An Incremental Constraint-Based Object Layout Algorithm and Its Implementation

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Gregory M. Oster

Fall 1995

PERMISSION TO USE

ABSTRACT

Visualization is gaining in popularity as a means of monitoring and debugging logic programs. A well-drawn picture can quickly reveal information that would take a significant amount of time and effort to uncover from text-based representations. The goal of the Wand research project at the University of Saskatchewan is to provide a set of graphical debugging tools which are language independent, easily tailorable, and useful for monitoring and debugging logic programming computations.

ICOLA (Incremental Constraint-based Object Layout Algorithm) is a graphics engine for the Wand system. ICOLA positions graphical objects according to object declarations and constraints specifying the relative positions of the objects. ICOLA is capable of drawing pictures from highly under-constrained constraint-based picture descriptions without requiring the specification of absolute object positions.

This thesis presents new capabilities of ICOLA designed and implemented as M.Sc. research work. All language features of ICOLA's original input language specification are now supported. Certain object attributes (properties) may also be specified. A new incremental object layout algorithm is the focal point of this research. This new algorithm supports both incremental insertions and incremental deletions of objects and constraints. As well, it produces consistent picture layouts from underspecified descriptions. The implementation shows the new algorithm to be quite successful. This work is important for not only the enhancements to the capabilities of ICOLA, but also for the completion of a declarative graphics language which provides a simple means of describing complex pictures.

This thesis outlines the purpose and importance of logic program visualization; gives a background on related issues in the areas of visualization, constraint satisfaction, and graphics; shows the usefulness of ICOLA; and presents the enhancements made to ICOLA and the results of those enhancements.

ACKNOWLEDGEMENTS

The completion of this thesis would not have been possible without the support of many people. Thanks must first go to my supervisor, Tony Kusalik, for countless hours of discussions and guidance.

Thanks also to my thesis Advisory Committee members, Mark Keil and Eric Neufeld; and to Ron Bolton, the External Examiner, for their comments on the thesis.

Thanks to my dad and mom, Jim and Wanda Oster, and to my brothers, Doug and Stephen, for moral support and relaxing weekends away from thesis work.

Thanks to Rob Morton for his comments on and testing of the implementation of ICOLA.

Thanks to the Natural Sciences and Engineering Research Council of Canada (NSERC) for providing the funding for this research.

Finally, thanks to the faculty, staff, and students of the Department of Computer Science, University of Saskatchewan, who made my graduate studies here an enjoyable and enriching experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Visualization is the portrayal of information in a graphical form. Using pictures or images to represent data is common in many areas, from supply and demand curves in economic forecasts, to contour maps in geographical drawings, to graphical user interfaces in computer software. The usefulness of visualization comes from the ability of humans to more rapidly obtain information from pictures than from text. As Mahling [MaFi90] notes, the visual capabilities of humans are not used to their full extent with strictly textual languages. In spite of this, however, text-based tools still prevail in many areas of computer science, including logic program development environments. As part of the ongoing research to add visual support to logic program development environments, this work extends the capabilities of ICOLA [KuOs93c, OsKu94], a graphics engine in a visualization framework for monitoring and debugging logic programs. The major contribution of this work is the construction of a new incremental object layout algorithm for ICOLA. This new algorithm is capable of producing layouts from highly under-constrained picture descriptions without requiring the specification of absolute object positions. The new algorithm supports not only the insertion of objects and constraints, but also the deletion of objects and constraints. A number of language features which are in the original specification of ICOLA's input language are also implemented. Certain object attributes (properties) can also be specified. The implementation also includes a number of minor extensions to ICOLA which improve the layout heuristics and result in consistent and accurate drawings.

The remainder of this thesis is organized as follows. Chapter 2 provides a background on the areas of visualization in logic programming, visual languages, constraint satisfaction, and graphics. Chapter 2 also discusses the state of ICOLA before this thesis work. Chapter 3 describes the updated picture description language, including the new features, syntax, and operational semantics. In Chapter 4 the new incremental algorithm is shown in detail. In Chapter 5 the results of the implementation of the new version of ICOLA are presented. Chapter 6 summarizes the highlights of this thesis and suggests some related future research topics. Appendix A contains picture descriptions for some of the larger examples presented in the thesis. Appendix B contains the raw performance values for the results presented in Chapter 5. A picture of the user-interface for the new version of ICOLA is found in Appendix C.

CHAPTER 2

BACKGROUND

ICOLA and its implementation draw on work done in many other research areas. This chapter discusses many of these areas, including visualization in logic programming, visual languages, constraint satisfaction systems, and graphics. Also examined is the state of ICOLA before this thesis research.

2.1 Motivation

A significant amount of work is being done in logic programming to provide tools for visually monitoring and debugging logic programs. Systems such as TPM [EiBr88], VISTA [Tick92], and VisAndOr [Car+93] have already proven to be quite useful. These systems, however, are typically preset in their capabilities and functionality, and are not readily changed to allow new debugging approaches [KuOs93c]. The main goal of Wand [KuOs93b, KuOs93c, OsKu94], a research project at the University of Saskatchewan, is to provide a set of graphical debugging tools which are language independent, easily tailorable, and useful for monitoring and debugging logic programming computations. As part of achieving this goal, ICOLA [KuOs93c, OsKu94] has been developed. ICOLA (Incremental Constraint-based Object Layout Algorithm) provides a graphics engine for the Wand system. The result of this thesis research is an enhanced version of ICOLA which includes all the features and facilities called for in its original design as well as additional capabilities.

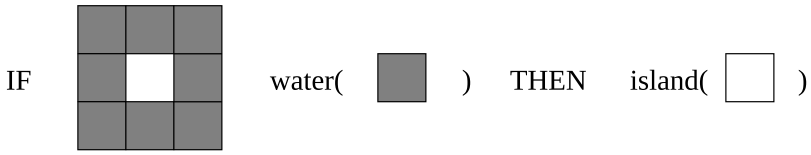
2.2 Visualization in Logic Programming

Visualization tools for debugging and monitoring logic programs are rapidly gaining popularity. The graphical tracing capabilities in these visualization tools vary in the kinds of computations they can represent and in the manner the information is presented. There are trace facilities for parallel logic programming languages (VisAndOr [Car+93], MUST [AlKa90]) and sequential logic programming languages (WAMTRACE [DiLu87], TPM [EiBr88]). Some systems, like GLP [KuOs93a], are post-mortem (i.e. are run after the program under development has generated all relevant debugging information) while others like TPM have tracing running concurrently with the execution being traced. The dynamic tracing facilities of MUST provide animations in contrast to the static traces of ParSee [PeKu94] and VISTA [Tick92]. While TPM is integrated directly into the development environment, other systems like GLP are free-standing components of the development environment.

Unfortunately the tailorability of these systems is often limited. Systems used for sequential Prolog tracing cannot typically be used for parallel Prolog tracing. Within the parallel Prolog realm, a tracer designed for only AND-tree computations cannot produce OR-tree traces. The nature and presentation of information obtained from the traces is also typically preset, and not easily changed. The Wand framework proposes a system which directly addresses these shortcomings. It is both easy to customize and easy to use.

2.3 Visual Languages

A *visual language* is a language for computer interaction which is pictorial in nature [HeMa90]. Chang [Chan87] suggests that there are two main classes of visual languages. *Visual information processing languages* deal with objects that have an inherent visual representation or deal with pictures that have some logical interpretation. These languages may be text-based or may have a visual form (i.e. use pictures to represent language constructs). They are used in applications like image processing, computer vision, and image database management. Languages in this class are typically traditional linear languages enhanced with support for visual objects. Hirakawa and Jungert [HiJu91] discuss a visual language of this type which is used for defining and retrieving information in an image database. Their language supports such visual constructs as:



and

These rules, for example, specify the visual representation of an "island", and the visual interpretation of "west_of". Their system, however, has a more visual form than most languages in this class, and can be used via icons.

Visual programming languages are the second class of visual languages that Chang suggests. These visual languages typically deal with objects that do not have an inherent visual representation. Traditional data types like arrays, stacks, and queues, and application data types like forms and documents are objects without inherent visual representations. The constructs and rules for the use of these objects are portrayed visually. These visual languages are used in computer graphics, user interface design, and database interface design. Holt [Holt90] describes a visual language of this class which is capable of representing variables and functions on those variables.

Chang also notes that in visual information processing languages or visual programming languages that visual language constructs may be visual or linear. Visual languages with visual constructs require special editors capable of handling the various images in the language. Visual languages with linear constructs can typically be handled easily by text-based editors as the visual objects and constructs can be represented using text.

Helm and Marriott [HeMa90] give four requirements they feel are necessary for any picture specification (visual) language:

1. It must be expressive enough to encode the relationships between the pictures and their meaning.

2. It must allow high-level abstractions similar to those people use for describing pictures.

3. It should have a formal, declarative semantics.

4. The generation and recognition algorithms should be efficiently implementable.

Helm et al. [Hel+90] note that the meaning of a picture produced by a visual language is generally not dependent on the absolute position, size, or orientation of the objects in the picture. Instead, the relative relationships, like relative position, are important.

Meyer [Meye93] gives a formal definition of picture logic programming, an augmentation of logic programming with *picture terms*. *Picture terms* are visual representations of pictures which behave and look very similar to Prolog terms. Picture terms can contain variables. This enables them to describe pictures whose contents may not be completely specified, a powerful concept considering most other description techniques require the entire picture to be known. Meyer also discusses a method for matching picture terms, and considers the positive effects of expressing pictures with picture terms if unification of picture terms is available. For example, the infinite recursion of a picture within itself could be easily defined with a simple recursive picture term. Meyer's work fits best into Chang's class of visual information processing languages.

2.3.1 Grammars

Visual languages are not possible without an underlying grammar. This section discusses grammars whose use is potentially applicable to declarative picture specifications. The most significant problem with typical grammars is that the languages they describe are one-dimensional and consist only of strings of text. Describing pictures, however, requires a grammar that can operate in two, or more, dimensions.

In 1967 Rosenfeld [Rose79] presented a number of 2-dimensional grammars, like array grammars and matrix grammars, as part of his work in recognizing pictures through picture languages.

Meyer [Meye93] discusses *picture clause grammars* (PCGs), which are to picture logic programming what *definite clause grammars* (DCGs) [PeWa80] are to Prolog. PCGs are formal visual specifications of visual languages and can be used both for parsing and for syntax-directed translation.

Constrained set grammars [Hel+90] have the interesting property that there is no natural parse sequence. As well, if the constraints used in a constrained set grammar are too powerful, it may be impossible to efficiently parse the language specified by the grammar.

Much work has also been done on visual language parsers [CoCh90, Cos+91, Cri+90, Fer+91, KoMy91, Ladk87, WiWe90, YuCh90]. A full discussion of the issues involved with parsing visual languages is beyond the scope of this thesis, as in this research grammars are required for picture generation, but not for parsing. It is noted, however, that parsing visual languages can be computationally expensive, and that a good compromise must be found between the expressiveness of the visual language and the computational cost to parse it [Fer+91].

2.4 Constraint Solvers

A natural way of specifying a graphical layout is with a visual language that is based on the idea of constraints. In general terms, constraints specify relationships which must be maintained. A *constraint satisfaction problem* (CSP) consists of a set of variables and a set of constraints. Each variable is over a domain specific to that variable. Each constraint either describes a relationship among the variables, or a restriction on a single variable. To solve the problem is to find a value for each variable such that all constraints are satisfied. General CSPs are *NP*-Complete, but fortunately many individual CSPs and even classes of CSPs can be solved in polynomial time. Examples of CSPs are found in game playing, scheduling, VLSI routing and layout, operations research, simulation, and many other areas. A set of linear equations over the domain of real numbers is a simple example of a CSP. A typical CSP in visualization might be to determine the absolute positions of a set of objects from a set of arithmetic equations which specify their relative positions. Positioning the set of objects in a graphical layout can be thought of as simply assigning values to variables in a CSP.

The following sections briefly discuss the main difficulties in solving constraint-based problems. The emphasis is primarily on incremental approaches to constraint solving since the incremental approach is well suited to the interactive nature of debugging and monitoring tools. Over-constrained and under-constrained systems are discussed along with incremental constraint satisfaction techniques, common solution strategies, and the internal representation of variables and constraints. As well, a number of specific CSP solvers are briefly investigated.

2.4.1 Over-constrained and Under-constrained Systems

A major problem in constraint satisfaction is *over-constrained* and *under-constrained* systems. An *over-constrained* system is inconsistent – there is no assignment of values to variables which will simultaneously satisfy all of the constraints. An *under-constrained* system has too little information to assign each variable only a single value – one or more variables could take any of a range or set of values. To deal with over-constrained systems the user is typically forced to relax some of the constraints before a solution can be found. The multiple solutions for under-constrained systems are typically avoided by applying heuristics to obtain values for the under-constrained variables. If such heuristics are not used, the solution may have to be presented to the user as a set of values, a set of intervals, or a set of symbolic expressions.

Borning et al. [Bor+87] advocate the use of a *constraint hierarchy*, to handle over-constrained and under-constrained systems. Each constraint is assigned to a certain level of a hierarchy. Each level must have as many constraints as possible satisfied before a constraint at a lower level can be satisfied. The highest level of constraints must always be satisfied. The lowest level of constraints, therefore, will rarely have any effect on the system, and are typically used only as default constraints for situations where the system would otherwise be under-constrained. A five level hierarchy has proven quite useful in ThingLab [Fre+90] and ThingLab II [Mal+89]. These constraint-based tools, ThingLab for conducting physics simulations and ThingLab II for interactive user-interface construction, are described further in Section 2.5.4.

Kamada and Kawai [KaKa91] handle under-constrained and over-constrained picture specifications differently in their layout package, COOL (COnstraint-based Object Layout). For an under-constrained system COOL adds two constraints which place the average position of all objects at the center of the picture or sub-picture. If adding these two constraints does not sufficiently restrict the variables, then COOL is unable to solve the problem. COOL uses a least-squares method to calculate an approximation to the solution if the system is over-constrained.

In other constraint solvers like CHIP (Constraint Handling In Prolog) [Din+88] the user is simply informed that there is no solution if the constraint system is over-constrained. If the system is under-constrained, CHIP gives all possible solutions to the user in the form of a symbolic, most general solution.

2.4.2 Incremental Issues

The term *incremental* can be used in two different ways to describe an algorithm. First, an algorithm is said to be "incremental" if it maintains an internal state, and each new input is processed in conjunction with the internal state. This definition of incremental, however, says nothing about how the internal state is updated, and refers only to the incremental nature of the interface of the algorithm. The second usage of "incremental" directly refers to the issue of updating the internal state. In particular, the previous solution to the problem (or the internal information used to derive that solution) is combined with the new information to produce the new solution. Consider two algorithms for computing a running average of a stream of numbers. An algorithm with only an incremental interface might simply add the next value from the stream to an array (the internal state), add all the values in the array to get the sum, and then divide by the number of values in the array to obtain the average. An algorithm which uses incremental update methods internally, on the other hand, could add the next value to the array, but would rely on the knowledge of the sum of the previously inserted values. Storing the previous sum in the internal state allows easy computation of the new sum, and then easy computation of the new average. As the number of values processed increases, the algorithm using incremental updating internally will outperform the algorithm with the incremental interface. While both systems appear to be incremental, only the second system implements a incremental algorithm in the true sense of the meaning. In this thesis the incremental nature of an algorithm is defined by how the algorithm behaves, not by the nature of the interface.

Constraint satisfaction systems can be classified as being incremental or non-incremental. In non-incremental systems, finding a new solution after a slight change to the variables or constraints requires a complete re-calculation by the solver. No use is made of the previous solution. Incremental systems, however, maintain an evolving set of variables and constraints over time, and each increment is combined with the previous solution (or internal information used in arriving at that solution) to produce a new solution. Incremental techniques for constraint satisfaction have proven quite useful in many systems [Fre+90, Hel+92, Mal+89, Men+93, Oste93, Stuc91a, vanH90, vaLe91]. They are considered superior for applications where the user interactively specifies the constraints, and the application is expected to react quickly [Hel+92]. Incremental solvers are faster than non-incremental solvers as long as each incremental update of the solution can be done faster than solving the entire current system of constraints from the beginning. For finding one-time solutions to a large system of constraints, however, non-incremental solvers may be faster since they do not have the overhead costs associated with maintaining the intermediate information.

Incremental systems may allow both incremental insertions and incremental deletions which add constraints and variables to, and, respectively, remove constraints and variables from, the constraint system's current knowledge base. While most constraint solvers allow both the set of variables and the set of constraints to change [Fre+90, Mal+89, MiFa90, OsKu94, San+93] there is no guarantee that incremental deletions will be allowed.

