Using Branch-and-Bound with Constraint Satisfaction in Optimization Problems

Stephen Beale
Computing Research Laboratory
Box 30001
New Mexico State University
Las Cruces, New Mexico 88003
sb@crl.nmsu.edu

Abstract

This work[1] integrates three related AI search techniques – constraint satisfaction, branch-and-bound and solution synthesis – and applies the result to constraint satisfaction problems for which optimal answers are required. This method has already been shown to work well in natural language semantic analysis (Beale, et al, 1996); here we extend the domain to optimizing graph coloring problems, which are abstractions of many common scheduling problems of interest. We demonstrate that the methods used here allow us to determine optimal answers to many types of problems without resorting to heuristic search, and, furthermore, can be combined with heuristic search methods for problems with excessive complexity.

Introduction

Optimization problems can be solved using a number of different techniques within the constraint satisfaction paradigm. Full lookahead (Haralick & Elliott, 1980) is popular because it is easy to implement and works on a large variety of problems. Tsang and Foster (1990) demonstrated that the Essex algorithms, a variation on Freuder's (1978) solution synthesis techniques, significantly outperform lookahead algorithms on the N-Queens problem. Heuristic methods such as (Minton, et al, 1990) hold great promise for finding solutions to CSPs, but their heuristic nature preclude them from being used when **optimal** solutions are required.

Our work follows along the line of Tsang and Foster and, earlier, Freuder, but instead of using constraints as the primary vehicle for reducing complexity at each stage of synthesis, we use branch-and-bound methods focused on the optimization aspect of the problem. Furthermore, we extend Tsang and Foster's use of variable ordering techniques such as "minimal bandwidth ordering" (MBO) to a more general partitioning of the input graph into fairly independent sub-problems. We demonstrate the utility of these techniques by solving various graph coloring problems to which optimality constraints are added (for example, use as many reds as possible). These graph color optimization problems are similar to many scheduling problems of interest.

 Research reported in this paper was supported in part by Contract MDA904-92-C-5189 from the U.S. Department of Defense.

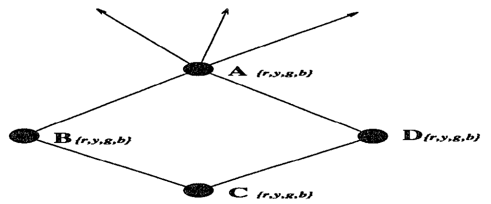

Figure 1: Subgraph: maximize the number of reds.

Optimization-Driven Solution Synthesis

The weakness of previous solution synthesis algorithms is that they do not **directly** decrease problem complexity. Constraint satisfaction methods used in combination with solution synthesis indirectly aid in decreasing complexity, and variable ordering techniques such as MBO try to direct this aid, but complexity is still driven by the number of exhaustive solutions available at each synthesis. In effect, solution synthesis has been used simply as a way to maximize the disambiguating power of constraint satisfaction for optimization problems.

Instead of concentrating on constraints, we focus on the optimization aspect and uses that to guide solution synthesis. The key technique used for optimization is branch-and-bound. Consider the subgraph of a coloring problem in Figure 1. In this subgraph, only vertex A has constraints outside the subgraph. What this tells us is that by examining only this subgraph, we cannot determine the correct value for vertex A, since there are constraints we are not taking into account. However, given a value for vertex A, we **can** optimize each of the other vertices with respect to that value.

The optimization is done in three steps. First, exhaustively determine all of the combinations of values for the vertices in the subgraph. Next, use constraint satisfaction to eliminate impossible combinations. Fi-

