Hunter-Gatherer:
Applying Constraint Satisfaction, Branch-and-Bound and Solution Synthesis to Computational Semantics

Stephen Beale
2 May 1997
CMU-LTI-97-XXX

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Committee:
Sergei Nirenburg (advisor)
Jaime Carbonell
Robert Frederking
Victor Raskin

Abstract

This work[1] integrates three related AI search techniques and applies the result to processing computational semantics, both in the analysis of source text to discover underlying semantics, as well as in the planning of target text from an input semantic representation. We summarize 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 together" all optimal solutions while avoiding exponential complexity.

Each of these three general AI techniques will be described. We will look at how how they have been used to solve a variety of problems. These general techniques were extended and used in novel ways in this project. We describe these extensions in detail and give examples of how they were applied to computational semantic processing. A major contribution of this work will also be in showing how and why Natural Language is a prime candidate for applying these methods, and how they can enable near-linear time processing. As part of this discussion, we demonstrate the important result that by converting Text Planning to a constraint satisfaction problem, Means-End type planning can be replaced by an efficient constraint-based search through a complex tree. We examine the results in the light of the Mikrokosmos Machine Translation project. This project is a large-scale Spanish-English MT system implemented at New Mexico State University. We will be able to evaluate the control mechanism presented here against a large corpus of sample texts. In particular, we will show that a search space in the billions or more can be reduced to a few thousand or less, with a corresponding decrease in run-time.

Finally, we try to explicitly uncover the benefits of the Hunter-Gatherer system. We apply HG to several different types of graph coloring constraint problems and examine its behavior. We discuss several properties of HG, including the important notions of soundness and completeness. Finally, we show how, in its most basic sense, HG operates by reducing the dimensionality of a problem, and why this produces near linear-time processing for the kinds of topologies found in problems of computational semantics.

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

Contents

1. Introduction

Fifty six million, six hundred eighty seven thousand, and forty. A big number, to be sure. This is the number of possible semantic analyses for an **average** sized sentence in the Mikrokosmos Machine Translation project. Complex sentences have gone past the trillions. If every combination could be accurately judged in one thousandth of a second, it would still take almost a day to analyze the average sentence. And you can forget about the hard ones.

And yet, understanding natural language sentences is intuitively not an exponential affair. Not every word in a sentence is dependent on every other word. Sentences can generally be broken up into relatively independent areas of self-contained meaning which then interact on a higher level to produce the meaning of the whole. This research aims to recognize that fact, analyze it, and apply appropriate AI techniques to take advantage of it.

This work integrates three related AI search techniques and applies the result to processing computational semantics, both in the analysis of source text to discover underlying semantics, as well as in the planning of target text using input semantics. We summarize 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 together" all optimal solutions while avoiding exponential complexity.

We will describe each of these general AI techniques and look at how how they have been used to solve a variety of problems. These general techniques were then extended or used in novel ways in this project. We will describe these extensions in detail and give examples of how they were applied to computational semantic processing. A major contribution of this work will also be in showing how and why Natural Language is a prime candidate for applying these methods, and how they can enable near-linear time processing. As part of this discussion, we will demonstrate the important result that by converting Text Planning to a constraint satisfaction problem, Means-End type planning can be replaced by an efficient constraint-based search through a complex tree. We will examine the results in the light of the Mikrokosmos Machine Translation project. This project is a large-scale Spanish-English

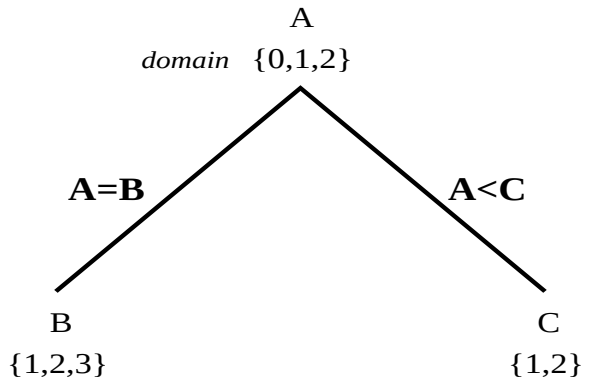


Figure 1: Basic Constraint Satisfaction Problem

MT system implemented at New Mexico State University. We will be able to evaluate the control mechanism presented here against a large corpus of sample texts. In particular, we will show that a search space in the billions or more can be reduced to hundreds, with a corresponding decrease in run-time.

Finally, we try to explicitly uncover the benefits of the Hunter-Gatherer system. We apply HG to several different types of graph coloring constraint problems and examine its behavior. We discuss several properties of HG, including the important notions of soundness and completeness. Finally, we show how, in its most basic sense, HG operates by reducing the dimensionality of a problem, and why this produces near linear-time processing for the kinds of topologies found in problems of computational semantics.

1.1. Section Overviews

This introduction will give brief descriptions of each of the main points to be covered in detail below. The interested reader will then be able to judge which sections of the report are of immediate interest. The text is divided into main sections as follows.

1.2. Hunters and Gatherers in AI: An introduction to constraint-based AI techniques

In recent years, Constraint Satisfaction Problems (CSP) have received a good deal of attention in the computer science world (see (Tsang, 93) for a detailed look at CSP). Con-

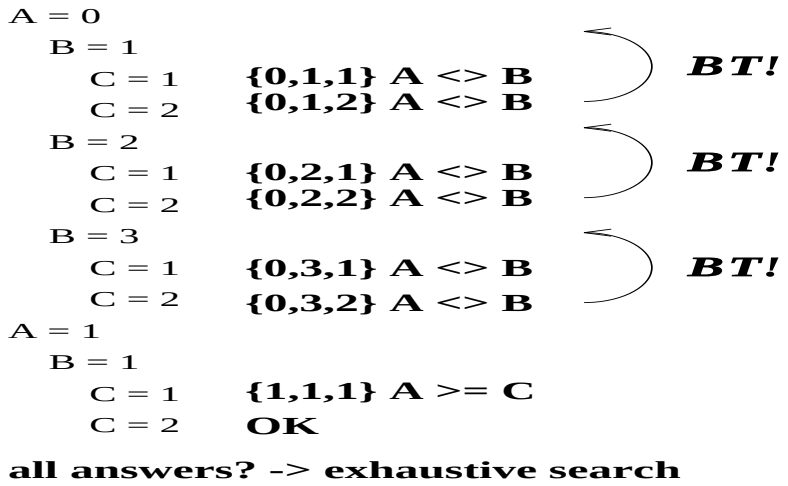


Figure 2: Backtracking in Constraint Satisfaction

straint satisfaction techniques enable search procedures to prune off substantial portions of the search tree by identifying solution sets/subsets that cannot meet input constraints. For example, in Figure 1, three variables, A, B and C, with the domains given, are to have values chosen subject to the two constraints: [A = B] and [A < C].

A naive parser would try every combination of A, B and C until it found one that met the constraints (Figure 2). If all correct answers are desired, an exhaustive search is required. Notice that certain combinations that always lead to failure (A=0, B=x) are tried again and again. Constraint satisfaction programming techniques can eliminate this unnecessary processing. For instance, it can reduce A's domain to {1,2}, since 0 as a value for A can never satisfy the [A = B] constraint (0 is not in B's domain). Actually, A's domain can be reduced to {1}, since 2 as a value for A can never satisfy the [A < C] constraint. Once A is reduced to {1}, then B can be reduced by similar reasoning to {1}, and C can be reduced to {2}. Only one choice for each variable is left; thus, the answer is achieved without search. In section 2.1, we will overview the algorithms used to achieve constraint "consistency" in a search.

Solution synthesis (section 2.2) is a method of generating **all** valid answers to a search problem. Instead of working from the top of the tree down,[2] solution synthesis attempts

[2]CSP does not assume tree shaped search spaces. Neither do we, for that matter, although computational semantics generally present as tangled trees, a fact we take advantage of (see below).

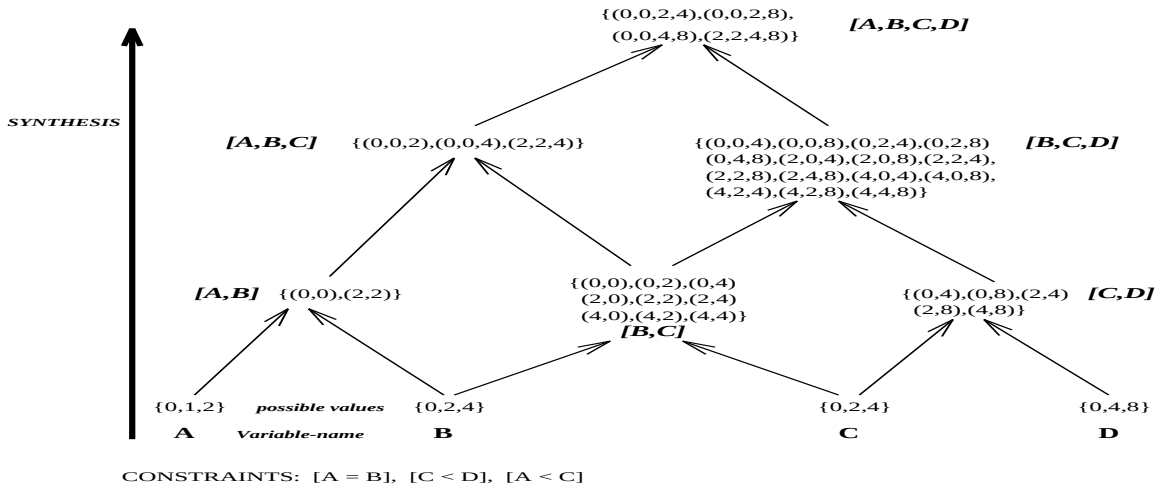


Figure 3: Basic Solution Synthesis Technique

to combine legal combinations of nodes from the bottom up. Figure 3 gives a graphical overview of the process. Used in conjunction with constraint satisfaction techniques, solution synthesis is a powerful method for avoiding exponential time requirements associated with conventional tree search.

Branch-and-bound techniques (section 2.3) can be used to find the optimal solution without resorting to heuristics.[3] The basic idea is displayed in Figure 4. In that Figure, a graph is displayed with the cost of each arc listed. The order of arc traversal used in a branch-and-bound algorithm is marked by circled numbers. The shortest total path accumulated at any point is always expanded next. If there is a tie, as is the case at the start node when no paths have been chosen yet, then the shortest next arc is taken. Arcs without circled numbers in Figure 4 were not tested, because they were extensions of paths of length greater than the length of a valid solution. Branch-and-bound identifies and eliminates paths which can be guaranteed to have more costly solutions than some valid solution.

Each of these AI methods - constraint satisfaction, solution synthesis and branch-and-bound - will be described and the appropriate literature reviewed. In addition, we will take a look at some related techniques useful in processing constraint satisfaction problems. An attempt is made to place Hunter-Gatherer in its proper relation to linear programming and nonserial dynamic programming methods. We will also compare HG with the well-known

[3]They can be used with heuristics as well, if desired.

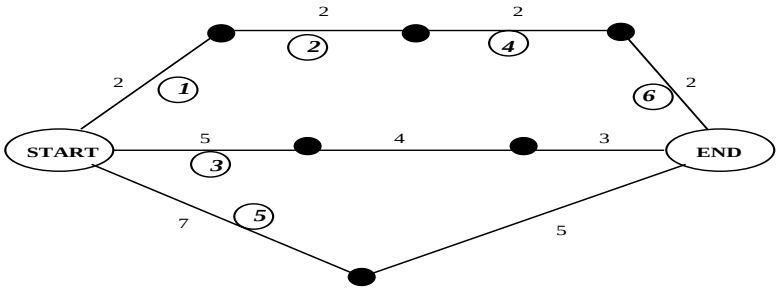


Figure 4: Branch-and-Bound Example

work of Steven Minton (Minton, et al 1990), who was able to solve certain types of complex constraint satisfaction problems in linear time. And finally, we overview the scheduling techniques used at the Robotics Institute of Carnegie Mellon University.

1.3. The Hunter-Gatherer Control Architecture

In this section, we will discuss how constraint satisfaction, solution synthesis and branch-and-bound techniques were modified and adapted in creating the Hunter-Gatherer control architecture. We will demonstrate how constraint satisfaction information allows us to identify "circuits" of inter-dependence in the input. Solutions for each circuit are synthesized apart from the rest of the problem.

Branch-and-bound techniques were refined for use with the solution synthesis and constraint satisfaction methods. As each of the "circuits" mentioned above are synthesized, certain variables will no longer be dependent on nodes outside of the circuit. These variables can be optimized, with non-optimal solutions "bound" and eliminated. This particular merging of techniques accounts for the majority of savings produced by our system.

1.4. The Mikrokosmos Machine Translation System

Here we attempt to give the reader the background necessary to understand the application of HG to problems in computational semantics. In particular, we present an outline of the Mikrokosmos Machine Translation project. Topics discussed include knowledge sources such as ontologies (semantic models of the world) and computational lexicons, and the principles

used in extracting and using semantic (and syntactic) constraints to determine the correct meaning of an input text.

1.5. Using Hunter-Gatherer in Semantic Analysis

In this section, we present the Hunter-Gatherer algorithm in the light of Computational Semantic problems. We exemplify the steps of the algorithm using semantic analysis problems taken from the Mikrokosmos corpus. Results from processing a wide variety of inputs are given.

1.6. Hunter-Gatherer in Natural Language Generation

The HG principles were capitalized on to change means-end text planning into a simpler and faster type of Constraint Satisfaction Problem (CSP), which can be solved using optimized CSP algorithms. Means-End planners typically are inefficient. We demonstrate in this section how we can set up our planner as a Constraint Satisfaction Problem, and then let HG techniques automatically identify macro-plans which can then be efficiently "searched" to find the optimal plan.

1.7. Natural-Language - a "Natural" Constraint Satisfaction Problem

Implicit in the work above is the idea that computational semantics naturally fits into the class of problems for which these types of constraint satisfaction techniques apply. We will introduce and discuss the notion of "local inter-dependence." We will show that natural language semantics fits this notion well. We will examine the effect of "long-distance dependencies" on this type of processing. We will look at the class of problems for which the methods described above do **not** work well and show that computational semantics typically does not conform to such problems.

1.8. Other Applications for the Hunter-Gatherer Technology

Hunter-Gatherer is a general purpose algorithm that can be applied to many types of problems. While it is not the intent of this report to describe such applications in detail, nor to assert that HG is superior to other methods for those applications (although this may be a topic for further research), we thought it was important to give a brief overview of at least one other application, namely, 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). It is quite easy to make up a large range of problems, from simple to complex, single dimensional to multi-dimensional. This fact allows us to compare HG to other techniques for a variety of problems. And finally, graph coloring examples provide a more suitable framework for discussing the concepts of graph topology, taken up in the next section.

1.9. Discussion

The final section of this work seeks to collect in one place several of the most important features of the Hunter-Gatherer control architecture. We begin by stating outright what we believe are the novel contributions made to the state of the art. We discuss the formal characteristics of HG, including the important notions of soundness and completeness. Finally, we point out some of the interesting aspects behind the success of HG. Included in this is a discussion on how HG implements island processing, and how the notion of graph topology can be used to explain the benefits (and limitations) of Hunter-Gatherer.

2. Hunters and Gatherers in AI

Search is the most common tool for finding solutions in artificial intelligence. That being true, the most common work in the AI smithery is aimed at refining that tool. Such work can be classified into two main areas:

1. Reducing the search space. Looking for sub-optimal or impossible solutions. Removing them. Killing them. "Hunting"

2. Efficiently extracting answer(s) from the search space. Collecting satisfactory answer(s). "Gathering"

The hunter is savage. She kills without mercy to serve the needs of the clan. The gatherer is gentle. He takes in all that supply the needs of the group.[4]

Much work has been done with regard to the hunters. Finding and using "heuristics" to guide search intelligently has been a major focus. Heuristics are necessary when other techniques cannot reduce the size of the search space to reasonable proportions. Under such circumstances, "guesses" have to be made to guide the search engine to the area of the search space most likely to contain acceptable answers. "Best-first" search (see, among many others, (Charniak, et al, 1987)) is an example of how to **use** heuristics. Such a methodology is almost always combined with some concept of "satisficing" (Newell & Simon, 1972), a determination of whether a given answer is "good enough," regardless of the fact that other "better" solutions may still be present in the search space.

Discovering appropriate heuristics for any given problem is another matter. Often "experts" in the field must be consulted, their methods for finding solutions analyzed, and means of implementing those methods computationally invented. MYCIN (Buchanan & Shortliffe, 1984) is a typical "expert system" whose search is based upon heuristic medical knowledge.

This research does not address heuristic search. Heuristics, by definition, are guesses, and thus can only lead to the most probable answers. In contrast, this work claims that by using

[4]This "hunter-gatherer" business is undoubtedly politically-incorrect. Be advised that we are in no way denigrating pre-industrialist cultures.

the modified CSP techniques presented below, the most optimal solution can be guaranteed for computational semantic problems, even under reasonable time constraints. This is not to say that heuristics cannot be valuable in computational semantics, and in fact, heuristics could be easily added to the methods described here.

Some further clarifications regarding heuristics must be made. Heuristics can be used more conservatively as a method to order the search, without resorting to "satisficing" determinations which may leave optimal solutions undiscovered. Combined with other methods (namely branch-and-bound and solution synthesis), these ordering heuristics may potentially provide substantial pruning of the search space. There are several general ordering heuristics that are common in constraint satisfaction algorithms. These heuristics are discussed below in the "Other Strategies for CSPs" section. In addition, knowledge sources which evaluate solutions can also be seen as heuristics. For example, the fact that the Mikrokosmos lexicon constrains the **AGENT** of a **SPEAK** event to be a **HUMAN** is simply a heuristic. Metonymic speech, such as *The White House said today ...* often overrides the basic constraint. In all that follows, we assume a given knowledge source with all of its inherent heuristics. The control mechanisms developed here then find the best solution, **given** that knowledge.

The "hunting" techniques applied in this research are most closely related to the field of CSPs; a detailed summary of this work is given below. "Branch-and-bound" methods have been modified and adapted for work with CSPs, and are thus also described below. The recent work of Steve Minton and colleagues also provides an interesting comparison and will be summarized, along with techniques in linear programming and the related field of nonserial dynamic programming. An overview with scheduling techniques used at the Robotics Institute of Carnegie Mellon University is also presented.

"Gathering" has been studied much less in AI. Most AI problems are content with a single "acceptable" answer. Heuristic search methods generally are sufficient. Certain classes of problems, however, demand **all** correct answers. Optimization problems, for instance, either need to examine all correct answers and select the most optimal, or they must utilize certain techniques such as branch-and-bound which allow them to ignore certain sections of the search space which can be guaranteed not to contain optimal answers. "Solution synthesis" addresses the need to determine all correct answers to a constraint satisfaction problem. Solution Synthesis techniques (Freuder, 1978; Tsang & Foster, 1990) iteratively combine

(gather) partial answers together to arrive at a complete list of all correct answers. Often, this list is then rated according to some separate criteria in order to pick the most suitable answer. Solution synthesis methods will be described below.

In sections 3 and following, the modifications and interactions of these "hunter-gatherers" utilized in this project will be developed. In particular, it will be shown that by combining branch-and-bound techniques with a novel solution synthesis method, the best solution for a computational semantic problem can be found in near-linear time. Also, the conversion of a means-end type text planner to a CSP will be described.

2.1. Constraint Satisfaction Problems

The seminal paper on CSP is (Mackworth, 1977). In this paper he describes the central concepts of CSP and gives the basic consistency algorithms described below. (Mackworth & Freuder, 1985) and, later, (Mohr and Henderson, 1986) improved on the basic algorithms. (Freuder, 1978) introduces solution synthesis and (Tsang and Foster, 1990) present improved methods. (Tsang, 1993) is an indispensable resource for anyone interested in CSP.

The simple Constraint Satisfaction Problem presented in the introduction is a good example of the pitfalls of uninformed backtracking. Figure 2 displays the ever-present fact-of-life inherent in uninformed search. In that problem, seven combinations of A,B and C were tested before one was found that met the input constraints. If all answers to the problem were required, an exhaustive search (18 combinations in this case) would be required. All of this despite the fact that no search was required at all! By applying the basic principles of CSPs, the single correct answer falls out without search. This is not to say that search is never required; in most problems it is. But for many types of problems, using CSP techniques can drastically reduce the amount of processing needed.

DEFINITIONS

The following terminology will be used throughout this report. A CSP consists of a set of **variables**, also called **nodes**, or, in connection with discussions about graphs, **vertices**. Each variable can take on a **value** taken from a set of values, called its **domain**. A variable's domain will always be presented between curly braces { }. An assignment of a value to a variable will be written <A,2>, meaning variable A has value 2. There are two types of

constraints. **Unary constraints** restrict the domain of a variable without reference to any other variable. For instance, [A > 3] is a unary constraint for variable A. It will be assumed that there is one (or zero) unary constraint per variable.[5]

Binary constraints[6] restrict the values a variable can take by comparing it to another variable. For instance, [A < B] is a binary constraint between A and B. Actual constraints will always be presented between straight brackets [ ]. They will also be represented as follows: C_A is a unary constraint for A, C_{AB} is a binary constraint between the arc AB. An **arc** is defined to be any pair of variables for which there is a binary constraint. It is assumed there is only one binary constraint per arc. In general, capital letters at the beginning of the alphabet (A,B,C ...) will represent variables, while those at the end (X,Y,Z) will represent unknown values.

A solution to a CSP is an assignment to each variable of a value taken from that variable's domain, such that all the unary and binary constraints are satisfied. A complete solution is the set of all such solutions for a CSP. A solution will be represented as a set of values enclosed within parentheses, such as (1,2,4,0), where 1 is the value of the first variable (generally A in these abstract examples), 2 is the value of the second variable, etc.. Alternatively, a solution sometimes is represented more explicitly as a set of variable-value assignments such as (<A,1>,<B,2>,<C,4>,<D,0>). A set of solutions will be a list of solutions enclosed in curly brackets: {(1,2,4,0),(1,2,4,1), ... }. Partial solutions will be represented similarly, with the assignment of values to variables determined by context. The fact that a partial solution satisfies a given constraint will be represented (X,Y) SAT C_{AB} for binary constraints,[7] and X SAT C_A for unary constraints.[8]

Figure 5A is a slightly more complex CSP. A few more possible values have been added to the domains of each variable to more clearly demonstrate the types of consistency discussed below. Unary constraints are also included. The three central "consistency" checks used for CSPs are node consistency, arc consistency and path consistency.

---

[5] If there is more than one, they can always be combined into one with ANDS: [A > 0 AND A < 10]

[6] Higher order constraints are also possible, but since they currently not used in Mikrokosmos, they will not be discussed here. Higher order constraints can be used by HG with no problems.

[7] Meaning that the partial solution which assigns value X to variable A and Y to B satisfies the binary constraint C_{AB}.

[8] The assignment of value X to variable A satisfies the unary constraint C_A.

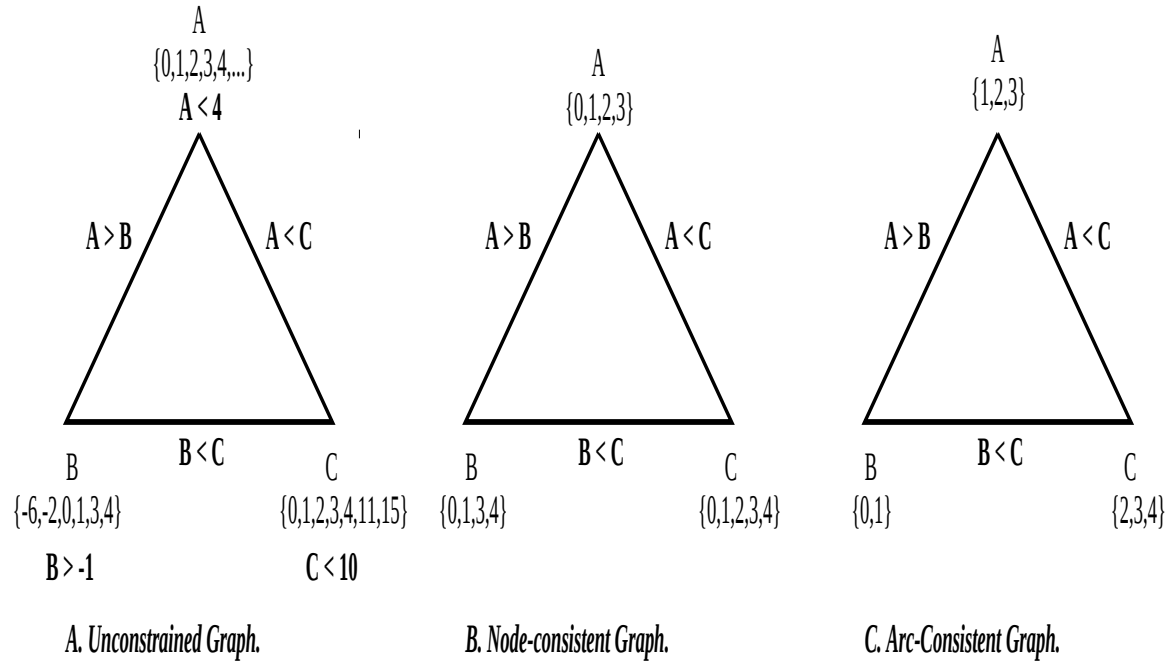


Figure 5: Constraint Satisfaction Problem - Consistency

1. Node Consistency.

   Node consistency (NC) is simply a state where the domains of each variable have been reduced to the set of all possible values that satisfy the unary constraints. In Figure 5A-B, the results of node consistency processing are displayed. For instance, the values 11 and 15 are removed from the domain of C since they do not satisfy C's unary constraint that C < 10.

   Node consistency can often be assumed, either because there are no unary constraints, or because the unary constraints were used in setting up the problem. For instance, in the Mikrokosmos semantic analyzer, the unary constraints correspond to picking the correct lexicon entries based on the root word and surrounding syntax.

   A simple algorithm for enforcing node consistency is as follows:

```
1 PROCEDURE NC-1
2     FOR each variable, V
3         C_V <- unary constraint for V
4         IF C_V is null ;; no unary constraints
5             THEN do nothing
6             ELSE
7                 FOR each value in V's domain, X
```

```
8               IF X SAT C_V
9                  THEN do nothing
10                 ELSE remove X from the domain of V
```

This algorithm has time-complexity *O(an)*, where a is the maximum size of the domains and n is the number of variables to be examined. In all that follows, our primary goal will be to reduce the time complexity with respect to n, the number of variables.[9] Thus, in this respect, NC-1 is linear in time.

2. Arc Consistency.

Arc consistency (AC) ensures that the binary constraints connecting any two nodes are satisfied. There have been various implementations of AC. AC-1, AC-2 and AC-3 are presented in (Mackworth, 1977). (Mohr & Henderson, 1986) present the optimal AC-4 algorithm. The naive approach, AC-1, is given below:

```
1 PROCEDURE AC-1
2    NC-1 ;; ensure NC first
3    REPEAT
4       Changed <- false
5       FOR each arc, AB
6          C_AB <- binary constraint between A and B
7          FOR each value in the domain of A, X
8             IF there is a value in the domain of B, Y
               such that (X,Y) SAT C_AB
9                THEN do nothing
10               ELSE
11                  REMOVE X from the domain of A
12                  Changed <- true
13   UNTIL NOT Changed
```

This algorithm cycles through each potential arc[10] to determine if for each value in the domain of the first variable there exists a value in the domain of the second variable that can satisfy the binary constraint on that arc. If a value is found for which this is not true, it is removed from the domain of the first variable. If any values are removed,

[9] In computational semantics, the maximum size of any domain is probably around 10, with rare exceptions (10 word senses for a single lexical item). However, the average case domain size is less than 3. Thus, terms which involve domain size will be insignificant compared to the number of variables (actual words) in a problem, which can get to 40 or higher in long sentences. Later, time and space complexity terms involving the maximum number of constraints per variable will be used. A similar argument applies.

[10] Note: the arcs must be identified before using this procedure. This can be accomplished by examining each constraint once. The algorithm above assumes separate arcs for both directions, i.e. AB and BA.

the process must be started again from the beginning, since the removal might cause some arc-inconsistency in an arc that was already tested.

In Figure 5B-C, the arc-consistency results are shown. For example, in Figure 5B, the domain of B is {0,1,3,4}. When the arc AB is tested with the binary constraint [A > B] and the value 0 for variable A is tested, it is discovered that no values of B remain such that [0 > B]. Thus 0 must be removed from the domain of A. Note that this will impact on the testing of arc CA. When the value 1 is tested for C, no values for A remain in its revised domain {1,2,3} such that [A < 1]. Thus 1 will need to be removed from the domain of C. This demonstrates the necessity of testing all arcs again after any value is removed, since if arc CA is tested before AB, the value of 1 for C is still consistent.

If a is the maximum size of the domains, e is the number of arcs and n is the number of variables, then there will be at most na values to be checked. If, in the worst case, one value is deleted for each iteration in the UNTIL loop, then the loop will be run na times. The first FOR loop (line 5) can be run a maximum e times per iteration of the REPEAT loop. The second FOR loop (line 7) can be called a maximum of a times per iteration of the first FOR loop, and the IF statement in line 8 may involve a maximum of a searches through the domain of B per each iteration of the second FOR loop. Thus, the complexity of the second FOR loop is $O(a^2)$, the complexity of the first FOR loop is $O(ea^2)$, and the complexity of the entire procedure is $O(nea^3)$. In unconstrained graphs, e can be at most n^2. However, in computational semantics, typically, e is proportional to linear n (the reasons for this will be discussed below). Thus AC-1 typically has time-complexity $O(n^2a^3)$, which is not linear with respect to the number of variables.

Various improvements to AC-1 have been offered. AC-4 (Mohr and Henderson, 1986) is the optimal worst-case algorithm, offering time complexity of $O(ea^2)$. In AC-1, whenever a value is removed from a variable's domain, every arc has to be rechecked. AC-4 recognizes the fact that a value in a variable's domain "supports" an identifiable list of values in other variables. For instance, in Figure 5C, the value 0 in B "supports" the value 1 in A. The constraint [A > B] is enabled by this support. Remove the value 0 from B's domain and the value 1 in A is no longer supported. Therefore, if a value is deleted in one variable, only the values in other variables that are supported by the

deleted value must be checked. Furthermore, the number of "supports" for a certain value can be tracked. Examining Figure 5C again, the two values in B, 0 and 1, **both** support the value 2 in A. This will be represented by saying SUPPORTS(B,0) = {(A,2,C_{AB}),...}[11] and SUPPORTS(B,1) = {(A,2,C_{AB}),...}. The fact that the value 2 in A has two values in B supporting C_{AB} is represented as *SUPPORT(A,2,*C_{AB}*) = 2*.[12] If one of the supporting values in B was removed, *SUPPORT(A,2,*C_{AB}*) = 1* would still be supported by one value. Thus, by recording how many supports each value-constraint pair has initially, and by subtracting one from that total each time a supporting value is removed, it can be determined when a value-constraint pair no longer can be supported. Whenever a value-constraint pair has no supports, the value must be deleted. An algorithm for AC-4 follows:

```
1 PROCEDURE AC-4
2     NC-1 ;; ensure node consistency first

      ;; Step I: INITIALIZATION
3     INITIALIZE array SUPPORT[V,X,AB] = 0
4     INITIALIZE SUPPORTS[V,X] = nil
5     INITIALIZE FAIL-LIST = nil

      ;; Step II: SETTING UP SUPPORT VARIABLES
6     FOR each arc, AB
7         FOR each value X in variable A
8             FOR each value Y in variable B
9                 IF (A,B) SAT C_AB
10                    THEN
11                        INCREMENT(SUPPORT[A,X,C_AB])
                          ;; increment the number of supporters for X
12                        APPEND(SUPPORTS[B,Y],(A,X,C_AB))
                          ;; record that Y supports X
13            IF (SUPPORT[A,X,C_AB] = 0)
              ;; X has no supports for this arc
14                THEN
15                    APPEND(FAIL-LIST,(A,X))
16                    REMOVE X from the domain of A

      ;; Step III: REMOVING UNSUPPORTED VALUES
17    WHILE FAIL-LIST <> { }
18        PICK first label (A,X) from FAIL-LIST
```

[11] That is, the assignment of value 0 to variable B supports the assignment of value 2 to variable A with respect to C_{AB}, along with any other supports.

[12] That is, the assignment of the value 2 to A has two supports for the C_{AB} constraint.

```
19      FAIL-LIST = FAIL-LIST - (A,X)
20      LABELS-SUPPORTED <- SUPPORTS[A,X]
21      FOR each label (B,Y,C_BA) in LABELS-SUPPORTED
        ;; determine if the supported value has any other supports left
22          DECREMENT(SUPPORT[B,Y,C_BA])
23          IF ((SUPPORT[B,Y,C_BA] = 0)
            AND NOT-MEMBER((B,Y),FAIL-LIST) ;; not pending failure already
            AND MEMBER(Y,DOMAIN(B))) ;; not already failed
24              APPEND(FAIL-LIST,(B,Y))
25              REMOVE Y from the domain of B
```

The INITIALIZATION stage is trivial, and, practically speaking, can be combined with step II. In step II, if the number of arcs is e, the maximum number of values in a variable is a, then it is easy to see that the inner-most IF statement (line 9) is executed a^2e times. In step III, in the worst case, the WHILE loop will be executed once for each value. If the number of variables is n, then there can be at most an values to be deleted. Finally, there can be at most a labels to be examined in the inner FOR loop (line 21), so that the total complexity of step III is $O(a^2n)$.[13] Note that n, while proportional to e, is always less than or equal to it in connected graphs. Therefore, the total complexity of AC-4 is $O(a^2n + a^2e) = O(a^2e)$. Again, in computational semantics e is typically proportional to n; thus, thus the time complexity for AC-4 is typically $O(a^2n)$.

Because space becomes an issue in AC-4, the space complexity will also be analyzed. The space complexity is dominated by the SUPPORTS array. There can be at most one entry in SUPPORTS for every variable-value pair, or an entries. Each entry can support every other variable-value pair, again an. Since there can be an entries, each of can hold up to an bits of information, the total complexity is $O(a^2n^2)$. Since e is equivalent to n^2, this is the same as $O(a^2e)$, which is what is reported in the literature. Again, it should be pointed out that in computational semantics, instead of every variable-value pair supporting every other variable-value pair (an), typically it is more on the line of ca, where c is a constant. Thus, implementations of AC-4 for natural

[13]Note that (Tsang, 1993) as well as (Mohr and Henderson, 1986) give the complexity of step 3 as $O(a^2e)$. They assume each arc can add a items to the FAIL-LIST, so that the WHILE loop can be executed ae times. It is easy to see, though, that this is excessive. A value can only be deleted once from its domain. The APPEND(FAIL-LIST,...) statements (lines 15 and 24) can be implemented so that duplicate variable-value pairs are not added, and the MEMBER(Y,DOMAIN(B)) statement (in line 23) prevents any variable-value pair that has been deleted from being re-added to FAIL-LIST. Thus, an is the correct limit for the WHILE loop.

language semantics typically have a space complexity of $O(a^2n)$, which is linear with respect to the number of variables.