For the remainder of this thesis, an "increment" refers to both insertions and deletions, unless explicitly specified otherwise. It is also required that for a constraint to be incrementally deleted, it must have been explicitly incrementally inserted earlier. This requirement ensures that the behavior of a constraint deletion is intuitive and well defined. As well, the deletion of a constraint does not necessarily mean that the negation of that constraint is true (e.g. if the constraint $a > 10$ is inserted, the deletion of the constraint $a > 10$ does not necessarily mean that $a \leq 10$ holds). Deletions always result in a less specific (or equally specific, if the deleted constraint was redundant) constraint system.

Systems of constraints which are being solved incrementally typically remain under-constrained until sufficient constraints are added to make the system uniquely solvable. Unfortunately, once a system reaches this point then all constraints added to the system which are not redundant and which do not introduce new variables will result in an over-constrained system. The system will become under-constrained, however, if a new constraint introduces new variables whose values are not uniquely determined.

An important issue in incremental constraint satisfaction is how the effects of a new constraint are propagated throughout the system. *Local constraint propagation* refers to a technique where the effects of a constraint are propagated only to a small, localized portion of the system (i.e. only a small number of variables or constraints are affected). An example of local propagation is seen in changing the value of a variable which occurs in a number of equations in a set of constraints. That set of constraints might be represented in an implementation by a graph with variables encoded as nodes, and constraints as arcs. Local propagation would mean that only a small number of nodes are visited by some propagation algorithm. If the variable occurs in many of the equations, then local propagation would be forced to visit a larger number of nodes, and the local propagation benefits diminish. A key deficiency of local propagation is that it often can only guarantee local consistency – that the system is consistent in each localized area (e.g. a subset of the constraints). Local consistency may not be sufficient to guarantee global consistency. With *global constraint propagation* the effects of the constraint must be distributed throughout the entire system. This has the benefit of allowing full consistency to be achieved, but the disadvantage of requiring a large amount of propagation. Incremental systems are more effective if most of the constraint propagation is localized, as, in general, the less propagation done, the faster the system. Davis [Davi87] discusses propagation issues in more depth.

2.4.3 Solution Strategies used in Constraint Solvers

Constraint solvers typically deal with constraints in two different ways – either symbolically or numerically. Symbolic solvers rely on manipulating the constraints to find a solution, and both the calculations and solutions are expressed symbolically. In CHIP [Din+88], for example, valid symbolic constraints are: $X > Y$, $X >= Y$, $X < Y$, $X =< Y$, $X = Y$, and $X\tilde{}= Y$. Actual values are inserted into the symbolic solution to get the final (numeric) solution. In the numerical (non-symbolic) solvers, algebraic computations (e.g. Gaussian Elimination or least-squares calculations) are used to find a solution.

There are two basic approaches to incrementally solving constraints. The *incremental search* approach typically involves a tree-like search strategy. At each increment, the new information is used to prune the search tree – i.e. to determine which parts do not need to be re-examined in the search for a new solution. The main goal is to re-use the old search tree as much as possible [MaSt89, vanH90, vaLe91]. The technique is similar to dependency-directed backtracking, where knowledge from the previous computation is used to guide the current search through the tree.

The *add-and-synthesize* approach to incrementally solving constraints behaves as its name indicates – new information is combined with the existing solution to form the new solution. The main idea is that the internal representation remains in the same form after each update. The constraint systems using the add-and-synthesize approach are ideally able to detect which variables are affected by the increment and update only their values. New values are found for the variables using techniques like local propagation or relaxation methods [Lele88]. The synthesis involves propagation or calculations which are not using explicit search strategies like those found in the incremental search approaches. For an incremental solver to be effective, the number of previously solved variables which need to be changed after an increment should be much smaller than the total number of variables in the system.

2.4.4 Internal Constraint Representations

There are many varying internal representations of constraint systems. In this section a number of the main representations are outlined. It should be noted that many systems use a combination of these approaches.

1. Trees. Tree-based backtracking approaches are used in many logic programming language implementations and constraint solvers as part of their incremental search techniques [MaSt89, vanH90, vaLe91]. While many of these tools conduct exhaustive searches, techniques such as dependency-directed backtracking are used to greatly improve efficiency.

2. Tableaus. A number of constraint solvers incrementally update an internal tableau. As each increment is added, the entries in this matrix-like table are recomputed as necessary. The incremental Simplex solver in CLP($\Re$) [Stuc91b] uses such an approach.

3. Graph structures. Many constraint systems [Bor+87, Meye93, Sann93, Seid81] encode the variables and constraints in a graph-based structure. This encoding is typically accomplished by either encoding variables and constraints in separate nodes (e.g. Meyer's "Pictures Depicting Pictures" [Meye93]) or by encoding variables as nodes and constraints as arcs (e.g. ThingLab [Fre+90] and ThingLab II [Bor+87]). In these structures, graph properties are often used to obtain more information about the constraints. A cycle in the constraint graph, for example, might indicate that the constraints are inconsistent.

2.4.5 Constraint Satisfaction Systems

This section discusses a number of different constraint satisfaction systems (constraint solvers) which are not specifically designed for graphical layout or visualization. These systems could be used, however, to solve constraints in graphical layout or visualization tools.

CHIP (Constraint Handling In Prolog) [Din+88] is a declarative logic-based language similar to Prolog. It has symbolic and numerical constraint solving capabilities and is designed to handle constrained search problems in the domains of finite terms (a discrete, finite search space), boolean terms (an exponential search space), and linear rational terms (an infinite number of points in the search space, with rational numbers used to model the points). CHIP returns symbolic solutions to queries, and these solutions are the most general ones. This allows CHIP to finitely represent an infinite number of solutions. While many constraint systems try to deal with over-constrained problems, CHIP simply informs the user that there are no solutions. Under-constrained problems have many possible solutions, and all of these are returned to the user in the form of a symbolic solution.

CLP($\Re$) [JaMi87] is a constraint logic programming engine for use in the domain of real numbers. An incremental Simplex algorithm [Stuc91b] provides linear arithmetic capabilities to CLP($\Re$). As each new constraint is encountered, a corresponding (new) equation is added to the existing set of equations which are in the form of an internal Simplex tableau. Over-constrained systems have no solution, and under-constrained systems are solved using slack variables as in the regular Simplex algorithm. No facility is available for deleting constraints in this system.

Geometric constraints in constraint logic programs are discussed by Dubé and Yap [DuYa93]. They claim that incorporating geometric constraints into logic programming is non-trivial, as general-purpose constraint solvers would typically be too inefficient to be practical.

2.5 Graphics

While constraint satisfaction systems do the actual work in solving layout problems, their efforts are directed by a number of issues in the areas of graphics and diagram layout. This section examines some prominent problems and techniques found in declarative and constraint-based graphics, and discusses a number of (constraint-based) graphics systems.

2.5.1 Interactive Graphics Issues

Interactive graphics tools and concurrent visualization systems have the very difficult task of satisfying the requests of a user in real-time. Helm et al. [Hel+92] note that the constraint solvers for interactive graphics-based tools must be incremental, the user feedback must be fast, and the rate of feedback should be high. Interactive graphics applications which allow users to specify the constraints must also be capable of gracefully handling under-constrained and over-constrained situations [Glei93].

The "Principle of Least Astonishment" [Hel+93] is very important in interactive graphics systems. This principle states that the transitions between consecutive incremental solutions should be smooth, such that the user is easily able to track the movement of the objects. Rowe et al. [Row+87] also acknowledge that stability between successive drawings is extremely important. If the diagram remains mostly unchanged, the user can retain a mental model of the general layout and picture structure. The findings of Rowe et al. come from first-hand experience with DAG, an interactive browser for directed graphs, where image consistency is especially important in dealing with incremental deletions of graph edges and nodes.

2.5.2 Graphical Layout Issues

Graphical presentation of information requires care that the images present the information in a clear, consistent, and unobtrusive way. In graph (or tree) drawing, for instance, there are a number of aesthetic criteria which should be considered when constructing a graph drawing algorithm. These include evenly distributing nodes in the picture, minimizing the number of edge crossings, making edge lengths uniform, and portraying any graph symmetry. Di Battista et al. [Bat+93] give an annotated bibliography of many graph drawing algorithms. Cohen et al. [Coh+92] discuss a framework for dynamic (incremental) graph drawing.

Other layout systems, like VLSI design packages [Roac84], have additional restrictions which must be met. These restrictions, for example, might be to minimize the total area of the circuit, or that each path on the circuit should be less than a certain length. These restrictions add extra difficulties for the constraint solver.

Automatic software diagram layout [Pro+91] presents issues relating to the positioning of the various data-flow diagram components. Included here are routing problems, ensuring clarity of the resulting diagram, and labeling of the various components. Software diagram layout has the interesting property that some components (e.g. data stores and interfaces) may be replicated (drawn in multiple locations on the diagram). A component is replicated if doing so would result in greater diagram clarity. Too much replication, of course, makes the diagram confusing and hard to read.

The layout issues presented in this section must be considered when designing any layout algorithm, as the paramount concern of any visualization tool is to display images in the best possible way.

2.5.3 Declarative Graphics

Declarative graphics systems specify picture layout with declarative constructs instead of with functions having side-effects. Many logic programming systems have some declarative graphics capabilities. KEGI in SEPIA [Kegi90], Graphics Manager (GM) in SICStus Prolog [SICS94], ProXT and ProXL in Quintus Prolog [Quin91], as well as the graphics facilities in Wild LIFE [Aït+94] are all examples of graphics support in logic programming systems. While all of these systems have reasonable graphics capabilities, the level of windowing system expertise required to use them varies greatly. ProXT and ProXL require a good grounding in programming concepts and techniques for the X Window System [Nye90]. This is due to the fact that the "declarative" semantics behave more like functions with side effects. Other systems like GM have a higher level of abstraction which allows the user to ignore the low-level details and concentrate on the interface layout and design.

Haarslev and Möller [HaMö90] give a declarative formalism for specifying graphical layout. Their "box-and-glue" approach is similar to the layout techniques used in TEX [Knut84] and is extended for use in specifying graphical user-interfaces.

Helm and Marriott [HeMa90] discuss the declarative specification of graphics in the context of logic programming. Their declarative picture specification language combines ideas from constraint satisfaction and grammar-based formalisms.

Johnson and Harrison [JoHa91] present PRELOG, a tool for Presenting and REndering LOGic specifications of interactive tools. PRELOG uses ideas from temporal logic to create a temporal, declarative graphics engine.

2.5.4 Constraint-based Graphical Tools

Constraint-based graphical tools are used in logic-program visualization, user interface construction, VLSI layout, and other applications. This section discusses a number of constraint-based graphical tools from some of these areas.

Sutherland's Sketchpad [Suth63] was a pioneer in the area of constraint-based graphical editors. Developed in 1963, Sketchpad was far ahead of its time and is still being praised for its contributions to visualization. Sketchpad had the ability to draw points, lines, and circles. These objects could be duplicated, deleted, rotated, or scaled. Mathematical constraints could be also be placed on the objects. The limited capabilities of the hardware, however, prevented it from becoming widely used.

ThingLab [Born81, Bor+87] and ThingLab II [Fre+90, Mal+89, San+93] are two tools which use constraint satisfaction techniques. ThingLab applies constraint programming techniques to interactive physics simulations. ThingLab II is an interactive user-interface construction tool. The DeltaBlue algorithm found in ThingLab II shows the power of incremental methods. As mentioned in Section 2.4.1, the constraint solvers in ThingLab and ThingLab II use a number of constraint hierarchies. From highest to lowest, the five levels of constraints in the constraint hierarchy are: "require", "strongly prefer", "prefer", "default", and "weak default". The "require" constraints are the constraints (user- or system-specified) which must always be satisfied. The "prefer" constraints correspond to the interface editing requests of the user. The "weak default" constraints are system defaults. The constraint satisfaction algorithms in ThingLab and ThingLab II have the concept of a solution being *locally-predicate-better*. A solution R_i is considered *locally-predicate-better* than solution R_j if for a level k in the constraint hierarchy every constraint satisfied by R_j is satisfied by R_i, and R_i satisfies at least one more. A constraint graph is used to internally represent the set of constraints. Nodes in the graph represent variables, and edges represent the constraints. When a new constraint is added to the system, each node which corresponds to a variable in the new constraint is checked to determine if having that node satisfy the new constraint would result in a *locally-predicate-better* solution. If such a "better" solution would result then the node is changed to satisfy the new constraint. Otherwise, the node remains unchanged. As the nodes of the internal constraint graph are updated, the effect is propagated throughout the graph until no further local propagation would result in a *locally-predicate-better* solution. If multiple equally good *locally-predicate-better* solutions exist then the algorithm will arbitrarily pick one of them.

The DeltaBlue algorithm of ThingLab II is an incremental version of the Blue algorithm found in ThingLab. Blue is part of a family of algorithms for solving constraint hierarchies. Other members of this family of constraint solvers include: Red, based on graph re-writing; Orange, based on the Simplex algorithm; Yellow, based on iterative hill-climbing techniques; and Green, for solving finite-domain constraints. The SkyBlue [Sann93] algorithm extends DeltaBlue to allow cycles of constraints and multiple values to be determined from a single formula (versus only a single value from a single formula, as in the DeltaBlue algorithm). The incremental ideas in DeltaBlue are also being used as a guide for the creation of DeltaRed and DeltaOrange, incremental versions of the Red and Orange algorithms, respectively.

The COOL (COnstraint-based Object Layout) system developed by Kamada and Kawai [Kama89, KaKa91] allows pictures to be specified by objects and constraints. The two fundamental rules which govern objects in COOL are [KaKa91]:

1. The reference points of the graphical objects must lie on a geometric curve (line, circle, ellipse, arc, or spline) at equal intervals.

2. The reference points of a graphical object may be connected by a geometric curve (line, arc, or spline) with or without an arrowhead.

COOL has two classes of constraints – rigid and pliable. As their names indicate, rigid constraints must be satisfied, while pliable constraints need not be satisfied. The graphical objects available include `box`, `diamond`, `circle`, `ellipse`, and `point`. Constraints include `above`, `below`, `leftof`, and `rightof`. COOL also supports other constructs for laying out a collection of objects in a specific pattern. For example, the `y_order` constraint specifies a list of objects which are to be drawn in a row horizontally in the order they occur in the list. Kamada and Kawai [KaKa91] give a complete list of the constructs supported by COOL.

Roach [Roac84] describes RPL (Rectangle Placement Language), a constraint-based symbolic layout language. This language includes the constructs `aligned`, `above`, `below`, `leftof`, and `rightof`, and also allows explicit specification of an object's *bounding box*[1] in constraints with `(LEFT X)` and `(RIGHT X)` LISP functions. RPL constraints are solved by a rule-based expert system which is used as part of an editor for VLSI diagram layout. The expert system also aids the VLSI designer by providing advice and information on properties of the current design layout. RPL also has *restraints*, which are evaluatable designer goals [Roac84] similar to user-defined, high-level constraints on the layout. Restraints have a lower priority than constraints. The goal of a restraint, for example, might be to minimize the area of the VLSI layout, while the goal of a constraint might be to minimize the size of a single object. RPL has a "completeness" restriction that any two variables (representing component (object) positions) must have either an explicit or an implicit relationship. There is no mention of Roach's system being incremental.

Helm et al. [Hel+92] describe a linear constraint technology as a foundation for user-interface construction tools. In this constraint system, constraints are in the form of equations. As each incremental constraint is added or deleted the resulting constraint system is encoded in an internal canonical form. The set of constraints in this canonical form is typically smaller than the original set of constraints. It contains no redundant constraints, restricts the set of values which can be assigned to variables, and makes all implicit constraints explicit [Hel+92]. A *projection* is done for each variable in the system. The projection determines the range of values assignable to each variable while still satisfying the constraints. Projecting a system of constraints onto a set of variables shows the inter-relationships among the variables for any solution to the constraints [Hel+92]. An interesting aspect of this technology is that for over-constrained systems, the interface construction tool presents the interface developer with a list of constraints which can be relaxed to make the constraint system solvable. This list of constraints also typically contains many *anchor* constraints. The purpose of *anchor* constraints is to attempt to make variables remain as constant as possible after each increment. The anchor constraints provide an initial structure to a developing picture, and must often be

[1] A *bounding box* is a rectangle which is just large enough to entirely surround an object.

removed as the user constraints begin to more fully define the picture. The anchor constraints also help ensure that changes to the image follow the "Principle of Least Astonishment". Under-constrained systems require the user to tighten some of the constraints, and again the interface construction tool is able to inform the user as to which changes to the constraints would be the most effective.