Step 1	Step 2	Step 3	
exhaustive	*const sat*	*b&b reduction*	
(A,B,C,D)	*(A,B,C,D)*	*(A,B,C,D)*	
(r,r,r,r)			
(r,r,r,y)			
...	...		
(r,y,r,y)	*(r,y,r,y)*	*(r,y,r,y)*	A=r
(r,y,g,y)	*(r,y,g,y)*		
(r,y,b,y)	*(r,y,b,y)*		
(r,y,b,b)			
...	...		
(y,r,r,r)			
(y,r,r,y)			
(y,r,y,r)	*(y,r,y,r)*	*(y,r,y,r)*	A=y
(y,r,y,g)	*(y,r,y,g)*		
(y,r,y,b)	*(y,r,y,b)*		
(y,g,y,y)			
(y,g,y,g)	*(y,g,y,g)*		
...	...		
(b,r,y,g)	*(b,r,y,g)*		
(b,r,y,r)	*(b,r,y,r)*	*(b,r,y,r)*	A=b
(b,r,r,g)			
(b,r,g,g)			
(b,r,g,y)	*(b,r,g,y)*		
...	...		
(g,y,g,y)	*(g,y,g,y)*		
(g,r,r,r)			
(g,r,g,r)	*(g,r,g,r)*	*(g,r,g,r)*	A=g
(g,y,g,y)	*(g,y,g,y)*		
...	...		
total: 256	*total: ??*	*total: 4*	

Figure 2: Three steps for subgraph optimization.

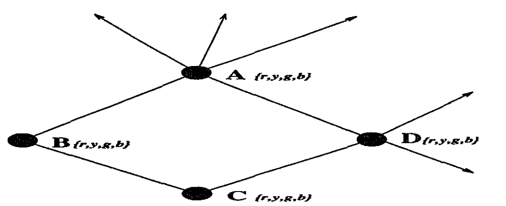

Figure 3: Subgraph: maximize the number of reds.

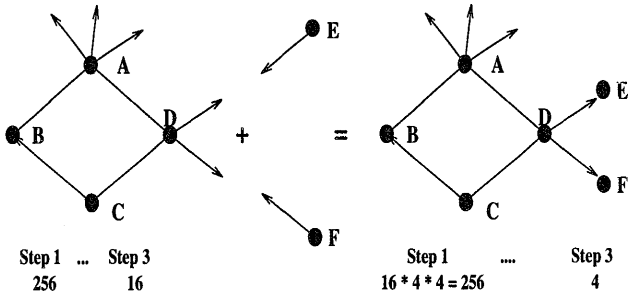

Figure 4: Solution Synthesis

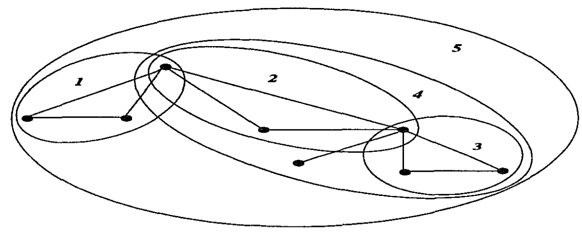

Figure 5: Subgraph Construction

nally, for each possible value of the vertex with constraints outside the subgraph, determine the optimal assignments for the other vertices. Figure 2 illustrates this process. Note that step 3 retains a combination for each possible value of vertex A, each optimized so that the maximum number of reds appear.

Consider what happens if, instead of having a single vertex constrained outside the subgraph, two vertices are constrained outside the subgraph, as in Figure 3. In this case, the assignment of correct values for both Vertices A **and** D must be deferred until later. In the branch-and-bound reduction phase, all possible combinations of values for vertices A and D must be retained, each of which can be optimized with respect to the other vertices. Phase three for this subgraph would yield 4*4=16 combinations.

Solution synthesis is used to combine results from subgraphs or to add individual vertices onto subgraphs. Figure 4 shows an example of combining the subgraph from Figure 3 with two additional vertices, E and F. Step 1 in the combination process is to take all of the outputs from the input subgraph, 16 in this case, and combine them exhaustively with all the possible values of the input vertices, or 4 each. This gives a total complexity for step 1 of 16 * 4 * 4 = 256. Constraint satisfaction can then be applied as usual. In the resulting synthesized subgraph, only vertex A has constraints outside the subgraph (assuming vertices E and F have no other constraints, as shown). The output of step 3, therefore, will contain 4 optimized answers, one for each of the possible values of A.