It should be noted for completeness that Bessiere and Cordier (1993) present another algorithm, AC-6, which improves upon AC-4 in two important ways. First, although it retains the same worst-case complexity of $O(a^2e)$, it has better average-case complexity for many problems. More importantly, AC-6 reduces the space complexity to $O(ae)$. Hunter-Gatherer, at present, employs an algorithm similar to AC-4. Upgrading to AC-6 may produce significant improvements, especially in the more constraint-oriented problems of graph coloring. Semantic analysis problems actually do not make use of constraints (for pruning, via an AC-4 algorithm) very heavily, for reasons discussed below; therefore, this improvement would not affect these types of problems.

3. Path Consistency.

Note that Figure 5C still contains some value combinations that theoretically could be removed ahead of time. For instance, the partial solution (B,C) = (1,2) is never possible. When B = 1 is assigned, A must be constrained to {2,3} to meet the [A > B] constraint. However, with C = 2, the [A < C] constraint could no longer be satisfied. Path consistency is an attempt to eliminate impossible partial solutions.

Path consistency requires that for any path[14] (A,B,C,...M), then for any value assignment A = X and M = Y, there must exist a value assignment for each of the variables B,C,...,L such that all binary constraints on **adjacent** variables are satisfied. In Figure 5C, the path in question is (B A C). When we assign B = 1 and C = 2, there is no value X for A left such that (X,B) SAT C_{AB} **AND** (X,C) SAT C_{AC}.

Path consistency (PC) algorithms can be found in (Tsang, 1993) and (Mohr and Hendersen, 1986). The optimal algorithm runs with time complexity $O(a^3n^3)$ and space complexity $O(a^3n^3)$. Obviously, these complexities are much higher than those for AC-4 and NC-1.

Fortunately, PC algorithms are not necessary. A dynamic application of AC algorithms performs the same function. To illustrate, assume that a pre-processor performs AC-4 on Figure 5A, yielding the CSP in Figure 5C. Next, submit the CSP to a AC-enabled search algorithm. Such an algorithm would choose one value for B, say B = 1. This,

[14] A path is a set of variables, each variable of which has an arc between itself and its adjacent variables.

in effect, creates an "artificial island",[15] with the domain of B = {1}. Since we have removed the value 0 from the domain of B, the AC-enabled search can dynamically eliminate values in A and C which are supported only by B = 0. This would remove A = 1. After removing A = 1, the AC-enabled search would also eliminate C = 2. Thus, a dynamic AC-enabled search algorithm performs the same function as PC algorithms. The algorithm for an AC-enabled algorithm will be given below.

2.2. Solution Synthesis

Solution synthesis is a method used to generate all solutions to a CSP. That is, all assignments of values to variables that satisfy the problem's constraints are produced by a solution synthesis algorithm. Often, this set of solutions can be further judged according to some separate criteria to obtain the optimal answer. For instance, in a modified traveling salesperson problem in which all cities must be visited, but in an order subject to certain constraints,[16] solution synthesis can generate the list of all possible answers that meet the constraints, from which the most optimal answer (presumably the one with the least total mileage) could be picked.

It is important to realize that solution synthesis, in itself, is simply a way to organize search that happens to be useful when all solutions are required. One could just as well perform a depth first search and, when a solution is found, backtrack and continue searching for more answers. Solution synthesis techniques remove the need for backtracking since all possible answers are calculated at each synthesis level. Furthermore, solution synthesis can be used to focus the effects of other methods, most notably constraint satisfaction. (Freuder, 1978) introduced solution synthesis and (Tsang & Foster, 1990) refined it. Both of their research will be described below. We extend the work of Tsang and Foster and combine it with branch-and-bound techniques, all of which will be described in the section 3.

[15] Discussion on "islands" follows below.

[16] For example, Baltimore must come first because the supplies need to be picked up, then the cities where the perishable goods are sold must be next, etc.

2.2.1. General Algorithm

Solution synthesis constructs the set of all possible solutions by iteratively combining smaller solutions while propagating constraints. A simplified algorithm[17] which implements solution synthesis follows:

```
1 PROCEDURE SS-1
     ;; initialize SOLUTION SET to set of order-1 solutions
2    SOLUTION-SET <- nil
3    FOR each variable, V
4       FOR each value, X in the domain of V that meets unary constraint C_V
5          SOLUTION-SET = SOLUTION-SET + (< V, X >)

6    FOR I = 2 to n, where n = number of variables
        ;;Create all partial solutions of length I
        ;;At this point, SOLUTION-SET contains all solutions of
        ;; length I-1
7       SOLUTION-SET <- SYNTHESIZE(SOLUTION-SET,I)

8 PROCEDURE SYNTHESIZE(SOLUTION-SET,I)
9    SOLUTION-SET-TEMP <- nil
10   FOR each SOLUTION-A in SOLUTION-SET
11      FOR each SOLUTION-B in SOLUTION-SET
12         IF SOLUTION-A and SOLUTION-B have I distinct variables
b          AND all like-variable assignments are the same THEN
13            POSSIBLE-SOLUTION <- UNION(SOLUTION-A,SOLUTION-B)
14            IF POSSIBLE-SOLUTION meets all I-ARY CONSTRAINTS
b             AND NOT-MEMBER(POSSIBLE-SOLUTION,SOLUTION-SET-TEMP)
c             AND SYNTHESIZED?(POSSIBLE-SOLUTION,SOLUTION-SET,I) THEN
15               SOLUTION-SET-TEMP <- SOLUTION-SET-TEMP + POSSIBLE-SOLUTION
16   RETURN SOLUTION-SET-TEMP

17 PROCEDURE SYNTHESIZED?(POSSIBLE-SOLUTION,SOLUTION-SET,I)
      ;; Make sure all sub-solutions of length I-1 of the new solution
      ;; are in SOLUTION-SET (which contains all valid solutions of order I-1.)
18    OK <- true
19    FOR all COMBOs of I-1 < V, X > pairs in POSSIBLE-SOLUTION
20       IF NOT-MEMBER(COMBO,SOLUTION-SET) THEN
21          OK <- false
22    RETURN OK
```

[17] Note that this is simplified; Freuder's algorithm will be described below.

Before giving an example, a walk-through of the algorithm would be instructive. At the most abstract level, SS-1 creates partial solution sets of order k[18] by combining solution sets of order k-1. The initial solution set of order 1 is created in lines 3-5; a solution is added for each value of each variable. From there, solution sets of higher order are created using SYNTHESIZE (lines 8-16).

In SYNTHESIZE, solution sets of order I are created. The input SOLUTION-SET contains all solutions of order I-1. POSSIBLE-SOLUTIONs are created by combining compatible solutions of order I-1 that differ only in one variable.[19] For example, a solution of variables (A,B,C) combined with a solution of variables (A,B,D) would combine to create a solution of variables (A,B,C,D).[20] "Compatible" combinations are those which have like-variables assigned similarly. For example, $(< A,1 >, < B,2 >)$ can combine with $(< A,1 >, < C,3 >)$ to create a POSSIBLE-SOLUTION $(< A,1 >, < B,2 >, < C,3 >)$, but $(< A,1 >, < B,2 >)$ cannot combine with $(< A,4 >, < C,3 >)$ because the assignments $< A,1 >$ and $< A,4 >$ are not compatible (line 12b checks for compatibility).

Each POSSIBLE-SOLUTION must meet three tests (lines 14a-c). First, all I-ary constraints must be met. For instance, for an order 2 POSSIBLE-SOLUTION involving variables A and B, the constraints C_{AB} and C_{BA} must be met. Order 3 possible solutions must meet 3-ary constraints, and so on. In computational semantics, there are only binary constraints, so order 3 solutions and above will always pass this test. Second, line 14b simply ensures that duplicate solutions are not added. For instance, when adding variable C onto a partial solution (A,B), solutions for (A,B,C) are obtained. Later, when adding on variable B to (A,C), the same solutions would be obtained. Line 14b prevents this. Line 14c ensures that solutions of length I meet the constraints already calculated for solutions of length I-1. This is the synthesizing step. A simple example will illustrate. In order to allow a POSSIBLE-SOLUTION = $(< A,0 >, < B,1 >, < C,2 >)$, a solution of order 3, the following order-2 solutions must exist:

$$\{(< A,0 >, < B,1 >), (< A,0 >, < C,2 >), (< B,1 >, < C,2 >)\}$$

[18] A solution of order k is an assignment of values to k variables; a solution set of order k is the set of all solutions of order k.

[19] Two solutions, each of order I-1, which differ in only one variable, will combine to give a solution of order I.

[20] Line 12a guarantees SOLUTION-A and SOLUTION-B differ by one variable. Line 13 performs the combination.

In other words, an order-N solution cannot have any sub-solutions of order N-1 that were not already identified. Because of the way order I-1 solutions are combined, most of these sub-solutions will be present. For instance, in the example above a solution of variables (A,B) was combined with a solution of variables (B,C) to create a solution of variables (A,B,C). Thus, we do not need to check the (A,B) or (B,C) sub-solutions. However, the sub-solution involving variables (A,C) was synthesized and must be checked. If, for this example, an order I-1 solution ($< A, 0 >, < C, 2 >$) does not exist, then this combination of values is unacceptable. The SYNTHESIZED? procedure in lines 17-22 performs this check. Again, this step ensures that any new order-I solution only contains order-(I-1) sub-solutions that had already been deemed legitimate.

In order to start moving this discussion towards natural language semantics, the solution synthesis algorithms will be exemplified using a simple computational semantic problem. Consider the following sentence:

(1) IBM acquired Jacob-Smith for ten-million-dollars.

For simplicity, assume that Jacob-Smith[21] and ten-million-dollars are phrasal entries in the lexicon. Also assume that we have an ontology,[22] or model of the world, that maps each of these words into the concepts[23] as shown in Figure 6. *IBM* (referred to below as I) maps into a single concept, **ORG**. *acquired* (referred to as A) maps into two possible concepts. The first, **TAKE-OVER**, constrains *IBM* to be an **ORG** and *Jacob-Smith* to be an **ORG**. Refer to Figure 6 for the rest of the possible assignments and constraints.

The SS-1 algorithm would proceed as follows. First, it would initialize SOLUTION-SET to the set of all order-1 nodes in lines 2-5. For convenience, we will present SOLUTION-SET as a group of subsets arranged according to the variables involved:[24]

---

[21] An imaginary company name, as well as a person's name.

[22] See Section 4 for a discussion of ontologies and their place in computational semantics

[23] Concepts will be in CAPS. We use very simple concepts symbolized with English words, and simply list the constraints that would be specified in an ontology. We also assume an inheritance hierarchy (HUMAN IS-A ANIMATE), and a mechanism for identifying metonymy; for instance (ORG IS-A ANIMATE) because an ORG has MEMBERS that are HUMAN, and (ORG IS-A INANIMATE), because an ORG has a BUILDING that is an INANIMATE. Metonymy will be discussed in more depth below.

[24] Here, for clarity, we present each solution as a set of variable-value pairs, i.e $N_I = \{(< I, ORG >)\}$. After this, we generally will present only the values, with the assignment to variables obvious in context, i.e $N_I = \{(\text{ORG})\}$.

WORD	CONCEPT	CONSTRAINTS	EXAMPLE
IBM (I)	ORG		
acquired (A)	TAKE-OVER (T-O)	[I=ORG TAKE-OVER J=ORG]	
	OBTAIN (OBT)	[I=ANIMATE OBTAIN J=INANIMATE]	
Jacob-Smith (J)	HUMAN (HUM)		
	ORG		
for (F)	COST	[A=EVENT FOR T=MONEY]	I bought it for $10
	BENEFIC (BEN)	[A=EVENT FOR T=ANIMAL]	I bought it for Sam
	PURPOSE (PUR)	[T=EVENT FOR T=EVENT]	I bought it for mowing the lawn.
	DURATION (DUR)	[T=EVENT FOR T=TIME]	I hid for 10 hours.
ten-million-dollars (T)	MONEY (MON)		

Figure 6: Concept assignments for SS example.

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T-O >), (< A, OBT >)\}$
$N_J = \{(< J, HUM >), (< J, ORG >)\}$
$N_F = \{(< F, COST >), (< F, BEN >), (< F, PUR >), (< F, DUR >)\}$
$N_T = \{(< T, MON >)\}$

For example, N_A has two possible solutions, the first of which assigns the concept **TAKE-OVER** (T-O) to A (*aquired*).

If necessary, unary constraints would be applied in line 3. In computational semantics, unary constraints correspond to selecting the appropriate word-senses from the lexicon based on the word used and the surrounding syntax. In this case, those constraints were applied before beginning the algorithm.

Before line 6 is executed, then:

SOLUTION-SET = APPEND(N_I, N_A, N_J, N_F, N_T)

Next, all higher order nodes, including the final solution, are created in steps 6 and 7. First, order-2 nodes are constructed. When SYNTHESIZE is called with I=2, SOLUTION-

SET contains all the solutions of order-1. In SYNTHESIZE, these order-1 solutions are combined into order-2 solutions. For instance, if, in lines 10 and 11:

SOLUTION-A = $(< I, ORG >)$
SOLUTION-B = $(< A, T-O >)$

Then, together, A and B have two distinct variables (I and A), checked in line 12, there are no like-variables, so line 12 is true, so in line 13:

POSSIBLE-SOLUTION = $(< I, ORG >, < A, T-O >)$

Line 14a is important only when I=2, as in this case, because computational semantic problems only have binary constraints. Two binary constraints must be examined, C_{IA} and C_{AI}. As shown in Figure 6, I does not constraint A for any value of A, but $< A, T-O >$ constrains I to be of type ORG. Since $< I, ORG >$ obviously meets this constraint, line 14a is true. No other solutions have been added for order-2 solutions, so line 14b is also true. Finally, since the only two sub-solutions of POSSIBLE-SOLUTION of order 1, $(< I, ORG >)$ and $(< A, T-O >)$, came directly from SOLUTION-SET, SYNTHESIZED? will obviously return true. In fact, SYNTHESIZED? will always be true for POSSIBLE-SOLUTIONS of order-2. It only becomes relevant for order-3 and higher nodes, when previously non-existent lower order sub-solutions are possible.

An example of when the binary constraint in line 14a rejects a POSSIBLE-SOLUTION occurs for

SOLUTION-A = $(< A, T-O >)$
SOLUTION-B = $(< J, HUM >)$

which yields

POSSIBLE-SOLUTION = $(< A, T-O >, < J, HUM >)$

The binary constraint, C_{AJ}, specifies that for the assignment $< A, T-O >$, J must be an ORG. However, the assignment $< J, HUM >$ does not meet this constraint, so this POSSIBLE-SOLUTION is rejected. Note that using literal constraints to reject solutions leads to problems in semantic analysis. HG overcomes these problems using branch-and-bound techniques as the primary vehicle for search pruning instead of constraint satisfaction.

See below for details, both on the problems created by using literal constraints, and how HG overcomes it.

A complete listing of order-2 solutions for this example is shown below, with solutions rejected by binary constraints identified:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{IJ} = {(ORG,HUM),(ORG,ORG)}
N_{IF} = {(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{IT} = {(ORG,MON)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} ;; eliminate (T-O,HUM),(OBT,HUM)
N_{AF} = {(T-O,COST),(T-O,BEN),(T-O,PUR),(T-O,DUR),
(OBT,COST),(OBT,BEN),(OBT,PUR),(OBT,DUR)}
N_{AT} = {(T-O,MON),(OBT,MON)}
N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{JT} = {(HUM,MON),(ORG,MON)}
N_{FT} = {(COST,MON)} ;; eliminate (BEN,MON),(PUR,MON),(DUR,MON)

Order-3 nodes are then synthesized by combining order-2 solutions. Note that from this point on, no reference needs to be made to constraints, because all the binary constraints information is implicit in the order-2 solution set. An example of synthesisizing an order-3 solution follows:

SOLUTION-A (from N_{IA}) = (ORG,T-O)
SOLUTION-B (from N_{AJ}) = (T-O,ORG)

Together, A and B have three distinct variables (I, A and J), and the only like-variable, A, has the same value assignment in both, so:

POSSIBLE-SOLUTION = $(< I, ORG >, < A, T-O >, < J, ORG >)$

Because there are no n-ary constraints for n $>$ 2, line 14a will be true from here on. It will also be assumed, starting now, that line 14b will prevent duplicate solutions from being added. Thus, the SYNTHESIZED? procedure called in line 14c is the only part left needing comment. In this case, the following sub-solutions of order-2 taken from POSSIBLE-SOLUTION are:

$(< I, ORG >, < A, T-O >)$
$(< A, T-O >, < J, ORG >)$
$(< I, ORG >, < J, ORG >)$

The first two come directly from SOLUTION-A and B. The third, however, was a result of the synthesis; therefore, it must be checked to see if it is a valid order-2 solution. A quick look at the list of order-2 solutions shows that this solution is present in N_{IJ}. Therefore, POSSIBLE-SOLUTION is a valid synthesis.

An example of a POSSIBLE-SOLUTION that does not meet the SYNTHESIZED? criterion occurs for:

SOLUTION-A (from N_{JF}) = (HUM,BEN)
SOLUTION-B (from N_{JT}) = (HUM,MON)

This gives:

POSSIBLE-SOLUTION = $(< J, HUM >, < F, BEN >, < T, MON >)$

Three sub-solutions of order-2 can be extracted from POSSIBLE-SOLUTION:

$(< J, HUM >, < F, BEN >)$
$(< J, HUM >, < T, MON >)$
$(< F, BEN >, < T, MON >)$

Again, the first two come directly from SOLUTION-A and B. The third, however, was synthesized. This time, though, the synthesized sub-solution cannot be found in the list of order-2 solutions, thus it cannot be allowed.

A complete list of order-3 solutions follows:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{IAF} = {(ORG,T-O,COST),(ORG,T-O,BEN),(ORG,T-O,PUR),(ORG,T-O,DUR),
(ORG,OBT,COST),(ORG,OBT,BEN),(ORG,OBT,PUR),(ORG,OBT,DUR)}
N_{IAT} = {(ORG,T-O,MON),(ORG,OBT,MON)}
N_{IJF} = {(ORG,HUM,COST),(ORG,HUM,BEN),(ORG,HUM,PUR),(ORG,HUM,DUR),
(ORG,ORG,COST),(ORG,ORG,BEN),(ORG,ORG,PUR),(ORG,ORG,DUR)}
N_{IJT} = {(ORG,HUM,MON),(ORG,ORG,MON)}
N_{IFT} = {(ORG,COST,MON)}
N_{AJF} = {(T-O,ORG,COST),(T-O,ORG,BEN),(T-O,ORG,PUR),(T-O,ORG,DUR),
(OBT,ORG,COST),(OBT,ORG,BEN),(OBT,ORG,PUR),(OBT,ORG,DUR)}
N_{AJT} = {(T-O,ORG,MON),(OBT,ORG,MON)}
N_{AFT} = {(T-O,COST,MON),(OBT,COST,MON)}
N_{JFT} = {(HUM,COST,MON),(ORG,COST,MON)}

It is interesting to note that the solution sets N_{IAF}, N_{IJF} and N_{AJF} all carry along values for F that, with a little bit of thought, could be eliminated. In N_{IJF}, N_{IJT} and N_{JFT}, inconsistent values for J are also kept. Freuder's algorithm, as well as Tsang's and our own, eliminate this.

Order-4 solutions are formed similarly:

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,T-O,ORG,BEN),(ORG,T-O,ORG,PUR),
(ORG,T-O,ORG,DUR),(ORG,OBT,ORG,COST),(ORG,OBT,ORG,BEN),
(ORG,OBT,ORG,PUR),(ORG,OBT,ORG,DUR)}
N_{IAJT} = {(ORG,T-O,ORG,MON),(ORG,OBT,ORG,MON)}
N_{IAFT} = {(ORG,T-O,COST,MON),(ORG,OBT,COST,MON)}
N_{IJFT} = {(ORG,HUM,COST,MON),(ORG,ORG,COST,MON)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

Finally, the order-5 solutions are synthesized:

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

At each stage, higher order solutions are created by combining lower order solutions, while ensuring no unacceptable lower-order sub-solutions are introduced. The order-n solution set contains all of the valid answers for the problem.

Although various improvements to this algorithm will be offered, all solution synthesis algorithms will have the same worst-case time complexity of $O(a^n)$. Assume a CSP for which all combinations of values for every variable meet all constraints. Furthermore, assume each variable has a values in its domain. For such a CSP, there exist a^n solutions. It is easy to see, therefore, that line 15 must be executed a^n times when the SYNTHESIZE procedure is called the last, n^{th} time. It is the nature of CSPs that worst-case (algorithmic) time behavior is always exponential because the number of possible solutions is always, theoretically, exponential. However, "worst-case" can be redefined non-algorithmically, and with respect to a certain class of problems, to mean the "typical" worst class complexity as measured experimentally. Such measurements will be described below. It should also be noted the complexity of HG's solution synthesis mechanism, combined with its branch-and-bound pruning, can be measured analytically (see section 3). This gives us the important ability to determine a given problem's actual complexity before processing.

2.2.2. Freuder's Algorithm

Freuder's algorithm (Freuder, 1978) eliminates some of the waste present in SS-1. Instead of only propagating constraints upward from lower-order nodes to higher order nodes, Freuder also propagates constraints downward, from higher order nodes to lower order nodes.

An example would clarify best. Suppose the following order-1 solutions were found for a simple CSP:

N_A = {(1),(2)}
N_B = {(3),(4)}
N_C = {(5),(6)}
N_D = {(7)}

Assume binary constraints were then used to acquire the following order-2 solutions:

N_{AB} = {(1,3),(2,3)}
N_{AC} = {(1,5),(1,6),(2,5)}
N_{AD} = {(1,7),(2,7)}
N_{BC} = {(3,5),(4,5),(4,6)}
N_{BD} = {(3,7),(4,7)}
N_{CD} = {(5,7),(6,7)}

At this point, it can be deduced that an assignment $< B, 4 >$ will never be compatible with any assignment for A. SS-1, however, blindly constructs the order-3 solutions:

N_{ABC} = {(1,3,5),(1,3,6),(2,3,5)}
N_{ABD} = {(1,3,7),(2,3,7)}
N_{BCD} = {(3,5,7),(4,5,7),(4,6,7)}

Clearly, (4,5,7) and (4,6,7) in N_{BCD} are impossible. SS-1 eventually gets the correct answer because it cannot construct any order-4 solutions with the assignment $< B, 4 >$, but it wastes much effort in doing it.

Freuder suggests allowing multiple downward propagation of constraints. For instance, once it can be determined that a solution set of order I contains no instances of a particular sub-solution of order I - 1, that sub-solution can be eliminated from the order I - 1 solutions. Above, since the sub-solution $(< B, 4 >)$ does not occur in N_{AB}, $(< B, 4 >)$ can be removed from the order-1 solutions:

$N_B = \{(3)\}$;; removed (4)

Whenever a solution is removed, all higher order solutions involving that solution must also be removed (upward propagation). In addition, whenever a solution is removed, downward propagation can occur again, possibly resulting in the removal of more solutions. In the example above, solutions of order-2 involving $< B, 4 >$ must be removed, giving:

$N_{BC} = \{(3,5)\}$;; removed (4,5) and (4,6)
$N_{BD} = \{(3,7)\}$;; removed (4,6)

Before creating new order-3 solutions, downward propagation can be repeated, since the assignment $< C, 6 >$ no longer is compatible with any assignment of B. Therefore, removing (6) from N_C yields:

$N_C = \{(5)\}$

which can be re-propagated upward to form the level-2 solutions:

$N_{AC} = \{(1,5),(2,5)\}$;; removed (1,6)
$N_{CD} = \{(5,7)\}$;; removed (6,7)

The resulting complete list of order-2 solutions is:

$N_{AB} = \{(1,3),(2,3)\}$
$N_{AC} = \{(1,5),(2,5)\}$
$N_{AD} = \{(1,7),(2,7)\}$
$N_{BC} = \{(3,5)\}$
$N_{BD} = \{(3,7)\}$
$N_{CD} = \{(5,7)\}$

Creating order-3 solutions from this yields the smaller set:

$N_{ABC} = \{(1,3,5),(2,3,5)\}$
$N_{ABD} = \{(1,3,7),(2,3,7)\}$
$N_{BCD} = \{(3,5,7)\}$

from which the order-4 solutions can be calculated easily:

$N_{ABCD} = \{(1,3,5,7),(2,3,5,7)\}$

SS-1 needs to be modified to include this downward propagation. One major change is that SOLUTION-SETs for all the previous levels need to be stored. Also, after a solution set is formed, it needs to be analyzed to see if it excludes any sub-solutions of the next lower order. If it does, these sub-solutions will need to be removed from the lower order SOLUTION-SET. Whenever a solution is removed from a SOLUTION-SET, higher order solutions utilizing it must also be removed, and the SOLUTION-SET it was removed from needs to be re-checked for downward propagation possibilities. We refer the reader to (Freuder, 1978) for implementation details.

To conclude the description of SS-FREUDER, the computational semantic example presented above will be re-worked. The order-1 solutions would be calculated the same as before, and are repeated here for convenience:

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T-O >), (< A, OBT >)\}$
$N_J = \{(< J, HUM >), (< J, ORG >)\}$
$N_F = \{(< F, COST >), (< F, BEN >), (< F, PUR >), (< F, DUR >)\}$
$N_T = \{(< T, MON >)\}$

The order-2 nodes are then calculated:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{IJ} = {(ORG,HUM),(ORG,ORG)}
N_{IF} = {(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{IT} = {(ORG,MON)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} N_{AF} = {(T-O,COST),(T-O,BEN),(T-O,PUR),(T-O,DUR),
(OBT,COST),(OBT,BEN),(OBT,PUR),(OBT,DUR)}
N_{AT} = {(T-O,MON),(OBT,MON)}
N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{JT} = {(HUM,MON),(ORG,MON)}
N_{FT} = {(COST,MON)}

In the DOWNWARD-PROPAGATE step, it will be determined that all assignments $< F, BEN >, < F, PUR > and < F, DUR >$ can never be combined with any values for T, and that the assignment $< J, HUM >$ is incompatible with any value of A. These facts will be propagated downward to the order-1 constraints:

$N_I = \{(< I, ORG >)\}$

$N_A = \{(< A, T-O >), (< A, OBT >)\}$
$N_J = \{(< J, ORG >)\}$;; removed $(< J, HUM >)$
$N_F = \{(< F, COST >)\}$;; removed $(< F, BEN >), (< F, PUR >), (< F, DUR >)$
$N_T = \{(< T, MON >)\}$

The removals can then be upward-propagated to order-2 nodes again:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{IJ} = {(ORG,ORG)} ;; removed (ORG,HUM)
N_{IF} = {(ORG,COST)} ;; removed (ORG,BEN),(ORG,PUR),(ORG,DUR)
N_{IT} = {(ORG,MON)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)}
N_{AF} = {(T-O,COST),(OBT,COST)}
;; removed(T-O,BEN),(T-O,PUR),(T-O,DUR),(OBT,BEN),(OBT,PUR),(OBT,DUR)
N_{AT} = {(T-O,MON),(OBT,MON)}
N_{JF} = {(ORG,COST)} ;; removed (HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{JT} = {(ORG,MON)} ;; removed (HUM,MON)
N_{FT} = {(COST,MON)}

No further downward propagation is possible, so the order-3 solutions are synthesized:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{IAF} = {(ORG,T-O,COST),(ORG,OBT,COST)}
N_{IAT} = {(ORG,T-O,MON),(ORG,OBT,MON)}
N_{IJF} = {(ORG,ORG,COST)}
N_{IJT} = {(ORG,ORG,MON)}
N_{IFT} = {(ORG,COST,MON)}
N_{AJF} = {(T-O,ORG,COST),(OBT,ORG,COST}
N_{AJT} = {(T-O,ORG,MON),(OBT,ORG,MON)}
N_{AFT} = {(T-O,COST,MON),(OBT,COST,MON)}
N_{JFT} = {(ORG,COST,MON)}

From here, order-4 and order-5 solutions are simple:

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,OBT,ORG,COST)}
N_{IAJT} = {(ORG,T-O,ORG,MON),(ORG,OBT,ORG,MON)}
N_{IAFT} = {(ORG,T-O,COST,MON),(ORG,OBT,COST,MON)}
N_{IJFT} = {(ORG,ORG,COST,MON)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

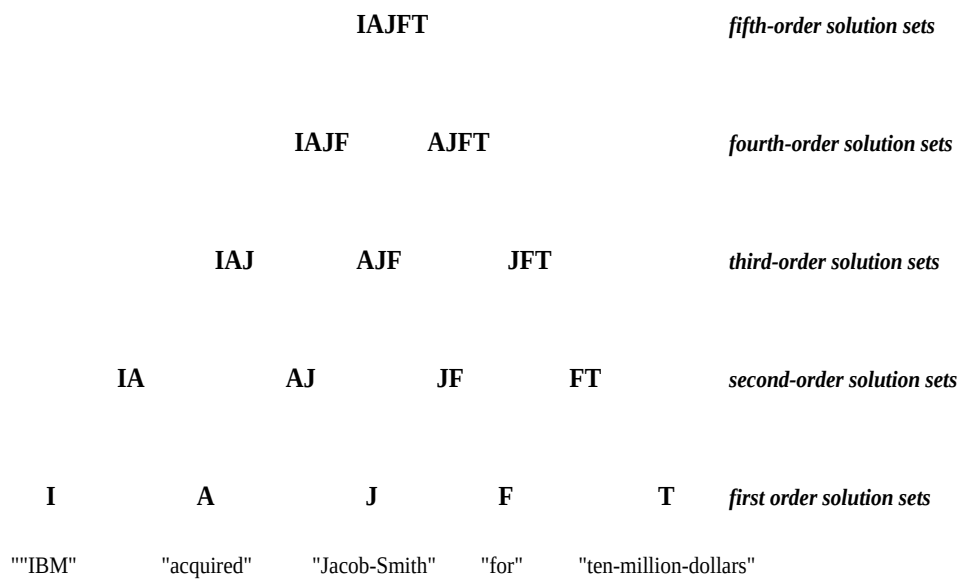


Figure 7: SS-TSANG Solution Set Construction.

2.2.3. Tsang's Algorithm

Tsang's algorithms, defined in (Tsang and Foster, 1990), also known as the Essex algorithms, improve upon SS-FREUDER by limiting the number of second-order solutions (and thus limiting the number of higher order solutions as well). SS-TSANG sets up an arbitrarily ordered list of the variables, and then constructs second-order solutions only for variable pairs that are adjacent in that list. Third-order solutions are then synthesized from "adjacent" second-order solution sets, etc.. Figure 7 displays the process for the computational semantic example.

SS-TSANG is exactly the same as SS-1, except that the number of order-2 solutions is restricted.[25] This restriction could be easily added by constructing a list of the variables, then only allowing order-2 solutions that involve adjacent variables in the list. Synthesizing higher-order solutions will proceed exactly as in SS-1.[26] Because they are so similar, with the changes fairly obvious, the algorithm will not be presented here.

[25]SS-TSANG can also be built upon SS-FREUDER if (limited) propagation is desired.

[26]Because synthesizing order-3 solutions occurs only between order-2 solutions with 3 distinct variables (SS-1 line 12a), this will only occur between "adjacent" (as shown in Figure 7) order-2 solutions, as all other combinations would lead to 4 distinct variables. Likewise, higher order solutions only are synthesized from adjacent solutions of the next lower order.

The computational semantic example solutions would be synthesized in the following manner. Order-1 nodes would be constructed first, exactly the same as before:

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T-O >), (< A, OBT >)\}$
$N_J = \{(< J, HUM >), (< J, ORG >)\}$
$N_F = \{(< F, COST >), (< F, BEN >), (< F, PUR >), (< F, DUR >)\}$
$N_T = \{(< T, MON >)\}$

Assuming a list of the variables as in Figure 7,[27] the order-2 solution set is constructed the same as for SS-1, except only solutions which utilize adjacent variables are allowed:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} ;; eliminate (T-O,HUM),(OBT,HUM)
N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}
N_{FT} = {(COST,MON)} ;; eliminate (BEN,MON),(PUR,MON),(DUR,MON)

Level 3 solutions are then synthesized from these:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{AJF} = {(T-O,ORG,COST),(T-O,ORG,BEN),(T-O,ORG,PUR),(T-O,ORG,DUR),
(OBT,ORG,COST),(OBT,ORG,BEN),(OBT,ORG,PUR),(OBT,ORG,DUR)}
N_{JFT} = {(HUM,COST,MON),(ORG,COST,MON)}

Level 4 solutions follow:

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,T-O,ORG,BEN),(ORG,T-O,ORG,PUR),
(ORG,T-O,ORG,DUR),(ORG,OBT,ORG,COST),(ORG,OBT,ORG,BEN),
(ORG,OBT,ORG,PUR),(ORG,OBT,ORG,DUR)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

Finally, the correct solution set is generated:

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

Propagation, as in SS-FREUDER, can be added. For example, once the order-2 solutions are calculated above, it can be deduced that the assignment $< J, HUM >$ is incompatible

[27] This ordering is arbitrary. The efficiency of SS-TSANG is greatly affected by the ordering chosen. This fact is recognized and expanded upon in our work.

with all assignments to variable A, and the assignments $< F, BEN >, < F, PUR > and < F, DUR >$ are incompatible with variable T. This information can be propagated downward to order-1 solutions to give:

$N_I = \{(< I, ORG >)\}$
$N_A = \{(< A, T-O >), (< A, OBT >)\}$
$N_J = \{(< J, ORG >)\}$
$N_F = \{(< F, COST >)\}$
$N_T = \{(< T, MON >)\}$

This can then be upward propagated to order-2 solutions to give:

N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} N_{JF} = {(ORG,COST)}
N_{FT} = {(COST,MON)}

Higher order solutions follow:

N_{IAJ} = {(ORG,T-O,ORG),(ORG,OBT,ORG)}
N_{AJF} = {(T-O,ORG,COST),(OBT,ORG,COST)}
N_{JFT} = {(ORG,COST,MON)}

N_{IAJF} = {(ORG,T-O,ORG,COST),(ORG,OBT,ORG,COST)}
N_{AJFT} = {(T-O,ORG,COST,MON),(OBT,ORG,COST,MON)}

N_{IAJFT} = {(ORG,T-O,ORG,COST,MON),(ORG,OBT,ORG,COST,MON)}

The major benefit of SS-TSANG is that the number of solution sets at each level is minimized. Also important is the fact that this algorithm is ideal for parallel implementations (see (Tsang and Foster, 1990)). There are two disadvantages:

1. Full downward propagation is impossible. Downward propagation in SS-FREUDER relies on having information about allowable combinations of sub-solutions. However, SS-TSANG only determines allowable combinations of adjacent nodes; therefore, propagation can only proceed from these nodes. A limited amount of propagation between adjacent nodes is possible. This drawback is related to the next problem.