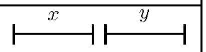
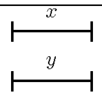
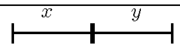
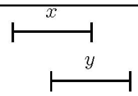
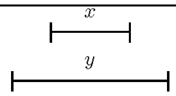
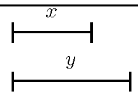
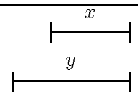
Cruz [Cruz93] discusses an *U-term language* for declarative specification of pictures. This language supports such graphical objects as `box`, `circle`, `arrow`, and `text`. *Cartesian landmarks* (like midwest, midnorth, mideast, midsouth, and center), *polar landmarks* (like first angle, second angle, inner radius, and outer radius), and *line landmarks* (the head and tail of a line) are used to specify the shape of an object. The constraints available are of two classes: *length constraints* and *overlap constraints*. The *length constraint* defines a distance between two landmarks, and the *overlap constraint* indicates which of two objects is drawn on top.

Donikian and Hégron [DoHé93] present a model for describing scenes in CAD systems. Their system is based on Allen's 13-valued interval algebra [Alle83] for maintaining knowledge about temporal intervals. These thirteen relationships (including the inverse relations), seen in Table 2.1, define the possible relationships between two intervals. The description of a scene is translated into a system of linear and non-linear inequalities. It is interesting to note that their algorithm does not attempt to detect all inconsistencies in the picture definition – it detects only those inconsistencies which can be identified easily in the linear inequalities.

Wilk [Wilk91] discusses Equate, an object-oriented constraint solver which determines the positions of the objects using an algorithm based on definite-clause backward chaining. Objects conform to the principles of encapsulation, and rules for a given object class are used to break down a constraint into smaller, more manageable constraints.

As seen in this section there are many issues and concerns which must be accounted for when constructing an interactive graphical layout tool. Not only must the tool be fast and efficient, but it must also produce a suitable image without overwhelming the user.

Table 2.1: Thirteen possible relationships between two objects x and y.

Relation	Symbol	Symbol for Inverse	Example
x before y	$<$	$>$	
x equal y	$=$	$=$	
x meets y	m	mi	
x overlaps y	o	oi	
x during y	d	di	
x starts y	s	si	
x finishes y	f	fi	

2.6 ICOLA

ICOLA (Incremental Constraint-based Object Layout Algorithm) [OsKu94] is a layout engine for a logic programming visualization tool under development at the University of Saskatchewan. ICOLA can produce images incrementally from graphical object specifications and constraints among those objects.

This section begins with a brief history of the development of ICOLA. An overview of ICOLA is given with a discussion of its position in the Wand framework. The picture description language (PDI - Picture Descriptions for ICOLA) is then examined, and the problems with ICOLA which prompted this thesis work are explained.

2.6.1 The History of ICOLA

There are four versions of ICOLA, referred to as ICOLA-1, ICOLA-2, ICOLA-3, and ICOLA-4. The picture description language of ICOLA-1, developed in early 1992, allowed specification of objects of types `ellipse`, `rectangle`, `circle`, and `square`. Relationships between the objects were specified using the constraints `above`, `below`, `left_of`, and `right_of`. Absolute coordinate positions were not required. The incremental constraint solution algorithm, implemented in Prolog, was quite simple, and outperformed a non-incremental constraint solution algorithm.

The picture description language for ICOLA-2 added support for objects of type `line`, `text`, and `point` and supported constraints between the edges of objects, not just between entire objects. The constraints `touching`, `horizontally_aligned`, `vertically_aligned`, and `connects` were added. ICOLA-2's constraint algorithm was a slight modification of ICOLA-1's algorithm. The constraint engine was written in C, and performed ten times faster than ICOLA-1.

In ICOLA-3 the input language became known as PDI – Picture Descriptions for ICOLA. The `box` object replaced `square` and `rectangle`, and the `oval` object replaced `ellipse` and `circle`. These replacements allowed similar functionality with a more robust constraint solution algorithm. The PDI specification called for the addition of the objects `triangle`, `bitmap`, and `arc`, and object attributes, but these were not implemented. The while being much better than older versions, ICOLA-3 was deficient in a number of areas (See Section 2.6.7) and a better object layout algorithm was desired.

It is with ICOLA-3 that this thesis work begins. The description of ICOLA in the following sections is a description of ICOLA-3. In Chapter 3 the new PDI language for ICOLA-4 is presented, and the new constraint algorithm is discussed in Chapter 4. For the remainder of this thesis, PDI-3 will refer to the PDI language supported by ICOLA-3, and PDI-4 will refer to the PDI language supported by ICOLA-4.

2.6.2 The Wand Framework

ICOLA (Incremental Constraint-based Object Layout Algorithm) is a graphics engine for Wand. Wand [KuOs93b] is a design framework for the development of monitoring and debugging tools for logic programming. The overall design of Wand is found in Figure 2.1. A meta-interpreter generates trace information from the system being monitored or debugged. A *transformation specification* provides the mapping between trace information and an internal knowledge base. The rules for converting the data in the internal knowledge base into a graphical form is contained in a *configuration specification* file. The configuration specification language is a constraint-based "display description language", similar in style to definite clause grammars. The terminals of this grammar are basic graphical objects which can be drawn by the underlying system. Non-terminals are graphical objects defined in terms of terminals, non-terminals, constraints, and attributes. These grammar rules are used as productions, with their arguments instantiated with information from the knowledge base. Labels are used to refer to objects in the constraints. Constraints, attributes, and labels can appear on either side of the production symbol.

Figure 2.2 shows an example of a rule in this display description language for specifying an "AND-tree" in a trace of a logic program.

This rule states that the "AND-tree" (or "AND-subtree") corresponding to the execution state described by `t( Parent, Children, Clause )` is

- an "AND-node" showing the information in `Parent`, plus
- a collection of "AND-subtrees", each corresponding to an element in `Children`,

where

- an "AND-edge" showing the information in `Clause` joins each "AND-subtree" to the parent "AND-node", and
- all the subtrees are horizontally aligned.

Unfortunately, the above grammar is far removed from the graphics extensions found in Prolog systems. To bridge this gap, the Wand design maps the display

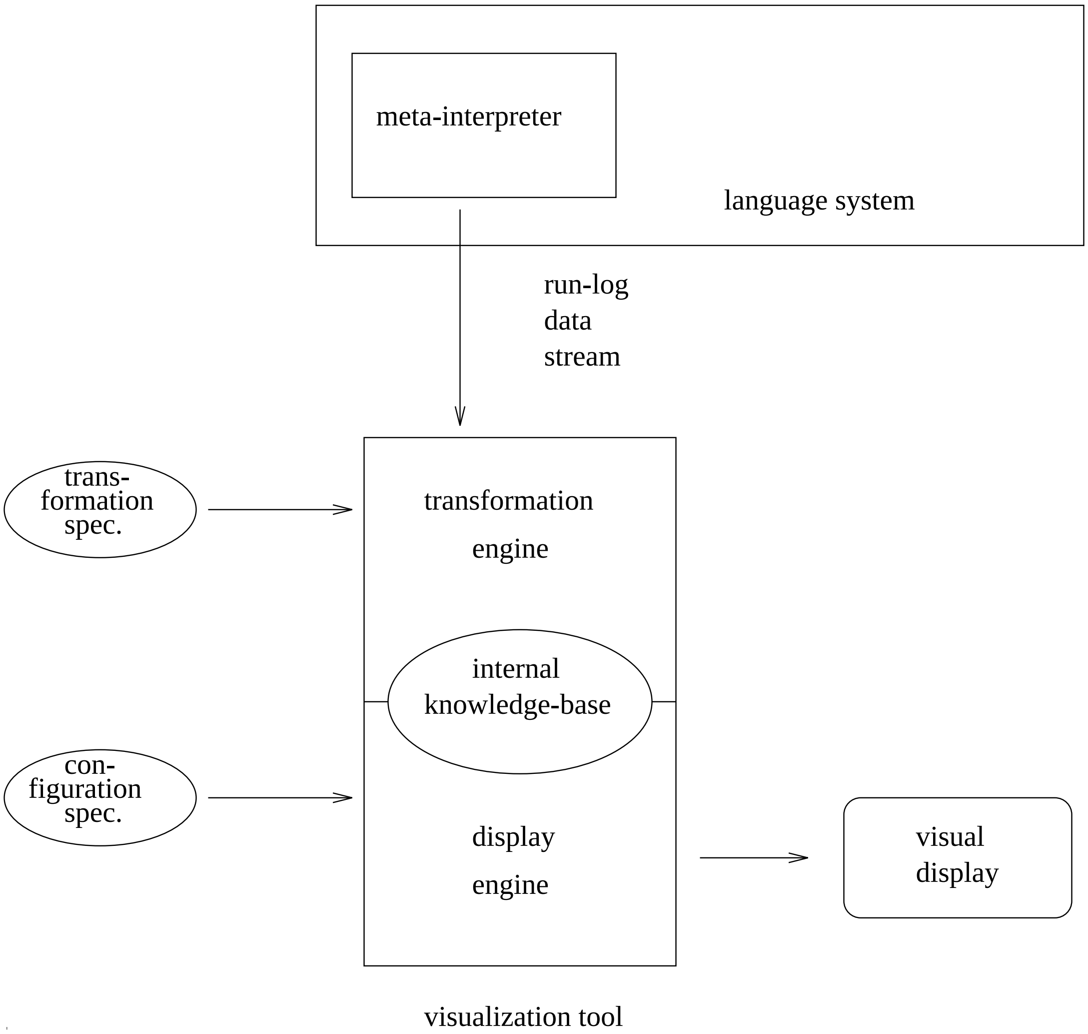


Figure 2.1: Overview of the Wand design.

```
and_tree( t( Parent, Children, Clause )) ==>
        and_node( Parent )<N>,
        collect( Child,
                ( and_edge( Clause )<E>,
                  and_tree( Child )<St>,
                  where E connects N to St ),
                Children )<C>,
        where horizontally_aligned C.
```

Figure 2.2: Configuration specification for an “AND-tree”.

description language to an intermediate language, as seen in Figure 2.3. This intermediate language is then mapped to graphics primitives which result in the graphics being drawn. In this case, the intermediate language is PDI (Picture Descriptions for ICOLA). Figure 2.4 shows how ICOLA fits into the Wand system.

configuration specification	→	intermediate	→	graphics primitives
(display description language)		language (PDI)		(Prolog)

Figure 2.3: Bridging the language gap.

It should be noted that ICOLA's use is not restricted to only the Wand framework. Any application which needs high-level constraint-based graphics capabilities and can produce a stream of PDI constructs is capable of using the facilities provided by ICOLA.

2.6.3 PDI-3 – Picture Descriptions for ICOLA-3

The declarative picture description language of ICOLA-3, PDI-3 (Picture Descriptions for ICOLA-3), consists of objects, constraints, attributes, and control constructs. All PDI language constructs are in the form of valid ground Prolog terms. "White space" in a picture description is ignored, as are "comments" (in the form of valid Prolog comments).

The origin of the coordinate system used in ICOLA-3 is the upper left-hand corner of the image. All coordinate values[2] are non-negative integers, and the maximum size of an image is theoretically infinite. An important feature of the PDI language is that pictures are easily specified with relative positioning constraints – absolute coordinate positions for the objects are not required.

The font styles and what they denote throughout the remainder of this paper are as follows:

- `typewriter` – actual language constructs

[2]In the current implementation the coordinate values are expressed in "units", where each "unit" is simply a "pixel". This need not be the case in future implementations.

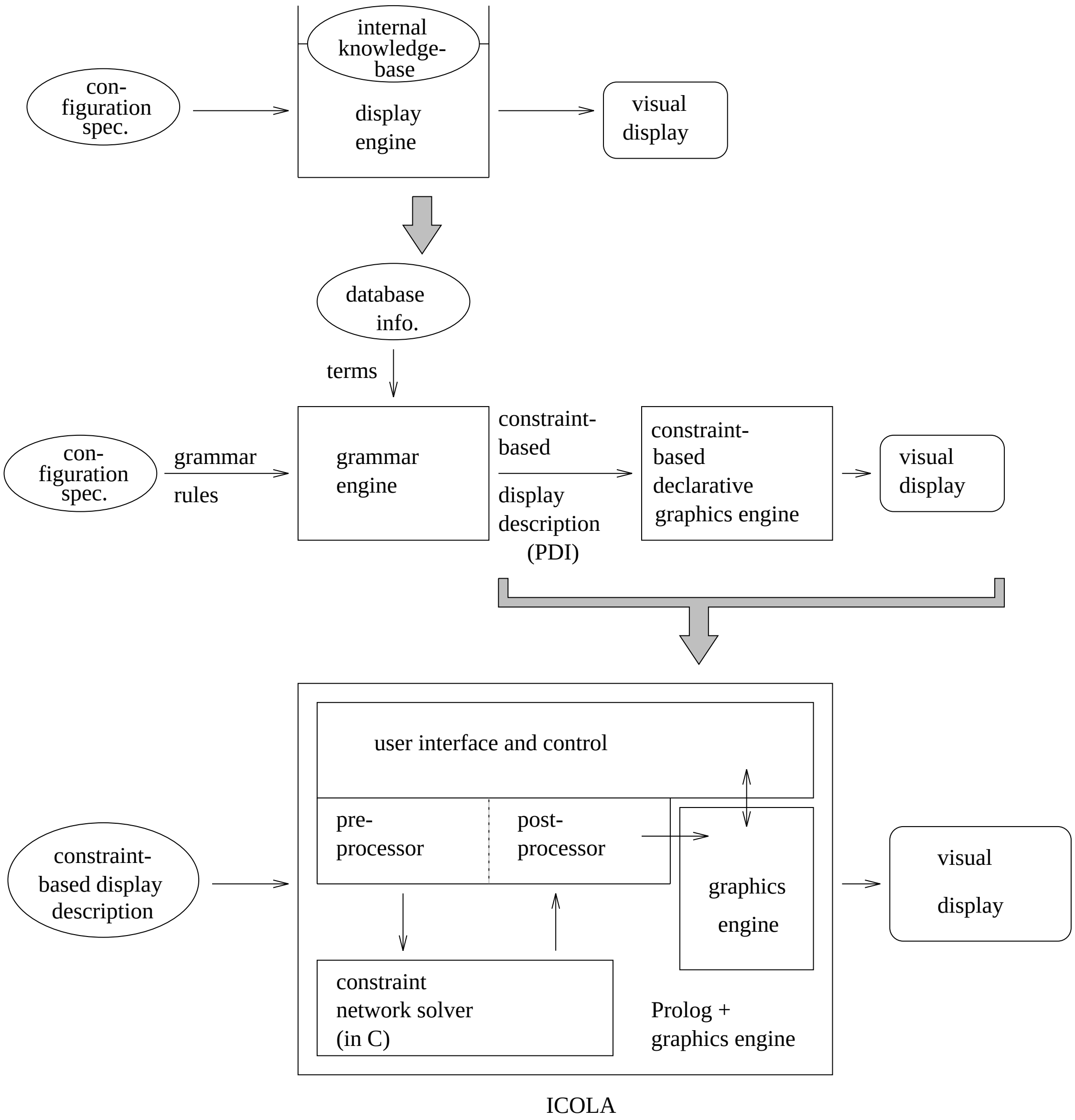


Figure 2.4: ICOLA in the Wand system.

- Sans Serif – user data
- **boldface** – text to be replaced with actual language constructs
- SMALL CAPS – object classes
- *italics* – emphasis

Objects

The PDI-3 language is based on the object hierarchy found in Table 2.2. To explain the hierarchy further, the following definitions of the classes are given:

- Obj – all objects are of this parent class.
- BB – objects having a bounding box.
- RegBB – objects having a regular bounding box: that is, other objects can be placed inside these objects, and the bounding box will increase in size to accommodate them.
- SpecialBB – objects having a special bounding box – these objects have either a fixed image, or are the special complex objects.
- Fixed – objects with a fixed image. This includes bitmap images and text.
- RectBB – objects having a strictly rectangular shape, as well as a (rectangular) bounding box.
- NonRectBB – objects having a non-rectangular shape, but which are assigned a rectangular bounding box.
- Point – objects which are represented by a single "dot".
- Bitmap – objects whose images are defined by bitmaps.
- Complex – objects used for grouping other objects together.
- Line – objects which are either lines or arcs.
- Text – objects which are text strings. For a given text string, the text image is fixed.

There are nine object types defined in the PDI language, six of which are supported by PDI-3 and implemented in ICOLA-3. Table 2.3 lists all the objects with a brief description for each of them.

Table 2.2: Object classes in PDI-3.