Synthesis of two (or more) sub-graphs into one proceeds similarly. All of the step 3 outputs of each input subgraph are exhaustively combined. After constraint satisfaction, step 3 reduction occurs, yielding the output of the synthesis. Figure 5 illustrates how subgraphs are created and processed by the solution synthesis algorithms. The smallest subgraphs, 1, 2 and 3, are processed first. Each is optimized as described above. Next, smaller subgraphs are combined, or synthesized into larger and larger subgraphs. In Figure 5, subgraphs 2 and 3 are combined to create subgraph 4. After each combination, the resulting subgraph typically has one or more additional vertices that are no longer constrained outside the new subgraph. Therefore, the subgraph can be re-optimized, with its fully reduced set of sub-answers being the input to the next higher level of synthesis. This process continues until two or more subgraphs are combined to yield the entire graph. In Figure 5, circles 1 and 4 are combined to yield circle 5, which contains the entire problem.

Note that the complexity of the processing described up to this point is dominated by the exhaustive listing of combinations in step 1. With this in mind, subgraphs are constructed to minimize this complexity (see below for a discussion of how this is accomplished in non-exponential time). For this reason, our solution synthesis will be directed at combining subgraphs created to minimize the total effect of all the step 1's. This involves limiting the number of inputs to the subgraph as well as maximizing the branch-and-bound reductions (which in turn will help minimize the inputs to the next higher subgraph). This outlook is the central difference between this methodology and previous efforts at solution synthesis, all of which attempted to maximize the effects of early constraint disambiguation. The pruning driven by this branch-and-bound method typically overwhelms the contribution of constraint satisfaction (see below for actual results).

The other difference between this approach and previous solution synthesis applications is the arrangements of inputs at each synthesis level. Tsang and Freuder both combine pairs of variables at the lowest levels. These are then combined with adjacent variable pairs at the second level of synthesis, and so on. We remove this artificial limitation. Subgraphs are created which maximize branch-and-bound reductions. Two or more subgraphs are then synthesized with the same goal - to maximize branch-and-bound reductions. In fact, single variables are often added to previously analyzed subgraphs to produce a new subgraph.

The Hunter-Gatherer Algorithm

The algorithm described below was introduced in (Beale, et al, 1996). We summarized the approach as "Hunter-Gatherer" (HG):

- branch-and-bound and constraint satisfaction allow us to "hunt down" non-optimal and impossible solutions and prune them from the search space.
- solution synthesis methods then "gather" all optimal solutions avoiding exponential complexity.

```
1 PROCEDURE SS-HG(Subgraphs)
2     FOR each Subgraph in Subgraph
3         PROCESS-SUB-GRAPH(Subgraph)
4     RETURN last value returnd by 3

5 PROCEDURE PROCESS-SUB-GRAPH(Subgraph)
      ;; assume Subgraph in form
      ;; (In-Vertices In-Subgraphs Constrained-Vertices)
6     Output-Combos <-- nil
      ;; STEP 1
7     Combos <--
         COMBINE-INPUTS(In-Vertices In-Subgraphs)
8     FOR each Combo in Combos
         ;; STEP 2
9        IF ARC-CONSISTENT(Combo) THEN
10          Output-Combos <--
               Output-Combos + Combo
      ;; STEP 3
11    REDUCE-COMBOS(Output-Combos Constr-Verts)
12    RECORD-COMBOS(Subgraph Output-Combos)
13    RETURN Output-Combos
```

A simple algorithm for HG is given. It accepts a list of subgraphs, ordered from smallest to largest, so that all input subgraphs are guaranteed to have been processed when needed in PROCESS-SUB-GRAPH. For example, for Figure 5, HG would be input the subgraphs in the order (1,2,3,4,5). Each subgraph is identified by a list of input vertices, a list of input subgraphs, and a list of vertices that are constrained outside the subgraph. Subgraph 1 would have three input-vertices, no input subgraphs, and a single vertex constrained outside the subgraph. Subgraph 4, on the other hand, has subgraphs 2 and 3 as input subgraphs, a single input vertex, and one vertex constrained outside the subgraph.

