solid) v. Hierarchic Random Clustering Default Size (Dotted)

"analytic" because it operates on a declarative description of a class of constraint problems (PCSP), rather than empirical data, or data obtained from problem solving experience. The analytic technique synthesizes the components of a hierarchic problem solver, i.e., an abstraction space, an abstract goal and a set of constrained variable range generators. Since these components can be used to solve many problem instances, the computational costs of analytic learning may be amortized over many problem solving episodes. In addition, this research has developed a novel way of utilizing clustering algorithms. In particular, it has showed how clustering algorithms can be used to construct hierarchies of abstraction spaces to speed the solution of constraint satisfaction problems.

Acknowledgments

The research reported in this paper is supported by the National Science Foundation under grants IRI-9017121 and IRI-9021607. This research has also benefited from discussions with Haym Hirsh and Christopher Tong. Thanks also to Saibal Patra for programming assistance.
Figure 11: Partition CPU Time: Hierarchic Bottom-Up Clustering Optimal Size (Solid) v. Hierarchic Random Clustering Optimal Size (Dotted)

References


Figure 12: Min Flow Cut CPU Time: Hierarchic Bottom-Up Clustering Optimal Size (Solid) v. Hierarchic Random Clustering Optimal Size (Dotted)