OBJ	=	BB $\bigcup$ LINE
	=	{ `box, oval, triangle, bitmap, point, text, complex, line, arc` }
BB	=	REGBB $\bigcup$ SPECIALBB
	=	{ `box, oval, triangle, bitmap, point, text, complex` }
REGBB	=	RECTBB $\bigcup$ NONRECTBB
	=	{ `box, oval, triangle` }
SPECIALBB	=	FIXED $\bigcup$ COMPLEX
	=	{ `bitmap, point, text, complex` }
FIXED	=	BITMAP $\bigcup$ POINT $\bigcup$ TEXT
	=	{ `bitmap, point, text` }
RECTBB	=	{ `box` }
NONRECTBB	=	{ `oval, triangle` }
POINT	=	{ `point` }
BITMAP	=	{ `bitmap` }
COMPLEX	=	{ `complex` }
LINE	=	{ `line, arc` }
TEXT	=	{ `text` }

Table 2.3: Descriptions for the objects in PDI-3.

Object Type	Description
`box`	an extensible rectangle.
`oval`	an extensible ellipse.
`point`	a single "dot".
`line`	a straight line between two objects.
`triangle`	an extensible triangle. (Not implemented.)
`bitmap`	a user-defined bitmap. (Not implemented.)
`arc`	a curved arc between two objects. (Not implemented.)
`complex`	a "grouping" object. It can be used to combine a number of objects. Any operation or constraint applied to a complex object applies to all members of the complex object as a unit.
`text`	a text string.

Objects in ICOLA must be given a unique name and a type before they can be referenced in a picture description. An object's name is represented by a Prolog atom. Scoping of all names is global. All objects except those of type `complex` and `text` are created using the syntax:

`object(` ObjectName, **ObjectType** `)`

where ObjectName is the name of the object and **ObjectType** is an object type (from Table 2.3) other than `complex` or `text`. Complex and text objects have a special syntax. Complex objects are created by:

```
object( ObjectName, complex ).
        body_of_complex_object
end_complex.
```

where body_of_complex_object is replaced with the objects and constraints that comprise the complex object. Complex objects may be nested, with each

`end_complex`

ending the scope of the innermost (when encountered) complex object. A special complex object called `root` denotes the entire picture. All objects are automatically contained within the `root` object. The object name `root` is reserved and can not be used as the name of any other object in a picture description.

A `text` object is declared using:

`object(` ObjectName, `text`, Text `)`

where Text is a string of characters enclosed in double quotes (""). This method of specifying the text string is a result of attributes not being implemented in ICOLA-3.

Selector functions are an important feature of the PDI language. Selector functions allow explicit specification of the left, right, top, or bottom sides of an object's bounding box. Selector functions can be applied to any object in class BB. There are two horizontal (`left` and `right`) and two vertical (`top` and `bottom`) selector functions. The left side of the bounding box of object A is specified by `left(A)`.

The right, top, and bottom sides of the bounding box of object `A` are specified by `right(A)`, `top(A)`, and `bottom(A)`, respectively. Figure 2.5 shows the edges of a bounding box for a hypothetical object and the related selector functions. Using horizontal selectors in vertical constraints, or using vertical selectors in horizontal constraints is not allowed. For the `point` object the bounding box is defined by a single point.

For the remainder of this thesis the following notational conventions will be used: **hs** will stand for "horizontal selector", and **vs** will stand for "vertical selector". Thus **hs**(`A`) can be replaced with `left(A)` or `right(A)`, and **vs**(`A`) can be replaced with `top(A)` or `bottom(A)` for some object `A` of class BB.

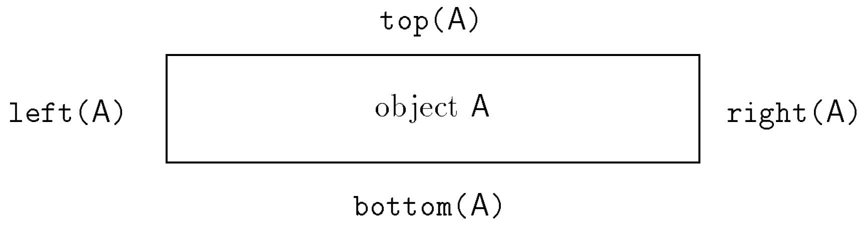


Figure 2.5: Selector functions for object A.

Constraints

Constraints in PDI specify positional relationships among the objects. A brief description of the constraints is found in Table 2.4. The constraints in PDI are categorized into two classes – *basic* constraints and *aliased* constraints. *Basic* constraints are the fundamental constraints of the PDI language. *Aliased* constraints map to one or more basic constraints. The basic constraints and their general usage are listed in Table 2.5, where A and B are objects of class BB, Distance is an integer value, and Edge is an object of type LINE.

For convenience, ICOLA provides default selector functions when object names are given as arguments but selector functions (applied to these objects) are expected. When a constraint is specified between two objects, the default selector functions are chosen so that the constraint is applied to the entirety of each object. Table 2.6

outlines these defaults for the basic constraints. If only one selector function is given, then a default is provided for the other argument.

Table 2.4: Constraints in PDI-3.

Constraint	Description
`left_of`	one object is to the left of another.
`right_of`	one object is to the right of another.
`horizontal_distance`	`left_of` by a specified number of "units".
`above`	one object is above another.
`below`	one object is below another.
`vertical_distance`	`above` by a specified number of "units".
`connects`	two object are connected by a `line`.
`inside`	one object is contained inside another.
`touching`	one object borders on another in a specified axis, such that the objects have no space between them.
`aligned`	align the specified edges of two objects so that they share a common horizontal or vertical axis.

Table 2.5: Basic constraints in PDI-3.

Constraint	Usage
`left_of`	`constraint( left_of, [ hs(A), hs(B) ]).`
`above`	`constraint( above, [ vs(A), vs(B) ]).`
`horizontal_distance`	`constraint( horizontal_distance, [ Distance, hs(A), hs(B) ]).`
`vertical_distance`	`constraint( vertical_distance, [ Distance, vs(A), vs(B) ]).`
`connects`	`constraint( connects, [ Edge, A, B ]).`

Table 2.7 shows the aliased constraints and their mapping to basic constraints. Aliased constraints are not required for the full functionality of the system, as all aliased constraints can be represented by one or more basic constraints. They do, however, make specifying certain relationships (like inside and touching) much easier. Note that aliased constraints are converted to basic constraints before the default selectors are applied.

Table 2.6: Constraints and the default selector function application.

Constraint → Constraint with default selector functions applied.
`constraint( left_of, [ A, B ]). →` `      constraint( left_of, [ right(A), left(B) ]).`
`constraint( above, [ A, B ]). →` `      constraint( above, [ bottom(A), top(B) ]).`

`constraint( horizontal_distance, [ D, A, B ]). →`
`      constraint( horizontal_distance,`
`                 [ D, right(A), left(B) ]).`

`constraint( vertical_distance, [ D, A, B ]). →`
`      constraint( vertical_distance,`
`                 [ D, bottom(A), top(B) ]).`

The `left_of`, `right_of`, `above`, and `below` constraints specify the corresponding intuitive spatial relationships between objects. Figure 2.6 shows a simple picture description illustrating the use of the `left_of` constraint. The picture produced by ICOLA-3 for this picture description is seen in Figure 2.7. For each figure, the actual image produced by ICOLA-3 is on the left side of the vertical bar, and an annotated version of the same image is provided on the right side of the vertical bar.

```
object( box_1, box ).
object( oval_1, oval ).
constraint( left_of, [ box_1, oval_1 ]).
```

Figure 2.6: Simple usage of the `left_of` constraint.

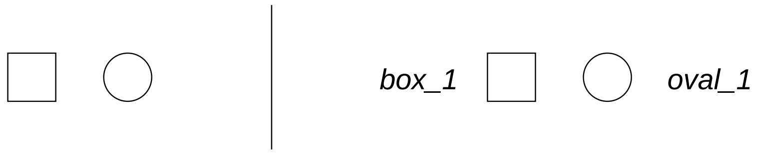


Figure 2.7: A simple image illustrating the `left_of` constraint.

The `horizontal_distance` and `vertical_distance` constraints specify the exact distance (in "units') between coordinate positions of edges of objects. A dis-

Table 2.7: Aliased constraints and their mapping to basic constraints.

Aliased constraint → Basic constraint(s)
`constraint( right_of, [ A, B ]).` →   `constraint( left_of, [ B, A ]).`
`constraint( below, [ A, B ]).` →   `constraint( above, [ B, A ]).`

`constraint( inside, [ A, B ]).` →
 `constraint( left_of, [ left(B), left(A) ]).`
 `constraint( left_of, [ right(A), right(B) ]).`
 `constraint( above, [ top(B), top(A) ]).`
 `constraint( above, [ bottom(A), bottom(B)]).`

`constraint( touching, [ hs(A), hs(B) ]).` →
 `constraint( horizontal_distance, [ 1, hs(A), hs(B) ]).`
 `constraint( above, [ top(A), bottom(B) ]).`
 `constraint( above, [ top(B), bottom(A) ]).`

`constraint( touching, [ vs(A), vs(B) ]).` →
 `constraint( vertical_distance, [ 1, vs(A), vs(B) ]).`
 `constraint( left_of, [ left( A ), right(B) ]).`
 `constraint( left_of, [ left( B ), right(A) ]).`

`constraint( aligned, [ hs(A), hs(B) ]).` →   `constraint( horizontal_distance, [ 0, hs(A), hs(B) ]).`
`constraint( aligned, [ vs(A), vs(B) ]).` →   `constraint( vertical_distance, [ 0, vs(A), vs(B) ]).`

tance value of 0 specifies that the coordinate positions are exactly the same. The `horizontal_distance` constraint:

```
constraint( horizontal_distance, [ Distance, A, B ])
```

also implies that A is to the left of B. The `vertical_distance` constraint behaves the same as the `horizontal_distance` constraint, except it constrains the position of objects along the vertical axis. The picture produced by ICOLA-3 for the picture description in Figure 2.8 is seen in Figure 2.9. The separation between `box_1` and `oval_1` in Figure 2.9 is larger than the separation between those two objects in Figure 2.6 since 100 is larger than the default distance between the objects.

```
object( box_1, box ).
object( oval_1, oval ).
constraint( horizontal_distance, [ 100, box_1, oval_1 ]).
```

Figure 2.8: Simple usage of the `horizontal_distance` constraint.

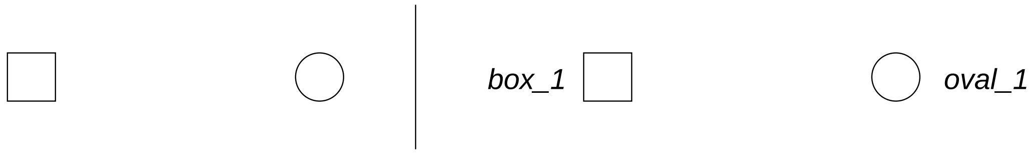


Figure 2.9: A simple image illustrating the `horizontal_distance` constraint.

The `aligned` constraint is used to align the specified edges of two objects. For the `aligned` constraint there are no default selector functions so the horizontal or vertical selectors must be given. Figure 2.11 shows the picture produced by ICOLA-3 for the picture description seen in Figure 2.10.

```
object( box_1, box ).
object( oval_1, oval ).
constraint( above, [ box_1, oval_1 ]).
constraint( aligned, [ right( box_1 ), left( oval_1 ) ]).
```

Figure 2.10: Aligning edges of objects.

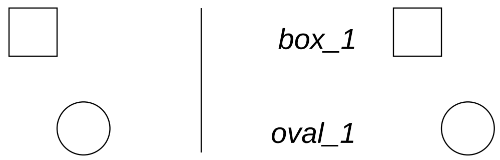


Figure 2.11: A simple image showing the behavior of the `aligned` constraint.

The `touching` constraint specifies that two edges border each other, such that there is no space between the two edges. As with the `aligned` constraint, the selector functions must be explicitly given for the `touching` constraint. Figure 2.13 shows the picture produced by ICOLA-3 for the picture description given in Figure 2.12.

```
object( box_1, box ).
object( oval_1, oval ).
constraint( above, [ box_1, oval_1 ]).
constraint( touching, [ bottom( box_1 ), top( oval_1 ) ]).
```

Figure 2.12: Touching the bottom of `box_1` to the top of `oval_1`.

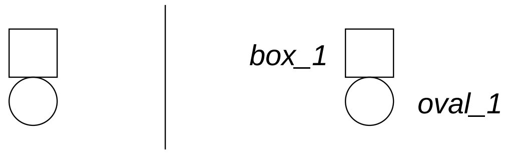


Figure 2.13: The image produced for the picture description in Figure 2.12.

The `inside` constraint is used to place an object of class BB inside the bounding box of an object of class RegBB. Horizontal and vertical selector functions must not be used when specifying the objects of the `inside` constraint. Figure 2.15 shows the picture produced by ICOLA-3 for the picture description in Figure 2.14.

```
object( box_1, box ).
object( oval_1, oval ).
constraint( inside, [ oval_1, box_1 ]).
```

Figure 2.14: Simple usage of the `inside` constraint.

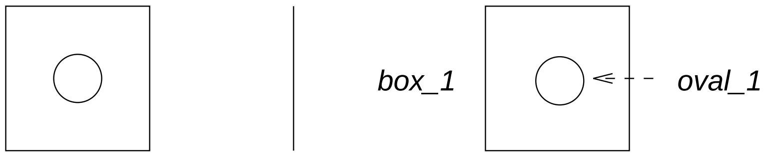


Figure 2.15: A simple example of the results of the `inside` constraint.

The `connects` constraint is used to specify which objects are the endpoints of an object of class Line. The syntax of the `connects` constraint is:

```
constraint( connects, [ Edge, ObjectA, ObjectB ])
```

where edge `Edge` is of class Line and `ObjectA` and `ObjectB` are each of class BB. Selector functions must not be used on `ObjectA` or `ObjectB`. A heuristic positions

Edge on a line between the bounding boxes of ObjectA and ObjectB. If the objects are completely above or below or to the left or right of each other, then the line connects the nearest corner points of the bounding boxes. Otherwise the line connects the midpoints of the nearest sides of the bounding boxes. Figure 2.17 shows the picture produced by ICOLA-3 for the picture description given in Figure 2.16. Figure 2.18 shows the general behavior of the line drawing heuristic for varying positions of objects which are the lines' endpoints.

```
object( box_1, box ).
object( oval_1, oval ).
object( edge_1, line ).
constraint( left_of, [ box_1, oval_1 ]).
constraint( connects, [ edge_1, box_1, oval_1 ]).
```

Figure 2.16: Simple usage of the `connects` constraint.

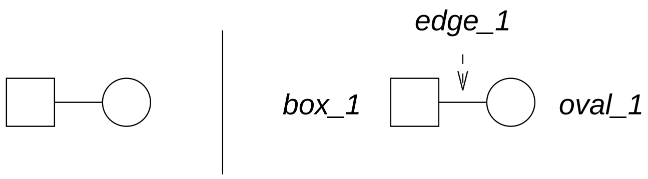


Figure 2.17: A simple image showing the use of the `connects` constraint.

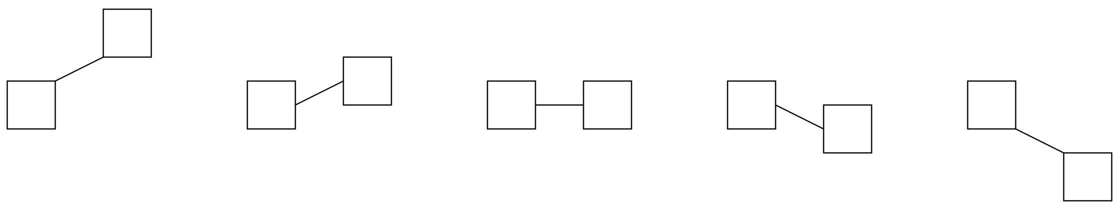


Figure 2.18: Images showing the behavior of the `line` drawing heuristic.

Attributes

In ICOLA, *attributes* are special object properties which can be modified. For example, the colour of an object might be an attribute. For a specific object the colour could be changed by setting that object's colour attribute to a certain value (e.g. 'red'). The implementation of ICOLA-3, however, does not support attributes.

Control Constructs

PDI-3 specifications can also contain a number of *control constructs* to control ICOLA-3's constraint engine. Table 2.8 gives a listing of the supported control constructs and their effects. The most significant of these is the `reset` construct which re-initializes the constraint system, removing from the system all objects and constraints encountered before the `reset`. The `mode_batch` construct selects the "batch" mode constraint solver. This solver delays evaluation of the current set of constraints until a `solve` directive is encountered. The `mode_increment` construct selects the incremental constraint solver which re-computes a new solution after each increment. The `solve` construct instructs the "batch" mode constraint solver to find a solution to the current set of constraints. Control constructs can be ignored when determining the "meaning" of the picture description.

Table 2.8: Control constructs in PDI-3.

Control Construct	Effect
`reset.`	resets the constraint engine.
`mode_batch.`	selects ICOLA-3's "batch" mode constraint solver.
`mode_increment.`	selects ICOLA-3's incremental constraint solver (the default).
`solve.`	forces the batch constraint solver to be run.