The COMBINE-INPUTS procedure (line 7) simply produces all combinations of value assignments for the input vertices and subgraphs. This procedure has complexity $O(s_1 * s_2 * ... * a^x)$, where x is the number of vertices in In-Vertices, a is the maximum number of values for a vertex, and s_i is the complexity of the input subgraph (which was already processed and reduced to a minimum by a previous call to PROCESS-SUBGRAPH). In the worst case, x will be n, the number of vertices; this is the case when the initial subgraph contains all the vertices in the graph. Of course this is simply an exhaustive search, not solution synthesis. In practice, each subgraph usually contains no more than two or three input vertices. In short, the complexity of Step 1 is the product of the complexity of the reduced outputs for the input subgraphs times the number of exhaustive combinations of input vertices. This complexity dominates the algorithm and will be what we seek to minimize below in the discussion of creating the input sub-graphs.

Lines 8-10 will obviously be executed the same number of times as the complexity for line 7. An arc consistency (line 9) routine similar to AC-4 (Mohr & Hendersen, 1986) is used. It has complexity $O(ea^2)$, where e is the number of edges in the graph. In the worst case, when the graph is a clique, e equals $n!$ because every vertex affects every other vertex. Fortunately, SS-HG is aimed at problems of lesser dimensionality. Cliques are going to have exponential complexity no matter how they are processed. For us, the only vertices that will have edges outside the subgraph are those in Constrained-Vertices. Propagation of constraint failures by the AC-4 algorithm is limited to these vertices, and, indirectly, beyond them by the degree of inter-connectedness of the graph. It should be stressed that this arc-consistency mechanism is **not** responsible for the bulk of the search space pruning, and, for certain types of problems with "fuzzy" constraints (such as semantic analysis), it is not even used. The pruning associated with the branch-and-bound optimization typically overwhelms any contribution by the constraint satisfaction techniques. See the Graph Coloring section for data comparing the efficiency of HG with and without arc-consistency.

The REDUCE-COMBOS procedure in line 11 simply goes through each combination, and keeps track of the best one for each combination of values of Constrained-Vertices. If there are two Constrained-Vertices, each of which has four possible values, REDUCE-COMBOS should return 16 combinations (unless one or more are impossible due to constraints), one for each of the possible combination of values of Constrained-Vertices, each optimized with respect to every other vertex in the subgraph. The complexity of line 12 is therefore the same as the complexity of COMBINE-INPUTS. We refer to this complexity as the "input complexity" for the subgraph, whereas the "output complexity" is the number of combinations returned by REDUCE-COMBOS, which is equal to $O(a^c)$, where c is the number of Constrained-Vertices.

With this in mind, we can re-figure the complexity of COMBINE-INPUTS. The output complexity of a subgraph is $O(a^c)$, which becomes a factor in the input complexity of the next higher subgraph. The input complexity of COMBINE-INPUTS is the product of $O(a^x)$ times the complexity of the input subgraphs. Taken together, the complete input complexity of COMBINE-INPUTS, and thus the overall complexity of PROCESS-SUBGRAPH, is $O(a^{x+c_{total}})$, where c_{total} is the total number of Constrained-Vertices in all of the input subgraphs. In simple terms, the exponent is the number of vertices in In-Vertices plus the total number of Constrained-Vertices in each of the input subgraphs. To simplify matters later, we will refer to input complexity as the exponent value $x + c_{total}$ and the output complexity as the number of vertices in Constrained-Vertices.

The complexity of SS-HG will be dominated by the PROCESS-SUBGRAPH procedure call with the highest input complexity. Thus, when creating the progression of subgraphs (see below) input to SS-HG, we will seek to minimize the highest input complexity, which we will refer to as Maximum-Input-Complexity. As will be shown, some fairly simple heuristics can simplify this subgraph creation process.

Space complexity is actually more limiting than time complexity for solution synthesis approaches. Although one can theoretically wait ten years for an answer, no answer is possible if the computer runs out of memory. The space complexity of SS-HG is dominated by the need to store results for subgraphs that will be used as inputs later. This need is minimized by carefully deleting all information once it is no longer needed; however, the storage requirements can still be quite high. The significant measurement in this area is, again, the maximum input complexity, because it determines the amount of combinations stored in line 7 of the algorithm. Again, it must be noted that this approach attempts to directly minimize the space complexity. Previous solution synthesis methods only reduced space (and time) complexity as an accidental by-product of the constraint satisfaction process. As will be shown below, their space complexity becomes unmanageable even for relatively small problems.