2. Ambiguity is carried forward needlessly if two constrained variables are far apart in list. Consider what would happen in the computational semantic example if the ordering of variables was changed to (T I A J F). The following order-2 solutions would be obtained:

N_{TI} = {(MON,ORG)} N_{IA} = {(ORG,T-O),(ORG,OBT)}
N_{AJ} = {(T-O,ORG),(OBT,ORG)} ;; eliminate (T-O,HUM),(OBT,HUM)
N_{JF} = {(HUM,COST),(HUM,BEN),(HUM,PUR),(HUM,DUR),
(ORG,COST),(ORG,BEN),(ORG,PUR),(ORG,DUR)}

Because the variable T is no longer adjacent to F, it will not be able to disambiguate it. Propagation will be of no help either, since it is dependent on discovering that certain values of F are not compatible with T. Thus, the ambiguity, under this ordering, will be carried all the way up until the final solution, the order-5 solutions, are synthesized.

Tsang (1993) states that the efficiency of his algorithm can be improved by giving the nodes a certain order. Preferred orderings, he suggests, can be obtained by applying some of the general variable ordering techniques discussed below in section 2.4. Specifically, he suggests using a Minimal Bandwidth Ordering (MBO). This idea is adopted and expanded upon in our work.

2.3. Branch-and-Bound

Branch-and-bound (BB) techniques can be used to reduce the amount of search needed to find the optimal solution. BB is based on a common-sense principle: do not keep trying a path that you already know is worse than the best answer.[28] One of the first articles on branch-and-bound was (Lawler and Wood, 1966). A more readable introduction is in (Winston, 1984). A simple algorithm describes the method (mostly from (Winston, 1984)):

1. Form a queue of partial paths. Let the initial queue consist of the zero-length, zero-step path from the root node to nowhere. Let the optimal-path be an infinite-length path.
2. Until the queue is empty or the first path in the queue has length longer than the optimal-path.
   (a) Remove the first path from the queue.

[28] AKA The "hit-your-head-against-the-wall phenomena"

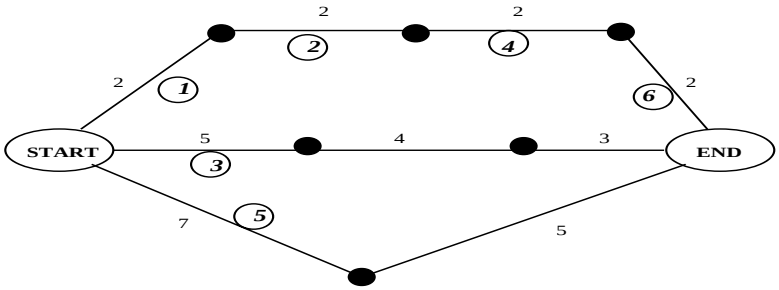


Figure 8: Branch-and-Bound Example

(b) Form new paths by extending the removed path one step, if possible.

(c) If any of the new paths reach the goal, and the shortest of these is shorter than the optimal path, set the optimal path to it. Remove all paths in the queue with length greater than this new optimal-path.

(d) add all the new paths (except any that reach the goal) with length less than the optimal-path length to the queue, sorting the queue with the smallest length paths in the front.

3. return optimal path

The example given in the introduction will be repeated and explained here. Figure 4 is repeated as Figure 8. Paths would be created for each of the initial arcs from the start. The path with the arc labeled with the circled 1 would be expanded first, since it has the shortest total length. Arc 2 is added to it, giving a total length of 4. Because this is still shorter than any other path, it is extended again by adding arc 3 to it, yielding a total path length of 6. At this point, the path with arc 4, with a length of 5, is shorter than the 1-2-3 path. It is expanded to give a path of length 9. The first path is again the shortest, so it is extended to give path 1-2-3-5, which reaches the goal in a length of 8. Optimal-path would, at this point, be set to this path. The path including arc 4 would be discarded now, since it already has length greater than the optimal-path. The path with arc 6, however, only has length 7. It is extended, yielding a path of length 12. Because there are no more paths with lengths less than 8, the process is terminated and optimal-path is returned.

This, of course, is a simplified example. Most search problems do not have all possible paths ending up at the goal. Most nodes have multiple branches. Nevertheless, the BB procedure can be used to find the optimal solution, despite these complications. Various other techniques, with constraint satisfaction being the most relevant here, can be used to

further optimize. Heuristics which estimate the distance from a node to the goal state can also be used,[29] if one is willing to risk non-optimal solutions.

BB is useful for searches where the optimal solution is needed. If **any** valid solution is desired, a depth-first search is indicated. If a solution with the smallest number of arcs is desired, a breadth-first search would be more advantageous. Below, we will demonstrate how BB techniques can be used in combination with solution synthesis and constraint satisfaction to produce an extremely efficient computational semantic problem solver.

2.4. Other Strategies for CSPs

CSP techniques can be used in conjunction with many other AI search strategies. The most basic of these strategies is called "lookahead." In fact, lookahead is not an additional strategy, but simply a full, dynamic application of consistency checking. At each decision point in a search, a value is assigned to a variable. This implicitly removes the other possible values of that variable; thus, other variables that depend on those values can be affected. By applying the consistency algorithm, these effects can be automatically propagated at each decision point.

There are various strategies used to handle backtracking. The most advantageous of these, used in conjunction with constraint analysis, is called "dependency-directed backtracking." When the need for backtracking arises, constraint analyses can help indicate what the source of the current conflict is. The search can then be backtracked directly to the source of the problem. This can potentially eliminate large areas of search that will not "fix" the current bottleneck. Because backtracking is eliminated in solution synthesis methods, the various backtracking strategies are not relevant to this research.

Heuristics can be used to great advantage in search. At any branching point, each choice can be analyzed using heuristic knowledge to estimate how much closer to a solution the choice brings the search. Choices with higher heuristic value can be followed first, a technique generally referred to as "best-first" search. Optimal solutions cannot be guaranteed with heuristic search, however, because local optima can obscure longer-term solutions. For

[29] For example, a potential path with a high estimated distance to goal can be excluded even though its total current distance is lower than the optimal path.

instance, turning south seems like a bad choice when your goal is north, unless there is a freeway one block to the south that can speedily take you on your way. Some types of problems require heuristic search to make them tractable. Prior to the current research, the Mikrokosmos project relied on heuristic search in its analysis of natural language semantics. It is the thesis of this report, however, that CSP techniques in combination with branch-and-bound and solution synthesis can deliver guaranteed optimal solutions in near-linear time for computational semantic problems. Therefore, heuristic search is not necessary. On the other hand, even with near-linear time speed, large problems, especially those involving discourse, cannot be considered "real-time." Best-first techniques can be added to those described below to improve this situation.

A different kind of heuristic can be used to optimally order the instantiations of variables and their values. These heuristics can be labeled more accurately as "strategies," because they involve general principles rather than world knowledge. These types of strategies are closely connected to constraint analysis; a good discussion of them can be found in (Tsang, 1993). The first is called "minimal width ordering" (MWO). In general terms, this strategy seeks to instantiate more highly constrained variables first, in hopes that backtracking will be reduced. For example, if variable A constrains variable B to value X and variable C to Y, it would be advantageous to instantiate variable A first. This would reduce the number of choices in B and C, which in turn might restrict choices elsewhere. If variable C was instantiated first, values for it might be tried that conflict with variable A.

Alternatively, a "minimal bandwidth ordering" (MBO) can be used. This ordering seeks to place constrained variables close together so that when backtracking is necessary, only a small distance will have to be backtracked, minimizing the amount of work that might have to be repeated. For example, if variable A constrains B to X, but B is instantiated first, followed by C, D, E and F, then when A is finally instantiated, the search will need to backtrack to B and then recalculate values for C, D, E and F as well. If, on the other hand, A was instantiated directly after B, backtracking would proceed directly to B, with no intervening variables affected.

Of course, a dynamic implementation of arc consistency reduces the importance of these types of orderings. First of all, before search even begins, conflicting values will be removed from any variables domain. In addition, during search, when a value for a variable is chosen, all variables affected by the choice can be dynamically processed, with all those effects

recursively propagated, etc. Thus, a dynamic implementation of arc-consistency inherently gives the benefits of MWO and MBO. This notwithstanding, these techniques can still give some advantage. If two or more variable instantiations work together to eliminate certain values of other variables,[30] it would be advantageous to process these "partners" early. Since ordering more constrained variables first maximizes this potential, a minimal width ordering would be helpful.

Solution synthesis, however, alters the picture somewhat. Solution synthesis combines small sub-solutions together to form larger and larger solutions. Interactions outside of the subsets being combined are not noticed. Because of this, it would be helpful to group variables together in such a way as to minimize the amount of interaction across sub-solutions. By grouping variables (and later sub-solutions) together that constrain each other maximally, ambiguity can be eliminated as early as possible. This type of variable grouping is an outgrowth of MBO. This project uses MBO concepts to group variables and synthesized solutions together in order to minimize ambiguity within the sub-solution. This goes well beyond a simple linear ordering of variables on which an SS-TSANG-like algorithm would work.

MWO and MBO help determine which order to instantiate variables. There are also techniques which help decide, given a variable, which values to try first. While variable ordering techniques seek to maximally constrain so that backtracking can be identified early, value ordering techniques seek to eliminate backtracking by trying the most likely values first. For problems in which all possible solutions are required (or the most optimal), these techniques are not helpful. All values that result in solutions must be attempted (unless it can be proved they will result in non-optimal solutions using, for instance, the branch-and-bound techniques described below); it does not matter which order they are found.

2.5. Using Linear Programming for Constraint Satisfaction Problems

Linear programming (LP) techniques have been around since the 1940's when G.B. Dantzig designed the so-called "simplex method" for solving linear planning problems for the U.S. Air Force. When such techniques work, they provide extremely fast answers to optimization

[30] Which is not detected by arc consistency until those variables are instantiated.

problems. It is tempting to try and apply such methods to computational semantics. Before discussing why LP cannot provide reliable answers for computational semantics (and similar problems), it will be instructive to give a short, elementary overview of the techniques used. (Chvátal, 1983) is an excellent introduction to linear programming.

The central idea used in LP stems from the algebra technique known as "Gaussian elimination," used for solving systems of equations. Given the following two equations:

$$\begin{aligned} 2x + 13y &= 22 \\ x + y &= 7 \end{aligned}$$

we can easily solve for the value of x in the second equation and then substitute it back into the first equation, which will then allow us to solve for y. By substituting this value for y back into one of the original equations, we can then solve for x.

$$\begin{aligned} x &= 7 - y \\ 2 * (7 - y) + 13y &= 22 \\ 14 - 2y + 13y &= 22 \\ 11y &= 8 \\ y &= 8/11 \\ x &= 7 - 8/11 \\ x &= 69/11 \end{aligned}$$

The simplex method for optimizing an equation subject to constraints extends the basic Gaussian elimination method. A short example follows. We make no attempt to explain the rationale behind the methodology here; the interested reader may refer to (Chvátal, 1983). Given the following problem:

$$\begin{aligned} maximize: \quad & 2x_1 + 2x_2 \\ subject-to: \quad & x_1 + 2x_2 \leq 4 \\ & 2x_1 + x_2 \leq 2 \\ & x_1, x_2 \geq 0 \end{aligned}$$

The simplex method starts by creating "slack" variables for each of the inequalities such that the slack variable must be ≥ 0. The variable z is then set to the equation to be maximized:

$$\begin{aligned} x_3 &= 4 - x_1 - 2x_2 \geq 0 && (1) \\ x_4 &= 2 - 2x_1 - x_2 \geq 0 && (2) \\ z &= 2x_1 + 2x_2 && (3) \end{aligned}$$

The simplex method works by starting out with a feasible solution, and then successively improving that solution. A feasible first solution in this case is $x_1, x_2 = 0$. Substituting this in for z, x_3 and x_4 gives:

$x_1 = 0, x_2 = 0, x_3 = 4, x_4 = 2, z = 0$

The key to the simplex method, then, is to pick one of the variables in equation 3 to increase so that z will increase. In this case, increasing either x_1 or x_2 will increase z, so we will pick x_1. Since we will keep $x_2 = 0$, equations 1 and 2 can give us an upper bound on how much we can increase x_1. Equation 1 gives us $x_1 \leq 4$, since $4 - x_1 \geq 0$, and equation 2 gives us $x_1 \leq 1$, since $2 - 2x_1 \geq 0$. The latter is more constraining, so we will adopt it as the starting point for the next intermediate solution. Substituting $x_1 = 1$ into equations 1, 2 and 3 gives us:

$x_1 = 1, x_2 = 0, x_3 = 3, x_4 = 0, z = 2$

The fact that the value of z increased confirms that this solution is better than the last one. We now wish to find an even better solution. The "trick" of the simplex method is to, after each intermediate solution, express the equations for variables with positive values in terms of those variables with 0 values. We can express x_1 in terms of x_2 and x_4 (the variables with 0 values) by solving equation 2 for x_1. We can then express x_3 and z in terms of x_2 and x_4 by substituting this equation for x_1 into equations 1 and 3:

$$\begin{aligned} x_1 &= 1 - \frac{x_4}{2} - \frac{x_2}{2} \geq 0 && (4) \\ x_3 &= 4 - x_1 - 2x_2 \geq 0 \\ &= 4 - (1 - \frac{x_4}{2} - \frac{x_2}{2}) - 2x_2 \geq 0 \end{aligned}$$

$$= 3 + \frac{x_4}{2} - \frac{3x_2}{2} \geq 0 \quad (5)$$

$$z = 2x_1 + 2x_2$$
$$= 2(1 - \frac{x_4}{2} - \frac{x_2}{2}) + 2x_2$$
$$= 2 - x_4 + x_2 \quad (6)$$

Again, we try to pick one of the variables in 6, which always will have values of 0 in the current solution, to increase so that the value of z will increase. In this case, increasing x_4 **decreases** z, so we don't want to try that. Increasing x_2, though, will increase z. Again, by holding $x_4 = 0$, we can read off the possible values for x_2 from equations 4 and 5. Equation 4 gives us $x_2 \leq 2$ and equation 5 gives $x_2 \leq \frac{8}{3}$. The former is more restrictive, so we substitute it into the equations giving:

$x_1 = 0, x_2 = 2, x_3 = 0, x_4 = 0, z = 4$

To start another iteration, we again need to express the non-zero variables in terms of the zero-valued variables. x_2 in terms of x_1 and x_4 can be obtained from equation 4, and then we can substitute this into equation 6:

$$x_2 = 2 - x_4 - 2x_1 \quad (7)$$

$$z = 2 - x_4 + (2 - x_4 - 2x_1)$$
$$= 4 - 2x_4 - 2x_1 \quad (8)$$

At this point, we again try to pick a variable from equation 8 that, when increased, will increase the value of z. In this case, however, increasing either x_1 or x_4 will decrease the value of z. This is when we know that we are finished. The correct solution to the original problem, then, is:

$x_1 = 0, x_2 = 2$

The simplex method for optimizing systems of linear equations is extremely powerful. There are, however, several problems.

1. Theory-internal problems.

(a) Initialization. The process described above assumed a feasible initialization ($x_1, x_2 = 0$). in practice, the initialization values may be difficult or impossible to find.

(b) Looping. It is possible to create a set of iterations that loop.

(c) Termination. Because of the looping problem, the simplex method may not terminate for some problems. In addition, some problems, although the answer might be found eventually, may require an exponential number of iterations.

2. Theory-external problems.

(a) Non-linear. The simplex method requires linear mathematical formulas. Many interesting real-world problems, including problems in computational semantics, are not linear.

(b) Non-decomposable. For complex problems, it is advantageous to break the original problem into small subproblems. Typically, only a subset of these subproblems has will have excessive complexity. An attractive paradigm for solving such problems would be to have a heuristic problem-solver work on the subproblems that are too complex for non-heuristic methods, and then integrate those solutions into the overall problem solution. LP methods do not fit well into this paradigm.

(c) Non-dynamic. If, after spending 30 minutes optimizing a complex series of linear equations, a single coefficient on one of the equations was changed, the whole process would have to be repeated. LP methods are not dynamic, and minimum perturbation re-planning is not possible.

The theory-internal problems will not be expounded upon further here. Various techniques have been created to minimize, or in some cases, eliminate their effects (Chvátal, 1983).

The problem of non-linear inputs is severe. Many real-world problems cannot be expressed by a system of linear equations. Many problems cannot be expressed with mathematical formulas at all. Semantic selectional constraints (see section 4) fit into this category. Figure 32, the example semantic analysis problem solved by Hunter-Gatherer in section 5 is previewed here in Figure 9.

The constraints between *adquirir* and *Dr-Andreu*, for this simplified example, can be represented as Table 1.

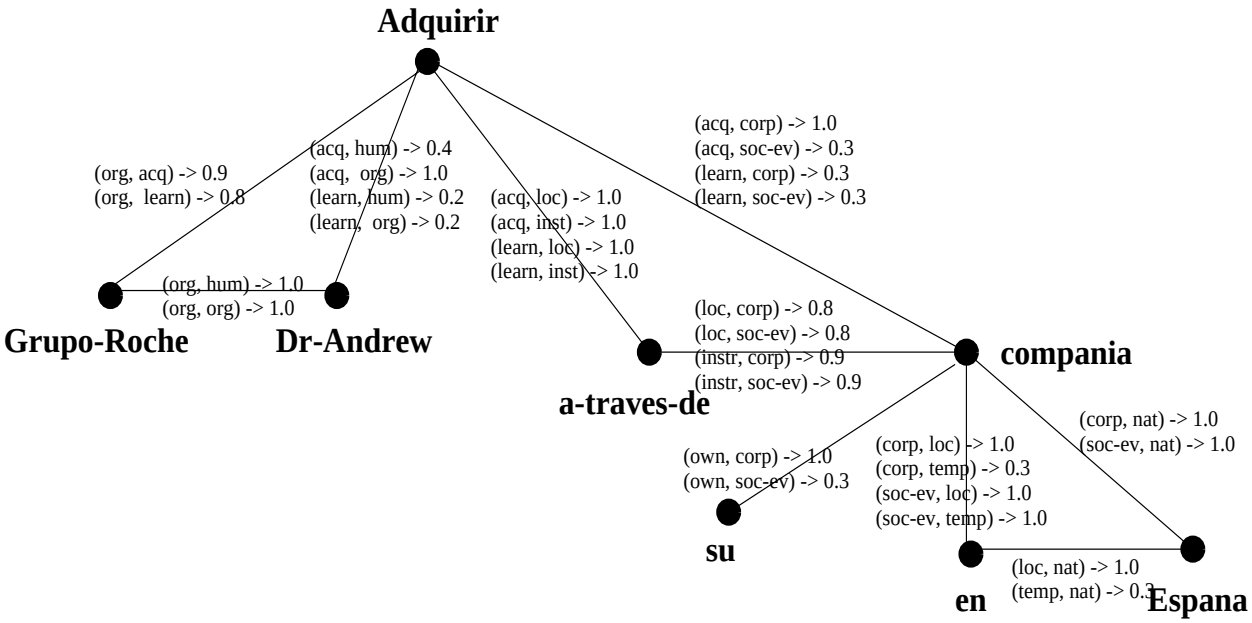


Figure 9: Constraint Dependencies in Sample Sentence

adquirir	Dr-Andreu	f_1
acq	human	0.4
acq	org	1.0
learn	human	0.2
learn	org	0.2

Table 1: Semantic constraints

The function, f_1, which maps from word sense combinations into an evaluation score is not linear. In the case of semantic selection restrictions, the scores correspond to a numerical rating of the ability of the one sense to fill a semantic role of the other (see section 4 for more details). The example in Figure 9 is very simplified. Typically, the function table contains two variables, each of which may contain up to 8 possible values (word senses). Furthermore, semantic constraints do not have to be binary. Tables with three or more variables might be used in certain cases. LP methods simply cannot handle these types of input functions.

The Hunter-Gatherer control architecture is based, in part, on decomposing the input problem into simpler subproblems which can be solved separately and then "synthesized" together to obtain the answer for the whole problem. As discussed in section 9, this methodology takes advantage of the topology of the input problem, separating "easier" sections of

the problem from potentially "harder" ones. One of the great strengths of HG is its ability to accept solutions to any subproblem from any source. If the particular subproblem has a complexity that is too high for non-heuristic methods such as HG to solve, it can be "sub-contracted" to other problem solvers. Furthermore, HG can identify these "hard" areas ahead of time, enabling a collaborative approach between heuristic and non-heuristic methods. Unfortunately, this problem solving approach does not even make sense for LP. It is possible for LP techniques to be used as one of the sub-contractors for a particular subpart of the problem, but LP cannot divide up a problem for itself. Neither can it decide ahead of time whether a given problem can be solved in a given amount of time. Assuming a problem is even of the type that **can** be solved using LP, these are the greatest drawbacks of the approach: one cannot determine ahead of time if the problem can be solved, nor can LP avail itself of other problem solvers to help with difficult parts of the problem.

The other serious limitation of LP is that it is a static problem solver. The methods solve a given system of equations. Change any of the inputs and the whole problem needs to be solved again. Hunter-Gatherer, on the other hand, is a constraint-based control architecture. Interactions between variables are tracked, and problem decomposition is based on these interactions. If any inputs are changed, the extent of their impact can determined. Minimal perturbation re-planning can then be accomplished by re-examining only those parts of the problem that are affected. In fact, future research is aimed at making Hunter-Gatherer functionally equivalent to a blackboard architecture. HG will be able to dynamically add values to the domain of a variable (for semantic analysis this is equivalent to adding new word senses on the fly in order to, for example, process figurative language), add in new constraint links between two variables (for instance, if two words are found to be coreferent, their meanings will be constrained to be equivalent) and/or change constraint valuations between two variables.

2.6. Nonserial Dynamic Programming

In the course of presenting this research at various conferences, it has come to our attention that previous work in nonserial dynamic programming (nonserial DP) has many parallels with our own that might be fruitfully investigated. An excellent introduction to this field can be found in (Bertelé & Brioschi, 1972). Another more recent article relating the work

in nonserial DP to partial k-trees is (Anborg & Proskurowski, 1988).

Nonserial DP is very similar to the Gaussian elimination procedures examined in the previous section. It operates by eliminating variables one by one[31] by, in essence, creating new functions that eliminate the variable. The added benefit nonserial DP gives is that it works on nonlinear systems in which no mathematical functions can be formulated.

The properties of nonserial DP are most concisely explained with an example taken from (but expanded upon here) (Bertelé & Brioschi, 1972). Consider a problem involving five variables, x_1 through x_5, each of which has a domain of two values, {0,1}. The goal of the problem is to minimize the total $f_1 + f_2 + f_3$, where the functions have nonlinear evaluations as shown in Table 2.

x_1	x_3	x_5	f_1
0	0	0	1
0	0	1	3
0	1	0	5
0	1	1	8
1	0	0	2
1	0	1	6
1	1	0	2
1	1	1	4

x_1	x_2	f_2
0	0	4
0	1	8
1	0	0
1	1	5

x_2	x_4	x_5	f_3
0	0	0	0
0	0	1	5
0	1	0	6
0	1	1	3
1	0	0	5
1	0	1	1
1	1	0	4
1	1	1	3

Table 2: Input evaluation functions

It should be pointed out that these evaluation functions are very similar to those found in semantic analysis, as discussed in the section on Linear Programming above.

To eliminate x_1, we need to create a new evaluation function, h_1, which combines all evaluation functions in which x_1 takes part. In this case, only f_1 and f_2 need to be considered. The new evaluation function in Table 3 is created by making a new table including all the variables present in f_1 and f_2.

The values for h_1 are calculated by combining scores (in this case, just adding them together) from the original functions. For instance, for the first entry, $x_1 = 0$, $x_3 = 0$ and $x_5 = 0$, so the score (from Table 2) for f_1 is 1. Additionally, $x_1 = 0$ and $x_2 = 0$, so the evaluation of f_2 is 4. This gives a total score for the first entry in Table 3 of 5. The other

[31] Actually nonserial DP introduces methods for eliminating groups of variables - see below

x_2	x_3	x_5	x_1	h_1
0	0	0	0	5
0	0	0	1	2 *
0	0	1	0	7
0	0	1	1	6 *
0	1	0	0	9
0	1	0	1	2 *
0	1	1	0	12
0	1	1	1	4 *
1	0	0	0	9
1	0	0	1	7 *
1	0	1	0	11 *
1	0	1	1	11
1	1	0	0	13
1	1	0	1	7 *
1	1	1	0	16
1	1	1	1	9 *

Table 3: A New Evaluation Function

values for h_1 are calculated similarly.

At this point, x_1 can be eliminated from h_1 by selecting the optimal value for x_1 for each combination of the other variables. h_1 will then be a function that maps from the remaining variables, into an evaluation value **and** the optimal value for x_1. The resulting table is shown in Table 4.

x_2	x_3	x_5	h_1	$x_1{}^*$
0	0	0	2	1
0	0	1	6	1
0	1	0	2	1
0	1	1	4	1
1	0	0	7	1
1	0	1	11	0
1	1	0	7	1
1	1	1	9	1

Table 4: h_1 optimized

At this point, h_1 replaces f_1 and f_2. x_2 can be eliminated by a similar process, combining h_1 and f_3 (which x_2 takes part in but was not replaced by h_1) to produce h_2, as shown in

Table 5.

x_3	x_4	x_5	h_2	$x_2{}^*$
0	0	0	2	0
0	0	1	11	0
0	1	0	8	0
0	1	1	9	0
1	0	0	2	0
1	0	1	9	0
1	1	0	8	0
1	1	1	7	0

Table 5: h_2 optimized

h_2 now replaces h_1 and f_3, and is the sole remaining evaluation function. x_3 can be eliminated next, creating a new evaluation function h_3, as shown in Table 6.

x_4	x_5	h_3	$x_3{}^*$
0	0	2	0
0	1	9	1
1	0	8	0
1	1	7	1

Table 6: h_3 optimized

h_3 replaces h_2, and is then used to eliminate x_4, as shown in Table 7.

x_5	h_4	$x_4{}^*$
0	2	0
1	7	1

Table 7: h_4 optimized

x_5 is then eliminated from h_4 to produce the function h_5 shown in Table 8.

The optimal answer to the problem is then obtained iteratively substituting into the equations. With $x_5 = 0$, the optimal value for $x_4 = 0$, which can be read off h_4. With the values of x_4 and x_5 determined, the optimal value for x_3 can be read off h_3, and so on. The optimal answer for the problem turns out to be: $x_5 = 0$, $x_4 = 0$, $x_3 = 0$, $x_2 = 0$ and $x_1 = 1$.

h_5	$x_5{}^*$
2	0

Table 8: h_5 optimized

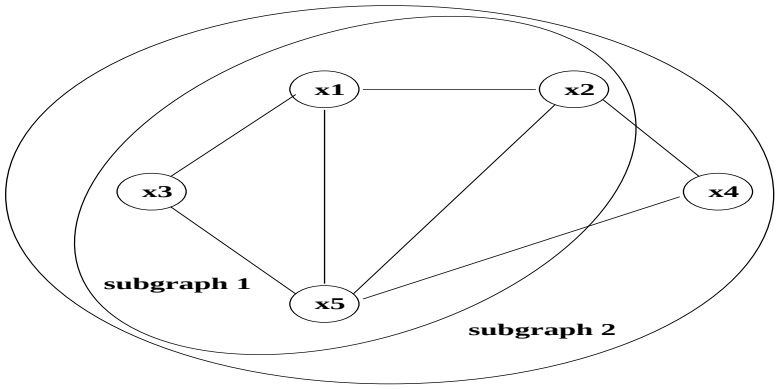


Figure 10: HG used to solve the nonserial DP example

This process required 46 functional evaluations and 78 table look-ups, as reported by Bertelé and Brioschi. "Functional evaluations" refer to the total number of combinations evaluated in creating the h_x functions. For example, in Table 3, 16 combinations had to be created and evaluated. "table look-ups" refers to the total number of times the tables had to be consulted. For instance, in creating each entry h_1 in Table 3, a value had to be looked up in both f_1 and f_2, for a total of 32 table look-ups. A different order of elimination, for instance x_3,x_4,x_1,x_2 and x_5 was more efficient, requiring only 30 functional evaluations and 42 table look-ups.

As will be seen below, the general idea behind this technique is quite similar to the Hunter-Gatherer methodology. The following comments will undoubtedly make more sense once the remainder of this work is read. HG can be used to eliminate one variable at a time in a similar fashion to what was just reported. HG could start off with a subgraph that contains all variables that constrain x_1, namely subgraph 1 = (x_1, x_2, x_3, x_5), as shown in Figure 10. All combinations of values for these variables, 16 in all, would need to be calculated. In subgraph 1, only x_2 and x_5 are constrained outside the subgraph, so the branch-and-bound reduction phase would optimize x_1 and x_3 for each of the four combinations of x_2 and x_4. In subgraph 2, the two values for x_4 would then be combined with the four outputs of

subgraph 1, yielding eight combinations. Since subgraph 2 contains the whole problem, the best answer would then be picked. A total of 24 combinations were created, which compares directly to the 46 (or 30) "functional evaluations" in the nonserial DP method.

In this case, HG outperforms the nonserial DP method primarily because HG eliminated 2 variables in the first subgraph and then the remaining three in the second subgraph (as opposed to one variable at a time for the DP method). To be fair, nonserial DP has techniques for eliminating blocks of variables which would yield the same number of functional evaluations as HG. HG has significant advantages, however, due to its use of solution synthesis rather than a Gaussian-like substitution. By utilizing solution synthesis techniques, we are able to not only eliminate blocks 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 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. Most importantly, Intermediate evaluations are associated with each subgraph, removing the need to recalculate intermediate functions. This results in far less reliance on the table look-ups. HG will refer to a given entry in the input function tables once and only one. In addition, no new tables are constructed. Thus, for the example problem above, HG will only need 20 table look-ups, as opposed to the 78 (or 42) needed by the non-serial DP methods.

Nonserial DP removes the problem of nonlinear inputs that prevented us from utilizing linear programming techniques on many problems of interest. Unfortunately, in addition to the efficiency issues discussed above, it does not remove the other problems associated with linear programming. In particular, a large advantage HG has over nonserial DP is its ability to seamlessly integrate heuristic solutions to subproblems with excessive complexity (see section 9.4). Since HG does not require a chain of intermediate functions, which may be difficult or impossible to create at certain points of a complex problem, it is able to accept and integrate sub-answers from different sources. Furthermore, HG can estimate the computational complexity for processing any given subgraph before processing begins. This allows collaboration with other heuristic problem solvers.

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 (like its "sister" linear programming) is 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 re-planning can be accomplished. This aspect of HG is a primary goal of our future research.

The body of work associated with nonserial DP may be of great value in further research directed at HG. In particular, Bertelé and Brioschi discuss heuristic methods for the "secondary optimization problem" - that of finding the correct order of elimination of the variables. HG has developed its own set of heuristics (see section 3.2), but does not make any claims about their optimality.

In summary, nonserial DP uses linear programming-like techniques to solve nonlinear problems. HG then improves on these techniques by generalizing them with the use of solution synthesis and branch-and-bound. HG's constraint-based organization also affords it a greater flexibility for handling dynamically changing problems.

2.7. Minton's Work on Heuristic Repair

(Minton, et al, 1990) presents a method for using heuristic repair to find solutions to certain types of constraint satisfaction problems. His approach is to generate a reasonable first guess at an answer, determine which variables in that answer participate in constraint violations, and then use repair heuristics to fix the problems. The basic heuristic used in (Minton, et al, 1990) is to reassign a variable that is in a conflict to a value that minimizes the number of conflicts.

Minton applies this simple heuristic to the N-Queens problem and reports the staggering result that the million-Queens program can be processed in minutes. Prior to his work, the 1000-Queens problem was practically unsolvable. Furthermore, Minton claims that the number of repairs made was a constant; that is, approximately the same number of repairs were necessary for the 100-Queens as the million-Queens.

Despite these encouraging results, the heuristic repair method has certain drawbacks. The first and foremost is that such methods only are able to return **a** solution that meets all the constraints. It is not able to return the most optimal answer that meets the constraints;

in fact, it is quite possible that it could return the **worst** such answer. Nor can it be used to generate a list of **all** answers to a CSP. If it could do that, then some optimizing criteria could be used to pick the best answer.

For the types of problems we are concerned with, this drawback is fatal. To begin with, the idea of constraints itself is blurred in computational semantics. As discussed below, semantic constraints are only tendencies, not firm yes or no restrictions. Thus, almost any answer can be "correct;" that is, it will receive a non-zero score. Finding a "correct" answer - one that meets all constraints - is therefore trivial. The goal of our work is to find the solution with the best score, which corresponds to the most probable meaning in the given context. Even if constraints were not "fuzzy," Minton's work (as it stands) could not be applied to optimization problems. Again, these repair methods only seek to find a single solution that meets constraints. Optimization problems of all kinds need to find the best such answer.

There are other problems with Minton's work which we will not delve into deeply here. His methods work best on uniform problems like the N-Queens. If certain constraints are more important, or lead to more breakdowns in the constraint network, the heuristic repair methods tend to work less efficiently (or not at all). Furthermore, the search space of the N-Queens problem lends itself to this type of analysis. As N increases, the probability that a given value assignment will conflict with another given value assignment decreases, even for variables "close" to one another. For instance, for the 4-Queens problem, the probability that adjacent variables (corresponding to adjacent columns) will conflict is greater than 1 in 2. For the million-Queens problem, the probability is 3 in a million! This local disjointedness is not a feature of most real life optimization problems.

The preceding should not be taken to mean that Minton's work does not hold promise for optimization problems in general, or computational semantics in particular. Repair methods that increase a solutions overall score could be adopted. Along with an adequate measure of "satisficing" (Newell & Simon, 1972), this could lead to a promising methodology for estimating near-optimal solutions.

2.8. Scheduling at CMU's Robotic Institute

Scheduling is an area related to the generic work on optimized planning presented here. In general, scheduling problems are much more difficult to process because of their multiple sources of costs (personnel, inventory, costs of delays, machinery, etc.), the interaction of these costs, multiple sources of priorities, and most importantly, the dynamic nature of the problems (new orders received, machinery malfunctions, resources delivered late, etc.). The Robotics Institute at Carnegie Mellon University has been central in formulating scheduling problems and developing methodologies to solve them. The first research to recognize scheduling problems as heuristic, constraint-directed search was the ISIS project (Fox, 1994). ISIS utilized a hierarchical scheduler which created levels of abstractions in the scheduling problem. The main problem encountered was "bottlenecks," or certain resources being needed (but unavailable) at certain critical stages of planning.

OPIS (Smith, 1994) was developed in response to the bottleneck problem. OPIS used a constraint-based **control** mechanism that identified and prioritized possible bottlenecks. OPIS also recognized the dynamic nature of scheduling. It was a **reactive** scheduler in the sense that it was able to respond to changes in requirements or resource availability, as well as recognize the appearance of new bottlenecks as they arose in the scheduling process. This ability to recognize and handle new situations automatically was termed **opportunistic** scheduling.