2.6.4 Implementation

The graphics and user-interface portions of ICOLA-3 are implemented in SICStus Prolog using the GM (Graphics Manager) extensions. These extensions provide high-level graphics support to Prolog. The constraint engine is implemented in C.

2.6.5 Examples in PDI-3

This section gives a number of larger examples of the usage of PDI-3, and the pictures produced by ICOLA-3. For most examples each figure contains two images separated by a vertical bar. The image on the left is the picture produced by ICOLA-3, and the image on the right is the same image, but annotated with the corresponding object identifiers from the picture description.

Figure 2.19 contains a picture description of a simple label inside a rectangle. The picture produced by ICOLA-3 is found in Figure 2.20.

```
object( box_a, box ).
object( label_a, text, "box a").
constraint( inside, [ label_a, box_a ]).
```

Figure 2.19: PDI-3 listing for a simple label inside a box.

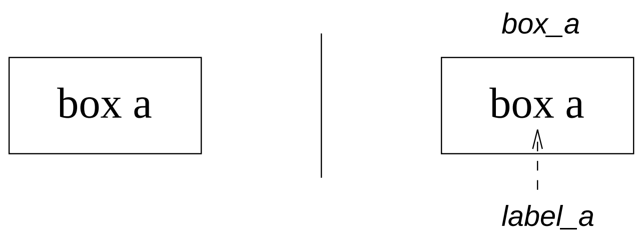


Figure 2.20: Example picture of a simple label inside a box.

Figure 2.21 shows an example PDI-3 listing for an "AND-tree". This example contains a parent goal, and three sub-goals. Figure 2.22 shows the picture created by ICOLA-3.

Figure 2.23 shows a simple example of nested `complex` objects and Figure 2.24 shows the resulting picture. Figure 2.25 shows the use of `line` and `point` objects in the construction of polygons. An annotated version of this figure is not shown.

As can be seen from these examples, the picture descriptions can become quite large, yet their individual components remain simple to understand. Fortunately, PDI is only an intermediate language and not meant to be input directly by the user. In the Wand framework, for example, the implementation of a Wand tool

```
object( goal, box ).
object( goal_text, text, "Goal 1" ).
constraint( inside, [ goal_text, goal ]).
object( subgoal1, oval ).
object( subgoal2, oval ).
object( subgoal3, oval ).
constraint( below, [ subgoal1, goal ]).
constraint( below, [ subgoal2, goal ]).
constraint( below, [ subgoal3, goal ]).
constraint( left_of, [ subgoal1, subgoal2 ]).
constraint( left_of, [ subgoal2, subgoal3 ]).
object( line1, line ).
object( line2, line ).
object( line3, line ).
constraint( connects, [ line1, goal, subgoal1 ]).
constraint( connects, [ line2, goal, subgoal2 ]).
constraint( connects, [ line3, goal, subgoal3 ]).
```

Figure 2.21: A PDI-3 listing for an "AND-tree".

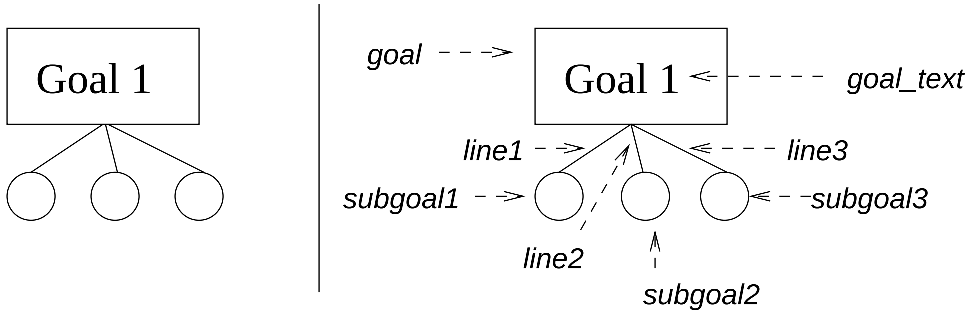


Figure 2.22: Example picture of an "AND-tree".

will hide PDI from the user. The user will instead be working with the "display description language" discussed in Section 2.6.2. For testing and demonstration purposes, however, PDI specifications are generated by hand. It would also be straightforward to create a simple tool in Prolog to automate the generation of PDI descriptions.

```
object( collection, complex ).
      object( box_1, box ).
      object( box_2, box ).
      constraint( below, [ box_2, box_1 ]).
      object( ovals, complex ).
            object( oval_1, oval ).
            object( oval_2, oval ).
            constraint( left_of, [ oval_1, oval_2 ]).
      end_complex.
      constraint( left_of, [ box_2, ovals ]).
      constraint( above, [ box_2, ovals ]).
end_complex.
object( oval_3, oval ).
constraint( right_of, [ oval_3, collection ]).
```

Figure 2.23: Example PDI-3 showing use of complex objects.

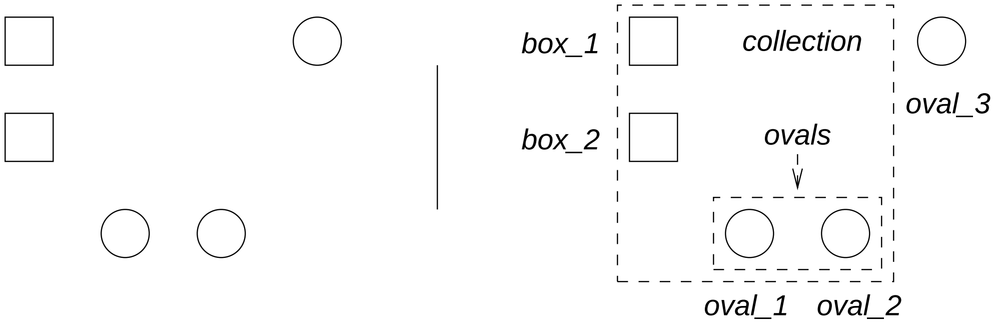


Figure 2.24: Objects grouped via a `complex` object.

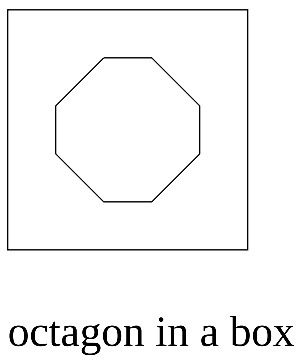


Figure 2.25: Example picture of a polygon.

2.6.6 ICOLA-3's Solution Strategy

In ICOLA-3 there is a single algorithm which is called in both incremental and batch modes. ICOLA-3's constraint system is encoded internally as two graphs. All objects are represented by nodes in these two graphs. The first graph represents the `left_of` relationship, and contains two nodes corresponding to the left and right sides of each object's bounding box. The second graph represents the `above` relationship, and contains two nodes which correspond to the top and bottom sides of each object's bounding box.

Constraints are encoded as arcs in the graphs. As each new object or constraint is added, corresponding nodes and/or arcs are added to the graphs. The absolute positions of the objects are computed by a breadth-first-like traversal (similar to the CPM (Critical Path Method) algorithm in project planning) through the graphs. Since the constraint system is typically severely under-constrained, ICOLA-3 relies on heuristics to find a solution. The two main heuristics of ICOLA-3 are that pictures will be drawn as compactly as possible, and all objects will be positioned as close as possible to the top and left edges of the picture. Over-constrained systems are not allowed. Any constraint which would cause such an inconsistency is reported to the user, and is not allowed to affect the constraint system. Inconsistencies are found by detecting cycles in the graphs. Attributes, as mentioned before, are not implemented. Oster and Kusalik [OsKu94] give the full details of ICOLA-3's constraint algorithm.

2.6.7 Deficiencies of ICOLA-3

ICOLA-3 has a number of deficiencies which are addressed in this thesis research. ICOLA-3 does not have a complete implementation of the PDI language. The original specification of the PDI language defined the objects `bitmap`, `triangle`, and `arc` but these were not implemented. It also specified object attributes (e.g. `colour` and `style`) which were also not implemented.

Once a complex (grouping) object is defined, there is no way to add any further objects or constraints to the group. This restricts the use of complex objects as it requires the entire contents of each complex object to be known when the complex object is defined.

As mentioned in Section 2.6.1, the solution strategy of ICOLA-1 and ICOLA-2 was purely incremental and worked quite well. ICOLA-3, which produces better pictures than its predecessors, suffers from the problem that its algorithm must update the entire constraint graph for each increment. This problem is analogous to having to do global constraint propagation instead of local constraint propagation. ICOLA-3, however, is still incremental, in the sense that it can find a solution after each increment without having to re-build the constraint graph. As well, the consistency check is incremental, and relies on the consistency previous solution.

The default behavior of the constraint algorithm of ICOLA is for the pictures it produces to be reasonably compact. To achieve this behavior in ICOLA-3, the heuristics used are that objects will tend to be small, and positioned as close as possible to the left and top edges of the picture. These two heuristics, however, are found to be in conflict for some picture specifications. The problem is that neither heuristic has priority, and certain situations can arise where there is no guarantee of a unique solution unless the heuristics are prioritized. Figures 2.26 and 2.27 show the two interpretations of a simple picture description which specifies that the right side of `oval_1` will be to the left of the right side of `box_1`. Figure 2.26 shows a picture where the heuristic that objects will be positioned as far left as possible is the dominant heuristic. Figure 2.27 shows a picture where the heuristic that objects will tend to be small is paramount. Since there is no guarantee in ICOLA-3 as to which heuristic has priority and since the heuristics are in conflict there is no way to guarantee the uniqueness of an image produced by a picture description.

Aliased constraints in ICOLA-3 are converted by macro expansion into one or more basic constraints (See Section 2.6.3). In ICOLA-3, these basic constraints are considered individually and independently of each other. This means that the inconsistency of one of the basic constraints will not affect the other basic constraints,

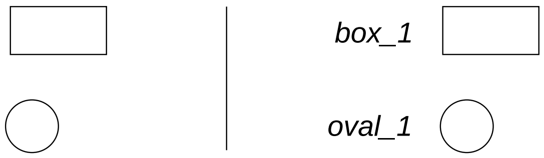


Figure 2.26: A picture showing conflicting heuristics – one interpretation.

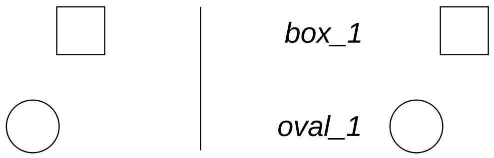


Figure 2.27: A picture showing conflicting heuristics – another interpretation.

and will not necessarily affect the consistency of the aliased constraint. The result is often partially enforced aliased constraints if one or more of the basic constraints are inconsistent.

2.7 Chapter Summary

It is seen in this chapter that ICOLA, despite a few shortcomings in the implementation of ICOLA-3, has the potential to be a powerful part of the Wand framework. It is with this potential in mind that this thesis work develops and expands ICOLA-3 to the new version, ICOLA-4. This new version and the results of the implementation are discussed in the next chapters.

CHAPTER 3

ICOLA – THE PICTURE DESCRIPTION LANGUAGE

This chapter presents the declarative picture description language, PDI (Picture Descriptions for ICOLA), in the new version of ICOLA, ICOLA-4. As with the previous versions of ICOLA, ICOLA-4's PDI language consists of objects, constraints, attributes, and control constructs. Pictures are easily specified with relative positioning constraints – absolute coordinate positions for the objects are not required. Both the incremental insertion and the incremental *deletion* of objects and constraints are supported ICOLA-4. This chapter begins with a summary of the differences between ICOLA-3 and ICOLA-4 in terms of the languages they support and in their behavior. A complete description of PDI-4 (PDI for ICOLA-4) is then presented. The chapter ends with a discussion of an operational semantics for PDI-4 and the behavior of ICOLA-4.

3.1 Changes from ICOLA-3

This section summarizes the differences between ICOLA-3 and ICOLA-4, in terms of the supported constructs, format, usage of their respective PDI languages, and the layout heuristics of the constraint solvers.

1. The most significant difference between the ICOLA-3 and ICOLA-4 is that ICOLA-4 has a new incremental algorithm which produces better and more

consistent images than ICOLA-3's algorithm. The incremental algorithm in ICOLA-4 is also faster than the one in ICOLA-3.

2. While ICOLA-3 supports only incremental insertions, ICOLA-4 allows both incremental insertions and incremental *deletions* of objects and constraints.

3. ICOLA-4 supports all the objects in the original specification of PDI. This includes the `bitmap`, `arc`, and `triangle` objects not implemented in ICOLA-3.

4. Attributes, which are not present in the implementation of ICOLA-3, are fully implemented in ICOLA-4. The supported attributes include `colour`, `visible`, `bitmap`, `text`, `style`, `rotation`, `arrowhead`, and `filled`. Table 3.10 gives a list of the attributes. The addition of attributes also results in a change to the way the `text` object is specified.

5. PDI-4 also allows the specification of attributes which affect the entire image, like `default_colour` and `scale`. Table 3.11 gives a complete list of the these attributes with a brief description of each.

6. In PDI-3, the `horizontal_distance` and `vertical_distance` constraints indicate a *fixed* distance between edges of objects. In PDI for ICOLA-4, these constraints indicate the *minimum* distance between edges of objects. This change allows PDI-4 to be less dependent on absolute positioning, and further supports the emphasis on relative positioning.

7. As discussed in Section 2.6.7, aliased constraints in ICOLA-3 are treated as macros in the implementation, and the basic constraints which comprise the aliased constraint are handled one at a time. ICOLA-4 ensures the consistency of an aliased constraint before any of its constituent basic constraints are enforced, and the problems encountered in ICOLA-3 with partially enforced aliased constraints do not appear in ICOLA-4.

8. The basic constraints which comprise the aliased constraint `touching` have been changed to provide a better definition of "touching". Table 3.7 shows the new definition.

9. Scoping of object names in PDI-3 is strictly global. In PDI-4, the naming convention adopted allows both local and global scoping of object names. The new naming convention is discussed in Section 3.3.

10. In PDI-3, `complex` objects cannot be modified once they are defined. In PDI-4, `complex` objects can be easily modified, either by adding subsequent objects and constraints to the group, or by deleting objects or constraints from the group. The syntax of the declaration of a `complex` object in PDI-4 has also changed from PDI-3.

11. The layout heuristic used in ICOLA-3 is that objects will tend to be small and positioned as far left as possible (similarly as close as possible to the top edge of the layout). In ICOLA-4, the heuristics are (in order of priority):

    (a) An object's right edge will be as far left as possible. Similarly, an object's bottom edge will be as close as possible to the top edge of the layout.

    (b) Objects will tend to be as small as possible.

    The difference in the heuristics between ICOLA-3 and ICOLA-4 is subtle, but the effect is significant.

As will be seen in the following sections, ICOLA-4's PDI language is more complete, more consistent, and more powerful than the PDI language in ICOLA-3. As well, the next two chapters will show ICOLA-4's constraint solution algorithm to be much better than the constraint algorithm in ICOLA-3.

3.2 The Coordinate System

The coordinate system used in ICOLA-4 is seen in Figure 3.1. The origin of the display is the upper left-hand corner. The maximum x and y coordinate values are theoretically infinite.

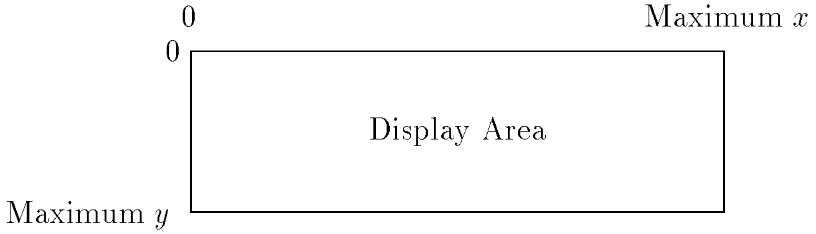


Figure 3.1: The coordinate system used in ICOLA-4.

3.3 Object Classes

The PDI language for ICOLA-4 is based on the object classes described in Table 3.1 and the object hierarchy found in Table 3.2. Table 3.3 lists the nine objects with a brief description of each. All language constructs in PDI are in the form of valid ground Prolog terms. All "white space" and Prolog comments in a picture description are ignored.

Before an object can be referenced in a picture description it must be given a unique name in the current (local) scope and a type. An object's name in the local scope is represented by a Prolog atom. If an object's local name is used in a constraint then the object must exist at the current scope for that constraint. An object's canonical name consists of a leading "/", followed by the name of the scope of the object, followed by the local name of the object. For example, `/group_1/box_a` is the canonical name of `box_a` whose scope is `group_1`. A "/" by itself represents the entire picture. A leading "/" indicates a canonical name, and each subsequent "/" in the name is a separator. A trailing separator, as in `/group_a/group_b/`, is not allowed. Only canonical names may contain "/" separators. If a canonical name is used, the specified object must exist at the specified scope.