Subgraph Construction

Subgraph decomposition is a field in itself (see, for instance, (Bosak, 1990)), and we make no claims regarding the optimality of our approach. In fact, further research in this area can potentially improve the results reported above by an order of magnitude or more. The novelty of our approach is that we use the complexity of the HG algorithm as the driving force behind the decomposition technique.

In step 1 of the algorithm, "seed" subgraphs are formed in various regions of the graph. These seeds are simply the smallest subgraphs in a region that cover at least one vertex (so that it is not constrained outside the subgraph). Seeds can be calculated in time $O(n)$.

From there, we try to expand subgraphs until one of them contains the entire input graph. First, possible expansions for each seed are calculated, and for each seed, the one that requires the minimum input complexity is kept. The seed with the lowest input complexity expansion is then allowed to expand. A type of "iterative deepening" approach is then used. The last created subgraph is allowed to expand as long as it can do so without increasing the maximum input complexity required so far. Once the current subgraph can no longer expand, new actions are calculated for any of the subgraphs that might have been affected by the expansion of previous subgraphs. These actions are then sorted with the remaining actions and the best one is taken. The process then repeats.

One major benefit of the overall approach used by HG is that, by examining the factors which influence the complexity of the main search, we can seek to minimize those factors in a secondary search. This secondary search can be carried out heuristically because the optimal answer, although beneficial, is not required, because it simply partitions the primary search space. Optimal answers to the primary search are guaranteed even if the search space was partitioned such that the actual search was not quite as efficient as it could be.

These decomposition techniques are similar to research in nonserial dynamic programming. Bertele and Brioschi (1972), for example, demonstrate variable elimination techniques that involve single variables or blocks of variables (corresponding to subgraph extensions in HG involving a single variable or multiple variables). A variable is eliminated by creating new functions in terms of related variables that map onto the optimal value for the variable being eliminated and return an overall score for the combination. Also discussed is the "secondary optimization problem" which determines the correct order of elimination of variables. Various heuristics are given for solving this secondary problem.

Despite the obvious similarities, we feel that HG offers significant advances. By utilizing solution synthesis techniques, we are able to not only eliminate groups of variables, but can more naturally decompose the problem into subgraphs, each of which can be analyzed separately. Additionally, intermediate functions do not have to be calculated in HG. All evaluation functions

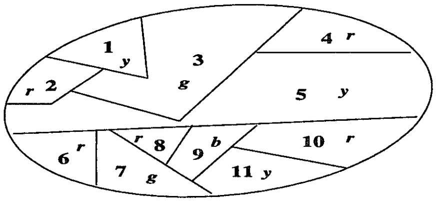

Figure 6: Graph Coloring Example

SUBGRAPH	INPUT VERTICES	INPUT SUBGRAPHS	VERTICES	CONSTRAINED VERTICES	INPUT COMPLEXITY	OUTPUT COMPLEXITY
A	1,2,3	none	1,2,3	2,3	3	2
B	5,6,7	none	5,6,7	5,7	3	2
C	none	A B	1,2,3,5,6,7	3,5,7	4	3
D	4	C	1,2,3,4,5,6,7	5,7	4	2
E	8,9,11	D	1,2,3,4,5,6,7,8,9,11	5,9,11	5	3
F	10	E	1,2,3,4,5,6,7,8,9,10,11	none	4	0