Opportunistic scheduling in OPIS was limited; it was required to finish scheduling all (or most of) an entire bottleneck before switching to another. Micro-Boss (Sadeh, 1994) overcame this limitation. Micro-Boss stops scheduling operations on a resource as soon as another resource is identified as more constraining. This ability is referred to as **micro-opportunistic** scheduling. Mikro-Boss utilizes a blackboard architecture which is able to support multiple control regimes, including highly user-interactive modes. Mikro-Boss also makes use of predefined scripts which specify a sequence of knowledge source activations and/or goals known to accomplish a specific task.

A slightly different area of work was undertaken by Muscettola (1994) in his attempt to integrate planning and scheduling. Historically, planning (i.e. which products to make and when) was separated from scheduling (which resources to use and when). This separation can cause problems. For instance, if two sequential operations require a different size drill bit, it

is necessary to spend time changing the bit in between the operations. It might be better to modify the schedule to allow interaction between plans such that resources can be optimally used with a minimum of setup activities. The major obstacle to this type of integration is the lack of a unified framework for planning and scheduling. Planning generally utilizes the typical STRIPS-type formalism which assumes instantaneous transitions from one state to another. Scheduling, on the other hand, exploits stronger structuring assumptions in that it explicitly represents resource utilization over extended periods. The HSTS system attempts to unify planning and scheduling. HSTS views plans as envelopes of behavior within which the scheduler is free to react to unexpected events.

All of the above use heuristic search techniques as the basis for implementing their respective systems. Hunter-Gatherer, in contrast, relies on problem decomposition and branch-and-bound pruning to minimize problem complexity. Hunter-Gatherer is uniquely prepared to handle many of the difficult problems facing schedulers. Future work is planned to give HG full dynamic re-planning capabilities. Combined with its ability to partition problems, identify particularly hard sub-problems which can be solved heuristically (see section 9.4), and its use of island constraints to help process bottlenecks (section 9.3), we intend for HG to become a significant tool in solving scheduling problems.

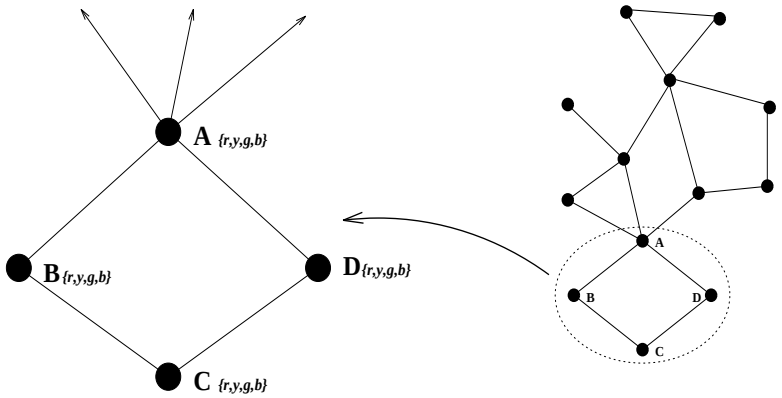


Figure 11: Graph Coloring Subproblem: maximize the number of reds.

3. The Hunter-Gatherer Control Architecture

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, HG focuses 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 shown on the left side of Figure 11. Coloring problems are constraint satisfaction problems with the simple constraint that no two adjacent nodes can have the same value (adjacent nodes are connected by arcs in this graph format). For our purposes, we can turn this type of problem into an optimization problem by requiring the solution to contain as many reds as possible. 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. Hunter-Gatherer, then, will partition a graph into subgraphs (see below for details), process and optimize each subgraph independently, and then use solution synthesis techniques to combine the results from the

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 12: Three steps for subgraph optimization.

subgraphs.

The optimization stage 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. Finally, for each possible value of the vertex with constraints outside the subgraph, determine the optimal assignments for the other vertices. Figure 12 illustrates this process. Step one exhaustively combined all of the possible values for each node in the subgraph. Notice that many of these combinations, such as (r,r,r,r), are not legal sub-answers since adjacent nodes have the same value. Step two used constraint satisfaction techniques to remove these illegal combinations (and possibly others, via a dy-

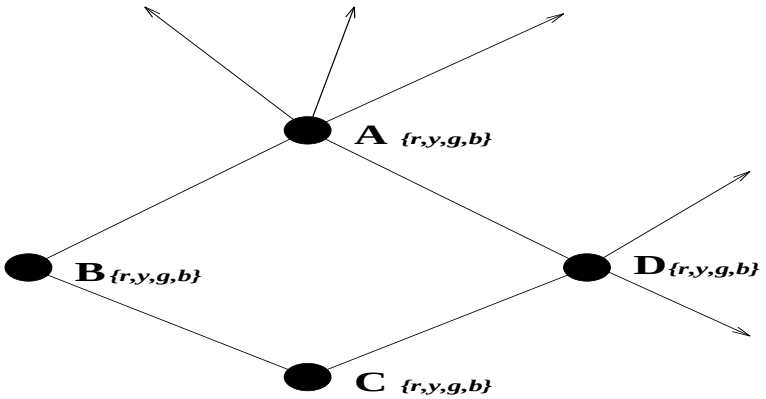


Figure 13: Subgraph: maximize the number of reds.

namic application of AC-4). Step 3 then retained a combination for each possible value of vertex A, each optimized so that the maximum number of reds appear. Now, no matter what value node A is assigned in the end, we know the optimal values for nodes B, C and D that correspond to it. After step 3, we say that the subgraph has been **reduced** to its optimal set of partial answers.

Now consider what happens if, instead of having a single vertex constrained outside the subgraph, two vertices are constrained outside the subgraph, as in Figure 13. 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 (actually less, since 4 of the 16 combinations assign the same value to A and D, violating the coloring constraint).

Solution synthesis is used to combine results from subgraphs or to add individual vertices onto subgraphs (we postpone the discussion of graph decomposition to section 3.2). Figure 14 shows an example of combining the subgraph from Figure 13 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,

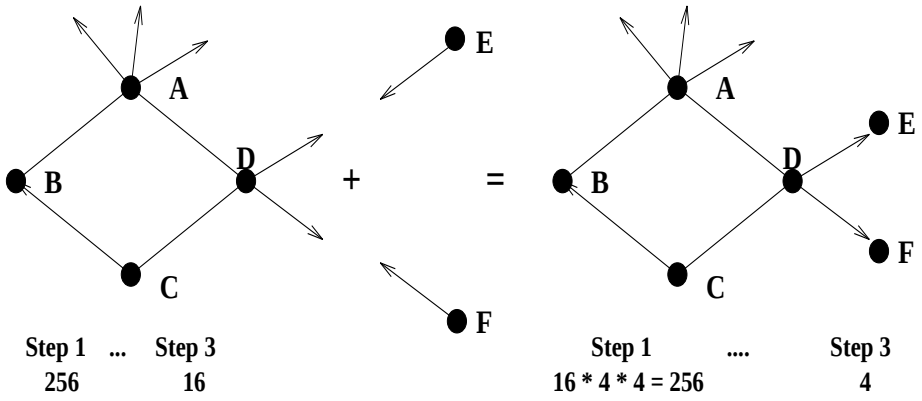


Figure 14: Using Solution Synthesis to Combine Subgraph with Vertices

one for each of the possible values of A.

Synthesis of two (or more) subgraphs 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 15 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 15, 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 15, subgraphs 1 and 4 are combined to yield subgraph 5, which contains the entire problem.

Note that the complexity[32] 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

[32] The complexity of the algorithm will be discussed in the next section.

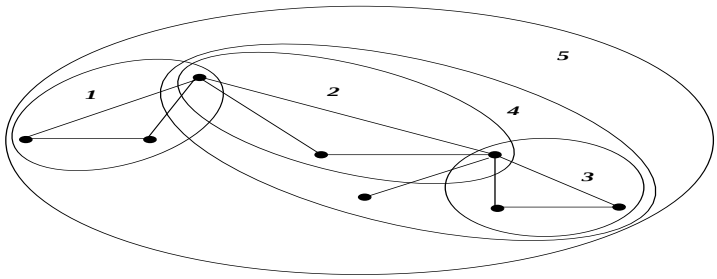


Figure 15: Subgraph Construction

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. HG removes 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.

3.1. The Hunter-Gatherer Algorithm

A simple algorithm for HG is shown below. 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-SUBGRAPH. For example, for Figure 15, 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. SS-HG simply calls PROCESS-SUBGRAPH for each of the input subgraphs. The last input subgraph is the

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

5 PROCEDURE PROCESS-SUBGRAPH(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 Constrained-Vertices)
12   RECORD-COMBOS(Subgraph Output-Combos)
13   RETURN Output-Combos
```

Figure 16: HG Algorithm

graph containing the whole problem. The answer returned by PROCESS-SUBGRAPH for that input will be the answer to the whole problem.

The COMBINE-INPUTS procedure (line 7) simply produces all combinations of value assignments for the input vertices and subgraphs. The following three cases are possible[33]:

1. The input subgraph contains no lower-level input-subgraphs. In this case, COMBINE-INPUTS returns all of the possible combinations of values for each of the In-Vertices. For instance, if two variables, A and B, each had domains {0,1}, COMBINE-INPUTS would return {(0,0),(0,1),(1,0),(1,1)}, where each combination is in the form (A,B).

2. The input subgraph contains one lower-level subgraph and one or more In-Vertices. This, in practice, is the most common way to extend a subgraph - simply by adding on extra vertices. COMBINE-INPUTS adds on the possible combination of In-Vertices

[33]Technically, there could be more possibilities, such as multiple In-Vertices combining with multiple lower-level subgraphs, but practically speaking, the Input-Complexity (see below) of these combinations is excessive)

to the combinations present in the lower-level subgraph. It must be stressed that this lower-level subgraph has already been optimized by a previous call to PROCESS-SUBGRAPH. For example, if the lower-level subgraph had two variables, C and D, each with the domain {2,3}, and had been reduced to the following combinations: {(2,2), (2,3)}, where each combination is in the form (C,D), and two In-Vertices, A and B (as above) were being added, then COMBINE-INPUTS would return the following combinations: {(0,0,2,2),(0,0,2,3),(0,1,2,2),(0,1,2,3),(1,0,2,2),(1,0,2,3), (1,1,2,2),(1,1,2,3)}, where each combination is in the form (A,B,C,D).

3. There are two lower-level subgraphs and no In-Vertices. For this case, there are two separate sub-cases:

   (a) The two lower level subgraphs do not have any vertices in common. In this case, COMBINE-INPUTS simply produces all of the combinations. If the first lower level subgraph had the two variables C and D as shown above, and the second lower level subgraph had two variables E and F, with domains {4,5}, and had been reduced to the following combinations: {(4,5),(5,5)}, then COMBINE-INPUTS would return the following combinations: {(2,2,4,5),(2,2,5,5),(2,3,4,5),(2,3,5,5)}, where each combination is in the form (C,D,E,F).

   (b) The two lower level subgraphs share one or more vertices. This is where the synthesis techniques come in. In this case, only compatible combinations are combined. For instance, if the first lower level subgraph had the variables C and D as shown above, and the second lower level subgraph had the variables C and G, each with the domain {2,3}, and had been reduced to {(2,2),(3,2)}, with each combination in the form (C,G), then COMBINE-INPUTS would return only the following two combinations: {(2,2,2),(2,3,2)}, where each combination is of the form (C,D,G). Notice that the output combinations only contain three variables. Also notice that the combination (3,2) from the second lower level subgraph, where C is assigned to 3, was not used, since none of the input combinations of the first lower level subgraph used that assignment.

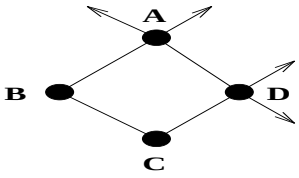
The COMBINE-INPUTS procedure has complexity $O(s_1 * s_2 * ... * a^x)$, where x is the number of vertices in In-Vertices, a is the maximum number of values in the domain of a vertex, and s_i is the number of combinations in the lower level subgraph (which was already processed

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 and no subgraphs. Of course, this is simply an exhaustive search, not Solution Synthesis. In practice, In-Vertices contains no more than two or three 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 subgraphs.

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 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.[34] 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 interconnectedness 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 section 8.1 for data comparing the efficiency of HG with and without arc-consistency.

The REDUCE-COMBOS procedure in line 11 simply goes through each Combos that passed arc consistency, and keeps track of the best one for each combination of values of Constrained-Vertices. The "best" is defined as the combination with the highest overall score. We discuss scoring, and how scores are tracked, in sections 4 and 5. If there are two Constrained-Vertices, each of which has four possible values, REDUCE-COMBOS should return 16 combinations (unless one or more combinations 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. This process is pictured in Figure 17. The complexity of line 11 is therefore the same as the complexity of COMBINE-INPUTS.

[34]For a graph with fan-out = k, $e \leq n^k$.



Constrained-Vertices = {A,D}

input combinations

A	B	C	D	score
0	0	0	0	.2
0	0	0	1	.4
0	0	1	0	.3
0	0	1	1	.8
0	1	0	0	.4
0	1	0	1	.5
0	1	1	0	.35
0	1	1	1	.7
1	0	0	0	.8
1	0	0	1	.75
1	0	1	0	.9
1	0	1	1	.65
1	1	0	0	1.0
1	1	0	1	.95
1	1	1	0	.5
1	1	1	1	.85

combinations of Constrained-Variables

A	D
0	0
0	1
1	0
1	1

Combinations of Constrained-Vertices with optimal values for non Constrained-Vertices

A	D	B	C	score
0	0	1	0	.4
0	1	0	1	.8
1	0	1	0	1.0
1	1	1	0	.95

Figure 17: Processing in REDUCE-COMBOS

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 vertices in 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 lower-level 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 lower-level 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 (details below) input to SS-HG, we will seek to minimize the highest input complexity, which we will refer to as Maximum-Input-Complexity. To be precise, Maximum-Input-Complexity $= x + c_{total}$ for the subgraph with highest input complexity. As will be shown, some fairly simple heuristics can simplify this subgraph creation process.

For completion, it should be noted that PROCESS-SUBGRAPH is called X times in SS-HG, where X is the number of subgraphs in Subgraphs. A Subgraph with input complexity less than Maximum-Input-Complexity will run in time negligible compared to those with Maximum-Input-Complexity. Thus, Number-At-Maximum-Input-Complexity is an important measure, and the overall complexity of SS-HG will be:
$O(Number\text{-}At\text{-}Maximum\text{-}Input\text{-}Complexity * a^{Maximum-Input-Complexity})$
Theoretically, one could create Subgraphs in such a way that there would be an exponential number of subgraphs. However, our algorithm below guarantees that there will be n or less subgraphs. Typically, only a portion of these have Maximum-Input-Complexity, thus the overall complexity is less than
$O(n * a^{Maximum-Input-Complexity})$. We will demonstrate below that, for problems in computational semantics, $a^{Maximum-Input-Complexity}$ is "nearly" a constant, and thus the complexity of HG will be linear with respect to the number of inputs, n.

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 the maximum output complexity, because it determines the amount of output combinations stored by the subgraph with the highest output complexity. The highest output complexity is bounded by the highest input complexity, and thus is proportional to $O(n * a^{Maximum-Input-Complexity})$. 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.

Please refer to section 5 for a detailed example of applying the HG algorithm to a semantic

analysis problem.

3.2. Subgraph Construction

Subgraph decomposition is a field in itself (see, for instance, (Bosák, 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 below 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. The overall complexity of HG is bounded by the maximum input complexity, as defined above. We attempt to minimize this feature in the algorithm given below:

1. Find "seed" subgraphs in various sections of the graph. These "seed" subgraphs are the smallest subgraphs in a region that "cover" at least one vertex. "Covering" a vertex with a subgraph, in this context, means that the vertex has no edges outside the subgraph.
   (a) Order the input vertices from those with the smallest number of vertices adjacent to those with the largest number of vertices adjacent. Set Vertices-Covered to nil. Set Subgraphs to nil.
   (b) For each vertex in the ordered list of vertices:
      - Set Subgraph to the vertex plus all its adjacent vertices.
      - Set Region to the Subgraph plus the set of all vertices adjacent to any vertex in Subgraph.
      - If the intersection of Region and Vertices-Covered is nil, then this is the smallest set of vertices that cover at least one vertex in this region of the graph.
        – add Subgraph to Subgraphs
        – append Region to Vertices-Covered
2. For each Subgraph in Subgraphs, determine the best action to take to expand the Subgraph. The "best" action is the one that requires the minimal input complexity. See below for a description of how to determine potential actions.
3. Order these "best" actions (one per subgraph) from minimal input complexity to maximum input complexity.
4. Take the first action, which results in the creation of a new subgraph. If this subgraph contains the entire input graph, exit. Else remove any other actions from the ordered list of actions that contain input subgraphs or vertices adjacent to vertices involved in the action (potentially, there is a better action now that we have created this new subgraph).
5. Compute a "best" action for the new subgraph.
   If its input complexity is not greater than the maximum input complexity used in any action thus far:

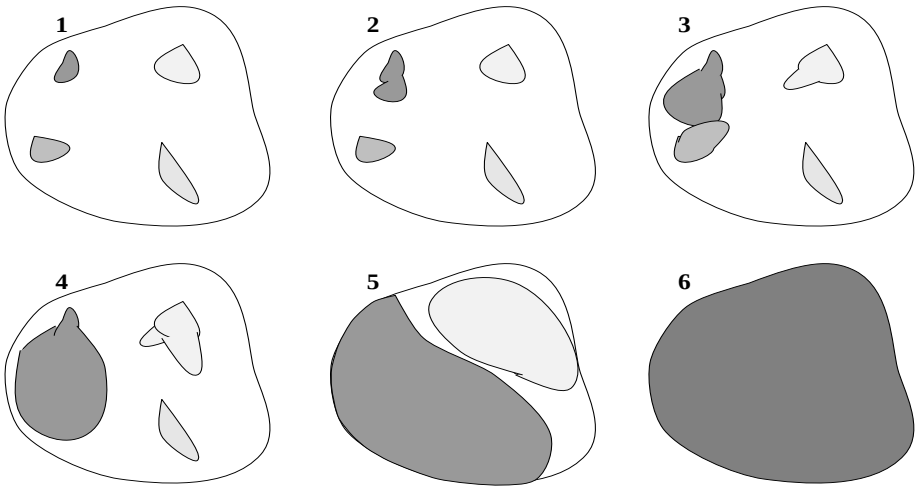


Figure 18: Overview of Subgraph Creation process

- then place the action at the front of the action list and repeat step 4.
- else determine new "best" actions for all of the subgraphs that have had actions removed since the last time this "else" was evaluated. Add these new best actions plus the action for the newly created subgraph to the action list and re-order it. Repeat step 4.
- Remove any subgraphs from this list of subgraphs created above that are not involved in creating the final subgraph which contains the entire input problem.

To find a list of potential actions to take on a subgraph:

1. Set Potential-Actions to nil.
2. For each Vertex in the subgraph that has an edge outside the subgraph, determine the list of vertices needed to be added to cover it. Make an action that adds these vertices and add it to Potential-Actions.
3. For each subgraph adjacent to any vertex in this subgraph, if combining the subgraphs results in more vertices covered than by the two subgraphs separately, create an action that combines the subgraphs and add it to Potential-Actions.

Figure 18 illustrates the process of subgraph creation. 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. 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 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.

Several measures are taken to reduce complexity. First, once a subgraph is expanded, it can no longer be used in other subgraph expansions. This keeps the number of subgraphs under consideration to a minimum. Second, we only recalculate actions for subgraphs directly affected by previous expansions. If subgraph expansion has been occurring in one portion of the graph, only subgraphs in that portion need to have actions recalculated. Finally, as stated above, we allow the subgraph created last to expand as long as it can do so without increasing the maximum input complexity. In practice, we find that, after some subgraph combinations in the early stages, expansion occurs almost entirely by adding individual vertices onto existing subgraphs. This occurs because the input complexity of combining subgraphs is usually high. Because expansion generally occurs in this manner, it makes sense to allow the last created subgraph to continue expanding, if possible.

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.

We also will demonstrate below that for any given problem, this subgraph creation process might be simplified. Problems in computational semantics, for instance, typically present as tangled trees (limited constraints between siblings and cousins superimposed on a basic tree structure). We can take advantage of this to create very simple algorithms that can partition the graph, sometimes better than this more complex version.

This section has been kept brief intentionally. Subgraph decomposition is not the focus of this research, although improvements can dramatically reduce the complexity of HG. In practice, the simple algorithm presented above produces excellent results; in all but the

two smaller problems the optimal maximum input complexity was found in time negligible compared to the actual HG processing. Other approaches might be able to deliver better overall performance, but these results were deemed acceptable for now. In the next sections, these principles of subgraph creation, along with the Hunter-Gatherer algorithm, will be exemplified using problems from semantic analysis.

4. The Mikrokosmos Machine Translation System

Implied information, background knowledge, ellipsis, coreference, figurative speech, ambiguity; these are a few of the immense challenges a natural language semantic system faces. And yet, humans process language in real time every day with very little misunderstanding. How can a computer do the same?

By constraining the problem. Fifty six million and some odd amount of thousands is, indeed, a large number. Two hundred and thirty five billion is much larger. These two numbers represent the number of choices an computational semantic system faces for a medium size and a slightly larger size problem. Come across a truly long sentence and the numbers soar past 10^{18}. And that only to determine basic semantic dependencies; add in ellipsis and coreference resolution possibilities and they increase even faster. Such exponential growth in the size of the problem must be constrained if serious work is to be accomplished.

In a "blocks" world, CSP techniques and solution synthesis are powerful mechanisms. Many "real-world" problems, however, have a more complex semantics: constraints are not "yes" or "no" but "maybe" and "sometimes." In computational semantics, certain word-sense combinations might make sense in one context but not in another. We need a method as powerful as CSP for this more complex environment. Our proposal in presenting HG is to 1) use constraint dependency information to partition problems into appropriate sub-problems, 2) combine (gather) results from these sub-problems using a new solution synthesis technique, and 3) prune (hunt) these results using, not constraint satisfaction, but branch-and-bound techniques.

This section provides the background information necessary to understand how HG applies these principles to semantic analysis. We begin by summarizing the Mikrokosmos Machine Translation system. Kavi Mahesh, Evelyne Viegas and Sergei Nirenburg are joint collaborators in this project and have contributed to this section.

In the Mikrokosmos (μK) project being developed by researchers at the Computing Research Laboratory (CRL) of New Mexico State University,[35] a comprehensive study of a

[35] Please see the CRL WWW home page for a more complete overview of the μK Project at http://crl.nmsu.edu/index.html

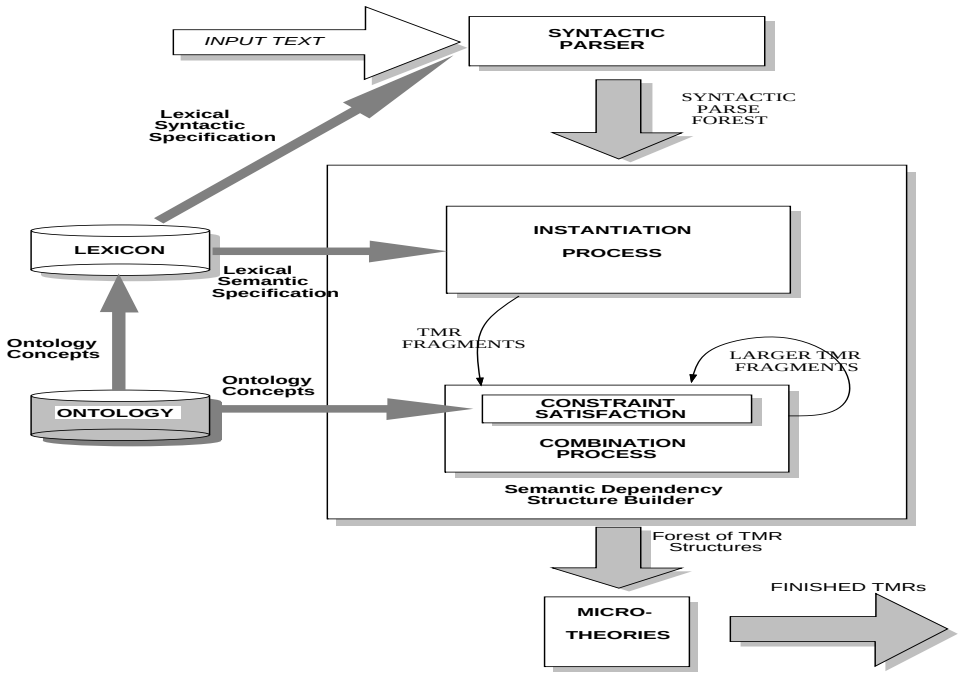


Figure 19: The Mikrokosmos NLP Architecture

variety of microtheories central to the support of KBMT systems is being carried out with the ultimate objective of defining a methodology for representing the meaning of natural language texts in a language-neutral interlingual format called a text meaning representation (TMR). The TMR represents the result of analysis of a given input text in any one of the languages supported by the KBMT system, and serves as input to the generation process. The meaning of the input text is represented in the TMR as elements of an independently motivated model of the world (or ontology). The link between the ontology and the TMR is provided by the lexicon, where the meanings of most open class lexical items are defined in terms of their mappings into ontological concepts and their resulting contributions to TMR structure. Information about the non-propositional components of text meaning such as speech acts, speaker attitudes and intentions, relations among text units, deictic references, etc. is also derived from the lexicon with inputs from other microtheories, and becomes part of the TMR. Figure 19 illustrates the μK architecture for analyzing input texts.

Initially, the project is concentrating on the microtheory of lexical-semantic dependency, the core microtheory underlying our approach to a comprehensive analysis of the meaning of texts, and the one in which the basic structure of events or states and their properties is spec-

ified. Additional microtheories are being developed for aspect, time, modalities, discourse relations, reference, event ellipsis and style.[36]

4.1. Text Meaning Representations

A TMR is a language-neutral description (an interlingua) of the meaning conveyed in a natural language text, and is derived by syntactic, semantic, and pragmatic analysis of the text. Because the TMR is intended to be language neutral, it is also deliberately syntax neutral, and avoids using terminology like clause, proposition, tense, etc., which are associated more closely with the syntactic structure of a particular language. In addition to providing information about the lexical-semantic dependencies in the text, the TMR represents stylistic factors, discourse relations, speaker attitudes, and other pragmatic factors present in the discourse structure. In doing so, the TMR captures not only the meaning of individual elements in the text, but also the relations between those elements, and captures both propositional and non-propositional components of textual meaning.

The results of analysis of an input text are represented in a formal, frame-based language. The meanings of most open-class lexical units are represented by instantiating, combining and constraining concepts available in the ontology. However, the intent of a text cannot fully be captured by instantiating ontological concepts alone; information about pragmatic and discourse related phenomena must be represented, and relations between components of meaning must also be expressed. To facilitate this, the TMR language contains special notation for representing attitudes, relations, speech acts, time, quantities, rates, and sets.

Figure 20 displays a portion of the TMR output for sentence 1:

1a. El grupo Roche, a través de su compañía en Espana, adquirió Doctor Andreu, se informó hoy aquí.

1b. The Roche group, through its company in Spain, acquired Doctor Andreu, it was announced today.

The central concept for the "acquire" clause is ACQUIRE-129. This maps various concepts into the AGENT, THEME and INSTRUMENT slots. The significance of these map-

[36] For example, see (Viegas and Nirenburg, 1995) for a treatment of verbal ellipsis.

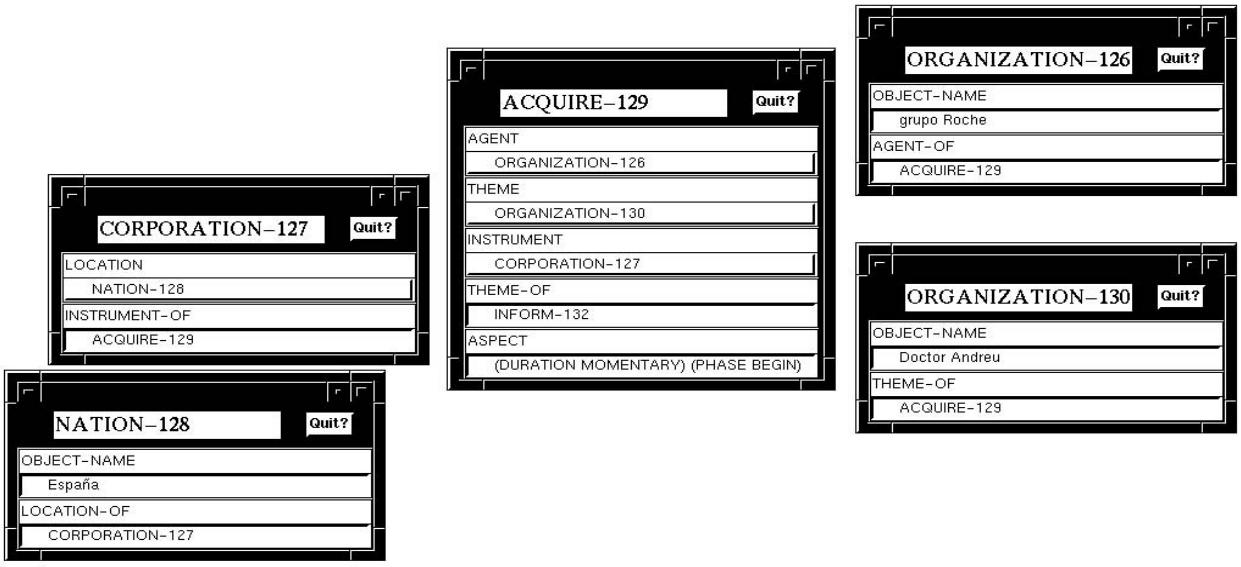


Figure 20: Partial Text Meaning Representation of Example Sentence.

pings and how they were selected will be detailed below.

4.2. Ontology

An ontology for NLP purposes (Mahesh and Nirenburg, 1995; Nirenburg, et al, 1995) is a body of knowledge about the world (or a domain) that a) is a repository of primitive symbols used in meaning representation; b) organizes these symbols in a tangled subsumption hierarchy; and c) further interconnects these symbols using a rich system of semantic relations defined among the concepts. In order for such an ontology to become a computational resource for solving problems such as ambiguity and reference resolution, it must be actually constructed, not merely defined formally. The ontology must be put into well-defined relations with other knowledge sources in the system. In this application, the ontology supplies world knowledge to lexical, syntactic and semantic processes, and other microtheories.

We have finished the initial acquisition of events and properties related to the domain of company mergers and acquisitions (Mahesh and Nirenburg, 1995). The μK ontology consists of over 5000 concepts organized in a tangled hierarchy with ample interconnection across the branches. The ontology emphasizes depth in organizing concepts and reaches depth 10 or more along a number of paths. The branching factor is kept much less than 10 at most

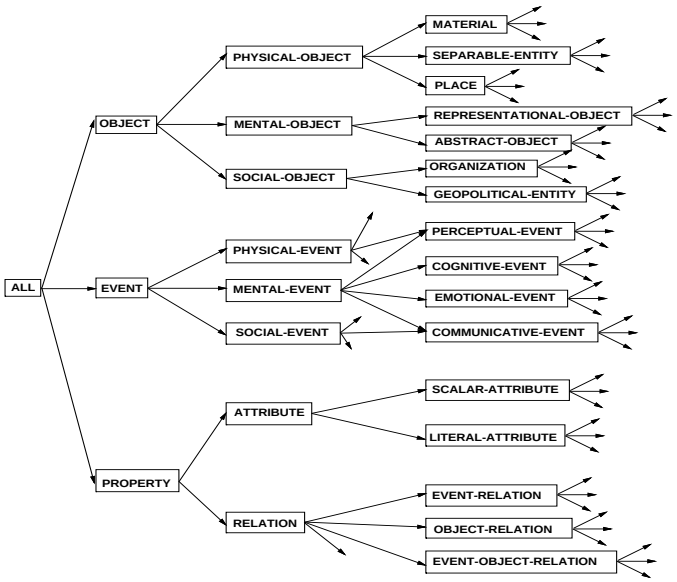


Figure 21: Top-Level Hierarchy of the Mikrokosmos Ontology Showing the First Three Levels of the Object, Event, and Property Taxonomies.

points. Each concept has, on average, 5 to 10 slots linking it to other concepts or literal constants. The top levels of the hierarchy (Figure 21) have proved very stable as we are continuing to acquire new concepts at the lower levels.

Unlike many other ontologies with a narrow focus, our ontology must cover a wide variety of concepts in the world. In particular, our ontology cannot stop at organizing terminological nouns into a taxonomy of objects and their properties; it must also represent a taxonomy of (possibly, complex) events and include many interconnections between objects and events to support a variety of disambiguation tasks. For example, in the sample text above, the analyzer must distinguish between two meanings of "adquirir": 1) ACQUIRE, and 2) LEARN, where ACQUIRE and LEARN are concepts in the ontology defined in Figure 22.

In our example sentence, the fact that the THEME of LEARN is constrained to be INFORMATION will be enough to eliminate it from consideration. Additional examples of disambiguation will be given below.

The ontology aids natural language processing in the following ways:

- It represents selectional preferences for relations between concepts. This knowledge is invaluable for resolving ambiguities by means of the constraint satisfaction process.

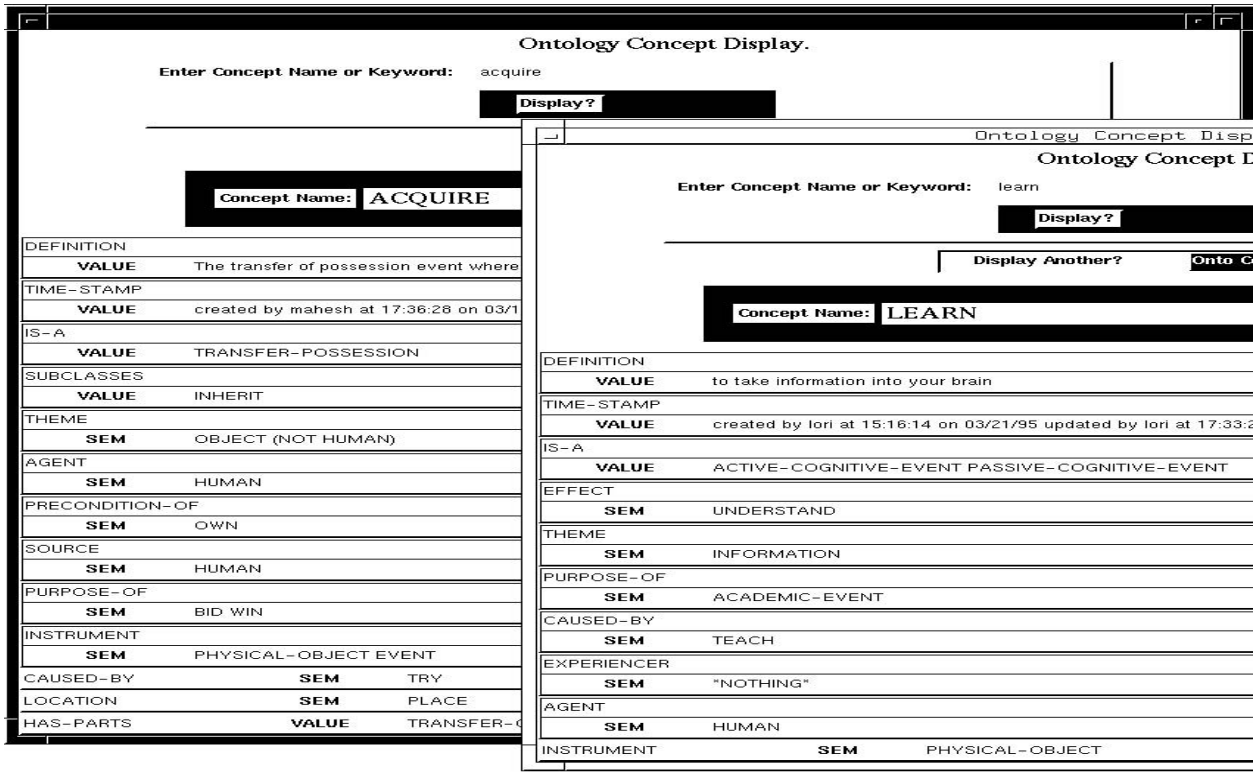


Figure 22: Ontological Definition of ACQUIRE and LEARN.

- It enables inferences to be made from the input text using knowledge contained in the concepts. This can help resolve ambiguities as well as fill gaps in the text meaning. A default value from the ontological concept can be filled in a slot, for example, when a text does not provide a specific value.

- It enables inferences to be made using the topology of the network, as in searching for the shortest path between two concepts. Such search-based inferences can support metonymy and metaphor processing, figuring out the meaning of a complex nominal or be used in constraint relaxation when the input cannot be treated with the available knowledge.

4.3. Semantic Lexicon

In the model of NLP adopted in a KBMT paradigm, the lexicon becomes the key locus and source of knowledge. Compared to many other computational lexicons, in our approach a

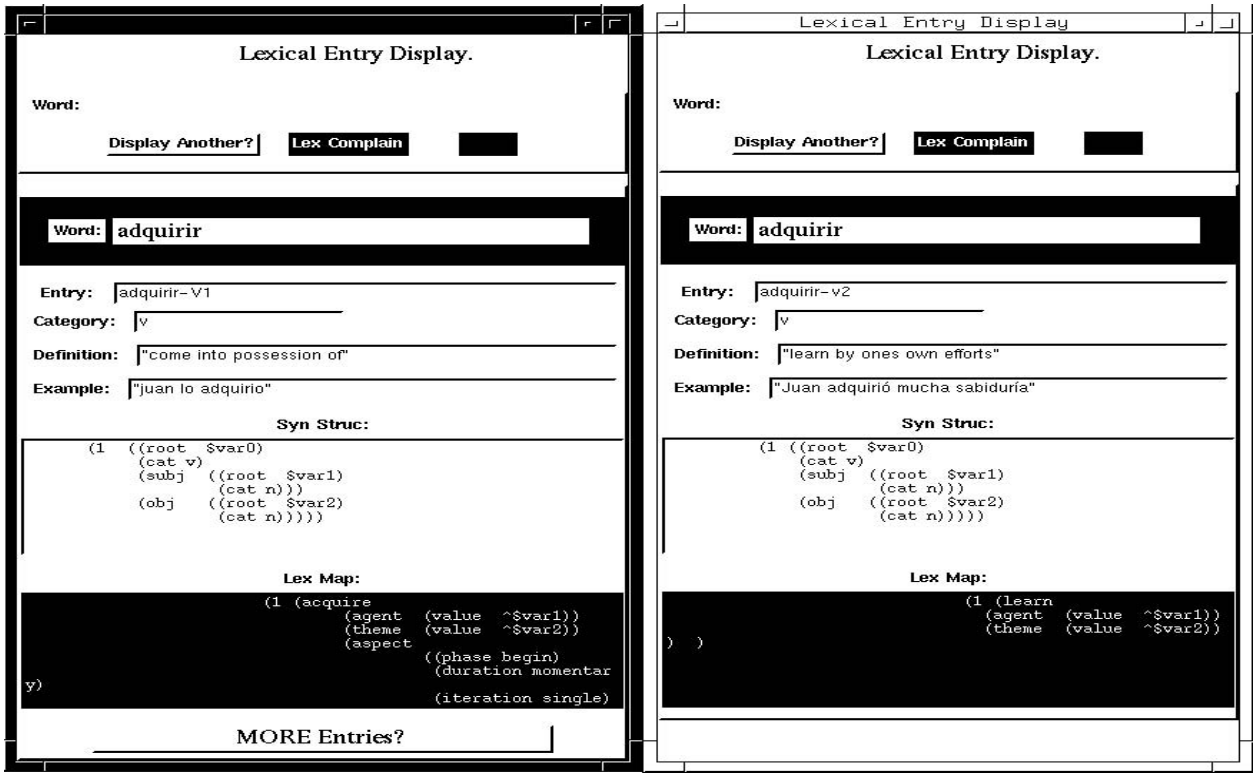


Figure 23: Lexicon Entries for "adquirir".

substantial amount of information is either directly located in the lexicon, or is indexed or referenced through the lexicon. Figure 23 depicts the lexicon entries for the Spanish word "adquirir". There are two entries corresponding to the two word-senses we currently identify. Sense 1 maps into an ACQUIRE concept while sense 2 maps into a LEARN concept.

An entry in the lexicon is comprised of a number of zones, integrating various levels of lexical information, from phonological and morphological to lexical-semantic and pragmatic information. Of particular interest to us here are the SYN-STRUC and SEM zones.

SYN-STRUC Zone. The content of the SYN-STRUC zone of a lexicon entry is an indication of where the lexeme may fit into the syntactic parse of a sentence. In addition, this zone provides the basis of the syntax-semantics interface. The information contained in this zone essentially amounts to an underspecified piece of a syntactic parse of a typical sentence using the lexeme; this piece contains the lexeme in question, and may include information from any number of embedded levels (but typically not more than two) above

or below the current lexeme. The information included reflects those levels and elements of the syntactic parse which the current lexeme syntactically selects for; in the current model, verbs select for all their arguments, modifiers select for their heads, prepositions select for their objects as well as for their node of attachment, etc. The SYN-STRUC zone thus determine such things as subcategorization (optionality is indicated, otherwise obligatoriness assumed), complements allowed, etc.

In the SYN-STRUC zone, we place variables at the ROOT positions selected for by the lexeme in question, which is identified by the variable var0. Subsequently numbered variables (var1, var2, ...) identify other syntactic nodes with which the current lexeme has syntactic or semantic dependencies. For example, the pattern below is appropriate for any regular mono-transitive verb:[37]