Table 3.1: Object class descriptions for PDI-4.

OBJ	the parent class "object".
BB	objects having a bounding box.
REGBB	objects having a regular bounding box. Other objects can be placed inside these objects, and the bounding box will increase in size to accommodate them.
SPECIALBB	objects having a special bounding box. These objects either have a fixed image, or are the special COMPLEX objects.
FIXED	objects with a fixed image, including bitmap images and text.
RECTBB	objects having a strictly rectangular shaped, and a (rectangular) bounding box.
NONRECTBB	objects having a non-rectangular shape, but which are given a rectangular bounding box.
POINT	objects which are represented by a single "dot".
BITMAP	objects whose images are defined by bitmaps.
COMPLEX	objects used for grouping other objects together.
LINE	objects which are either lines or arcs.
TEXT	objects which are text strings. For a given text string, the text image is fixed.

All objects except those of type `complex` are created using the syntax:

`object(` ObjectName, **ObjectType** `)`

where ObjectName is the local name of the object and **ObjectType** is an object type (from Table 3.3) other than type `complex`. ObjectName must not be a canonical name.

Complex objects (groups) are created with the following special syntax:

```
group( ObjectName ).
    body_of_complex_object
end_group( ObjectName ).
```

where body_of_complex_object is replaced with the various objects, constraints, and attributes that comprise the complex object or group. ObjectName is the local name

Table 3.2: Object classes for PDI-4.

OBJ	=	BB $\bigcup$ LINE
	=	{ `box`, `oval`, `triangle`, `bitmap`, `point`, `text`, `complex`, `line`, `arc` }
BB	=	REGBB $\bigcup$ SPECIALBB
	=	{ `box`, `oval`, `triangle`, `bitmap`, `point`, `text`, `complex` }
REGBB	=	RECTBB $\bigcup$ NONRECTBB
	=	{ `box`, `oval`, `triangle` }
SPECIALBB	=	FIXED $\bigcup$ COMPLEX
	=	{ `bitmap`, `point`, `text`, `complex` }
FIXED	=	BITMAP $\bigcup$ POINT $\bigcup$ TEXT
	=	{ `bitmap`, `point`, `text` }
RECTBB	=	{ `box` }
NONRECTBB	=	{ `oval`, `triangle` }
POINT	=	{ `point` }
BITMAP	=	{ `bitmap` }
COMPLEX	=	{ `complex` }
LINE	=	{ `line`, `arc` }
TEXT	=	{ `text` }

Table 3.3: Descriptions for the objects in PDI-4.

Object Type	Description
`box`	an extensible rectangle.
`oval`	an extensible ellipse.
`point`	a single "dot".
`line`	a straight line between two objects.
`triangle`	an extensible triangle.
`bitmap`	a user-defined bitmap.
`arc`	a curved arc between two objects.
`complex`	a "grouping" object. It can be used to combine a number of objects. Any operation or constraint applied to a complex object applies to all members of the complex object as a unit.
`text`	a "text string".

of the object, and must not be a canonical name. This syntax is new in ICOLA-4 and is a cosmetic replacement of the old `complex` object declaration in ICOLA-3. Complex objects may be nested. If complex object A is contained within complex object B, then the scope of A must be ended before the scope of B is ended.

A new feature in ICOLA-4 is the ability to add to the contents of a previously defined complex object. This is done by re-entering the scope of the complex object to be changed. Figure 3.2 shows an example of the addition of a new `box` object called `box_1` to the group `group_b` which is within group `group_a`.

```
group( group_a ).
      ⋮
      group( group_b ).
            ⋮
      end_group( group_b ).
      ⋮
end_group( group_a ).
      ⋮
group( group_a ).
      group( group_b ).
            object( box_1, box ).
      end_group( group_b ).
end_group( group_a ).
```

Figure 3.2: Incrementally changing the contents of a `complex` object.

The object "/", i.e. the entire picture, is a `complex` object which automatically contains all other objects. "/" can be thought of as the "root" (or "primary") picture.

Objects of type `bitmap` and `text` produce no image when initially declared. In ICOLA-3, the text for a `text` object was specified in the declaration of the object (See Section 2.6.3). In ICOLA-4, the `text` attribute of the `text` object is used to specify the text string. Similarly, the `bitmap` attribute of the `bitmap` object is used to specify the contents of the bitmap image. The full description of these attributes is in Section 3.6.

3.3.1 Selector Functions

Selector functions allow the explicit specification of the left, right, top, or bottom sides of an object's bounding box. Selector functions can be used with any object in class BB. There are two horizontal (`left` and `right`) and two vertical (`top` and `bottom`) selector functions. The left side of the bounding box of object A is specified by `left(A)`. Similarly, the right, top, and bottom sides of the bounding box are specified by `right(A)`, `top(A)`, and `bottom(A)`, respectively.

3.4 Constraints

Constraints in PDI specify positional relationships among the objects. Table 3.4 gives a list of the constraints with a brief description for each. The constraints in ICOLA-4 are divided into two categories – *basic* and *aliased*. *Basic* constraints are the fundamental constraints in the PDI language. *Aliased* constraints map to one or more basic constraints. The basic constraints and their general usage are listed in Table 3.5, where A and B are objects of class BB, Distance is a non-negative integer value, and Edge is an object of class LINE. For convenience, ICOLA provides default selector functions when object names are given instead of selector functions applied to object names. For a given object which does not have a selector function in a constraint, a default selector function is chosen so that the constraint applies to the entire object. For instance, in the constraint

```
constraint( left_of, [ left(A), B ])
```

the default selector function chosen for B is `left(B)` so that the `left_of` constraint applies to all of object B. After this default is applied the constraint is:

```
constraint( left_of, [ left(A), left(B) ]).
```

The default selector functions for the basic constraints are shown in Table 3.6.

Table 3.7 shows the aliased constraints and their mapping to basic constraints. Since all aliased constraints can be represented by basic constraints, aliased constraints are not needed for the full functionality of the system. They do, however,

Table 3.4: Constraints in PDI-4.

Constraint	Description
`left_of`	one object is to the left of another.
`right_of`	one object is to the right of another.
`horizontal_distance`	`left_of` by at least a specified amount.
`above`	one object is above another.
`below`	one object is below another.
`vertical_distance`	`above` by at least a specified amount.
`connects`	two object are connected by a `line` or `arc`.
`inside`	one object is contained inside another.
`touching`	one object borders on another in a specified axis, such that the objects have no space between them.
`aligned`	align the specified edges of two objects so that they share a common horizontal or vertical axis.

Table 3.5: Basic Constraints in PDI-4.

Constraint	Usage
`left_of`	`constraint( left_of, [ hs(A), hs(B) ]).`
`above`	`constraint( above, [ vs(A ), vs(B) ]).`
`horizontal_distance`	`constraint( horizontal_distance, [ Distance, hs(A), hs(B) ]).`
`vertical_distance`	`constraint( vertical_distance, [ Distance, vs(A), vs(B) ]).`
`connects`	`constraint( connects, [ Edge, A, B ]).`

make specifying certain relationships (like inside and touching) much easier. Note that aliased constraints are converted to basic constraints before the default selector functions are applied.

In ICOLA-3, the `horizontal_distance` and `vertical_distance` constraints specify a *fixed* distance (in "units") between two objects. In ICOLA-4, these constraints specify the *minimum* distance between objects. This change allows a simpler constraint solution algorithm which still provides layout capabilities similar to those found in ICOLA-3. As in ICOLA-3, a distance value of 0 specifies that the coordi-

Table 3.6: Constraints in PDI-4 and their default behavior in ICOLA-4.

Constraint → Default behavior.
`constraint( left_of, [ A, B ]). →` `      constraint( left_of, [ right(A), left(B) ]).`
`constraint( above, [ A, B ]). →` `      constraint( above, [ bottom(A), top(B) ]).`

`constraint( horizontal_distance, [ D, A, B ]). →`
`      constraint( horizontal_distance,`
`            [ D, right(A), left(B) ]).`

`constraint( vertical_distance, [ D, A, B ]). →`
`      constraint( vertical_distance,`
`            [ D, bottom(A), top(B) ]).`

nate positions of the edges of the objects are the same. As well, "units" in ICOLA-4 are simply "pixels".

In ICOLA-3 the PDI language handled only incremental insertions of objects and constraints. ICOLA-4 can handle both incremental insertions and incremental *deletions* of objects and constraints. Insertion of objects and constraints can either be implicit, as in previous examples, or explicit, using a new `insert` operator. For example:

```
insert object( box_1, box )
```

is equivalent to:

```
object( box_1, box ).
```

The following declaration is also a valid insertion of an object, and is equivalent to the above examples:

```
insert( object( box_1, box ) ).
```

The same syntactic conventions are also used for the insertion of constraints.

Table 3.7: Aliased constraints in PDI-4 and the basic constraints they map to.

Aliased constraint → Basic constraint(s)
`constraint( right_of, [ A, B ]).` →   `constraint( left_of, [ B, A ]).`
`constraint( below, [ A, B ]).` →   `constraint( above, [ B, A ]).`

`constraint( inside, [ A, B ]).` →
 `constraint( left_of, [ left(B), left(A) ]).`
 `constraint( left_of, [ right(A), right(B)]).`
 `constraint( above, [ top(B), top(A) ]).`
 `constraint( above, [ bottom(A), bottom(B) ]).`

`constraint( touching, [` **hs**`(A),` **hs**`(B) ]).` →
 `constraint( horizontal_distance, [ 1,` **hs**`(A),` **hs**`(B) ]).`
 `constraint( vertical_distance, [ 0, top(A), bottom(B) ]).`
 `constraint( vertical_distance, [ 0, top(B), bottom(A) ]).`

`constraint( touching, [` **vs**`(A),` **vs**`(B) ]).` →
 `constraint( vertical_distance, [ 1,` **vs**`(A),` **vs**`(B) ]).`
 `constraint( horizontal_distance, [ 0, left(A), right(B) ]).`
 `constraint( horizontal_distance, [ 0, left(B), right(A) ]).`

`constraint( aligned, [` **hs**`(A),` **hs**`(B) ]).` →   `constraint( horizontal_distance, [ 0,` **hs**`(A),` **hs**`(B) ]).`
`constraint( aligned, [` **vs**`(A),` **vs**`(B) ]).` →   `constraint( vertical_distance, [ 0,` **vs**`(A),` **vs**`(B) ]).`

3.4.1 Examples

The picture description in Figure 3.3 instructs ICOLA-4 to produce the image seen in Figure 3.4. The re-use of local object names, the changing of the contents of a group object, the use of canonical names, and the use of a triangle object are all features supported in ICOLA-4 but not supported in ICOLA-3.

3.5 Deleting Objects and Constraints

The syntax for object and constraint deletion is similar to the syntax for their insertion. To incrementally delete an object or constraint, however, that object or constraint must have been previously inserted, and the deletion must be done at the

```
group( outer_bunch ).
     object( box_1, box ).
     object( box_2, box ).
     constraint( below, [ box_2, box_1 ]).
     constraint( left_of, [ box_2, box_1 ]).
     object( edge_1, line ).
     constraint( connects, [ edge_1, box_1, box_2 ]).
     group( inner_bunch ).
          object( oval_1, oval ).
          object( oval_2, oval ).
     end_group( inner_bunch ).
     constraint( inside, [ inner_bunch, box_1 ]).
end_group( outer_bunch ).
constraint( left_of, [  /outer_bunch/inner_bunch/oval_1,
                        /outer_bunch/inner_bunch/oval_2 ]).
object( oval_3, oval ).
constraint( left_of, [ oval_3, outer_bunch ]).
group( outer_bunch ).
     object( triangle_1, triangle ).
     constraint( inside, [ triangle_1, box_2 ]).
end_group( outer_bunch ).
```

Figure 3.3: PDI-4 listing showing various constraints and nested complex objects.

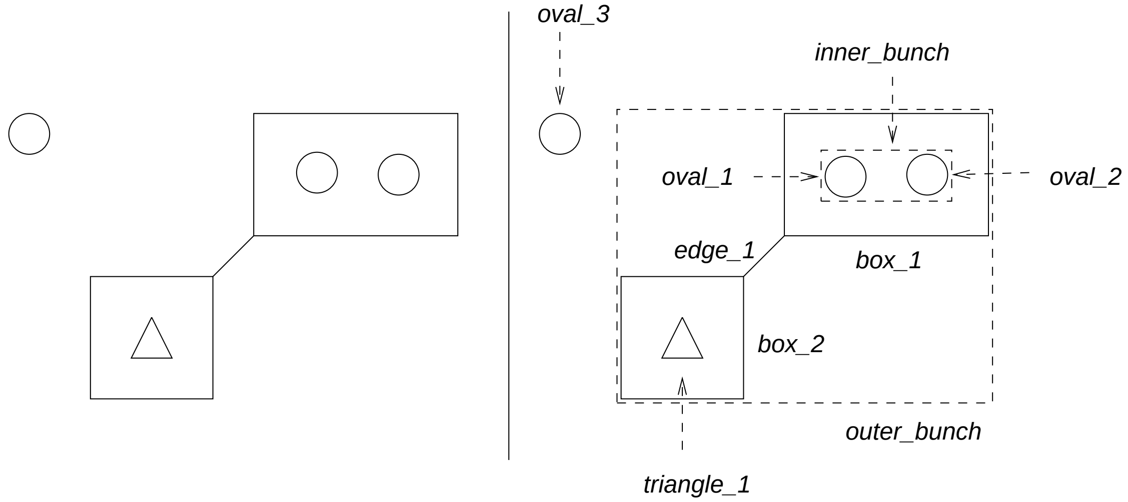


Figure 3.4: Image produced from the picture description in Figure 3.3.

same scope as the insertion. As with inserting objects, the deletion:

```
delete object( box_1, box )
```

is equivalent to:

```
delete( object( box_1, box ) ).
```

It is required that all constraints associated with the object `box_1` are deleted before the object `box_1` is deleted.

For constraints, defaults are applied to deletions before the constraint is removed. For example, the deletion:

```
delete constraint( left_of, [ box_1, box_2 ])
```

will cause the removal of the constraint

```
constraint( left_of, [ right( box_1 ), left( box_2 ) ])
```

which must have been previously inserted. A basic constraint added as the result of adding an aliased constraint can also be explicitly deleted. A subsequent deletion of that aliased constraint, however, will fail to find the deleted basic constraint, and the deletion of that aliased constraint will fail.

To delete a complex object, only

```
delete group( ObjectName )
```

is necessary. It is required, however, that all constituent objects and constraints inside and related to ObjectName are explicitly deleted prior to the group ObjectName being deleted.

A design goal of the constraint solution system is that the image produced after a deletion will be exactly the same as the image produced if the original insertion and deletion were never used in the PDI listing. The picture description in Figure 3.5 is the same as Figure 3.6, except that the `aligned` constraint insertion and deletions are removed in Figure 3.6. Both PDI listings, however, yield exactly the same final picture (Figure 3.7). As will be seen in Chapter 4 there is only one image which satisfies the heuristics and the constraints, and so the above design goal has been achieved.

```
object( box_1, box ).
object( oval_1, oval ).
constraint( aligned, [ bottom( box_1 ), top( oval_1 ) ]).
constraint( left_of, [ box_1, oval_1 ]).
delete constraint( aligned, [ bottom( box_1 ), top( oval_1 ) ]).
```

Figure 3.5: Example picture description with a simple constraint deletion.

```
object( box_1, box ).
object( oval_1, oval ).
constraint( left_of, [ box_1, oval_1 ]).
```

Figure 3.6: Example picture description with the constraint insertion and the constraint deletion removed.

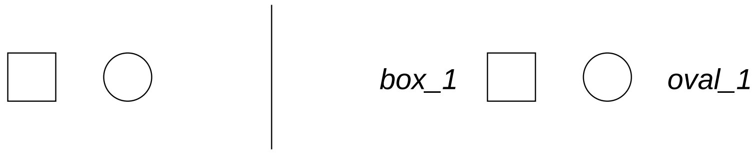


Figure 3.7: Picture of simple use of the `left_of` constraint.

3.6 Attributes

In ICOLA, *attributes* are special object properties that can be modified. The colour of an object, for example, might be an attribute. The picture description could specify a colour for that object by setting its colour attribute to a certain value (e.g. 'red'). Table 3.8 lists the attributes supported by ICOLA-4, along with a brief description each. The attributes, their possible settings, and their default settings are found in Table 3.9.

The default colour of an object is determined by the `default_colour` attribute of the entire picture (See Section 3.6.1). The value of the `curve` attribute can be any positive or negative integer. Positive values indicate a clockwise curvature, and negative values indicate a counter-clockwise curvature. Figure 3.8 contains a picture description of simple arcs with differing amounts of curvature. Figure 3.9 shows the resulting image.