Figure 7: HG Subgraph Processing

STEP 1: exahustive combos	STEP 2: const sat	STEP 3: B&B reduction
(r,r,r) (r,r,y) (r,r,g) (r,r,b)		
(r,y,r) (r,y,y) (r,y,g) (r,y,b)	(r,y,g) (r,y,b)	(r,y,g) (r,y,b)
(r,g,r) (r,g,y) (r,g,g) (r,g,b)	(r,g,y) (r,g,b)	(r,g,y) (r,g,b)
(r,b,r) (r,b,y) (r,b,g) (r,b,b)	(r,b,y) (r,b,g)	(r,b,y) (r,b,g)
(y,r,r) (y,r,y) (y,r,g) (y,r,b)	(y,r,g) (y,r,b)	(y,r,g) (y,r,b)
(y,y,r) (y,y,y) (y,y,g) (y,y,b)		
(y,g,r) (y,g,y) (y,g,g) (y,g,b)	(y,g,r) (y,g,b)	(y,g,r)
(y,b,r) (y,b,y) (y,b,g) (y,b,b)	(y,b,r) (y,b,g)	(y,b,r)
(g,r,r) (g,r,y) (g,r,g) (g,r,b)	(g,r,y) (g,r,b)	(g,r,y)
(g,y,r) (g,y,y) (g,y,g) (g,y,b)	(g,y,r) (g,y,b)	(g,y,r)
(g,g,r) (g,g,y) (g,g,g) (g,g,b)		
(g,b,r) (g,b,y) (g,b,g) (g,b,b)	(g,b,r) (g,b,y)	
(b,r,r) (b,r,y) (b,r,g) (b,r,b)	(b,r,y) (b,r,g)	
(b,y,t) (b,y,y) (b,y,g) (b,y,b)	(b,y,r) (b,y,g)	
(b,g,r) (b,g,y) (b,g,g) (b,g,b)	(b,g,r) (b,g,y)	
(b,b,r) (b,b,y) (b,b,g) (b,b,b)		

Figure 8: Subgraph A Processing

that can apply to a particular subgraph are simply used and deleted. Evaluations that require variables not in the subgraph are simply delayed, with the optimal values for those variables determined in later subgraphs. Intermediate evaluations are associated with each subgraph, removing the need to recalculate intermediate functions. Perhaps the most important advantage of all is HG's ability to react dynamically to changing preconditions, goals or constraints. Despite its name, "dynamic" programming, DP techniques are not very flexible. Change one input or add one extra constraint and all of the intermediate functions will have to be redone. Because HG is constraint-based, all interactions of a certain change can be tracked down and minimal perturbation replanning can be accomplished. This aspect of HG is a primary goal of our future research.

Graph Coloring

Graph coloring is an interesting application to look at for several reasons. First, it is in the class of problems known as NP-Complete, even for planar graphs (see for example, (Even, 1979)). In addition, a large number of practical problems can be formulated in terms of coloring a graph, including many scheduling problems (Gondran & Minoux, 1984). Finally, it is quite easy to make up a large range of problems.

Figure 6 is a simple problem (with solution) of the type: find a graph coloring that uses four colors such that red is used as often as possible. It should be noted that even this "simple" problem produces 4^{11} exhaustive combinations, or over 4 million.

Figure 7 shows the list of input subgraphs given to SS-HG and follows the processing. The subgraphs input to SS-HG along with their composition are given in the first three columns. The vertices included in the subgraph are in column 4. The vertices constrained outside of the subgraph are listed in column 5, and the input and output complexities of processing are given in the last two columns. Figure 8 shows steps 1, 2 and 3 for subgraph A.

The most important column in Figure 7 is the input complexity. The largest value in this column, 5 for subgraph E, defines the complexity for the whole problem. This problem will be solved in time proportional to 4^5, where 4 is the number of values possible for each vertex (red, yellow, green and blue), and 5 is the maximum input complexity. This complexity compares very favorably to an exhaustive search, and to other types of algorithms, as shown below.

It is interesting to experiment with SS-HG using several different-sized (and different dimensionality) problems and compare the results to other approaches. Figure 9 presents the results of such experiments. Six different problems are solved. Each of the six problems are run (when possible) using four different algorithms:

1. The full HUNTER-GATHERER algorithm, with constraint satisfaction and branch-and-bound.
2. HG minus constraint satisfaction. This, in combination with the next, gives an indication of the relative contributions of branch-and-bound versus constraint satisfaction.
3. HG minus branch and bound.
4. Tsang and Foster's (1990) solution synthesis algorithm with minimal bandwidth ordering (MBO).