```
((root var0)
 (subj ((root var1) (cat n)))
 (obj ((root var2) (cat n))))
```

For instance, in our example sentence, while processing the "adquirir" lexicon entry, "Grupo Roche" will be bound to var1 as the SUBJ, while "Dr. Andreu" will be bound to var2. If a SYN-STRUC requires a syntactic pattern not found in the current sentence, then that word sense is not used.[38]

The variable bindings introduced in the SYN-STRUC provide an interaction with the meaning pattern from the SEM zone in that certain portions of the meaning pattern for a phrase or clause are regularly and compositionally determined by the semantics of the components (Principle of Compositionality); the structure of the resulting meaning pattern is determined not only by the semantic meaning patterns of each of the components, but also by their syntactic relationship in the SYN-STRUC zone.

SEM Zone. The SEM zone provides the mapping to the output semantics. Each SEM zone is basically an underspecified TMR fragment which includes as much meaning as can

[37] An LFG-like syntactic description is used.

[38] Except possibly in failure recovery situations, not discussed here.

be extracted from the word being processed. The interaction of SEM zones from all the words in the sentence[39] result in the final TMR outputs.

Referring to Figure 23 again, the adquirir-v1 SEM zone creates an ACQUIRE concept with AGENT and THEME slots that will be filled by the TMR names that are produced by "grupo Roche" (var1) and "Dr. Andreu" (var2), respectively. Other words in the sentence can fill in additional information in the ACQUIRE TMR. One of the meanings of "a través de," treated as a phrasal entry, will add an INSTRUMENT slot.

In addition to specifying TMR fragments, the SEM zone can add in language-specific semantic constraints which add to or override the language-neutral constraints provided by the ontology.[40] For example, the English verb "to taxi," as in "the jet taxied to the gate" maps into a GROUND-CONTACT-MOTION, but further specifies that its INSTRUMENT must have AIRCRAFT semantics. These "constrained mappings" from language-specific definitions to language-neutral concepts arise because the ontology does not attempt to provide concepts for every conceivable event,[41] nor is its goal to predict all of the idiosyncratic constraints found in different natural languages.[42]

4.4. The Semantic Analyzer

The semantic analyzer is charged with the task of combining the knowledge contained in the ontology and lexicon and applying it to the current input to produce output TMRs. The central tasks involved in this endeavor are to retrieve the appropriate semantic constraints for each possible word sense, test each in context, and construct the output TMRs by instantiating the SEM zones of the word senses which, taken together, best satisfy the combination of constraints. Below, we will examine the steps taken to choose the ACQUIRE meaning of "adquirir" over the LEARN meaning. We will then briefly trace out the other decisions made and provide a summary of the computational methods applied in the analysis process.

---

[39] along with information added by other microtheories

[40] The next section describes how the analyzer retrieves and applies constraints from the ontology

[41] For example, a South-American Indian language has a single word for "she carries water down to the river", but our ontology sadly cannot map directly into such an event.

[42] For example, one word for a human drinking, another word for animals drinking.

Grupo-Roche	a-traves-de	su	compania	en	espana	adquirir	Dr. Andrew
ORGANIZATION	*LOCATION* *INSTRUMENT*	*OWNER*	*CORPORATION* *SOCIAL-EVENT*	*LOCATION* *TEMPORAL*	*NATION*	*ACQUIRE* *LEARN*	*HUMAN* *ORGANIZATION*

Figure 24: Possible Word Senses for Example Sentence.

Generating Constraints. Step 0 in the semantic analysis process is acquiring the syntactic analysis of the input sentence. To avoid duplication of effort, μK uses the output of the Pangloss MT (Frederking, et al, 1994) syntactic analysis module (or Panglyzer (Farwell, et al, 1994)), also developed at NMSU.[43] Since, at the present time, the Panglyzer makes all attachment decisions, μK is limited to deciding between word sense meanings.[44]

The first real step for μK is to gather up all of the possible lexicon entries for each of the words. Figure 24 gives a simplified list of word-sense mapping possibilities for the example sentence. For "adquirir," the two lexicon entries shown in Figure 23 are retrieved, with mappings into ACQUIRE and LEARN word senses. For each word sense, the SYN-STRUC zone must be examined to see if it fits the current sentence. If it does, then the VARs must be bound to their corresponding word instances in the current input sentence. For "adquirir," both word senses have identical SYN-STRUC zones, so the variable binding process displayed in Figure 25 applies to both.

After variable binding, the semantic analyzer examines the SEM zone of each word sense in order to construct a list of constraints that must be satisfied for that word sense. Constraints can arise from five sources:

1. The ontological definition of the current word-sense restricts the semantics of its slot fillers. The definitions for ACQUIRE and LEARN are shown in Figure 22. ACQUIRE and LEARN both require a HUMAN AGENT. ACQUIRE requires a non-HUMAN OBJECT for its THEME, while LEARN requires an INFORMATION THEME.

[43] We would prefer to interleave semantics in the syntactic analysis process. Currently, we are investigating ways to modify the Pangloss-μK interface to provide a level of interleaving, especially with regards to PP attachments.

[44] Again, this limitation will be removed shortly. Choosing between attachments will proceed in the same constraint-satisfaction paradigm as described for word senses, with some possible inputs from attachment microtheories such as "minimal attachment", etc..

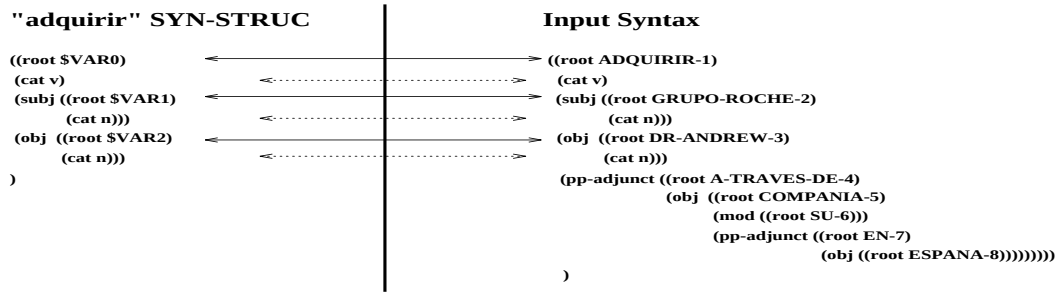


Figure 25: Variable binding for "adquirir".

2. The ontological definition of the word-sense that will **fill** the slot restricts the kind of slots it may be the filler of. Type 1 constraints ask "What kind of fillers do I allow?" Type 2 constraints ask about the **fillers**, "What kind of concepts can this filler modify with the given slot?" For instance, HAMMER, when used as the filler for an INSTRUMENT slot usually modifies some sort of BUILD event. In the example, ORGANIZATION (from Grupo-Roche-1) as an AGENT filler currently[45] does not select for any specific type of event, nor do ORGANIZATION (Dr-Andrew-1) or HUMAN (Dr-Andrew-2) as THEMEs select for a specific event.

3. The ontological definition of the **slot** (the property name that is being added) restricts what its DOMAIN and RANGE can be. Sometimes, in the absence of more specific constraints from 1 and 2 above, μK can find default values by looking up the slot itself in the ontology. An AGENT slot requires its DOMAIN (adquirir, in this case) to be an EVENT and its RANGE (Grupo-Roche) to be HUMAN.[46] A THEME slot REQUIRES an EVENT for the DOMAIN (adquirir) and any OBJECT or EVENT for its RANGE (Dr-Andrew). These constraints are always very general, but still can help eliminate wrong attachments and word meanings.

4. The lexicon entry explicitly includes constraints that override or add to the above ontological constraints. In our example, however, the two word-senses for "adquirir" have no explicit constraints in their lexicon entries.

[45] It is clear that ORGANIZATION as AGENT or THEME should select different types of EVENTS than, say, HUMAN. As the μK ontology is refined, such knowledge will be added.

[46] Please see the discussion of metonymy in the next section to understand how ORGANIZATION (grupo-roche's meaning) can meet a HUMAN constraint.

5. Other structures in the sentence that are not explicitly specified by the lexicon entry can nonetheless modify the word in question. For instance, adjectives and PPs typically add slots to the TMR corresponding to the word they modify, even though they rarely are included in its lexicon entry explicitly. In this case, "adquirir" is modified by "a-traves-de," which, depending on the meaning used, will either add a LOCATION slot or an INSTRUMENT slot to the TMR resulting from adquirir's analysis. In both cases, the slot will be filled by the TMR that results from "compania,"[47] which maps into either a CORPORATION or a SOCIAL-EVENT (as in "companionship"). The only interesting constraints that arise out of these combinations is that for the LOCATION meaning of "a-traves-de," the DOMAIN (adquirir) must be a PHYSICAL-OBJECT (which it is not), whereas the INSTRUMENT meaning requires an EVENT. Although the LOCATION meaning of "a-traves-de" can be eliminated using these constraints, it does not help to further disambiguate "adquirir."

Applying Constraints. μK employs an ontological graph search function (Onyshkevych, 1997) to check constraints. This function determines relevant paths between two concepts and returns a score based on their degree of closeness. For example, **check-onto-con**(ACQUIRE EVENT)[48] returns a score of 1.0 (out of 1.0) since ACQUIRE is a type of EVENT. However, **check-onto-con**(ORGANIZATION HUMAN) returns a score of 0.9 along with the path (ORGANIZATION HAS-MEMBER HUMAN). This indicates that ORGANIZATION can stand in the place of HUMAN because it has HUMAN members. This and other types of metonymy are frequent in natural language and are detected automatically by μK.

Determining the Best Combination of Word Senses. The early versions of the μK analyzer at this point simply tried all of the possible combinations of word senses. Each combination activates the applicable constraints, which are combined into a total score for the combination. The combination with the best total score is chosen as the basic Semantic Dependency Analysis, the core TMRs to which other microtheories (such as aspect and coreference) can be applied. In the example sentence, the following choices were made:

[47] The lexicon entries for "a-traves-de" needs to be consulted to determine these facts.

[48] Which asks "Is ACQUIRE an EVENT?"

1. "a-traves-de" is INSTRUMENT, since its LOCATION meaning would require "adquirir" to be a PHYSICAL-OBJECT.
2. "en" is LOCATION, since its TEMPORAL meaning requires "espana" to be a TEMPORAL-OBJECT.
3. "adquirir" maps into ACQUIRE, since its LEARN sense requires "Dr-Andrew" to be INFORMATION.
4. "Dr-Andrew" is an ORGANIZATION, since its HUMAN meaning cannot be the THEME of an ACQUIRE concept.
5. μK currently has trouble choosing between the CORPORATION and SOCIAL-EVENT meaning of "compania," the object of the "a-traves-de" PP. Both can have locations in Spain, and both can be INSTRUMENTS of EVENTs. At this point, μK needs to add information into the ontology that ORGANIZATIONS can typically fill the INSTRUMENT slot of ACQUIRE acts, but SOCIAL-EVENTS cannot.

It must be stressed that all of these choices resulted from the fact that the combination of all scores from all the semantic constraints for this combination of word senses was judged superior to any other combination of word senses.

The Mikrokosmos Project at NMSU is one of the first, large-scale attempts at a knowledge-based machine translation system. We have successfully implemented the first and central stage of Semantic-Dependency-Structure building. This involved the creation of a large, language independent ontology which interacts with the Spanish semantic lexicon. The μK analyzer extracts semantic constraints from these two sources, analyzes them using a sophisticated graph search function, and determines the combination of choices that leads to the best overall score.

Previous to the work on Hunter-Gatherer, the semantic analyzer performed its search through an exhaustive listing of combinations of word senses. This approach worked for many inputs, but the process was tedious, and, for some longer sentences, the processing could consume several days. For two or three sentences, no results could be obtained at all.

4.5. Using Hunter-Gatherer in Semantic Analysis - The Results

Table 9 shows the latest disambiguation results from Mikrokosmos. These are results from analyzing four real-world texts which had, on the average, 17 sentences with over 21 words per

sentence. Constraints were drawn from the μK ontology, consisting of about 5000 concepts, and lexicon, with about 6000 entries.

It can be seen that, on an average, we get 91% correct disambiguation of open class words, counting only ambiguous words. That is, of all the ambiguous open-class words in the texts, Mikrokosmos selects the right sense 91% of the time. If we count all open class words, the percentage correct goes up to about 97%.

Text:	1. Roche	2. R-R	3. Matra	4. Comercio Bras	Avg
# words	347	385	370	353	364
# sentences	21	16	14	17	17
words/sentence	16.5	24.0	26.4	20.8	21.4
# open-class	183	167	177	177	176
# ambiguous	57	42	57	35	48
# resolved by syntax	21	19	20	12	18
# ambiguous after syntax	36	23	37	23	30
# correctly resolved	30	22	25	22	25
% ambiguous correct	89%	98%	79%	97%	91%
% correct overall	97%	99%	93%	99%	97%

Table 9: Mikrokosmos Word Sense Disambiguation Results

It is obvious from the Table that the performance on the first and third texts (Roche and Matra-Hachette) was worse than the performance on the other two texts. The first and third texts had longer sentences, many more ambiguous words, and constructs that make disambiguation hard (e.g., ambiguous words embedded in appositions). Moreover, just a handful of difficult words led to significantly worse performance in these texts. For example, the word *operacion* occurred several times in these texts and was hard to disambiguate between its WORK-ACTIVITY, MILITARY-OPERATION, SURGERY, and FINANCIAL-TRANSACTION senses.

It can also be noted from Table 1 that syntactic information contributed to about 38% of word sense disambiguation (18 of 48, on an average, were disambiguated by syntax). Often, syntactic binding eliminates word senses and makes an ambiguous word unambiguous in its syntactic context.

One potential flaw with these tests was that the four texts were the same ones used in lexicon and ontology acquisition. This flaw is not as great as it may seem, though. In

research such as statistical modeling, when the data is used to actually train the statistical weights in order to obtain the best coverage, testing on the training data does not make any sense. In our work, however, we do not "tweak" parameters in order to obtain more favorable results. Revisions occur almost exclusively when some lexicon or ontology entry is found to be incorrect. For instance, in the lexicon, an incorrect or incomplete syntactic specification could be eliminating one or more senses of a word, or a mapping to an incorrect concept might be causing unnecessarily low constraint scores. These are the kinds of errors the Mikrokosmos team focused on when developing the knowledge sources.

In order to confirm that our revision techniques were not unduly affecting the results, we tested the Mikrokosmos analyzer on a new text, previously unseen by us. The results are very promising and quite comparable to the ones above.

```
total number of open class words: 104
number of words with only one sense: 78
number of words with only one senses after syntax: 87
number ambiguous words after syntax: 17
number ambiguous words correctly disambiguated: 14
number ambiguous words incorrectly disambiguated: 3
```

i.e., 88.5% correct counting only ambiguous open class words and 97.1% correct counting all open class words. These numbers are almost the same as the numbers for training texts shown earlier in Table 1.

With regards to the Hunter-Gatherer system, these results are presented as evidence that this research can be used profitably in real-life problems. Before HG was implemented, the testing/revision cycle in the project was extremely tedious. Analyzing even small sentences or phrases often took several minutes, and some of the sentences could not be analyzed in whole at all. After HG was implemented, the testing cycle was improved greatly. Not only did the speed increase, but "what-if" experiments were then simplified, and to a large extent, made practical. Furthermore, a 97% success rate in disambiguating open-class words can be taken as strong proof of the reliability of the Hunter-Gatherer system.

Grupo Roche	a traves de	su	compania	en	espana	adquirir	Dr. Andreu
ORGANIZATION	LOCATION INSTRUMENT	OWNER	CORPORATION SOCIAL-EVENT	LOCATION DURING	NATION	ACQUIRE LEARN	HUMAN ORGANIZATION

Figure 26: Possible Word Senses for Example Sentence

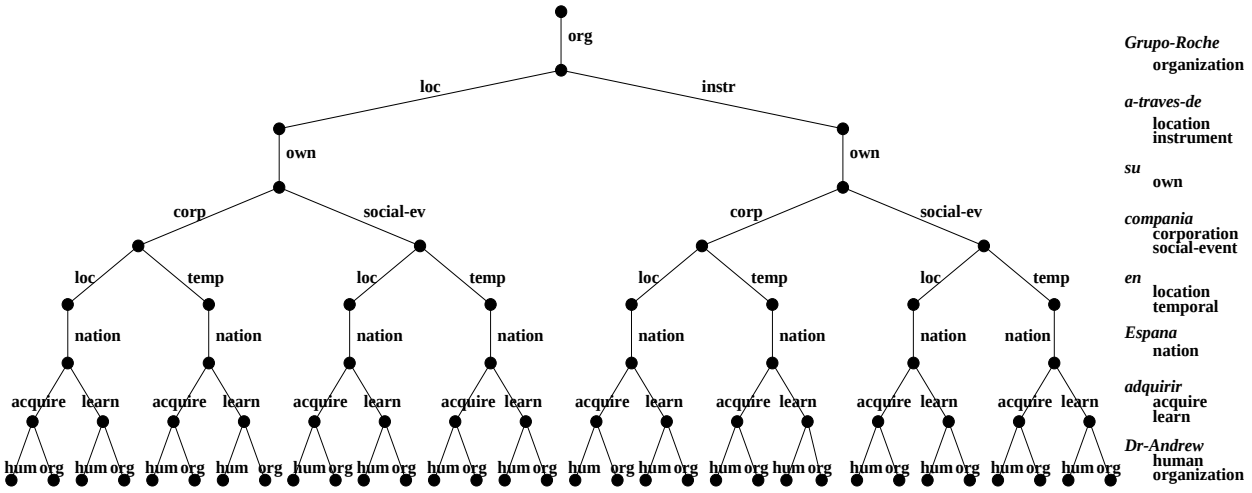


Figure 27: Decision Tree for Example Sentence

5. Using Hunter-Gatherer in Semantic Analysis

This section will examine Hunter-Gatherer's use in computational semantic problems. As will be argued in section 7, these types of problems typically consist of bundles of tightly constrained sub-problems, each of which combine at higher, relatively constraint-free levels to produce the complete solution. The algorithms that identify these bundles, or subgraphs, along with their constraint dependencies, will be demonstrated here. A simple, "literal," semantic problem will be solved using CSP consistency algorithms. The problem of non-literal meanings will be presented, with the solution to this problem to be presented in the "Using Branch-and-Bound in an Uncertain World" section.

Consider Figure 26, an example of a very simple Spanish sentence analyzed by the Mikrokosmos semantic analyzer. The Spanish sentence along with possible word senses are shown. Figure 27 shows the decision tree for this sentence. The paths leading to leaves at

the bottom of the tree represent possible solutions. In this case, there are 32 such solutions.

When some fairly obvious "literal" constraints are imposed, most of these paths can be eliminated. Obviously, CSP techniques can help immensely in this literal-interpretation paradigm. Some of the constraints that could be used to eliminate sub-trees in Figure 27 are as follows:

- the **location** sense for *a traves de* is ruled out because neither of the two senses of *compania* – {**corporation,social-event**} - are descendants of **place** in the ontology.
- *adquirir* is not used in its **learn** sense because **learn** requires an object of type **knowledge**, which *Dr. Andreu* = {**human, organization**} is not.
- *Dr. Andreu* is not **human** because only **organizations** can be **acquired** in our ontology.

Unfortunately, a literal imposition of constraints does not work in computational semantics. For example, *a-traves-de* could very well be **location**, because corporation names are often used metonymically to stand for "the place of the location:"

I walked to IBM.
I walked to where IBM's building is.

Therefore, the fact that *compania* is not literally a **place** only biases the result towards a different interpretation. In fact, under certain contexts, the **location** interpretation might be preferred; certainly it cannot be ruled out completely. Constraint satisfaction techniques such as arc-consistency, therefore, will be of limited value. This will be true of any problem whose constraints cannot be stated in terms of yes/no answers. HG overcomes this limitation by using branch-and-bound as the primary mechanism for pruning the search space.

Figure 28 gives a different view of this same problem by graphically displaying the constraint dependencies present in Figure 26. These constraint dependencies can be identified simply by iterating through the list of constraints[49]

Figure 28 clearly shows three sub-problems, or subgraphs, that are relatively independent. If these subgraphs could be identified, the processing involved in finding a complete solution could be decomposed into three sub-problems. Figure 29 displays the results of such a decomposition. The three subgraphs represent sub-problems with 4, 8 and 4 possible

[49]The semantic constraints are drawn from the lexicon and ontology as described in the previous section.

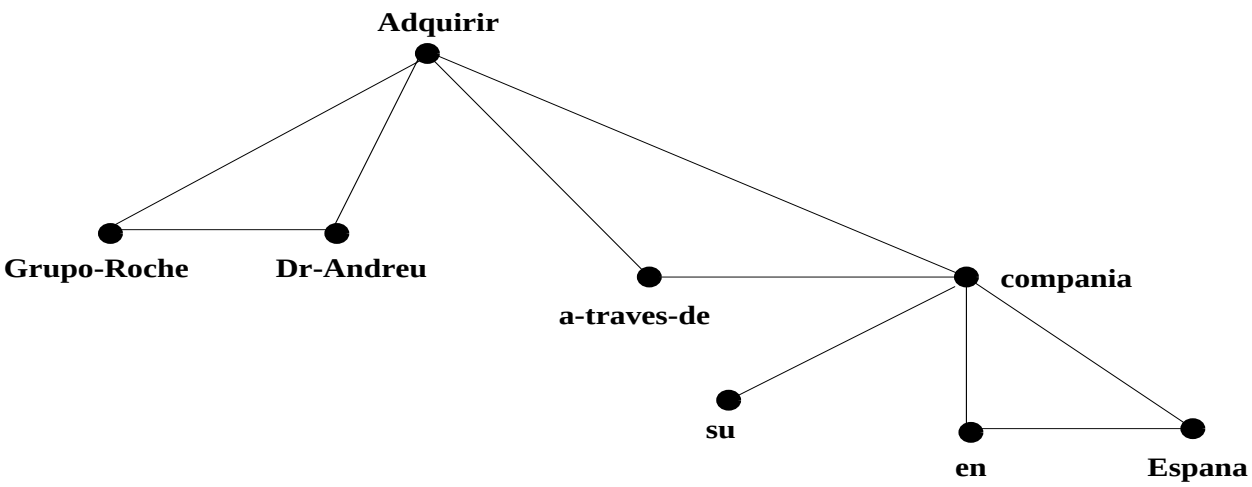


Figure 28: Constraint Dependencies in Sample Sentence

solutions, respectively. Notice that there are still dependencies between the subgraphs. The first two subgraphs are connected because each contains the word *adquirir*; the second two are connected because they both contain the word *compania*.

Section 5.2 will describe how the solution synthesis algorithms can be used to combine results from subgraphs to form larger and larger solutions, the largest of which will be the solution to the whole problem. Section 5.3 then goes through the branch-and-bound optimization stages for this example. But first, section 5.1 is devoted to the decomposition of the original problem into subgraphs which are the inputs to the later stages of processing.

5.1. Identifying Subgraphs of Dependency

Section 3.2 presented an efficient algorithm for partitioning a graph into subgraphs for use by HG. This partitioning algorithm tries to create subgraphs that minimize the complexity of the main HG algorithms. The algorithm began by creating "seed" subgraphs in various regions of the graph. A potential seed is created for each vertex, or variable, in the input simply by listing all of the vertices that are adjacent to it. For instance, for *Grupo-Roche* (G), *adquirir* (A) and *Dr-Andreu* (D) are adjacent. This gives one potential seed (A,D,G). Potential seeds are generated for each variable, with the results as follows:

```
G (Grupo-Roche):   (A,D,G)
```

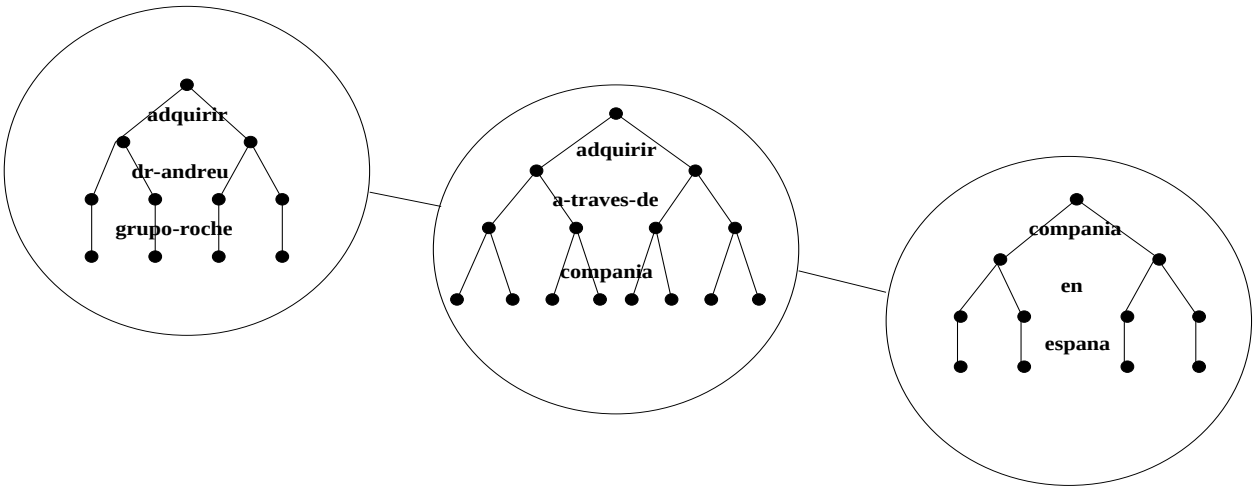


Figure 29: Problem Decomposition in Sample Sentence

```
D (Dr-Andreu):     (A,D,G)
A (adquirir):      (A,D,G,ATD,C)
ATD (a-traves-de): (A,ATD,C)
C (compania):      (A,ATD,C,E,ESP,S)
S (su):            (C,S)
E (en):            (C,E,ESP)
ESP (Espana):      (C,E,ESP)
```

These seeds are then ordered with the smallest first (and duplicates removed):

```
(C,S)
(A,D,G)
(C,E,ESP)
(A,ATD,C)
(A,D,G,ATD,C)
(A,ATD,C,E,ESP,S)
```

Our goal now is to keep only one seed for a given region of the graph. So we simply go through this list in order, and delete any that are in a region included by a previously accepted seed. We define the region of a seed to be the variables in the seed, plus any variables adjacent to the seed. The regions for each seed are given below:

```
(C,S):             (C,S,A,ATD,E,ESP)
(A,D,G):           (A,D,G,ATD,C)
(C,E,ESP):         (C,E,ESP,S,ATD,A)
```

```
(A,ATD,C):          (A,ATD,C,G,D,S,E,ESP)
(A,D,G,ATD,C):      (A,ATD,C,G,D,S,E,ESP)
(A,ATD,C,E,ESP,S): (A,ATD,C,G,D,S,E,ESP)
```

We always accept the first seed on the list. Having done this, for this example, all of the other seeds are rejected because they all contain ATD in their region, which is included in the (C,S) seed. This makes sense because this is a simplified example with a small input graph. Larger inputs result in more than one seed, but the process is identical. This concludes step 1 of the algorithm in section 3.2.

HG requires subgraphs to identify the following information: its input variables (that are not involved in any input subgraphs), its input subgraphs, and all of its variables that are constrained outside the subgraph. For our seed, (C,S), this information can be represented as ((C,S), nil, (C)).

Now, for each seed (in this case only one), we determine the best possible action to take to expand the seed. Possible actions to extend a subgraph are either:

1. The addition of variables onto the subgraph so that at least one additional variable in the seed is now covered (has no constraints outside the new subgraph).

2. The combination of two subgraphs so that at least one new variable that wasn't covered in either of the two subgraphs alone is now covered.

The best possible action is the action that requires the lowest input-complexity. As defined in section 3.1, the input complexity is the sum of all the output complexities of input subgraphs, plus the number of individual variables added.

Since, in this case, there are no other subgraphs, only option 1 is possible. There is only 1 variable in the (C,S) seed that is not already covered, namely C. Therefore, the only action possible is to add on all the variables necessary to cover C. This results in the subgraph described by: ((A,ATD,E,ESP), (C,S), (A)). In other words, by adding on the variables A, ATD, E and ESP to the input subgraph (C,S), we create a new subgraph whose only variable constrained outside the subgraph is A. To keep track of this subgraph, we name it (A,ATD,E,ESP,(C,S)). Since this is the only possible action, we take it, "officially" producing the subgraph (A,ATD,E,ESP,(C,S)).

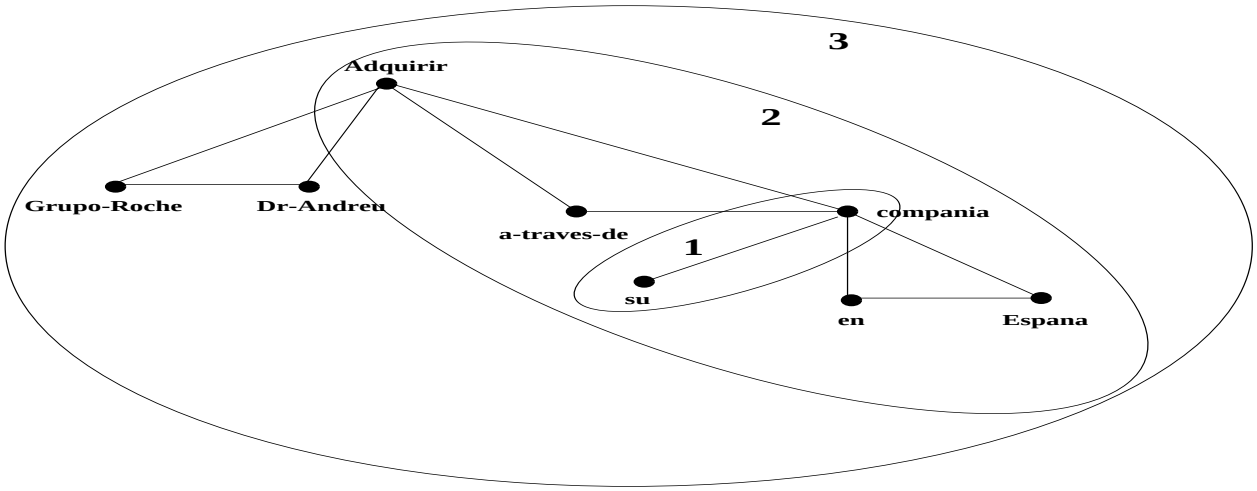


Figure 30: Subgraph Construction - Method 1

Next we try to expand this subgraph. Similar to our first expansion, there is only one variable constrained outside, namely A. Again, there are no other subgraphs available to combine with, so we simply need to add on variables to cover A. This results in the subgraph ((G,D), (A,ATD,E,ESP,(C,S)), nil). That is, by adding on the variables G and D to the subgraph (A,ATD,E,ESP,(C,S)), we produce a subgraph that has no variables affected outside the subgraph. Since this is the only action possible, we take it, producing a subgraph that contains the entire problem. Thus, we are done.

Figure 30 graphically displays the subgraphs created by this process. The input complexity of subgraph 1 is 2, since it contains two input variables. Its output complexity is 1 because only one of its variables are constrained outside the subgraph. The input complexity of subgraph 2 is the output complexity of subgraph 1 (1), plus the number of variables added (4), for a total of 5. Its output complexity is again 1. The input complexity of subgraph 3 is the output complexity of subgraph 2 (1), plus the number of variables added (2), for a total of 3. The maximum input complexity, which will dominate the complexity of HG for this problem, is therefore 5. Assuming each variable had two values in its domain for this simplified problem, we can therefore expect to examine on the order of $2^5 = 32$ combinations it.

It is easy to see that this is not the best way to partition this particular graph. This emphasizes the point made earlier that the algorithm in section 3.2 is a **heuristic** algorithm

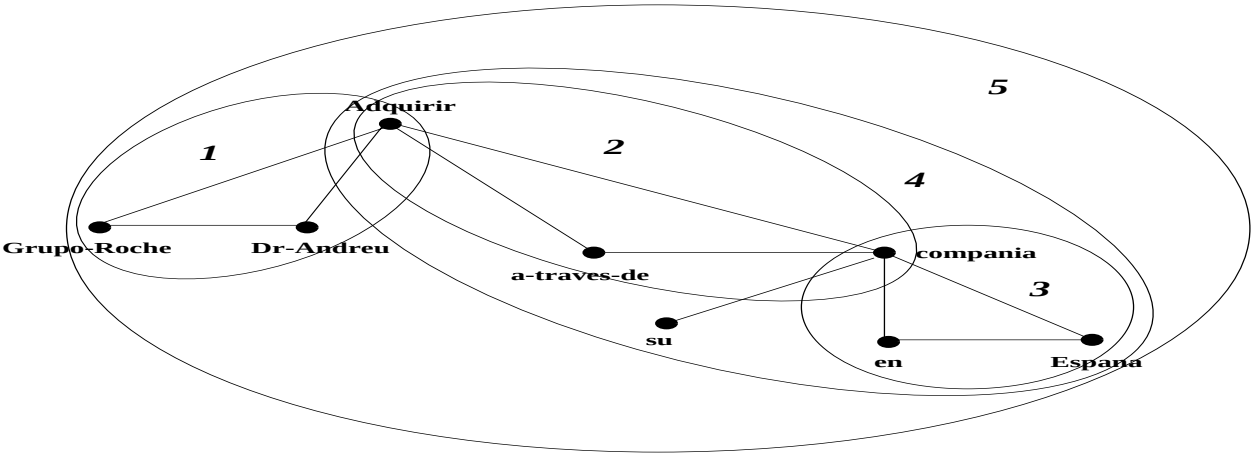


Figure 31: Subgraph Construction - Method 2 For Tree-like Inputs

that is designed to quickly partition a graph in a reasonable manner with respect to the needs of the main HG algorithm. Even though the optimal partitioning was not achieved, HG still guarantees the optimal answer, even if it does take longer.

We have developed additional algorithms to partition specific types of problems. In particular, we can take advantage of the tree-shaped bias in computational semantics (see section 7) to produce, even more easily, the partition in Figure 31. The maximum input complexity for this partitioning is 4. Actually, partitions with maximum input complexity of 3 can be found for this input; however, since partitioning is not the focus of our research, we will not present these additional algorithms. The point is that the graph partitioning methodology, although certainly general enough to apply to many problems using the algorithm in section 3.2, can be further optimized for specific problem types. See section 9.4 for further discussion on graph partitioning, its relation to the input graph topology, and how it can be used to more simply describe how and why HG works.

5.2. Solution Synthesis for Semantic Analysis

Once the input subgraphs are determined, Hunter-Gatherer begins processing. We will assume the input subgraphs of Figure 31. Below we outline how our solution synthesis mechanism is used to combine results from previous subgraphs in this example. Following this is a detailed example of how the branch-and-bound optimization stage is carried out.

As described above, Freuder introduced solution synthesis as a means to "gather up" all solutions for a CSP without resorting to traditional search methods. Freuder's algorithm created a set of two-variable nodes that contained combinations of **every** two variables. These two variable nodes were then combined into three variable nodes, and so on, until a node containing all the variables, i.e. the solution, was synthesized. At each step, constraints were propagated down and then back up the "tree" of synthesized nodes.

Tsang improved greatly on this scheme with the Essex Algorithms. These algorithms assumed that a list of the variables could be made, after which two-variable nodes were created only between adjacent variables in the list. Higher order nodes were then synthesized as usual, starting from the two-variable nodes. Tsang noted that some orderings of the original list would prove more efficient than others, most notably a Minimal Bandwidth Ordering" (MBO), which seeks to minimize the distance between constrained variables.

HG extends the concept of MBO and carries it to a higher level. The whole concept of synthesizing solution sets one order higher than their immediate ancestors is thrown out. Synthesis, here, is aimed at maximally interacting groupings of variables, of any order. Furthermore, this process of using maximally interacting groupings, or subgraphs, extends to the highest levels of synthesizing. Tsang only creates second order nodes from adjacent variables in a list, with the list possibly ordered to maximize second order interactions. After that, third and higher order nodes are blindly created from combinations of second order nodes. In this work, in a sense, MBO is continued on to the higher levels. The subgraphs of co-constrained variables described in the previous section guide the synthesis process from beginning to end.

One of the main improvement of this approach comes from a recognition that much of the work in SS-FREUDER and SS-TSANG was wasted on finding valid combinations of variables which were not related. For example, in the example worked out in section 2.2, the words "IBM" (I), "Jacob-Smith" (J) and "ten-million-dollars" (T) (among others) are not directly related by any constraints. Despite this, valid combinations for $N_{IJ}, N_{IT}, N_{JT} and N_{IJT}$ are calculated by SS-FREUDER and, depending on the ordering used, could also be calculated by SS-TSANG. Furthermore, the SS-TSANG algorithm tends to carry along unneeded ambiguity. As shown above, if two related variables are not adjacent in the original list, their disambiguating power will not be applied until they happen to co-occur in a higher-order synthesis. It should be noted that Freuder's algorithm does not have this disadvantage, be-

cause **all** combinations of variables are created.[50] The current work combines the efficiency of Tsang's algorithms with the early-disambiguation power of Freuder.

For our example, the Tsang algorithm would first use an MBO ordering of the input variables: (G D A ATD C E ESP S). Second order combinations variables adjacent in this list would then be synthesized. G-D, D-A, A-ATD, etc., would be generated at this step. Third order combinations, G-D-A, D-A-ATD, etc., would then be synthesized, and so on, until the eighth order solution containing the entire problem was synthesized. Even if our constraints were of the yes/no variety, enabling a certain amount of pruning at each synthesis, there would still be wasted processing because some constraint interactions would not be addressed until the higher-order syntheses. See section 8.1 for a direct comparison of Tsang's solution synthesis algorithm with our own in contexts where constraints are of this type. The fact is, however, that constraints in semantic analysis are more complex so that straightforward constraint satisfaction and solution synthesis techniques such as those used by Tsang are useless.

In Figure 31, there are only five input subgraphs which guide the synthesis process. The smaller subgraphs, 1, 2 and 3, are processed first. Each is optimized (see section 5.3 for examples) based on the principles in section 3.1. Next, subgraphs 2 and 3 are synthesized into subgraph 4. This involves combining all pairs of plans in subgraphs 2 and 3, as long as those plans are consistent (for instance, if the subgraph 2 plan assigns a value to a variable, the subgraph 3 plan, if it assigns a value to that variable at all, must assign the same value). Subgraph 4 is then optimized, before being combined with subgraph 1 to produce the answer for the whole problem. The bulk of the optimization occurs in the lower order subgraphs which were chosen to maximize this phenomena. Focusing the synthesizer on subgraphs that will maximally optimize produces large savings while still guaranteeing the correct solution.

5.3. Using Branch-and-Bound in the Uncertain World of Semantic Analysis

So far, for this example we have simply modified solution synthesis algorithms to use maximally interacting subgraphs of variables in order to exploit their optimization power as early as possible. Combined with CSP techniques, this give fast and completely reliable answers

[50] But only at great expense.

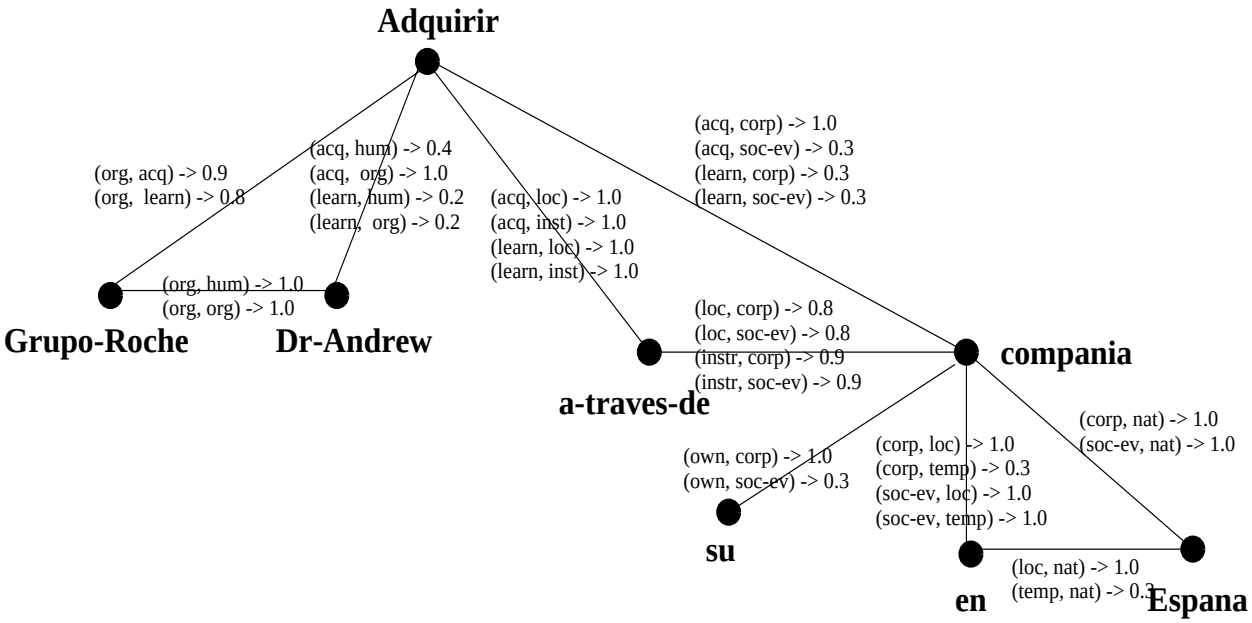


Figure 32: Constraint Scores

to simplified computational semantic problems. The difficulty is that the problems CSP techniques work well on **are** simplified. They require yes/no evaluations. In computational semantics, however, semantic constraints must be used only as **tendencies**, or **preferences**, not sure-and-fast answers. Therefore, for computational semantics, bare CSP techniques are not that helpful. In this section, we will demonstrate that branch-and-bound techniques can be combined with solution synthesis to overcome this problem.

A more complex version of Figure 28 is repeated here as Figure 32. In this figure, constraint "tendencies" are given for each possible combination of value assignments on an arc. Each tendency is rated on a scale of 0 to 1, with 1 being a perfect (literal) semantic match. Some of the values assigned in Figure 32 are explained below:[51]

- Grupo-Roche - Adquirir: Grupo-Roche has only one meaning, ORGANIZATION. As the AGENT of an ACQUIRE event (the (org, acq) arc), it is given a rating of 0.9. This reflects the fact that ORG is not literally a HUMAN (the expected AGENT of an ACQUIRE event), but through metonymy processing, it can be matched. As the AGENT of a LEARN event (the (org, learn) arc), it receives a slightly lower score, reflecting a slightly less promising metonymic relationship in that context.

[51]The main point here is not **why** the scores given were assigned. See section 4 for a discussion of constraints and how scores are determined. These scores are only a fairly intuitive assignment of values that will be used to demonstrate the algorithms below.

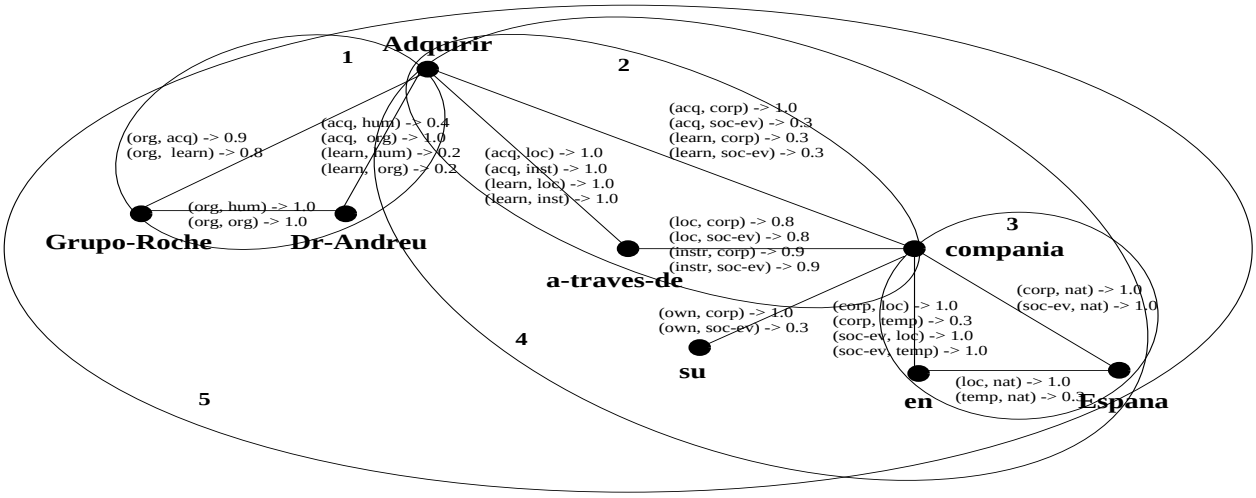


Figure 33: Constraint Subgraphs in Sample Sentence

- Grupo-Roche - Dr-Andrew: These two nouns do not, in this case, constrain each other. They are both given a score of 1.0. In some cases, the grammatical subject and grammatical object of a clause could constrain each other through collocational constraints.
- Adquirir - Dr-Andrew: Each of these words have two meanings, yielding the four possible combinations given. LEARN usually takes some sort of INFORMATION as its THEME; neither HUMAN nor ORG fit this very well, resulting in low scores. ACQUIRE generally takes an OBJECT for its THEME, which ORG is. ACQUIRE generally does not take HUMAN as its THEME, which results in a lower score for the (acq, hum) arc.
- Compania - En: Again, each word can have two meanings. The CORP meaning of Compania can have a LOC specified, but not usually a time (TEMPORAL). On the other hand, SOCIAL-EVENT can easily have either a LOC or a time specified.

The key observation that enables the application of branch-and-bound to solution synthesis problems is that some variables in a synthesis group, or subgraph, are unaffected by variables outside the subgraph. For example, in Subgraph 1 of Figure 33, (Adquirir, Grupo-Roche, Dr-Andrew), both *Grupo-Roche* and *Dr-Andrew* are not connected (by constraints) to any other variables outside the subgraph. This will allow us to optimize, or reduce, Subgraph 1 with respect to these two variables. Branch-and-bound techniques are used in this reduction.

Recall that when creating subgraphs, all nodes constrained outside the subgraph were

identified. To implement HG with branch-and-bound, we add this list of constrained variables to the inputs. That is, HG accepts as input a list of Subgraphs. Each subgraph is in the form:

(In-Vertices In-Subgraphs Constrained-Vertices)

This list is ordered from smaller subgraphs to larger subgraphs. For the example sentence, the following would be the list of input subgraphs:

```
(
 (subgraph-name: A
                 In-Vertices: (adquirir grupo-roche dr-andrew)
                 In-Subgraphs: NIL
                 Constrained-Vertices: (adquirir))
 (subgraph-name: B
                 In-Vertices: (adquirir a-traves-de compania)
                 In-Subgraphs: NIL
                 Constrained-Vertices: (adquirir compania))
 (subgraph-name: C
                 In-Vertices: (en compania espana)
                 In-Subgraphs: NIL
                 Constrained-Vertices: (compania))
 (subgraph-name: D
                 In-Vertices: (su)
                 In-Subgraphs: (B C)
                 Constrained-Vertices: (adquirir))
 (subgraph-name: E
                 In-Vertices: NIL
                 In-Subgraphs: (A D)
                 Constrained-Vertices: NIL)
)
```

In the last subgraph, no variables are identified in the List-of-Constrained-Vars. This last subgraph contains the entire problem and thus has no variables outside of it. The result of processing it will be the optimal solution for all variables. The second to last subgraph identifies only *adquirir* in the List-of-Constrained-Vars. Its solution set will contain plans for each possible value of *adquirir*, with each plan being optimized with respect to all its other variables.

5.3.1. An Example Application

This is how HG works for the example sentence. Refer to Figure 16 for references to variable and function names. The first Subgraph input to PROCESS-SUBGRAPH is

```
((adquirir grupo-roche dr-andrew)
  NIL
  (adquirir))
```

That is, the In-Vertices are (adquirir grupo-roche dr-andrew), there are no In-Subgraphs, and the list of Constrained-Vertices is (adquirir). The COMBINE-INPUTS function has no input subgraphs to combine, so it simply creates an exhaustive list of possible combinations of the In-Vertices:

```
((<A,acq>,<G,org>,<D,hum>)
 (<A,acq>,<G,org>,<D,org>)
 (<A,learn>,<G,org>,<D,hum>)
 (<A,learn>,<G,org>,<D,org>))
```

We will refer in what follows to any possible combination of values as a plan. For each plan, all of the constraints relevant to it are extracted and evaluated. For instance, in Figure 33, we can see there is a constraint between *adquirir* and *Dr-Andreu*. When the assignments $< A, acq >$ (the ACQUIRE meaning of *adquirir*) and $< D, hum >$ (the HUMAN meaning of *Dr-Andreu*) are made, the score is 0.4. Again, this reflects the fact that HUMANs are not usually the THEME of ACQUIRE events. Each of the scores for each plan are combined[52] with the results shown below:

```
 (<A,acq>,<G,org>,<D,hum>)   [0.9 * 0.4 * 1.0 = 0.36]
 (<A,acq>,<G,org>,<D,org>)   [0.9 * 1.0 * 1.0 = 0.9]
 (<A,learn>,<G,org>,<D,hum>) [0.8 * 0.2 * 1.0 = 0.16]
 (<A,learn>,<G,org>,<D,org>) [0.8 * 0.2 * 1.0 = 0.16]
```

Arc consistency functions are run at this point. For semantic analysis, however, constraints give rise to scores between 0 and 1. These "fuzzy" constraint values cannot be used

[52] Combining scores for a list of constraints is complicated. To simplify, we assume here that all individual constraint scores are multiplied together.

by traditional arc consistency routines; however, a threshold could be set to some reasonable value (possibly 0.5), below which a constraint score is considered a definite violation. Arc consistency routines could then be used to propagate these failures. For this example we will assume that arc consistency is not used.

Because *adquirir* is the only Constrained-Vertices, only combinations of value assignments involving it will be returned by REDUCE-COMBOS. Only two such possible assignments exist: $< A, acq >$ *and* $< A, learn >$. The plan $(< A, acq >, < G, org >, < D, org >)$ maximizes $< A, acq >$ with a score of 0.9. The two possible plans for $< A, learn >$ have identical scores. Both could be kept; the algorithm above simply chooses the first.[53] Therefore, the list of Output-Plans for the first subgraph is:

```
((<A,acq>,<G,org>,<D,org>)
 (<A,learn>,<G,org>,<D,hum>))
```

The other plans are discarded. It must be stressed here that discarding the other plans in no way incurs risk of finding sub-optimal solutions. The only variable that could be effected further outside this subgraph is *adquirir*, so plans that maximize each of its possible assignments were chosen. Nothing can happen later on to cause, for instance, the $(< A, acq >, < G, org >, < D, hum >)$ plan to become better than the $(< A, acq >, < G, org >, < D, org >)$ plan, since all interactions involving G and D have been accounted for.

The other subgraphs proceed along the same lines. The (adquirir a-traves-de compania) subgraph produces the following list of Output-Combos (in this case, *adquirir* and *compania* are both effected outside the subgraph, so we need to find optimal plans for all combinations of their possible values):

```
((<A,acq>,<ATD,instr>,<C,corp>)
 (<A,acq>,<ATD,instr>,<C,soc-ev>)
 (<A,learn>,<ATD,instr>,<C,corp>)
 (<A,learn>,<ATD,instr>,<C,soc-ev>))
```

The (en compania espana) subgraph produces:

[53] The best way to handle identical scores is to keep the first plan and save the second in a list of "synonyms." In practice, identical scores only seem to occur when all constraints are met literally (a total score of 1.0).

```
((<E,loc>,<C,corp>,<ESP,nat>)
 (<E,loc>,<C,soc-ev>,<ESP,nat>))
```

It becomes more interesting when smaller subgraphs are synthesized into larger ones. For the following subgraph:

```
((su)
 ((adquirir a-traves-de compania) (en compania espana))
 (adquirir))
```

there are two In-Subgraphs, (adquirir a-traves-de compania) and (en compania espana), and one In-Vertices, *su*. COMBINE-INPUTS synthesizes compatible plans for the smaller subgraphs into larger subgraphs. Recall that the Output-Combos for the two subgraphs were:

```
((<A,acq>,<ATD,instr>,<C,corp>)    and ((<E,loc>,<C,corp>,<ESP,nat>)
 (<A,acq>,<ATD,instr>,<C,soc-ev>)       (<E,loc>,<C,soc-ev>,<ESP,nat>))
 (<A,learn>,<ATD,instr>,<C,corp>)
 (<A,learn>,<ATD,instr>,<C,soc-ev>))
```

"Compatible" plans are then synthesized. "Compatible" plans include all those for which like-variables have the same assignment. For instance, $(< A, acq >, < ATD, instr >, < C, corp >)$ and $(< E, loc >, < C, soc - ev >, < ESP, nat >)$ are **not** compatible because a different value is assigned for C. The result is showed below:

```
((<A,acq>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>)
 (<A,acq>,<ATD,instr>,<C,soc-ev>,<E,loc>,<ESP,nat>)
 (<A,learn>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>)
 (<A,learn>,<ATD,instr>,<C,soc-ev>,<E,loc>,<ESP,nat>))
```

Each of these combinations is then combined with the single In-Vertices. This will, in effect, add the assignment $< S, own >$ onto each of the combinations:

```
((<A,acq>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>,<S,own>)
 (<A,acq>,<ATD,instr>,<C,soc-ev>,<E,loc>,<ESP,nat>,<S,own>)
 (<A,learn>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>,<S,own>)
 (<A,learn>,<ATD,instr>,<C,soc-ev>,<E,loc>,<ESP,nat>,<S,own>))
```

For the input subgraph ($< A, acq >, < ATD, instr >, < C, corp >$) only ATD was not in Constrained-Vertices, while for ($< E, loc >, < C, soc-ev >, < ESP, nat >$) both E and ESP were not Constrained-Vertices. In the plans input to REDUCE-COMBOS for this synthesis, those NON Constrained-Vertices are already reduced to optimal values for the plans they are in. For the output of this synthesis, only A (adquirir) is identified in the input Subgraph description as a member of Constrained-Vertices. Therefore, REDUCE-COMBOS should return only two plans, corresponding to the two possible values of A, each of which will have all the other variables optimized with respect to the value of A chosen.

REDUCE-COMBOS calculates the combined score for each of the constraints in the subgraph. In practice, a list of scores for input Subgraphs should be maintained so that each test is not repeated for each Plan. Only the constraints involving In-Vertices and constraints **between** input subgraphs should have to be calculated. In this case, all constraints involving S (su) need to be calculated since it was not involved in any input subgraphs. If there were any cross-constraints between subgraphs, such as between A (adquirir) and E (en), those constraints would need to be added. In summary, PROCESS-SUBGRAPH, and thus REDUCE-COMBOS, should only need to calculate constraint scores for constraint interactions new to the subgraph.

The score for the first plan is calculated as:

```
(<A,acq>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>,<S,own>)
  = (0.9 * 1.0 * 1.0) = 0.9
```

where 0.9 is the score for the ($< A, acq >, < ATD, instr >, < C, corp >$) subgraph (calculated above), the first 1.0 is the score for the ($< C, corp >, < E, loc >, < ESP, nat >$) subgraph, and the second 1.0 is the score of the constraint added by S (su).

The second input plan to REDUCE-COMBOS is ($< A, acq >, < ATD, instr >, < C, soc - ev >, < E, loc >, < ESP, nat >, < S, own >$). The score for this plan is calculated as

```
(<A,acq>,<ATD,instr>,<C,soc-ev>,<E,loc>,<ESP,nat>,<S,own>)
  = (0.27 * 1.0 * 1.0) = 0.27
```

where the score 0.27 is the score of the ($< A, acq >, < ATD, instr >, < C, soc - ev >$) subgraph, the first 1.0 is the score of the ($< C, soc - ev >, < E, loc >, < ESP, nat >$)

subgraph, and, again, the second 1.0 is the score of the constraint added by S (su). Because this plan has the same value assignment for A as the previous plan, and it has a lower score, it is discarded.

The third input plan, $(< A, learn >, < ATD, instr >, < C, corp >, < E, loc >, < ESP, nat >, < S, own >)$, receives a total score of 0.27.

The last input plan, $(< A, learn >, < ATD, instr >, < C, soc - ev >, < E, loc >, < ESP, nat >, < S, own >)$, has the same value assignment for A as the third plan, and receives a total score of 0.081. It is, therefore, discarded, since it has a lower score than the previous plan with the same assignment of A.

The output of REDUCE-COMBOS, and therefore the output of PROCESS-SUBGRAPH for this subgraph, is:

```
((<A,acq>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>,<S,own>)
 (<A,learn>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>,<S,own>))
```

The only variable that was not explicitly maximized in these plans is *adquirir*. This makes sense because *adquirir* is the only variable that interacts outside the subgraph.

The final subgraph, the solution to the whole problem, is then synthesized from Subgraph 1 and Subgraph 4, the subgraph just created. PROCESS-SUBGRAPH, in this case, combines all the compatible plans from Subgraph 1 and Subgraph 4, then, because the arc consistency will do nothing, sends them on to REDUCE-COMBOS. Because all of the variables except *adquirir* were optimized for the plans they were in, only *adquirir* must be reduced here. This produces a single optimal plan:

```
(<A,acq>,<G,org>,<D,org>,<ATD,instr>,<C,corp>,<E,loc>,<ESP,nat>,<S,own>)
```

5.3.2. Results of Using Hunter-Gatherer for Semantic Analysis

To illustrate how branch-and-bound dramatically reduces the search space, consider the results of applying it to the sample sentence.

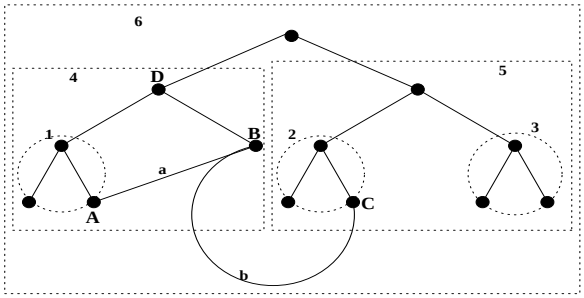


Figure 34: Cross Dependencies

Subgraph	Input-Subgraphs	Input-Combos	Reduced-Combos
1	none	2*2*1 = 4	2
2	none	2*2*2 = 8	4
3	none	2*2*1 = 4	2
4	2 and 3	synth only	2
5	1 and 4	synth only	1

The total number of combinations examined is the sum of the input combos; in this case 4+8+4=16. Compare this to an exhaustive search, which would examine (2*1*2*2*2*1*2*1) = 32 combinations. As the input problem size increases, the savings are even more dramatic. This happens because the problem is broken up into manageable sub-parts; the total complexity of the problem is the **addition** of the individual complexities. Without these techniques, the overall complexity is the **product** of the individual complexities.

The only way multiplicative growth can occur is when there are constraints across trees, as in Figure 34. In that Figure, several of the subgraphs cannot be fully reduced due to interactions outside the subgraph. Variable A in Subgraph 1 cannot be fully reduced[54] because of Arc a. Note, however, that when Subgraph 4 is synthesized, Variable A **can** be reduced because, at that point, it does not interact outside the larger subgraph. In Subgraph 4, Variable B cannot be reduced because it interacts with Variable C. Likewise, Variable C

[54]By "fully reduced" we mean all child variables maximized with respect to a single parent, which cannot be reduced because it connects higher up in the tree.

	Problem 1	Problem 2	Problem 3
Number word senses	79	95	119
exhaustive combos	7,864,320	56,687,040	235,092,492,288
HG combos	179	254	327

Near-Linear Processing for Computational Semantics

Figure 35: Results of Semantic Analysis of Various Sized Problems

cannot be reduced in Subgraph 2 because of its interaction with Variable B. In all of these cases, ambiguity must be carried along until no interactions outside the subgraph exist. For Variables B and C, that does not occur until Subgraph 6, the entire problem, is processed.

As is argued below, in computational semantic problems interactions such as Arc a and Arc b generally do not occur.[55] "Governed" interactions such as Variable D directly constraining Variable A can occasionally occur, but these only delay reduction to the next higher subgraph. Thus, some local multiplicative effects can occur, but over the whole problem, the complexity is additive.

To illustrate this point, consider what happens as the size of the problem increases. The table in Figure 35 shows actual results of analyses of various sized problems.

It is interesting to note that a 20% increase in the number of total plans[56] (79 to 95) results in a 626% increase (7.8M to 56M) in the number of exhaustive combinations possible, but only a 42% increase (179 to 254) in the number of combinations considered by HG. As one moves on to even more complex problems, a 25% increase (95 to 119) in the number of plans catapults the exhaustive complexity 414,600% (56M to 235B) and yet only increases the HG complexity 29% (254 to 327). As the problem size increases, the minor effects of "local multiplicative" influences diminishes with respect to the size of the problem. We expect, therefore, the behavior of this algorithm to move even closer to linear with larger problems (i.e. discourse). And, again, it is important to note that HG is guaranteed to produce the same results as an exhaustive search.

[55] "Long distance" dependencies do exist, but are relatively rare.

[56] The total number of plans corresponds to the total number of words senses for all the words in the sentence.

Although time measurements are often misleading, it is important to state the practical outcome of this type of control advancement. Prior to implementing HG, all computational attempts to process larger sentences failed. The largest sentence above was analyzed for more than a **day** with no results. This is the nature of exponential search space. Using HG, on the other hand, the same problem was finished in 17 **seconds**.[57] It must be pointed out as well that this is not an artificially inflated example. It is a real sentence occurring in natural text; and not an overly large sentence at that. Techniques such as HG must be employed to process real-life problems. Knowledge-based semantics has been severely limited until now, subject to arguments that it only works in "toy" environments. HG will enable large-scale investigations in the knowledge-based paradigm.

[57] Using a SPARC 20 with 140MB memory, implemented in compiled Allegro Common Lisp.

6. Hunter-Gatherer in Natural Language Generation

Many text planning systems are being used quite successfully today. Their success, however, has come about as a result of several compromises. Constraining the domain and text types are the most obvious. Related to this, however, are several control issues that have been hidden by the simplified nature of previous systems but are now becoming important as those simplifications are lifted:

> " Most current discourse planners ... rely on customized planning algorithms with procedural semantics for the purpose of solving specific text-planning problems. ... careful analysis of these programs show that there is nothing in their semantics to prevent them from generating incorrect plans, generating plans with redundant steps, or failing to find plans in situations where they exist. To the extent that these planners have been able to avoid these problems, they have done so by severely limiting the expressive power of action descriptions and/or requiring the designer of action descriptions to hand-craft each description to fit correctly into the ad hoc semantics of the specific plan for which the action is intended." (Young & Moore, 1994)

> "With simple state-based representations, complete search strategies will generally be exponential as a function of solution length. With more expressive representations ... determining if solutions to arbitrary problems exist is an undecidable problem. Such disconcerting results have led several researchers to abandon the use of explicit or declarative problem representations. However, it appears that doing so requires that the goals of the agent be within a narrow range that are hard-coded into the problem representation." (Tenenberg, 1991).

> "Time to impact?" (Captain J.L. Picard)

The last quote above graphically illustrates what the first two quotes are talking about. When Capt. Picard asks how long his spacecraft has until it is obliterated by alien fire, he needs to know **NOW**. Furthermore, he needs to know **CORRECTLY**. Unfortunately, the current generation of text planners are not able to process real-life problems quickly, nor are they guaranteed to process them correctly.

Tenenberg states the obvious problem for all AI applications: basic search strategies have exponential time complexity. Young and Moore point out that most text planning systems currently are neither **sound** (guaranteed to give correct answers) nor **complete** (guaranteed to find correct answers). Both citations agree that current approaches sidestep these problems by abandoning declarative knowledge in favor of ad hoc procedures. Young

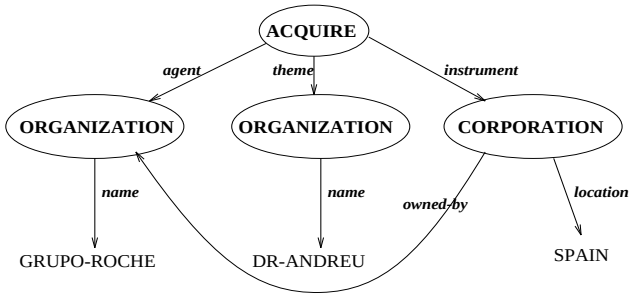


Figure 36: Example Semantic Representation

and Moore go on to argue that the proliferation of procedural knowledge leads to unsoundness and incompleteness, and both papers conclude that such an approach can only be successful on a narrow range of limited problems.

Young and Moore, in their paper, go on to introduce the DPOCL text planning system. The main goal of that research was to ensure soundness and completeness. In this they no doubt succeeded; however, no claims were made concerning the efficiency of their work. Tenenberg, on the other hand, addressed efficiency in his discussion of abstraction in planning. We agree with Young and Moore's conclusion that ad hoc procedures contribute to unsoundness and incompleteness. Declarative knowledge which clearly marks preconditions and effects must be used, along with a control mechanism that ensures soundness and completeness. PICARD, the Mikrokosmos Text Planner, can be viewed as an attempt to add efficiency to this type of control paradigm by applying techniques similar to Tenenberg's abstraction. The work builds on the HUNTER-GATHERER analysis system described above. That system employs constraint satisfaction, branch-and-bound and solution synthesis techniques to produce near linear-time processing for knowledge-based semantic analysis in the Mikrokosmos Machine Translation Project. PICARD enables similar results for text planning by recasting localized means-end planning instances into abstractions[58] connected by usage constraints that allow HUNTER-GATHERER to process the global problem as a simple constraint satisfaction problem.

6.1. Text Planning for Machine Translation

Figure 36 is a representation of the semantic content of a simple natural language sentence.

[58]Or macros, or subgraphs, or sub-groups, depending on your background.

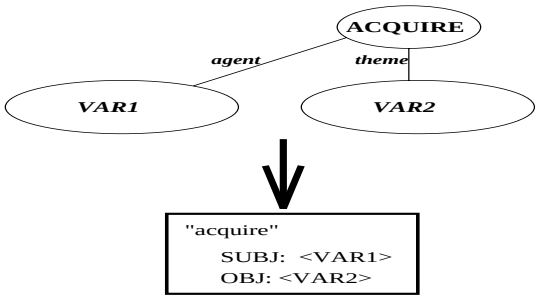


Figure 37: Lexicon Entry for *acquire*

In English the sentence could be rendered "Grupo Roche acquired Dr. Andreu through a subsidiary in Spain." The node names are semantic concepts taken from a language-independent ontology. Arc labels correspond to relations between concepts. The ontology defines for each concept the set of arcs that are allowed/expected, as well as the appropriate filler concepts. For simplicity, additional semantic information such as temporal relationships are not shown. Please consult (Beale, Nirenburg & Mahesh, 1995; Onyshkevych & Nirenburg, 1994 and Mahesh & Nirenburg, 1995) for more information about semantic representation in the Mikrokosmos system. For our purposes, the details of the semantic representation and generation lexicon entries to follow are unimportant; they serve only as simple examples of control concepts that will apply to more complex problems.

Generation lexicon entries attempt to match parts of the input semantic structures and map them into target language surface structures. For instance, a lexicon entry for the concept **ACQUIRE** might look like Figure 37. The VARs in the entry will be bound to the corresponding semantic structures in the input, and their target realization will be planned separately and inserted into the output structure as shown. Typically, lexicon entries also contain semantic and pragmatic constraints. For instance, VAR1 might be constrained to be HUMAN. The entry could also be constrained to apply only to texts with certain stylistic characteristics. Collocational constraints are also important in generation. Any of these constraints can apply locally or can be propagated down to the VARs. The interplay of constraints is a major factor in determining the best overall plan.

Planning for Machine Translation comes in when we try to combine information in various lexicon entries to best match the input semantics with as little redundancy as possible and maximal adherence to the constraints. Figures 38, 39 and 40 represent some possible lexicon

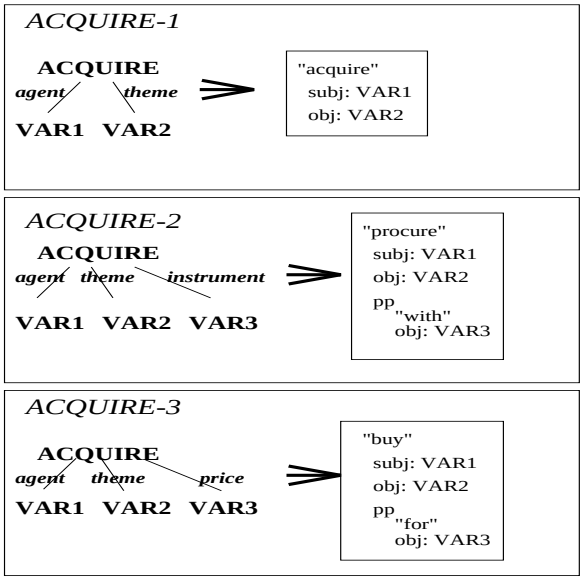


Figure 38: Three entries for ACQUIRE

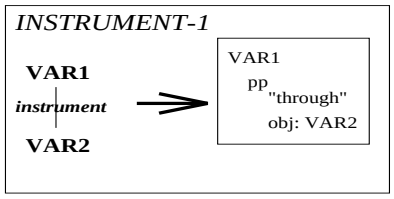


Figure 39: An entry for INSTRUMENT

entries that might be used in planning target English sentences for the structure in Figure 36.

A typical means-end, hierarchical planner[59] uses the following algorithm:

```
PROC PLAN(SEMANTIC-CONTENT)
1  Pick one PLAN that implements base
   meaning in SEMANTIC-CONTENT
2  FOR each PRECONDITION in PLAN not
   already satisfied
3      PLAN(PRECONDITION)
4  FOR each unrealized VAR in PLAN
5      PLAN(VAR)
```

[59] A similar algorithm can be used for non-hierarchical inputs. PICARD does not require hierarchical semantic inputs.

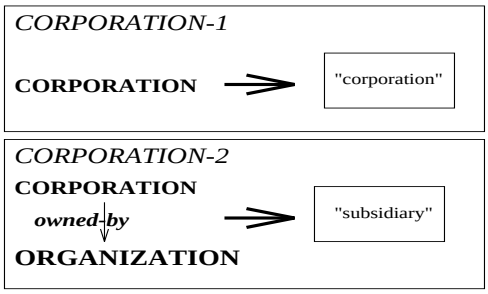


Figure 40: Two entries for CORPORATION