Table 3.8: Attributes in PDI-4.

Attribute	Description
`colour`	the colour of the object.
`visible`	indicates whether an object is displayed or not.
`filled`	indicates whether an object of class REGBB is filled in with colour (i.e. solid).
`bitmap`	the filename of a graphical bitmap.
`text`	a string of text.
`arrowhead`	indicates if the object of class LINE has an arrowhead at either or both of its endpoints.
`style`	indicates the style of an object of class LINE.
`curve`	the amount of curvature of an `arc`.
`rotation`	the orientation of a `triangle` object.

Table 3.9: Attribute values.

Attribute	Possible Values	Default Value
`colour`	any X11 colour	`default_colour`
`visible`	`true` or `false`	`true`
`filled`	`true` or `false`	`false`
`bitmap`	bitmap filename	no filename (“”)
`text`	text string	an empty string (“”)
`arrowhead`	`none`, `head`, `tail`, or `both`	`none`
`style`	`regular`, `dashed`, or `bold`	`regular`
`curve`	positive or negative integer	5
`rotation`	`0`, `90`, `180`, or `270`	`0`

Table 3.10 shows the object classes and associated attributes. Note that complex objects do not have attributes in ICOLA-4. There are inherent ambiguities with having attributes for complex objects since the complex object is simply a means of collecting a group of objects and positioning them as a single entity. Since a complex object has no image of its own, setting a `colour` attribute, for example, would only be appropriate for setting the colour of member objects. Resolution of conflicts, however, between the colour of a complex object and the colour of a member object

```
object( point_1, point ).
object( point_2, point ).
constraint( horizontal_distance, [ 100, point_1, point_2 ]).
object( arc_1, arc ).
object( arc_2, arc ).
constraint( connects, [ arc_1, point_1, point_2 ]).
constraint( connects, [ arc_2, point_1, point_2 ]).
attribute( arc_1, curve, 7 ).
attribute( arc_2, curve, -15 ).
attribute( arc_1, arrowhead, head ).
attribute( arc_2, arrowhead, head ).
```

Figure 3.8: Defining the curvature of `arc` objects.

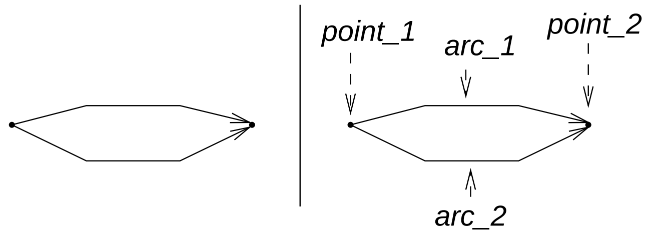


Figure 3.9: Picture of arcs with distinct curves.

is difficult unless arbitrary restrictions are placed on the effects of changing the `colour` attribute. The `visible` attribute could also be used for complex objects, but would have problems similar to those encountered with the `colour` attribute.

Table 3.10: Objects and their associated attributes.

Object	Attributes
`box`	`colour`, `visible`, and `filled`.
`oval`	`colour`, `visible`, and `filled`.
`point`	`colour` and `visible`.
`triangle`	`colour`, `visible`, `filled`, and `rotation`.
`bitmap`	`colour`, `visible`, and `bitmap`.
`text`	`colour`, `visible`, and `text`.
`line`	`colour`, `visible`, `style`, and `arrowhead`.
`arc`	`colour`, `visible`, `style`, `arrowhead`, and `curve`.

The following is the format for specifying attributes:

attribute(ObjectName, **AttributeName**, AttributeValue)

where ObjectName is the name of the object involved, **AttributeName** is a valid attribute for that type of object, and AttributeValue is a valid value for that attribute. ObjectName can be a canonical name or a local name.

Figure 3.10 contains a PDI-4 listing showing the simple use of attributes. Figure 3.11 shows the resulting picture.

```
object( box_1, box ).
object( oval_1, oval ).
attribute( oval_1, filled, true ).
constraint( horizontal_distance, [ 40, box_1, oval_1 ]).
object( line_1, line ).
constraint( connects, [ line_1, box_1, oval_1 ]).
attribute( line_1, arrowhead, both ).
attribute( line_1, style, dashed ).
```

Figure 3.10: Example showing simple use of attributes.

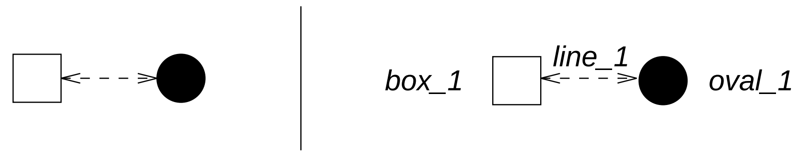


Figure 3.11: The picture resulting from the picture description in Figure 3.10.

Attributes cannot be deleted in a PDI description: they can only be changed. Initial values for all attributes of a given object are assigned when that object is created. The effect of an increment which changes an attribute is that the new attribute value replaces the old one.

3.6.1 Other Attributes

The attributes for the "root" object are found in Table 3.11. The `default_distance` attribute in Table 3.11 refers to the default size of objects, and the default distance between objects if no minimum size is otherwise specified. The default distance value must be a non-negative integer, and is specified in "units". The "scale factor" specifies the scale used to draw the image. A "scale factor" of 1.0 means that the internal "units" directly map to pixels in the resulting image. Scale values less than 1.0 indicate smaller images, and the absolute pixel distances on the drawn image will be less than the internal "unit" distances. Scale values greater than 1.0 indicate larger images, with one internal "unit" distance corresponding to more than one pixel.

Table 3.11: Special attributes of the special "root" object.

Attribute	Description	Default Value
`default_colour`	the default colour for all the objects.	black
`scale`	the "scale factor" for the entire picture.	1.0
`default_distance`	the "default distance" value.	20

3.7 Control Constructs

PDI-4 specifications can also contain a number of *control constructs* to control ICOLA-4's constraint engine. Table 3.12 gives a listing of the supported control constructs in ICOLA-4 and a brief summary of their effects.

Table 3.12: Control constructs in PDI-4.

Control Construct	Effect
`reset.`	resets the constraint engine.
`mode_batch.`	selects ICOLA's "batch" mode constraint solver.
`mode_increment.`	selects ICOLA's incremental constraint solver (the default).
`solve.`	forces the batch constraint solver to be run.

The `reset` construct re-initializes the constraint system, removing from the system all objects and constraints encountered since the previous `reset`.

The `mode_batch` construct selects the batch (non-incremental) mode of the constraint solver. In "batch" mode the constraint solver checks the consistency of each new constraint (relative to the current set of objects and constraints at that point), but delays finding a solution to the current set of constraints until a `solve` directive is encountered.

The `mode_increment` construct selects the incremental mode (the default) of the constraint solver. In incremental mode, a solution is found to the current set of constraints after each increment.

A picture description may be solved in "batch" mode, incremental mode, or an arbitrary combination of the two. The `mode_batch` and `mode_increment` constructs switch between these two modes of operation. If the constraint solver is in "batch" mode when the `mode_increment` directive is encountered then there is no guarantee that the current solution corresponds to the current set of constraints. To ensure the incremental solver has a correctly updated solution with which to work, the `mode_increment` construct also results in a call to the batch solver.

The `solve` directive instructs the `batch` solver to determine a solution to the current set of constraints. Note that only the batch solver is invoked with the `solve` construct – the incremental solver runs only as necessary and cannot be invoked using a `solve`.

The control constructs can be ignored when determining the "meaning" of the picture description.

3.8 Examples in PDI-4

This section presents a number of example pictures produced using the PDI constructs in ICOLA-4. Figure 3.12 shows a picture of a simple tree. If solid lines represent successful branches, dashed lines represent failed branches, and bold lines represent a the current branch, then this tree could be a simple representation of an

"OR-trace" of a Prolog computation. While ICOLA-3 could produce the relatively simple layout of the nodes of this tree, the various edge styles can only be produced by ICOLA-4. Appendix A contains the PDI-4 listing for this image.

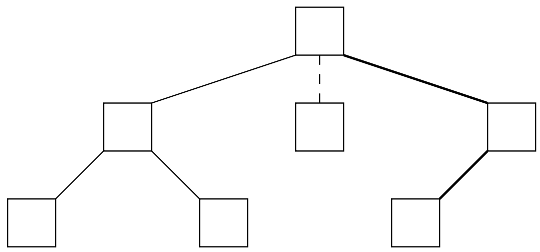


Figure 3.12: An simple "OR-trace" of a Prolog computation.

Figure 3.13 is a simple flow diagram illustrating the use of various objects, constraints, and attributes. Of importance in this diagram is the use of filled objects (the triangle), arrowheads (on the lines), and the `bitmap` object (the "happy face"). Appendix A also contains the PDI-4 listing for this image.

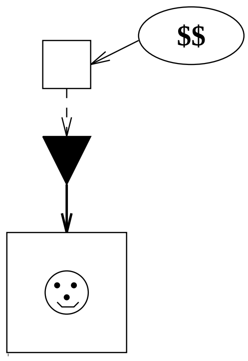


Figure 3.13: A simple flow diagram produced by a picture description.

3.9 Converting PDI to Graphics Primitives

ICOLA-4 must convert the picture descriptions to graphics primitives before an image can be produced. The PDI input stream, as shown in Figure 3.14, goes through a pre-processing phase where it is divided into two distinct data streams. The first stream contains the variables and constraints. These variables and constraints are used by the incremental constraint solver to determine the absolute positions of all the objects. The absolute positions of all the objects are then passed to the post-processor. The second stream is comprised of abstract object descriptions which contain non-positional information about the objects (e.g. attribute information). The post-processing phase merges the contents of the two streams and produces the graphics function calls which draw the resulting picture.

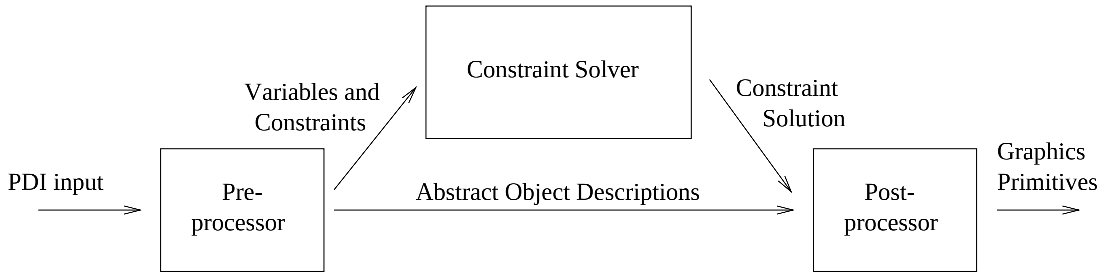


Figure 3.14: The high-level internal operation of ICOLA-4.

The pre-processing phase ensures that the input is syntactically well-formed and valid. The syntactic specification for PDI-4 results in some validity tests which might otherwise be performed at a semantic checking level. For example, the syntax checker is used to detect illegal use of constraints such as attempting to place a `box` inside a `line`, or specifying an alternate size for a `point` object. As well, a constraint such as

```
constraint( left_of, [ left( a ), top( b ) ])
```

is not allowed to pass through the pre-processing phase as it contains an improper selector function for the constraint (See Table 3.5). Semantic analysis at a later phase could detect these kind of problems. It is advantageous, however, to catch them earlier as incorrectly formed statements. The pre-processor also applies all default selector functions, assigns a unique name to each object, and ensures the existence of referenced objects. As well, the pre-processor is responsible for verifying that the class of an object being deleted matches the class of the object when it was inserted. This is to prevent, for example,

```
delete object( node_1, oval )
```

causing the removal of an object called `node_1` which is of a different type.

The pre-processing phase also converts object names to an internal form for the constraint solver. References to objects in constraints are converted into an internal "dot notation" form. In the PDI-4 language, the canonical object name "/" corresponds to the entire picture, or the "root" object. Under the "dot notation", the "/" object is referred to as `root`, and an object such as "/object_a" is referred to as `root.object_a` where the "." is the usual *cons* notational constructor. As well, `root` is the scope of `root.object_a`, and `root.object_a` is the scope of `root.object_a.object_b`. Object names in object declarations, which are not in canonical form, are not converted to the "dot notation" form when being passed to the constraint solver. Object names in object declarations are, however, converted to their canonical name for the abstract object description stream. Objects in this stream (Figure 3.14) are of the form

`object`(A, Attributes)

where the A is the canonical name of the object and Attributes is a representation of the attributes of the object. The post-processing is responsible for mapping the "dot notation" object names produced by the constraint solver into canonical names. The post-processing phase then merges the (absolute) coordinate values of the objects with the abstract object description and produces the graphics primitives required to draw the picture.

The format of the variables, constraints, and the solution in the constraint solver are discussed in the following sections.

3.9.1 The Constraint Solver

The constraint solver in ICOLA-4 operates on a system of constraints which is comprised of set of variables and a set of constraints. The variables in the system of constraints are of the form A_{left}, A_{right}, A_{top}, and A_{bottom} where A is the name of some object in the proper "dot notation" form. These variables represent the left, right, top, and bottom edges, respectively, of object A. The domain of all variables is the set of non-negative integers. Let V denote the set of all variables in the current system of constraints.

The constraints in the constraint system are formulae of the form $x + \alpha \leq_O y$ or $x+\alpha \leq_C y$, where $x, y \in V$ and $0 \leq \alpha$. The $x+\alpha \leq_O y$ formulae represent constraints induced by object declarations, while $x + \alpha \leq_C y$ formulae represent constraints induced by constraint declarations. These constraints encode the information that the specified edges of objects are at least a certain distance apart. The constraint $\texttt{root.a}_{right} + 15 \leq_C \texttt{root.b}_{right}$, for example, indicates that the right edge of object `root.a` is at least 15 "units" to the left of the right edge of object `root.b`. The constraint

```
constraint( left_of, [ left( /a ), right( /b )])
```

is represented as $\texttt{root.a}_{left} + \delta \leq_C \texttt{root.b}_{right}$, where δ is the default distance. The formula $x + \alpha \leq y$ is used to refer to formulae either of the form $x + \alpha \leq_C y$ or $x + \alpha \leq_O y$. Let C denote the current set of constraints.

A solution to the set of constraints is a set of variable assignments which satisfies the constraints. These variable assignments are the absolute coordinate values which are merged with the abstract object descriptions in the post-processing phase. Let the set S contain the solution to the current set of constraints as equations of the form $x = \gamma$, where $x \in V$ and $0 \leq \gamma$. S also contains equations of the form $x_{low} = \alpha$, where $x \in V$ and $0 \leq \alpha$. The value of x_{low} is the lowest possible value that can be assigned to the variable x. Both the x and x_{low} values are required by the incremental constraint satisfaction algorithm.

3.10 Operational Semantics

The following discussion of the operational semantics will concentrate on the constraint solver, which is the most important component of ICOLA-4. The design of other components is intuitive, and their implementations are straightforward.

This semantics does not include a detailed discussion of the insertion and deletion of objects of class LINE nor the associated `connects` constraint. This is because the position of a line (or arc) is entirely dependent on the position of the line's (arc's) endpoints – knowledge of the position of a line (arc) is not relevant to the positioning of any other object. Hence objects of class LINE and their related constraints need not involve the constraint solver, and the subject is not discussed further here. The positioning of objects of class LINE is discussed in Section 4.6.2.

The discussion of the operational semantics is based on the operational semantics framework used by Jaffar and Maher [JaMa94]. The operational semantics is presented as a series of transitions on states. There is a special state, denoted by $fail$, which indicates that the system is unable to solve the constraints. All other states are represented by tuples, and transitions from state to state are governed by a *computation rule*. This computation rule determines the next state based on the transition rules and the next input (increment) from the pre-processing stage.

Each state (except the $fail$ state) is represented by a tuple $\langle p,\ V,\ C,\ S \rangle$. In this tuple, p is the group object which contains all objects declared at the current scope. The name of that group object is synonymous with the name of the current scope. The set V contains all variables in the constraint system at the current state. V must not contain duplicates. C is the set of constraints over variables in V and may contain duplicates[1]. S is the current solution set. The contents of V, C, and S are as previously discussed in Section 3.9.1.

In general the state transitions are of the form

$$\langle p,\ V,\ C,\ S \rangle \rightarrow \langle p',\ V',\ C',\ S' \rangle$$

where p', V', C', and S' are updated versions of p, V, C, and S, respectively. The notation for transitions is $State \xrightarrow{Increment}_{Type} State'$ where $State$ is the current state, $Increment$ is the increment which induces the transition, and $State'$ is the next state. $Type$ is the type of the transition, and is one of i, d, a, or f. The $\rightarrow_i$ and

[1] If duplicates are not preserved in C then a single constraint deletion would be sufficient to remove the effects of two or more identical constraint insertions. Such behavior is undesirable.