A number of comments are in order. The numbers given in the graph represent the number of intermediate combinations produced by the algorithms. This measure is consistent with that used by Tsang and Foster (1990) when they report that their algorithm outperforms several others, including backtracking, full and partial lookahead and forward checking. "DNF"

	# vertices					
	5	11	27	83	100-tree	36-square
full HG	272	848	9248	286080	2416	137904
HG - CS	272	1232	23824	1394496	2528	287616
HG - BB	784	16592	DNF	DNF	DNF	DNF
Tsang	460	18748	DNF	DNF	DNF	DNF
exhaustive	1024	10**6	10**16	10**50	10**60	10**21

Figure 9: HG Processing Results

is listed for those instances when the program was not able to run to completion because of memory limitations (more than 100MB were available).

The first fact, although obvious, is worth stating. HG was able to process all of the problems. This, in itself, is a significant step forward, as problems of such complexity have been intractable up until now. HG performed significantly better than Tsang's algorithm for every problem considered. Especially noteworthy is the 100-tree problem (a binary tree with 100 nodes). HG's branch-and-bound optimization rendered this problem (as all trees) simple, while Tsang's technique was unable to solve it.

Comparing HG without constraint satisfaction to HG without branch-and-bound is significant. Removing constraint satisfaction degrades performance fairly severely, especially in the higher complexity problems. Disabling branch-and-bound, though, has a much worse effect. Complexity soars without branch-and-bound. This confirms that our approach, aimed at optimization rather than constraints, is effective.

We attempted a seventh problem - a cube with 64 vertices. The subgraph construction algorithm described below was able to suggest input subgraphs with a maximum input complexity of 15. The HG algorithm was unable to solve this problem, since over a billion combinations would need to be computed (and stored). This underlines the point that, although HG can substantially reduce complexity, higher dimensional problems are still intractable. Nevertheless, we expect that many practical problems will become solvable using these techniques. Furthermore, we can easily estimate a problem's dimensionality using the techniques described above. This gives us a new measure of complexity which can be used evaluate problems before solutions are sought and may help in formulating the problem in the first place.

Combining HG with Heuristic Methods

One of the most interesting aspects of this work is the topological view it gives to problem spaces. The dimension of a problem becomes very important when determining its complexity. HG processes 1 dimensional (trees), and 2 (squares), 3 (trees) and higher dimensional problems by, in essence, "squeezing down" their dimensionality. Typically, real problems do not present as perfect trees, squares or cubes (or higher-dimensional objects). Natural language semantic analysis, for instance, is tree-shaped for the most part with scattered two-dimensional portions. The overall complexity of a problem, even with HG, will be dominated by the higher dimensional subproblems. This suggests the following approach to generalized problem solving:

- Analyze the problem topology.
- Solve higher dimensional portions heuristically (if their input complexity is prohibitive).
- Use HG to combine the heuristic answers for the higher dimensionality portions with the rest of the problem. This relegates possibly non-optimal heuristic processing only to the most difficult sections of the problem, while allowing efficient, yet optimal, processing of the rest.

Although it remains to be shown, the author believes that many real-world problems can be partitioned in such a way as to limit the necessity of the heuristic problem solvers in this paradigm. Natural language semantic analysis is an example of a complex problem which can be solved in near-linear time using HG.

References

Beale, S., S. Nirenburg and K. Mahesh. 1996. Hunter-Gatherer: Three Search Techniques Integrated for Natural Language Semantics. In Proc. AAAI-96. Portland, OR.

Bertele, U. and F. Brioschi. 1972. *Nonserial Dynamic Programming.* New York: Academic Press.

Bosák, J. 1990. *Decomposition of Graphs.* Mass.: Kluwer.

Even, S. 1979. *Graph Algorithms.* Maryland: Computer Science Press.

Freuder, E.C. 1978. Synthesizing Constraint Expressions. *Communications ACM* 21(11): 958-966.

Gondran, M. and M. Minoux. 1984. *Graphs and Algorithms.* Chichester: Wiley.

Haralick, R. and G. Elliott. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 14:263-313.

Minton, S., M. Johnston, A. Philips and P. Laird. 1990. Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In Proc. AAAI-90, Boston, MA.

Mohr, R. and Henderson, T.C. 1986. Arc and Path Consistency Revisited. *AI* 28: 225-233.

Tsang, E. and Foster, N. 1990. Solution Synthesis in the Constraint Satisfaction Problem, Tech Report. CSM-142, Dept. of Computer Science, Univ. of Essex.