```
6  FOR each unplanned RELATION in
       SEMANTIC-CONTENT
7      PLAN(RELATION)
8  IF FAILURE THEN BACKTRACK
```

For example, to generate text for Figure 36, the **ACQUIRE** concept would be passed to PLAN. Three possible entries exist for **ACQUIRE** (Figure 38). The first, *ACQUIRE-1*, expects a semantic environment with *agent* and *theme* relations, both of which are in the input semantics. There are no preconditions in these simplified entries, so the procedure skips to line 4. There are two VARs that are not realized in the first entry. These VARs are bound to the input semantics, VAR1 to a **GRUPO-ROCHE** instance of an **ORGANIZATION** concept, VAR2 to a **DR-ANDREU** instance of **ORGANIZATION**, and PLAN is called recursively for each, leading eventually to surface strings like *Grupo Roche* and *Dr. Andreu*. There is still an unplanned relation of the input **ACQUIRE** node, namely the *instrument* link to a **CORPORATION** concept. This relation is therefore recursively planned in line 7. "FAILURE" in line 8 can refer to a number of possible outcomes, such as over-generation of semantic content or an inability to plan one of the VARs or RELATIONs. This type of planner can be made to find all possible solutions by storing successful overall plans and then backtracking.

Depending on which lexicon entries are used, plans will be more or less complex. The second entry that realizes an **ACQUIRE** concept, *ACQUIRE-2*, has a "built-in" *instrument* relation. Because of this, there will be no unplanned RELATIONs in line 6. Similarly, the second **CORPORATION** entry incorporates the *ownership* relation. The first entries for both **ACQUIRE** and **CORPORATION** need to specifically plan for those links. *ACQUIRE-3* is an example of over-generation. It expects a *price* relation in the input se-

mantics. This can either be made to cause FAILURE (line 8) or simply penalize any plans that utilize it.

The optimal overall plan can be determined by scoring all of the constraints present in each of the sub-plans, adding in penalties for over-generation, and adding in rewards for shorter plans. The best English sentence using the entries given on the example semantic input would combine *ACQUIRE-2* and *CORPORATION-2* to give something like "Grupo Roche procured Dr. Andreu with a subsidiary in Spain." [60] This would be better than using *ACQUIRE-1*, which requires the extra *INSTRUMENT-1* plan and *CORPORATION-1*, which requires an extra plan for the *ownership* link. *ACQUIRE-3* contains unwanted semantics (a *price* relation) and would thus be penalized.

There are two problems with PLAN. First, it cannot be guaranteed to be sound. Preconditions satisfied at a higher level of processing can be undone by side effects at lower levels. This is the problem that Young and Moore tackled with DPOCL. Second, PLAN is horribly inefficient. Local solutions are planned again and again as backtracking moves up and down the input semantic tree. Preconditions and constraints must be continually rechecked because each combination of sub-plans may be different. This is the problem tackled by Tenenberg with abstractions. PICARD identifies local areas of dependency and plans them separately. It uses constraint satisfaction techniques to ensure soundness. It recasts the means-end planning paradigm into an abstract system of independent sub-plans connected by usage constraints, so that efficient solution synthesis procedures can combine them. It is this last concept that is explored in the remainder of this section.

It must be noted that text planning for machine translation is somewhat easier than for many Natural Language text planning problems, primarily because the semantic content is given. The main goals of an MT text planner are lexical choice and word and sentence ordering. In general, communicative goals are inherent in the input semantic content, although pragmatic features must be taken into account to a greater or lesser degree. In addition, appropriate generation of discourse structure, figurative language, anaphora and ellipsis serve to complicate matters. Text planners for question-answering systems have the added complexity of starting from communicative goals. This makes for more complex planning; nevertheless, the PICARD principles to be explained below can be applied in exactly

[60] We emphasize that this is the best **computationally**, given the lexicon and semantic inputs in this simplified example.

the same manner.

6.2. Using Constraint Satisfaction to Enable Abstractions

It would be useful if we could divide text planning problems into relatively independent sub-problems and use HUNTER-GATHERER's solution synthesis to efficiently combine the smaller solutions. The problem is that solution synthesis requires an unchanging, orderly set of variables to start with. In the introduction to Solution Synthesis, Figure 3 shows 4 variables, A, B, C and D. Each one of these variables has a set of possible solutions. Three second order nodes are created, AB, BC and CD. From these, the ABC and BCD third order nodes are created and, finally, the answer, ABCD, is synthesized.

In text planning, as in all types of means-end planning systems, there is no fixed number of variables. "Variable," in this context, refers to a set of possible plans from which one **must** be chosen. A variable can be set up for **ACQUIRE**, which has three possible plans. One of them must be chosen. On the other hand, sometimes a plan for *instrument* is needed, and sometimes not. For instance, if *ACQUIRE-1* (Figure 38) is used, a separate sub-plan must be made for the *instrument* relation. Two "variables" would be needed, one for **ACQUIRE** and one for *instrument*. If the *ACQUIRE-2* is used, the *instrument* plan and variable are unnecessary. Lexicon entries which have different set of VARS, different preconditions and/or contain more or fewer relations all create differing amounts of sub-plans. These differences are compounded as different paths through the space of possible plans are taken.

PICARD solves this problem in a simple way. Means-end planning is carried out **locally** to determine, for each lexicon entry, the additional sub-plans that are needed. Again, these sub-plans correspond to VARs and missing relations or preconditions in the lexicon entry. For instance, the *ACQUIRE-1* entry requires a sub-plan for the missing *instrument* relation. For each needed sub-plan, a "usage constraint" is added to the lexicon entry that will "request" some "non-dummy"[61] sub-plan to be used that fulfills the need. The *ACQUIRE-1* entry, for example, would receive a usage constraint that requires it to use one of the sub-plans for *instrument*. In addition, for each of the sub-plans that can fill the need, a usage constraint is added such that those entries can only be used if "requested" by some other plan.

[61] "Dummy" plans are explained next.

For each semantic concept and relation that **is** included in the lexicon entry, a dummy sub-plan is created. For instance, in the *ACQUIRE-2* entry, a dummy *instrument* sub-plan is created and added to the list of other *instrument* sub-plans. The *ACQUIRE-2* entry then receives a usage constraint that "requests" the use of the dummy sub-plan. The dummy sub-plan receives a usage constraint that it be used only if "requested." The fact that ACQUIRE-2 does not "request" one of the non-dummy *instrument* plans will prevent them from being used.

The main benefit this gives is that a stable set of "variables" can be created. There will be an **ACQUIRE** variable, from which one of the three lexicon entries must be selected. There will be an *instrument* variable, from which either the entry shown in Figure 39 will be used or the newly created dummy entry. These variables can then be processed by a solution synthesis algorithm. Whenever a choice is made, for instance selecting *ACQUIRE-1* for **AQUIRE**, the constraint satisfaction mechanism in HUNTER-GATHERER will eliminate all conflicting sub-plans. Picking entry *ACQUIRE-1* will eliminate the dummy entry for *instrument*. Choosing entry *ACQUIRE-2* will eliminate all of the non-dummy *instrument* plans, as well as all the sub-plans that are created by the *instrument* plans. In this way, local means-end plans can be linked together, but can be processed globally by an efficient solution synthesis control. Figure 41 graphically displays the usage constraints for a portion of the example problem. The dotted lines connecting plans indicate compatible usage constraints.

Usage constraints are implemented by adding a series of preconditions and effects to each lexicon entry. For instance, for *ACQUIRE-1* to "request" that an *instrument* slot be filled, the following effect is added to it:

EFFECT: (FILL (ACQUIRE *instrument*))

Each of the non-dummy plans for *instrument* - only one in this case - then receive a precondition:

PRECONDITION: (FILL (ACQUIRE *instrument*))

This precondition cannot be fulfilled unless another plan with the corresponding effect is used.

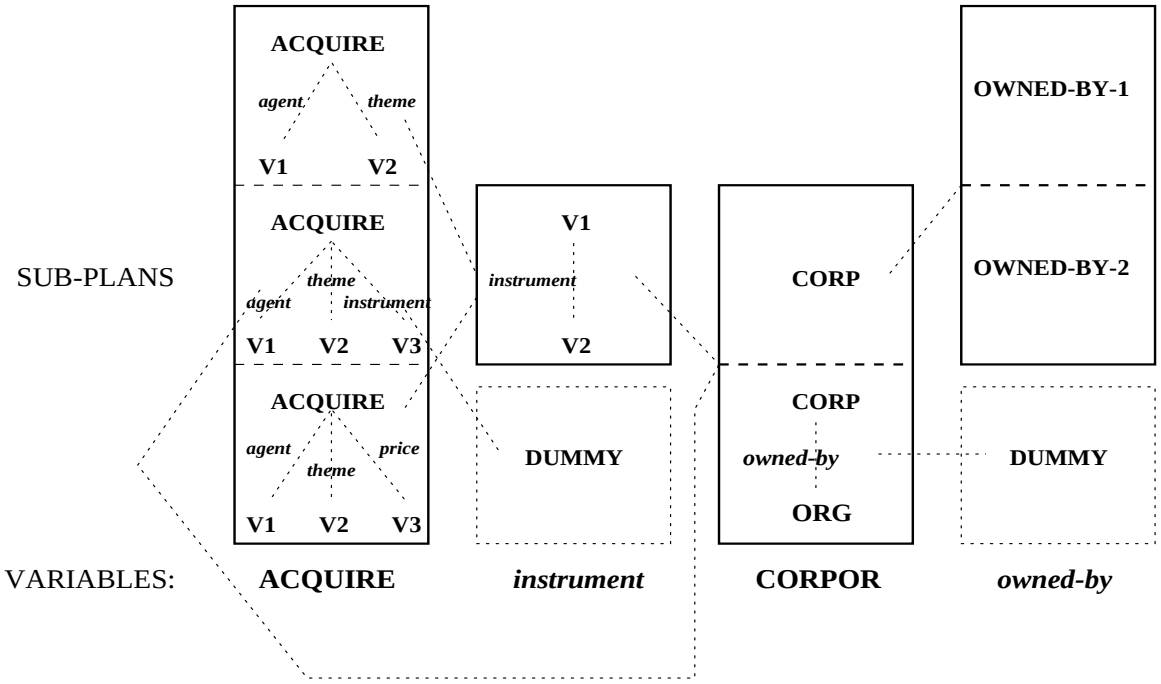


Figure 41: Usage Constraints Indicate Locally Optimal Combinations

To request a dummy filler, an effect like this is added:

EFFECT: (FILL (ACQUIRE *instrument-dummy*))

The dummy *instrument* then is given this precondition:

PRECONDITION: (FILL (ACQUIRE *instrument-dummy*))

Each concept (like **ACQUIRE**) and relation (like *instrument*) is linked to actual, uniquely-named structures in the input semantic representation. For example, (FILL (ACQUIRE *instrument*)) would actually look like (FILL (ACQUIRE-21 *instrument-23*)). This prevents confusion when more than one of the same concepts or relations is present.

Effects and preconditions can also be added to prevent redundant planning. For instance, if there was a **CORPORATION** lexicon entry that had a "built in" *instrument* link, combining this with *ACQUIRE-2* would over-generate the *instrument* meaning. Constraints can be added to ensure no input relation or concept is used twice.

To summarize, a means-end planner is used **locally** to set up possible sub-plans. The sub-plans are connected with a system of usage constraints that inhibit or allow usage depending on the other sub-plans being used. The HUNTER-GATHERER system can then efficiently process the collection of sub-plans to find the best overall plan. Constraint satisfaction techniques automatically control the combination of sub-plans. Constraint satisfaction also ensures the soundness of **all** preconditions used in the lexicon entries, including those which are not related to the ideas presented above. Efficiency is gained by restricting the means-end planning component to local sub-problems. Solutions to these sub-problems are then combined, utilizing solution synthesis, branch-and-bound and constraint satisfaction, by HUNTER-GATHERER.

Generation in the Mikrokosmos project is a relatively new development. Currently we are developing methods to reverse multilingual analysis lexicons (Viegas & Beale, 1996). PICARD has been used to back-translate the semantic analyses of the Mikrokosmos analyzer using these reversed lexicons. Efficiency results similar to those reported for HUNTER-GATHERER above were obtained.

The HUNTER-GATHERER algorithms are complete with respect to the set of monotonic solutions. Currently, solutions with plans that temporarily violate preconditions of other plans (with the "violation" corrected by a later plan) will not be allowed.[62] Other than this limitation, HUNTER-GATHERER is guaranteed to find the same solution(s) as an exhaustive search. In addition, the constraint satisfaction component of HUNTER-GATHERER ensures soundness. By converting means-end planners into a format that can be used by HUNTER-GATHER, PICARD achieves efficient processing with guaranteed soundness and completeness without sacrificing the generality of means-end planning.

[62] Problems in semantic analysis and generation do not require such plans. Other types of planning related to Natural Language, such as planning the content of a reply to a database query, might require them.

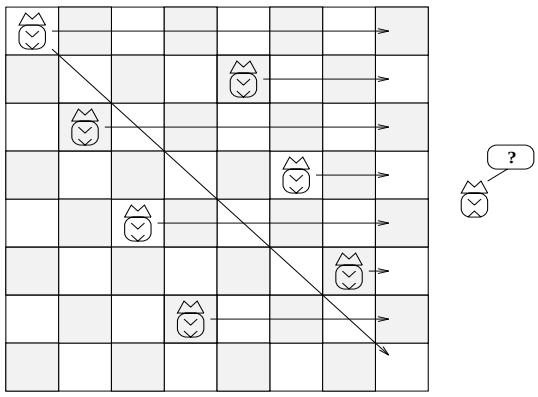


Figure 42: The 8-Queens Problem

7. Natural Language - a "Natural" CSP

There are two kinds of problems in the AI world:[63] the naughty and the nice. The naughty problems are those where everything depends on everything else. The N-Queens problem in Figure 42 is a good example. Even if some poor unfortunate is able to place N-1 Queens legally,[64] placement of the N^{th} Queen can be foiled by any of the N-1 already placed. These naughty problems typically are what people have in mind when they think about CSPs. Massive amounts of constraints going every which way. Look at just about any CSP reference (for example, (Tsang, 1993)) and you will see a preponderance of problems for which there will be no presents under the tree.

Fortunately (or perhaps unfortunately - see below), real world problems are not naughty, but nice. Often a real world problem gains its complexity not from dense interconnections of constraints, but simply from the large number of possible solutions. Natural Language is a perfect example of the latter. Given a sentence of length 23 (an average-looking sentence), and assuming each word can have two meanings, this yields 2^{23} possible combinations of word senses, or over eight million.

As strange as it may seem, though, in the CSP world "naughty" problems are actually "nice," and "nice" problems can be extremely "naughty." Figure 43 reproduces a table from

[63] As in Santa's.

[64] The object of N-Queens is to place N Queens on an N X N chessboard such that no Queen attacks any other Queen.

Solutions Required	Tightness of the Problem	
	Loosely Constrained	Tightly Constrained
Single Solution Required	Problem is easy by nature; brute force search (e.g. simple backtracking) would be sufficient	Problem reduction (i.e. CSP) helps to prune off search space, hence could be used to improve search efficiency
All Solutions Required	When the search space is large, the problem is hard by nature.	Problem reduction helps to prune off search space; solution synthesis has greater potential in these problems than in loosely constrained.

Figure 43: CSP Problem Types

(Tsang, 1993). "Naughty" problems are those on the right-hand side, tightly constrained. For a human, keeping track of a large number of interacting constraints makes the problem difficult, that is why we tend to think of them as "naughty." For a computer, though, the constraints actually help, using CSP, to make the problem easier. The loosely constrained problems on the left side for which all solutions are required from among a large set of possible solutions are described in the table as "hard by nature."

The first major point of this chapter is to show that computational semantic problems are, in fact "naughty," and thus are "nice" in the CSP world. More precisely, we will demonstrate that computational semantic problems **are** tightly constrained **locally**, and CSP techniques can be used to great advantage in identifying these local interactions and determining their solutions. Perhaps even more important, we can use the fact that certain parts of the problem are **not** constrained[65] to guide solution synthesis most efficiently.

Another aspect of computational semantics that can cause problems with regard to applying CSP techniques is the fact that the constraints often do not have "yes" or "no" answers. CSP relies on definite answers to prune away inconsistent solutions. In natural language semantics, however, nothing is ever straightforward. Is the "White House" a HUMAN? No, it is a building, and yet we can say "The White House said today that ..." without even thinking about calling the Ghost Busters. Does this eliminate computational semantics from the domain of CSP problems? In one sense, it does. A straightforward application of CSP consistency algorithms would yield little, since constraints in computational semantics are only

[65] Determining "unconstrainedness" is a peculiar, but potentially powerful outcome of constraint analysis.

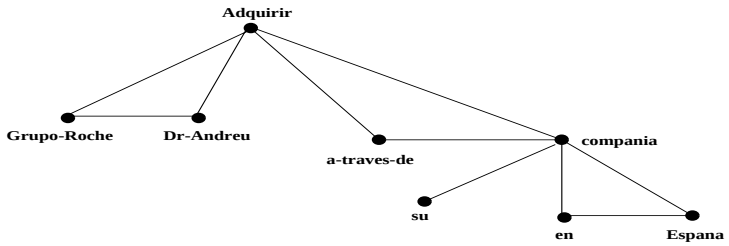


Figure 44: Constraint Dependencies in Sample Sentence

tendencies; metonymy and figurative language often override these tendencies. However, branch-and-bound techniques combined with solution synthesis can be used to efficiently drive a computational semantic search engine. The efficiency, though, as shown above, is derived from information gained by analyzing constraint dependencies. This is the second major point of this chapter: that the branch-and-bound / solution synthesis paradigm is a type of constraint analysis problem, with its efficiency coming from analysis of dependencies within the problem.

Finally, we demonstrated that one of the central tools in Natural Language Generation, the means-end planner, can be more efficiently implemented by framing the planning process in terms of a CSP, and then using straightforward CSP consistency algorithms to determine possible solutions. This is the third, and last, major point. Thus, the computational semantic problem itself, as well as one of the major tools used in solving it, can best be seen as being a type of CSP. It's a "natural" fit.

7.1. Local Dependency in Computational Semantics

It is evident from a quick look at Figure 28, repeated here as Figure 44, that constraints in that sentence are locally bundled. *Grupo-Roche* has nothing to do with *Espana*. The meaning of the one can be determined independently of the other. That is not to say that they cannot influence each other. For instance, *Espana* helps the analyzer choose the **location** meaning of *en*, which through a series of other interactions, could possibly influence the choice of meaning for *adquirir*, which, finally, could influence *Grupo-Roche*. However, this chain of dependencies is exactly what constraint-based mechanisms handles. As far as direct dependencies, though, the two words are not linked.

This is the general state of affairs in Natural Language. Government and Binding theory (Haegeman, 1991) is built on the assumption that one part of a text **governs** another, and interactions can only occur under this relationship. Government is restricted, among other things, to nodes that syntactically **command** other nodes. A node commands another node (again, among other things), if it is higher in the syntactic tree, and both are on the same path to the topmost node. This constraining property of syntax excludes non-governing relationships, which, in effect, partitions sentences into independent bundles. Smaller bundles can be combined into larger bundles as one moves higher up in the syntactic tree, where government domains become larger. This fact will be used to great advantage when constructing the subgraphs for solution synthesis, as described above.

It is fairly obvious that, at the sentence level, computational semantics tends to bundle dependencies into these subgraphs. What about larger sections of text? This research claims to be a step toward implementation of practical, large-scale, "real" computational semantic systems. Such systems eventually will address discourse issues. Can the claim that dependencies are locally bundled be maintained?

Yes, and no. (Grosz and Sidner, 1986) identify three aspects of discourse: the linguistic structure, the intentional structure and the attentional state. The first two seek to identify segments of text and give their purpose. In function, they are very similar to Rhetorical Structures (Mann and Thompson, 1986). The attentional state, on the other hand, is an abstraction of the participants' focus of attention. This can be either global, or local (Grosz, 1981). Knowledge of the attentional state is needed in reference resolution (and generation).

The rhetorical structure of a text links together chunks of text and identifies the function of the composition. In Rhetorical Structure Theory (RST), a nuclear section of text is joined to a satellite. Constraints between the nucleus and satellite are typically constraints between the main events of the main clauses of the sub-texts involved. For a constraint-based analyzer/planner, this simply adds an extra constraint link between the two sub-texts. In practice, this may cause final decisions at the sentence level to be delayed[66] until later in the discourse. This, of course, is a desirable situation. Often decisions cannot be made until the global purpose of a text or sub-text has been determined. In fact, this discourse oriented processing is a main driving force behind a constraint-based approach. If a 23 word

[66]Unless all other decisions can be ruled out locally using the techniques described above.

sentence produces millions of combinations, how many combinations would a thousand word text produce? Constraints must be used to intelligently prune the space of possibilities to a minimum, limiting interaction between sentences only to the bare minimum.

Attempts at processing the attentional state have concentrated on local focus. "Centering" theory (Grosz, et al. 1986) is an attempt to constrain reference resolution to the immediate context. Such efforts have proven to be effective for many texts; however, it is recognized that local focus alone cannot solve all reference problems.

(Elhadad, 1990) claims that conversations are locally constrained. He uses a constraint-based[67] paradigm to generate turns in a conversation. Each turn is linked to the previous turn by five types of local constraints. He argues that the most important characteristic of dialog is that it is locally managed.

In practice, local focus and local dialog constraints can be tracked independently of the main analyzer/planner. Before each sentence is processed, the focus and dialog constraints can be calculated for that sentence. These constraints can then be added to the local processing of each sentence. Tracking global focus can also be added to this independent mechanism. Thus, these phenomena do not pose a problem for CSP techniques.

On the other hand, certain aspects of Text Generation such as planning sentence length are heavily influenced by global considerations. What has come before and what comes next, the complexity of the preceding text, the surface length of the realizations of sub-parts of the current sentence, as well as global considerations of style; all these impact on sentence boundary decisions. Some of these factors can be tracked independently, similar to focus and local dialog constraints. The complexity and surface length of the current text, and of the text yet to be processed, however, are difficult to measure until the surface forms have been generated. For instance, a precondition such as "Sentence length < 25" cannot be satisfied by a single effect, but only from the combination of many effects. Effects could be created that increment a global variable, which is then referenced by the precondition; however, this creates a situation where one sub-plan is constrained by every other sub-plan, which destroys the computational efficiency of HUNTER-GATHERER (see the "Classes of Problems" section below.) In these cases, constraint-based planners hold little advantage.

[67] Elhadad uses functional unification to enforce constraint satisfaction. Although this certainly works, it has none of the efficiency advantages presented here.

The best solution to this problem probably lies in a post-processor that can examine the output text and suggest revisions based on measures of global surface features (Robin, 1994; Inui et al., 1992). Another possible solution would be to eliminate decisions based on surface features, such as the number of words planned, and replace these constraints with semantically based ones, such as the number of concepts. Or, perhaps, with a little more thought, using the global variable approach described above might be implemented in a way that does not impact efficiency. We leave these matters of handling surface constraints for future research.

7.2. Classes of Problems for which HUNTER-GATHERER is Beneficial

Several different features must come together to produce a problem for which HUNTER-GATHERER is the preferred processing methodology:

- Constraint-based. HUNTER-GATHERER gains its efficiency by identifying and processing independently parts of the problem that are tightly constrained. A knowledge of constraints between various parts of the problem is therefore essential.

- Constraints are tendencies or preferences. Traditional CSP techniques can be used for problems whose constraints are of the straightforward yes/no variety (although HG is still more efficient - see section 8.1). The N-Queens problem is a good example of such a problem. Many "real-world" problems, however, do not have such a simple semantics. Context often decides whether a certain decision is preferred (or even relevant) or not. The branch-and-bound methods used by HUNTER-GATHERER can be viewed as constraint satisfaction for "fuzzy" constraints. Constraints are combined in context, with value assignment combinations that are guaranteed to be sub-optimal removed. Solution synthesis provides for efficient combination of partial solutions, while knowledge of constraint dependencies recorded in the subgraphs guides the synthesis.

- Relatively independent subgraphs. Imagine a new N-Queens' problem where the constraints were "fuzzy." For instance, consider setting up constraints such that queens in odd-numbered columns attacking queens in other odd-numbered columns was relatively bad (possibly a score of 0.2), odd-numbered attacking even-numbered was not quite so bad (0.5), even-numbered combatants were not bad at all (a score of 0.9),

and queens not attacking any other queens (pacifists) were given a score of 0.1. This violent version of N-Queens could then have the goal of finding the highest scored placement of queens. Obviously, straight-forward CSP techniques would be useless in this problem. HUNTER-GATHERER, although able to find the solution, cannot take advantage of its solution synthesis mechanism. A simple branch-and-bound algorithm would be enough to handle this, but it might take several centuries to process a 20-Queens' problem.

The key reason the new N-Queens' problem is so difficult is because it is impossible to divide it into relatively independent sub-problems. HUNTER-GATHERER gains its efficiency in its ability, at each level of synthesis, to find at least one variable that is not effected outside the current synthesis subgraph. This variable can be optimized in all the possible combinations of variables that are effected outside the subgraph. In the new N-Queens' problem, every variable affects every other variable, making it impossible to perform this optimization.

In terms of constraint graphs, the prototypical form of suitable problems will be tree-shaped. Subgraphs are formed from the leaves up, by combining all children with their parent into a subgraph. At each synthesis, then, all of the children nodes will be optimized, since only the parent node is effected outside the subgraph.

The prototypical form of unsuitable constraint graphs is the clique. In a clique, each variable affects every other variable. Subgraphs cannot be constructed in this situation which will enable optimization at any level of synthesis.

Of course, there is a wide spectrum of problems in between these two extremes. The next section briefly reports on our work applying HG to graph coloring problems. These types of problems are halfway between the mostly tree-shaped inputs found in computational semantics and the clique of the N-Queens problem. Section 9.4 goes even further, examining how the input topology of a problem affects HG's partitioning and, consequently, its overall complexity.

In summary, constraint satisfaction finds the subgraphs, branch-and-bound optimizes "fuzzy" constraints, and solution synthesis combines together partial solutions. These functions, in turn, depend on, or take advantage of, the availability of constraints, the "fuzzy" nature of the constraints, and the localized nature of the interactions.

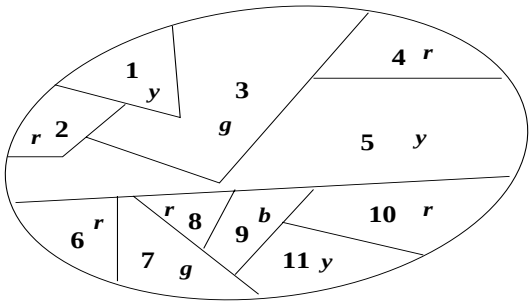


Figure 45: Graph Coloring Example

8. Other Applications for the Hunter-Gatherer Technology

8.1. 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). Quite a bit of research has been extended in this area, including the well-known theorem that every planar graph can be 4-colored (Appel & Haken, 1976). Finally, it is quite easy to make up a large range of problems, from simple to complex, single dimensional to multi-dimensional.

Figure 45 is a simple variant (with solution) of the first graph coloring problem in the introduction: 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 46 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 covered by 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 47 shows steps 1, 2 and 3 for subgraph A.

The most important column in Figure 46 is the input complexity. The largest value in this column, 5 for subgraph E, defines the complexity for the whole problem. Because there

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 46: HG Subgraph Processing

is only one subgraph with the maximum input complexity, this problem will be solved in time proportional to $1 * 4^5$, 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 must be noted again that the partitioning algorithm presented earlier is not guaranteed to give the optimal set of subgraphs. It is designed to give near-optimal results quickly (near optimal with respect to the eventual complexity of SS-HG. SS-HG is guaranteed to give the optimal result no matter what the subgraphs are, as long as they legally combine to cover the whole input problem). In this example, if subgraph E had added only vertices 8 and 9, it would have had input complexity 4 and output complexity 3. We could have then created an intermediate subgraph to which vertex 11 was added to E, with an input complexity of 4 and an output complexity of 3, from which the final subgraph could be created by adding vertex 10, with an input complexity of 4. In that case, the maximum input complexity would be 4. Since there would be 5 subgraphs with that complexity, the total complexity would be to $5 * 4^4$, which happens to be greater than the complexity of the solution actually used. The fact that it turns out like that in this case is irrelevant; the algorithm for creating the input subgraphs **can** lead to slightly higher complexities than theoretically possible. This tradeoff was chosen to simplify the subgraph creation (which is not the focus of this research), ensuring that the subgraph creation complexity doesn't overwhelm the complexity of SS-HG, yet still providing SS-HG with reasonable inputs.

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 47: Subgraph A Processing

It is interesting to experiment with SS-HG using several different-sized (and different dimensionality - see below) problems and compare the results to other approaches. Figure 48 presents the results of such experiments. Six different problems are solved. The last two, the 100-tree and the 36-square, are included in anticipation of the discussion below on graph topology. 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. In addition to comparisons with the above, this result can be compared to other solution synthesis algorithms to give an idea of the merits of our subgraph inputs to solution synthesis.

	# 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 48: HG Graph Coloring Results

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" 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

	# vertices						
	5	11	27	83	100-tree	36-square	64-cube
HG - CS Combos	272	1232	23824	1394496	2528	287616	DNF
Max Input Complexity	4	5	6	8	3	7	15
Number at Max Input Complexity	1	1	4	12	10	16	7
Predicted Combos	1*(4**4) 256	1*(4**5) 1024	4*(4**6) 16384	12*(4**8) 786432	10*(4**3) 640	16*(4**7) 262144	7*(4**15) 7516192000

Figure 49: Actual vs. Predicted Complexity

effect. Problem complexity soars without branch-and-bound. This confirms that our approach, aimed at optimization rather than constraints, is effective.

HG without branch-and-bound is very similar to previous solution synthesis algorithms, including Tsang. Tsang (as well as Freuder (1978)) require a linear ordering of vertices which are then combined into two-vertex solutions, then three, etc.. HG, on the other hand, uses the subgraphs to guide the synthesis process, an extension to the minimal bandwidth ordering principle used by Tsang. We expected this extension to benefit even without branch-and-bound, and our expectations were confirmed. Tsang's algorithm outperformed HG without branch-and-bound only for the simplest problem, for which an MBO ordering turned out to be a superior way to partition the problem. When the number of vertices was raised even a small amount, the subgraph approach became better. We therefore propose that a subgraph decomposition approach such as used in HG is superior to the linear ordering approach used by Tsang.

We can also compare the actual performance of these algorithms with the predicted behavior. Figure 49 shows the actual number of combinations built by HG **without** constraint satisfaction. Obviously the actual behavior aligns closely with the complexity predicted by max-input-complexity, as described above. The relatively small differences are due almost entirely to subgraphs with input complexity one less than the maximum input complexity. For example, the 83 vertex problem had 34 subgraphs with input complexity of 7. If added to the combinations for only subgraphs of input complexity 8, the total is within 3% of the actual total. Programs are available that can give the exact number of combinations that

will be required (without constraint satisfaction) for any problem.

We have added a seventh problem to the chart in Figure 49 - 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, as discussed in section 9.4. This can lead us to 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.

9. Discussion

In this section we try to sum up some of the important characteristics of the Hunter-Gatherer control architecture. First we set out the novel contributions HG brings to the state of the art. Next, we consider some of the formal properties of HG, namely the soundness and completeness of its planning capabilities. The third section discusses the notion of island processing, and how HG implements it. We conclude by discussing graph topology, and how it can be used to explain how and why (and when) HG works.

9.1. Novel Contributions to the State of the Art

In summary, this research provides the machinery for a fast, complete and sound search through many types of constraint problems, including natural language semantics. Previously in computational semantics work, researchers needed to make one or more of three simplifications:

1. Reduce the size of the inputs and/or knowledge base (i.e. one-to-one mappings).
2. Use (possibly faulty) heuristics to speed up search.
3. Make (possibly faulty) assumptions about the constraint interactions in the search space.

Each of these simplifications often leads to non-optimal solutions. In addition, generalized problems such as graph-coloring do not allow, or severely limit, such simplifications. HUNTER-GATHERER removes the need for them by providing a method for organizing and processing search using minimally interacting sub-problems.

This research advances three central theoretical contributions. First is the recognition that solution synthesis methods should not be limited to combining, at the lowest level, pairs of variables, as in Freuder and Tsang, nor need it be content to blindly synthesize combinations of these pairs. HUNTER-GATHERER organizes the search space into maximally independent subgroups of any size, and then guides the synthesis process as it combines results from these subgroups. A related contribution is the process by which HUNTER-GATHERER produces these input subgraphs and synthesis plans in non-exponential time.

The second major theoretical contribution is the use of branch-and-bound optimization techniques to prune the results of synthesis. Previously, only constraint satisfaction techniques were used in this role. The use of branch-and-bound techniques was precipitated by the fact that natural language semantics often does not allow straightforward yes-or-no constraints of the type needed by previous solution synthesis methods. It turns out that, even in more general problems such as graph coloring in which yes-or-no constraints can be used, branch-and-bound pruning significantly outperforms constraint-based pruning. HUNTER-GATHERER, in fact, uses both.

The final major theoretical contribution combines aspects from the first two. Tsang suggests using a "minimal bandwidth ordering" (MBO) algorithm to order the input variables in such a way as to maximize the benefits of early constraint pruning. HUNTER-GATHERER extends the concept of MBO to its branch-and-bound optimization. Input subgroups are constructed in such a way as to minimize the interaction outside of the subgraphs, which in turn allow maximal pruning based on branch-and-bound techniques.

A peripheral contribution of this work is the recognition that solution synthesis techniques could be applied to these types of problems. Previously, solution synthesis was used exclusively for constraint satisfaction problems. By substituting branch-and-bound techniques for constraint satisfaction, a whole new set of problems can utilize solution synthesis techniques.

9.2. Formal Properties of HUNTER-GATHER: Soundness and Completeness

The PICARD Natural Language Text Planner is an example of a planning system that utilizes HG. In a sense, though, HG is, at its core, a planner. Even for applications such as semantic analysis, HG can be seen as a processor planning the best combination of word senses to describe the input text. When discussing planners, two important concepts arise: soundness and completeness. The following sections discuss these important topics.

9.2.1. Soundness

(Chapman 87) discusses precondition "clobbering", the state where a rule that had been instantiated previously on the basis of some precondition, later had that precondition removed, invalidating the rule. Many previous text planners were not "sound", in that they

did not detect this kind of situation. To prove more formally that this system is sound, let me state the soundness criterion which it claims to adhere to:

Soundness Criterion: At all times, every rule that is not marked as currently failed must, for each of its preconditions, have at least one active, non-failed plan that has an effect that satisfies the precondition if it is a positive constraint (i.e. some state **must** exist for the precondition to be satisfied, identified by a "check-con" in the precondition field), or have no plans active that have an effect equal to the precondition if it is a negative constraint (i.e. the state may **not** exist, identified by a "check-not-con").

Thus, if plan 1 has a precondition that states that constraint A must be true, then whenever plan 1 is not marked failed, there must be at least one other active plan that has an effect that produces A. On the other hand, if plan 1 has a precondition that states that precondition B must not be true, then there can be no plans active that produce B while rule 1 is not failed.

Note that this is slightly stronger than a general soundness condition, which might allow a planner to go through unsound states as long as the final result is sound. These "non-monotonic" plans, though valid, are currently excluded by HUNTER-GATHERER. We do not feel that this is a big drawback, since, in practice, a planner that is guaranteed to be sound at the end most likely will be sound throughout. This is all the more true for computational semantic planners which do not have complex preconditions and effects.

This system meets the soundness criterion given above as follows. After some basic precondition application, which removes any rules that do not meet unary constraints (for example, "only apply this rule in formal contexts"), the dependencies between all the remaining rules are analyzed. The following information is recorded:

1. for each precondition of each plan, the plans are recorded that, if instantiated, would produce constraints that conflict with the precondition [68]

2. the inverse of the above: for each plan, if it were instantiated, record all the preconditions that conflict with it.

[68] A constraint A is considered conflicting when the precondition requires that constraint A **not** be set. Also judged as conflicting is the plan that sets constraint A when the precondition requires constraint -A

3. for each precondition that requires a constraint A, record all of the plans that, if instantiated, would set constraint A.

4. the inverse of the above: for every plan, if it were instantiated, record all the preconditions that would be satisfied by it.

Before solution synthesis is initiated, every plan that has a precondition for which there were no plans found in number 3 above is failed. Every time a plan is failed, both during this initial process and during the subsequent syntheses, all of the preconditions it can satisfy (from number 4 above) are retrieved. For each of these preconditions, all of the plans that could satisfy the precondition (from number 3 above) are retrieved, and it is checked that at least one of these plans is still active. If none are, then the plan corresponding to the precondition is failed. This process ensures that no plans are ever deleted that are the sole suppliers of preconditions of other active plans (or more accurately, that any such active plans are failed if their sole supplier is failed).

This does not yet fail preconditions that conflict with constraints. Whenever a variable has only one valid plan left[69], all of the preconditions that conflict with the plan (from number 2 above) are retrieved and the corresponding plan(s) are failed. The plans affected by these failures are then checked as described above.

During the solution synthesis, valid combinations of plans are chosen for each variable in the synthesis. In a combination, each variable has one plan chosen and the others are failed. The effects of failing the other plans are checked, and the effects of instantiating the plan that is used are checked. In general, whenever a plan is failed, the plans for which it supplied preconditions are checked. Whenever an island is created, the plans that conflict with the island are failed.

The fact that the system is sound is good in itself. But this is not where the benefits end. The whole process of ensuring soundness as described above, combined with island processing (see below), creates a very efficient text planner. All conflicting rules and impossible rules are eliminated at the earliest point possible. Subgraphs are created for the solution synthesizer

[69] This can happen because there was only one plan to start with, or because the soundness procedure above eliminated one or more plans in a node, or the process being described here eliminated one or more plans in a node, or a combination of the last two processes left only one valid plan, or, finally, the solution synthesis mechanism chose one plan and "failed" all the others.

that will maximize this effect. A plan is not ever failed, and then, after further processing, a different plan found to be invalid as a result. As soon as a plan is failed or an island created, all the other plans that can no longer be valid are removed immediately. Removing these plans then may lead to further removals by the same process. This process feeds on itself to remove as many of the possible plans as possible.

9.2.2. Completeness

HUNTER-GATHERER is complete[70] because it guarantees the same results as an exhaustive planner. The solution synthesizer, at every step, exhaustively calculates all of the valid combinations of plans for that synthesis. The only combinations that are removed are:

1. Combinations that can be guaranteed, by branch-and-bound techniques, to produce complete solutions that are not optimal.
2. Combinations that contain constraint conflicts.

Exhaustive planners are obviously complete, so HUNTER-GATHERER must be as well.

Two exceptions exist, however. The first has to do with the types of constraints allowed by HUNTER-GATHERER. One of the drawbacks inherent in a text generator like Penman (Mann, 1983) is the inability of the modules to communicate with each other. One of the main characteristics driving this project is the inter-action of choices available at different levels. HUNTER-GATHERER specifically allows for these inter-actions. There is, however, a class of constraints that this system cannot at present address. It is not able to use constraints that arise from the combination of plan effects. Thus a constraint such as "if the clause is already 30 words long, make a sentence boundary" cannot be used. This limitation exists because simple precondition-effect pairs cannot be set up. A precondition like "sentence 30 words long?" is satisfied by combining the effects of many plans together. Section 7 addressed this problem and suggested a number of possible alternative methods for handling it.

The second limitation with regards to completeness has already been mentioned. Non-monotonic plans are not allowed. Non-monotonic plans are those which at intermediate

[70] Except for the non-monotonic plans and surface constraint types described below.

stages contain constraint conflicts which are later resolved in the overall plan. While we recognize that this may be a severe limitation in generalized AI planning, we do not feel it presents much of a problem for computational semantics, where preconditions and effects are generally fairly simple. As we look into using HUNTER-GATHERER in other applications, we plan on investigating this limitation further.

9.3. Planning and Island Processing

One of the central characteristics of almost any natural, complex problem is that parts of its solution are fixed by available resources, while other parts may have a wide variability in possible solutions. For instance, in planning a cross-country trip, a traveler might have some general goals such as "visit as many places as possible", "enjoy the vacation", etc. There may be some specific goals such as "see the Grand Canyon". Unfortunately, there will be some general constraints as well: "spend less than 1000 dollars", "get back home in two weeks". There probably will be some very specific constraints as well. For example, "go to the meeting in Phoenix on Monday, 4-13, at 10 AM (so you can write-off the vacation)" and "visit Aunt Millie on her birthday". Some other constraints are fairly restrictive when combined with other constraints: "visit Cousin Fred while I am in Phoenix".

When planning, the smart traveler will first determine the areas where he has no choices. The other decisions will then be made in relation to the fixed **islands of certainty** in the schedule. Upon making an initial assessment of the island constraints, other islands may appear, such as "visit Fred while in Phoenix" in the context of "go to the meeting in Phoenix on 4-13...". Islands constraints can pop up at even with the most general of constraints. For instance, if the piggy bank is empty after the Grand Canyon, there is only one place to go.

Considering that island driving is such a central component of human planning, it is surprising that it receives so little emphasis in the planning literature. This project seeks to remedy that situation with respect to computational semantic systems. HUNTER-GATHERER automatically uses its constraint satisfaction engine to take advantage of islands. Whenever a variable's plan, or value, is failed for any reason, constraint satisfaction will fail any other plans that critically[71] depend on effects from the failed plan. If a variable has all of its

[71]No other non-failed plans exist that could also satisfy its constraints.

possible plans failed except one, forming an island, then only plans that do not critically rely on those failed plans will remain. In addition, when the non-failed plan that forms the island is processed, all plans that conflict with it will also be failed. Thus, by dynamically implementing constraint satisfaction techniques, islands are automatically identified and their effects propagated.

The second aspect to island driving in this project concerns the artificial creation of islands by the solution synthesis processor. In the solution synthesizer, valid combinations of plans are created and tested. For each combination, one of the plans, or values, is chosen and instantiated for each variable in the combination. This, in effect, creates artificial islands at each of these variables. The "island effects" can then be propagated out for each. Again, this might cause other plans to fail, creating other islands that are also artificial, in the sense that they were created by a combination of constraints imposed by planning choices rather than by the problem itself.

An interesting side-point in this discussion is that we treat text generation and semantic analysis equivalently; that is, they are both instances of planning. The "variables" in analysis are words, the "plans" are word senses. The analyzer then tries to plan the combination of word senses that best describes the semantics of the input. In generation, the "variables" are semantic concepts or relations that need to be realized, the "plans" are textual directions for implementing those "variables," and the generator plans the combination of textual directions that best implements the input semantics. With the exception of the "usage constraints" introduced by PICARD which enable HUNTER-GATHERER's solution synthesis mechanism in the slightly more "fluid" world of generation, analysis and generation are processed equivalently.

9.4. Exploiting Graph Topology for Optimization Problems

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. The last three columns of Figure 49 gave the complexity information for problems of three different dimensions: a one dimensional tree, a two dimensional square, and a three dimensional cube. The maximum input complexity of these problems in terms of the number of vertices, n, is (at worst):

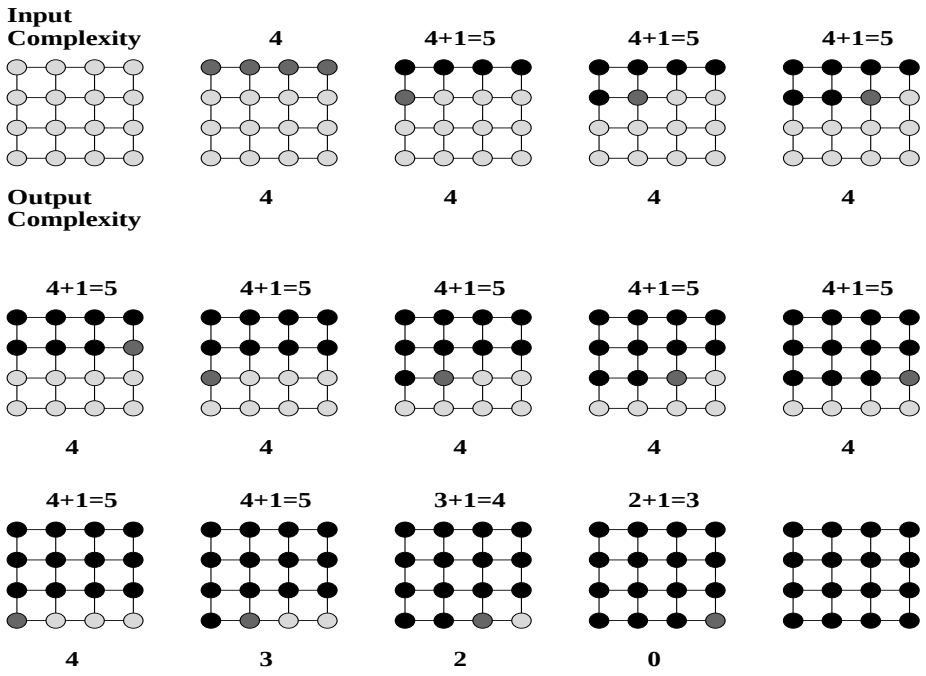


Figure 50: Partitioning a Square.

- a constant for the tree. Any tree can be partitioned with a maximum input complexity of 2 (our simplified subgraph creation algorithm uses 3). A 1000 node binary tree would still have a maximum input complexity of 2. Furthermore, any tree of arbitrary branching factor can be partitioned such that it will have a maximum input complexity of 2.
- approximately $n^{1/2}+1$ for squares.
- approximately $n^{2/3}+1$ for cubes.
- approximately $n^{3/4}+1$ for perfect 4-dimensional problems, etc..

To understand why this is true, refer to Figure 50. This Figure shows a very simple way to partition a square for input to HG (The subgraph creation algorithm above actually does a much better job, reducing the number of subgraphs with maximum input complexity of 5 to a minimum.). The first step is to create a subgraph using all of the vertices in the top row. This has input complexity 4 and output complexity 4. The next subgraph will be a combination of that subgraph and the first vertex in the second row. The input complexity of this subgraph is 5 (the output complexity of the last subgraph plus one vertex), and the

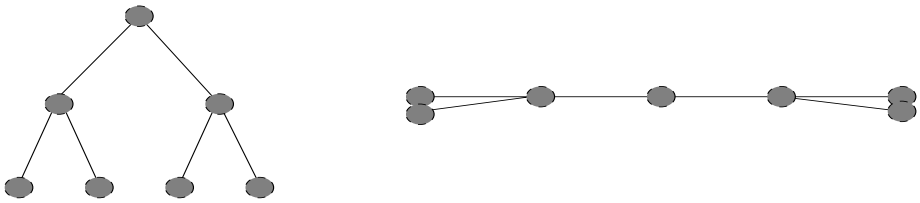


Figure 51: Topology of a Tree.

output complexity is 4, because the top-left-hand vertex no longer has an edge outside this subgraph. Subgraph creation continues, adding one vertex each time while maintaining a maximum input complexity of 5, until the entire graph is covered.

Thus, to solve this problem, HG will require complexity proportional to $O(a^5)$. In essence, the problem was reduced from a 4-by-4 square with exhaustive complexity of 16 to a "line" of length 4 (plus a constant 1). A 100-by-100 square would be reduced to a line of length 10. The input complexity is the square root of the number of vertices. There will be (n-$n^{1/2}$) such subgraphs giving a total HG complexity of (n-$n^{1/2}$)*$a^{n^{1/2}}$ (for this simplified method; our results are better).

Cubes perform similarly. The first step in the decomposition of a 3-by-3-by-3 cube is to make a subgraph of the first 3-by-3 square, giving an input complexity of 9. Then larger subgraphs can be constructed by adding a single vertex. The input complexity of a x-by-x-by-x cube is x-by-x (plus a constant 1), or $n^{2/3}$, since n=x^3. There will be (n-$n^{2/3}$) of these subgraphs (for this simplified method), yielding a total complexity of (n-$n^{2/3}$)*$a^{n^{2/3}}$.

Trees are basically one-dimensional objects, as shown in Figure 51. By starting at the ends, subgraphs can always be created by joining a leaf with its parent, giving an input complexity of 2 and output complexity 1. At each branching point, these subgraphs can be combined one at a time, each time with input complexity 2 and output complexity 1. This continues from the endpoints in until the whole graph is covered. There will be n-1 such subgraphs, giving the HG complexity for any tree to be (n-1)*a^2.

Figure 52 summarizes these results. HG can be seen as a process that "squeezes" down the dimensionality of a problem. Trees are squeezed into a point, squares into a line and cubes into a square.

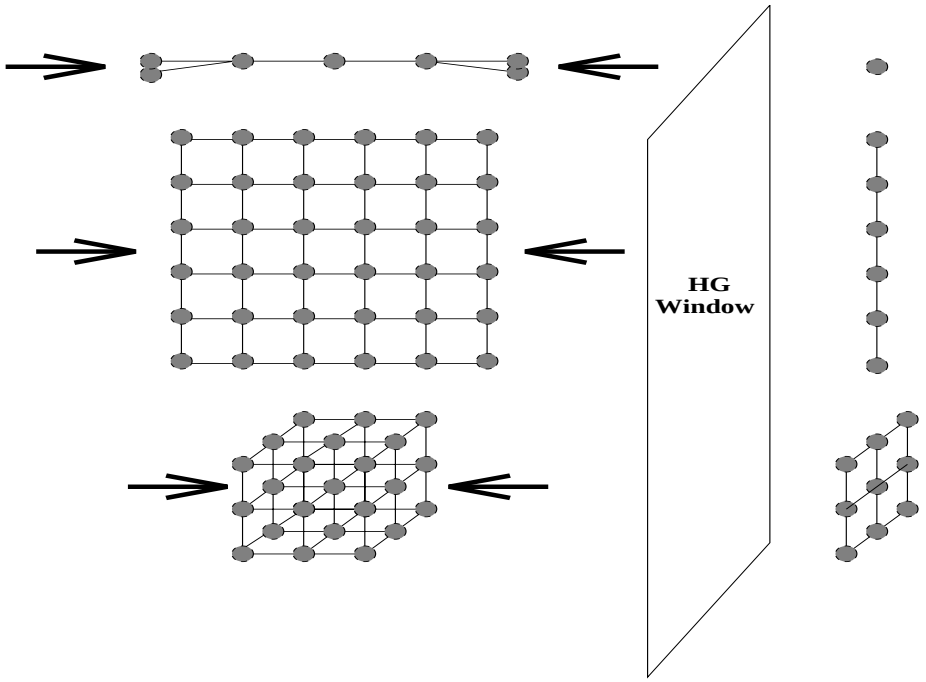


Figure 52: HG Operates on Graph Topology.

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. Figure 53 represents a problem with one, two and three dimensional aspects. Using HG's "window", we can reduce the complexity to zero, one and two dimensions. Unless they are very small, the overall complexity, 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.

This discussion raises the possibility of a new measure of complexity that might be useful.

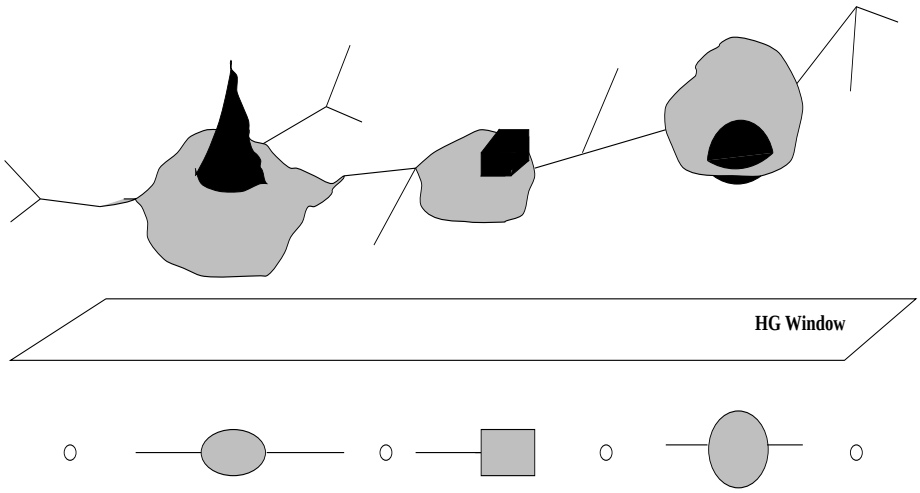


Figure 53: HG's Effect on a Typical Problem.

Theoretically, the exponents 0, 1/2 and 2/3 for the tree, square and cube, up to 1 for a clique, would be good measures. Unfortunately, these are hard to calculate for problems with different dimensionalities. An easy alternative measure would be to use the maximum-input-complexity. This gives a direct indication of how long it will take to solve a given problem. Functions are available that can determine this measure quickly. In addition, the topology of a problem can be analyzed using this complexity measure, with the most difficult subsections identified. Sections with maximum-input-complexity over a given value can be treated with heuristic search methods.

10. Conclusion

We have presented a new control environment for processing computational semantics. By combining and extending the AI techniques known as constraint satisfaction, solution synthesis and branch-and-bound, we have reduced the search space from billions or more to thousands or less. We have argued that the search problems encountered in computational semantics fit nicely into the class of problems that this control paradigm handle well.

In the past, the utility of knowledge-based semantics has been limited, subject to arguments that it only works in "toy" environments. Recent efforts at increasing the size of knowledge bases, however, have created an imbalance with existing control techniques which are unable to handle the explosion of information. We believe that this methodology will enable such work. Furthermore, we believe that it is applicable to a wide variety of real-life problems. Our work in graph coloring is a beginning step towards using this research in other areas.

Acknowledgments

I would like to thank Sergei Nirenburg for his continual encouragement and contributions to this work. He is the most patient man on the face of this earth and has put up with my foibles, along with several batches of ill health, in a most understanding way. I am also indebted to the other members of my committee, Jaime Carbonell, Robert Frederking and Victor Raskin. They have been remarkably flexible with this long-distance dissertation, in addition to offering much encouragement and advice. Also appreciated are Kavi Mahesh and Evelyne Viegas for their work on the Mikrokosmos project as well as their helpful suggestions with regards to this research. And finally, I would also like to acknowledge and thank David Evans and Barbara DiEugenio, who, along with Sergei Nirenburg, were my advisors at Carnegie Mellon where research on an earlier version of this work was done.

References

[1] K. Appel and W. Haken. 1976. Every Planar Map is Four Colorable. *Bulletin American Math Society*, Vol 82, pp. 711-712.

[2] S. Arnborg and A. Proskurowski. 1986. Characterization and Recognition of Partial k-trees. *SIAM Journal on Algebraic and Discrete Mathematics,* 7: 305-314.

[3] Stephen Beale. 1997. Using Branch-and-Bound with Constraint Satisfaction in Optimization Problems. In *Proc. Fourteenth National Conference on Artificial Intelligence (AAAI97)*, Providence, Rhode Island.

[4] Stephen Beale. 1994. Dependency-Directed Text Generation. Technical Report, MCCS-94-272, Computing Research Lab, New Mexico State Univ.

[5] Stephen Beale and Sergei Nirenburg. 1995. Dependency-Directed Text Planning. In *Proc. of the 1995 International Joint Conference on Artificial Intelligence, Workshop on Multilingual Text Generation*, 13-21. Montreal, Quebec.

[6] Stephen Beale, Sergei Nirenburg and Kavi Mahesh. 1996. HUNTER-GATHERER: Three Search Techniques Integrated for Natural Language Semantics. In *Proc. Thirteenth National Conference on Artificial Intelligence (AAAI96)*, Portland, Oregon.

[7] Stephen Beale, Sergei Nirenburg and Kavi Mahesh. 1995. Semantic Analysis in the Mikrokosmos Machine Translation Project. In *Proc. of the 2nd Symposium on Natural Language Processing*, 297-307. Bangkok, Thailand.

[8] U. Bertelé and F. Brioschi. 1972. *Nonserial Dynamic Programming.* Academic Press, New York.

[9] Christian Bessiere and Marie-Odile Cordier. 1993. Arc-Consistency and Arc-Consistency Again. In *Proc. Tenth National Conference on Artificial Intelligence (AAAI93)*, Washington, D.C.

[10] J. Bosák 1990. *Decomposition of Graphs.* Kluwer, Norwell, MA.

[11] B.G. Buchanan and E.H. Shortliffe, eds. 1984. Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley, Reading, MA.

[12] D. Chapman. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32.

[13] E. Charniak, C.K. Riesbeck, D.V McDermott and J.R. Meehan. 1987. *Artificial Intelligence Programming.* Erlbaum, Hillsdale, NJ.

[14] V. Chvátal 1983. *Linear Programming.* W.H. Freeman and Company, New York.

[15] Michael Elhadad. 1990. Constraint-based Text Generation. Technical Report, CUCS-003-90, Dept. of Computer Science, Columbia Univ.

[16] Shimon Even. 1979. *Graph Algorithms.* Computer Science Press, Maryland.

[17] David Farwell, S. Helmreich, W. Jin, M. Casper, J. Hargrave, H. Molina, F. Weng. 1994. PANGLYZER: Spanish Language Analysis System. In *Proc. First Conference of the Association for Machine Translation in the Americas.* Columbia, Maryland.

[18] Mark Fox. 1994. ISIS: A Retrospective. In *Intelligent Scheduling*, M. Zweben and M. Fox, Eds., Morgan Kaufmann Publishers, San Francisco.

[19] Robert Frederking, S. Nirenburg, D. Farwell, S. Helmreich, E. Hovy, K. Knight, S. Beale, C. Domashnev, D. Attardo, D. Grannes and R. Brown. 1994. Integrating Translations from Multiple Sources within the Pangloss Mark III Machine Translation System. In *Proc. First Conference of the Association for Machine Translation in the Americas.* Columbia, Maryland.

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

[21] M. Gondran and M. Minoux. 1984. *Graphs and Algorithms.* Wiley, Chichester.

[22] Barbara J. Grosz and Candace L. Sidner. 1986. Attentions, Intentions, and the Structure of Discourse. *Computational Linguistics* 12(3): 175-204.

[23] Barbara J. Grosz, A. Joshi and S. Weinstein. 1986. Towards a Computational Theory of Discourse Interpretation. Unpublished manuscript.

[24] Liliane Haegeman. 1991. *An Introduction to Government and Binding Theory.* Blackwell Publishers, Oxford, U.K.

[25] K. Inui, T. Tokunaga and H. Tanaka. 1992. Text Revision: a Model and its Implementation. In *Aspects of Automated Natural Language Generation*, R. Dale, E. Hovy, D. Roesner and O. Stock, editors. Springler-Verlag.

[26] E.W. Lawler and D.E. Wood. 1966. Branch-and-Bound Methods: a Survey. *Operations Research* 14: 699-719.

[27] A.K. Mackworth. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8(1): 99-118.

[28] A.K. Mackworth and E.C. Freuder. 1985. The Complexity of Some Polynomial Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence* 25: 65-74.

[29] Kavi Mahesh and Sergei Nirenburg. 1995. A situated ontology for practical NLP. In *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing, International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada.

[30] W.C. Mann. 1983. An Overview of the Penman Text Generation System. In *Proc. of the National Conference on Artificial Intelligence*, pp. 261-265.

[31] W.C. Mann and S.A. Thompson. 1986. Rhetorical Structure Theory: Description and Construction of Text Structures. In *Natural Language Generation: New Results in Artificial Intelligence, Psychology, and Linguistics.*, G. Kempen, Ed., Kluwer Academic Publishers, Boston.

[32] S. Minton, M. Johnston, A. Philips and P. Laird. 1990. Solving Large-scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method. In *Proc. Seventh National Conference on Artificial Intelligence (AAAI90)*, Boston.

[33] R. Mohr and T.C. Henderson. 1986. Arc and Path Consistency Revisited. *Artificial Intelligence* 28: 225-233.

[34] Allen Newell and Herbert Simon. 1972. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, N.J.

[35] S. Nirenburg, V. Raskin and B. Onyshkevych. 1995. Apologiae Ontologiae. In *Proc of the Conference on Theoretical and Methodical Issues in Machine Translation.* Leuven, Belgium.

[36] B. Onyshkevych. (1997). An Ontological-Semantic Framework for Text Analysis. Ph.D. Diss., Program in Language and Information Technologies, School of Computer Science, Carnegie Mellon University.

[37] Boyan Onyshkevych and Sergei Nirenburg. 1994. The Lexicon in the Scheme of KBMT Things. Technical Report MCCS-94-277, Computing Research Lab, New Mexico State University.

[38] Jacques Robin. 1994. Revision-Based Generation of Natural Language Summaries Providing Historical Background. Technical Report CUCS-034-94, Columbia University.

[39] Norman Sadeh. 1994. Micro-opportunistic Scheduling: The Mikro-Boss Factory Scheduler. In *Intelligent Scheduling*, M. Zweben and M. Fox, Eds., Morgan Kaufmann Publishers, San Francisco.

[40] Stephen Smith. 1994. OPIS: A Methodology and Architecture for Reactive Scheduling. In *Intelligent Scheduling*, M. Zweben and M. Fox, Eds., Morgan Kaufmann Publishers, San Francisco.

[41] Josh D. Tenenberg. 1991. Abstraction in Planning. In *Reasoning about Plans.* James A. Allen, Henry A. Kautz, Richard N. Pelavin and Josh D. Tenenberg, eds. Morgan Kaufmann Publishers, San Mateo, Ca.

[42] Edward Tsang. 1993. *Foundations of Constraint Satisfaction.* Academic Press, London.

[43] Edward Tsang and Nigel Foster. 1990. Solution Synthesis in the Constraint Satisfaction Problem. Technical Report, CSM-142, Dept. of Computer Science, Univ. of Essex.

[44] Evelyne Viegas and Stephen Beale. 1996. Multilinguality and Reversibility in Computational Semantic Lexicons. In *Proc. to the 8th International Workshop on Natural Language Generation, Poster Session*, Sussex, UK.

[45] E. Viegas and S. Nirenburg. 1995. The Semantic Recovery of Event Ellipsis: its Computational Treatment. In *Proc. IJCAI-95 Workshop on Context in NLP.* Montreal, Canada.

[46] P.H. Winston. 1984. *Artificial Intelligence* Addison-Wesley, Readig, MA.

[47] R. Michael Young and Johanna D. Moore. 1994. DPOCL: A Principled Approach to Discourse Planning. In *Proceedings of the Seventh International Workshop on Natural Language Generation*, Kennebunkport, ME.