$\rightarrow_d$ transitions deal with the insertions and deletions, respectively, of objects and constraints. The insertion and deletion of constraints due to the `text` and `bitmap` attributes are handled by the $\rightarrow_a$ transitions. The $\rightarrow_f$ transition is a transition to the $fail$ state.

Let δ denote the default distance between edges of objects. δ is both an inter-object and an intra-object default distance. Let the operator $\uplus$ denote a set union operation which preserves duplicates. Note that when a state such as $\langle p,\ V,\ C',\ S\rangle$ matches the left side of a transition, say:

$$\langle p,\ V,\ C \uplus \Delta_C,\ S\rangle \rightarrow_d \langle p,\ V,\ C,\ S'\rangle,$$

it is required that $C' = C \uplus \Delta_C$ for some C and Δ_C. Note that the contents of either C or Δ_C will always be completely defined and therefore that the $\uplus$ operation is deterministic.

There are a number of functions used in the transitions. The first of these is the function $consistent(\Delta_C,\ C)$, which is used to test the consistency of $C \uplus \Delta_C$. Let each constraint $(x + \alpha \leq y) \in C$ be represented by an arc of weight α from node x to node y in some weighted, directed acyclic graph G. If there is a path from x to y, and a path from y to x, then the constraint system is considered inconsistent, since the value for x would be dependent on y when the value of y is already dependent on x. If no such cycle exists, then all variables can be assigned a value. Given a new constraint $p + \gamma \leq q$, an incremental consistency check can be performed by simply checking for the existence of a path from q to p in G. If such a path does not exist then $p + \gamma \leq q$ can be safely added to C and the resulting set of constraints will be consistent. Let C_{temp} be equal to the current set of constraints. The consistency of each constraint $(p + \gamma \leq q) \in \Delta_C$ is checked, one at a time, with the constraints in C_{temp}. If the new constraint is consistent with C_{temp}, then it is added to C_{temp} and the next constraint in Δ_C is considered. If any constraint in Δ_C is inconsistent, then the consistency check fails, and none of the constraints in Δ_C will be added to C. If C_{temp} equals $C \uplus \Delta_C$ after all constraints in Δ_C are considered, then the consistency check succeeds, and all the constraints in Δ_C can be safely added to C. The system of constraints must be consistent before each increment. If an increment causes an inconsistency, the state transitions will halt in the *fail* state.

The function $solve\_insert(C,\ \Delta_C,\ S)$ is used to determine a new solution S' based on $C \uplus \Delta_C$ and S, where C is the current set of constraints, and S is the solution for C. The function $solve\_delete(C,\ \Delta_C,\ S)$ determines a new solution S' based on the removal of the constraints in Δ_C from C and based on S, the solution to C. The $solve\_insert()$ and $solve\_delete()$ functions are discussed in Section 3.10.3. Note that the solution S is required for the incremental computation of the new solution S'.

3.10.1 The Initial State

The initial state of the system is:

$$\langle p,\ V,\ C,\ S\rangle$$

where

$$\begin{aligned}
&p = \texttt{root},\\
&V = \{\ \texttt{root}_{left},\ \texttt{root}_{right},\ \texttt{root}_{top},\ \texttt{root}_{bottom}\},\\
&C = \{\ \texttt{root}_{left} + 0 \leq_O \texttt{root}_{right},\ \texttt{root}_{top} + 0 \leq_O \texttt{root}_{bottom}\}, \text{ and}\\
&S = \{\ \texttt{root}_{left_{low}} = 0,\ \texttt{root}_{left} = 0,\ \texttt{root}_{right_{low}} = 0,\ \texttt{root}_{right} = 0,\\
&\qquad \texttt{root}_{top_{low}} = 0,\ \texttt{root}_{top} = 0,\ \texttt{root}_{bottom_{low}} = 0,\ \texttt{root}_{bottom} = 0\}.
\end{aligned}$$

The solution is trivially correct.

3.10.2 The Transitions

The first types of transitions considered deal with the insertion and deletion of objects. These transitions add (or remove) the internal variables and constraints associated with the objects. As objects are added or removed, a solution is found to the current set of constraints. The computation rule selects the correct transition based on the class of the object being added or removed.

The objects `box`, `oval`, and `triangle` are the only objects for which the function *solve_insert*() must be called for insertions. All insertions of these objects of class RegBB are handled by a single transition. If the increment is:[2]

```
object( A, T )
```

and T is of class RegBB, then the transition is:

$$\langle p,\ V,\ C,\ S\rangle \overset{\mathtt{object(A,T)}}{\longrightarrow_i} \langle p,\ V \uplus \Delta_V,\ C \uplus \Delta_{C1} \uplus \Delta_{C2},\ S'\rangle$$

where

$$\begin{aligned}
\Delta_V &= \{\ p.\mathsf{A}_{left},\ p.\mathsf{A}_{right},\ p.\mathsf{A}_{top},\ p.\mathsf{A}_{bottom}\},\\
\Delta_{C1} &= \{\ p_{left} + 0 \le_O p.\mathsf{A}_{left},\ \ p.\mathsf{A}_{right} + 0 \le_O p_{right},\\
&\qquad p_{top} + 0 \le_O p.\mathsf{A}_{top},\ \ p.\mathsf{A}_{bottom} + 0 \le_O p_{bottom}\},\\
\Delta_{C2} &= \{\ p.\mathsf{A}_{left} + \delta \le_O p.\mathsf{A}_{right},\ \ p.\mathsf{A}_{top} + \delta \le_O p.\mathsf{A}_{bottom}\},\\
N &= \{\ p.\mathsf{A}_{left_{low}} = p_{left_{low}},\ p.\mathsf{A}_{left} = p_{left_{low}},\\
&\qquad p.\mathsf{A}_{right_{low}} = p_{left_{low}},\ p.\mathsf{A}_{right} = p_{left_{low}},\\
&\qquad p.\mathsf{A}_{top_{low}} = p_{top_{low}},\ p.\mathsf{A}_{top} = p_{top_{low}},\\
&\qquad p.\mathsf{A}_{bottom_{low}} = p_{top_{low}},\ p.\mathsf{A}_{bottom} = p_{top_{low}}\},\\
S' &= \ \mathit{solve_insert}(C \uplus \Delta_{C1},\ \Delta_{C2},\ S \uplus N),
\end{aligned}$$

and $\Delta_V \cap V = \emptyset$. This last restriction ensures that object identifiers are not duplicated. The immediate addition of the variables in Δ_V to the set V is required as in any further transition V must contain all the variables occurring in the constraints in $C \uplus \Delta_{C1} \uplus \Delta_{C2}$. The introduction of Δ_{C1} and Δ_{C2} to C can never make the constraint system inconsistent. This must be true as both $p_{left} + 0 \le p_{right}$ and $p_{top} + 0 \le p_{bottom}$ are already known to be true by the existence of p, C is consistent, the formulae introduced in Δ_{C1} and Δ_{C2} do not introduce a cycle from p_{bottom} to p_{top} nor from p_{right} to p_{left} (by examination), and $\Delta_{C1} \uplus \Delta_{C2}$ does not contain any cycles. The set N contains the initial values for the new variables. Observe that the initial value for $p.\mathsf{A}_{right_{low}}$ and $p.\mathsf{A}_{right}$ is $p_{left_{low}}$, as the value of $p.\mathsf{A}_{right_{low}}$ must be as low

[2] Recall that "`object( A, T )`" is equivalent to "`insert object( A, T )`".

as possible, and $p.\mathsf{A}_{right}$ must equal $p.\mathsf{A}_{right_{low}}$. The initial values for $p.\mathsf{A}_{bottom_{low}}$ and $p.\mathsf{A}_{bottom}$ are assigned in a similar fashion. These values are immediately updated to their correct values by the invocation of the *solve_insert*() function. Note that there is no reason to individually solve each of the constraints in Δ_{C1} as these constraints are satisfied by the initial solution in N.

If the increment is:

```
delete object( A, T )
```

where T is of class RegBB, then the transition is:

$$\langle p,\ V \uplus \Delta_V,\ C \uplus \Delta_{C1} \uplus \Delta_{C2},\ S\rangle \overset{\texttt{delete object(A,T)}}{\longrightarrow_d} \langle p,\ V,\ C,\ S' - N\rangle$$

where

$$\begin{aligned} N &= \{\ x_{low} = \alpha,\ x = \omega \mid x \in \Delta_V\}, \\ S' &= \ \ solve\_delete(C \uplus \Delta_{C1} \uplus \Delta_{C2},\ \Delta_{C2},\ S), \end{aligned}$$

and Δ_V, Δ_{C1}, and Δ_{C2} are defined as for the insertion of an object called A of class RegBB. It must also be the case that $\forall(x + \gamma \leq y) \in C$, $x \notin \Delta_V$ and $y \notin \Delta_V$. This restriction ensures that an object is only removed if all constraints associated with the variables of that object have already been removed. Once the constraints in Δ_{C2} are removed from $C \uplus \Delta_{C1} \uplus \Delta_{C2}$, the constraints in Δ_{C1} can be removed from $C \uplus \Delta_{C1}$ without requiring any further solving. This is possible as the values of the variables in Δ_V are made as small as possible by the *solve_delete*(). The values of $p.\mathsf{A}_{right_{low}}$, $p.\mathsf{A}_{right}$, $p.\mathsf{A}_{bottom_{low}}$, and $p.\mathsf{A}_{bottom}$ will be 0. As well, $p.\mathsf{A}_{left_{low}} = p_{left_{low}}$, $p.\mathsf{A}_{left} = p_{left}$, $p.\mathsf{A}_{top_{low}} = p_{top_{low}}$, and $p.\mathsf{A}_{top} = p_{top}$. This means that the constraints in Δ_{C1} can be trivially removed. Note that the *solve_delete*() requires the equations in N to be present when determining the new solution, and hence N can only be removed from the new solution after *solve_delete*() is finished.

All objects of classes Fixed and Complex have an initial size of 0, which differs from an initial size of δ for objects of class RegBB. This difference is also the reason why there is no need to call the *solve_insert*() function for insertions of objects of class FIXED or COMPLEX. Objects of class FIXED are considered first. If the increment is:

```
object( A, T )
```

and T is of class FIXED, then the transition is:

$$\langle p,\ V,\ C,\ S\rangle \overset{\mathtt{object(A,T)}}{\to_i} \langle p,\ V \uplus \Delta_V, C \uplus \Delta_C,\ S \uplus N\rangle$$

where

$$\begin{aligned}
\Delta_V &= \{\ p.\mathsf{A}_{left},\ p.\mathsf{A}_{right},\ p.\mathsf{A}_{top},\ p.\mathsf{A}_{bottom}\},\\
\Delta_C &= \{\ p_{left} + 0 \le_O p.\mathsf{A}_{left},\quad p.\mathsf{A}_{right} + 0 \le_O p_{right},\\
&\qquad p_{top} + 0 \le_O p.\mathsf{A}_{top},\quad p.\mathsf{A}_{bottom} + 0 \le_O p_{bottom},\\
&\qquad p.\mathsf{A}_{left} + 0 \le_O p.\mathsf{A}_{right},\quad p.\mathsf{A}_{top} + 0 \le_O p.\mathsf{A}_{bottom}\},\text{ and}\\
N &= \{\ p.\mathsf{A}_{left_{low}} = p_{left_{low}},\ p.\mathsf{A}_{left} = p_{left_{low}},\\
&\qquad p.\mathsf{A}_{right_{low}} = p_{left_{low}},\ p.\mathsf{A}_{right} = p_{left_{low}},\\
&\qquad p.\mathsf{A}_{top_{low}} = p_{top_{low}},\ p.\mathsf{A}_{top} = p_{top_{low}},\\
&\qquad p.\mathsf{A}_{bottom_{low}} = p_{top_{low}},\ p.\mathsf{A}_{bottom} = p_{top_{low}}\}.
\end{aligned}$$

The solution in N is available immediately as adding the new constraints in Δ_C will not change the values of p_{left}, p_{right}, p_{top} or p_{bottom}. This is true as these constraints simply imply that $p_{left} + 0 \le p_{right}$ and that $p_{top} + 0 \le p_{bottom}$, which is already the case.

The deletion of objects of class FIXED requires special consideration due to the way that `bitmap` and `text` attributes change the system of constraints. The transition for the deletion of an object of type `point` is considered first. If the increment is:

```
delete object( A, T )
```

where T is `point`, then the transition is:

$$\langle p,\ V \uplus \Delta_V,\ C \uplus \Delta_C,\ S \uplus N\rangle \overset{\mathtt{delete\ object(A,T)}}{\to_d} \langle p,\ V,\ C,\ S\rangle$$

where

$$N \;\; = \{\ x_{low} = \alpha,\ \ x = \omega \mid x \in \Delta_V\}$$

and Δ_V and Δ_C are defined as for the insertion of an object called A of class FIXED. It must also hold that $\forall (x + \gamma \leq y) \in C$, $x \notin \Delta_V$ and $y \notin \Delta_V$. This last restriction ensures that the object is only removed if all constraints associated with that object have already been removed.

The deletion of an object of type `bitmap` or `text` is considered next. If the increment is:

```
delete object( A, T )
```

where T is `bitmap` or `text`, then the transition is:

$$\langle p,\ V \uplus \Delta_V,\ C \uplus \Delta_{C1} \uplus \Delta_{C2},\ S\rangle \overset{\texttt{delete object(A,T)}}{\rightarrow_d} \langle p,\ V,\ C,\ S' - N\rangle$$

where

$$\begin{aligned}
\Delta_{C1} &= \{\ p_{left} + 0 \leq_O p.\mathsf{A}_{left},\quad p.\mathsf{A}_{right} + 0 \leq_O p_{right}, \\
&\qquad p_{top} + 0 \leq_O p.\mathsf{A}_{top},\quad p.\mathsf{A}_{bottom} + 0 \leq_O p_{bottom}\}, \\
\Delta_{C2} &= \{\ p.\mathsf{A}_{left} + \sigma \leq_O p.\mathsf{A}_{right},\quad p.\mathsf{A}_{top} + \psi \leq_O p.\mathsf{A}_{bottom}\}, \\
N &= \{\ x_{low} = \alpha,\ \ x = \omega \mid x \in \Delta_V\}, \\
S' &= \ \ solve_delete(C \uplus \Delta_{C1} \uplus \Delta_{C2},\ \Delta_{C2},\ S),
\end{aligned}$$

and Δ_V is defined as for the insertion of an object called A of class FIXED. Note that σ and ψ are the current width and height, respectively, of the `bitmap` or `text` object. This special case is required for objects of type `bitmap` and `text` due to the changes to the constraint system induced by their `bitmap` and `text` attributes, respectively. Again, it must also hold that $\forall (x + \gamma \leq y) \in C$, $x \notin \Delta_V$ and $y \notin \Delta_V$. This last restriction ensures that the object is only removed if all constraints associated with that object have already been removed.

The insertion of an object of class COMPLEX requires special consideration, since

```
group( A )
```

in a picture description is used to both begin a new group and to re-open an existing group. The first of these two cases is if the group object does not exist. In this situation, if the increment is:

```
group( A )
```

then the transition is:

$$\langle p,\ V,\ C,\ S\rangle \stackrel{\texttt{group}(\mathsf{A})}{\rightarrow_i} \langle p.\mathsf{A},\ V \uplus \Delta_V, C \uplus \Delta_C,\ S \uplus N\rangle$$

where Δ_V, Δ_C and N are defined as for the insertion of an object of class Fixed and $\Delta_V \cap V = \emptyset$. Note that the current scope changes from p to $p.\mathsf{A}$, as reflected by the first argument of the tuple.

If the group object already exists and the increment is:

```
group( A )
```

then the transition is:

$$\langle p,\ V,\ C,\ S\rangle \stackrel{\texttt{group}(\mathsf{A})}{\rightarrow_i} \langle p.\mathsf{A},\ V,\ C,\ S\rangle$$

where $p.\mathsf{A}_{left}, p.\mathsf{A}_{right}, p.\mathsf{A}_{top}, p.\mathsf{A}_{bottom} \in V$. In this transition only the current scope changes.

If the increment is:

```
end_group( A )
```

then in all cases the transition is:

$$\langle p.\mathsf{A},\ V,\ C,\ S\rangle \stackrel{\texttt{end\_group}(\mathsf{A})}{\rightarrow_i} \langle p,\ V,\ C,\ S\rangle$$

where $p.\mathsf{A}_{left}, p.\mathsf{A}_{right}, p.\mathsf{A}_{top}, p.\mathsf{A}_{bottom} \in V$. Observe that the current scope has changed from $p.\mathsf{A}$ to p.

The transition for group deletions requires that all constituent objects of that group and all constraints of the form $x + \alpha \leq_C y$ associated with that group are removed first. If the increment is:

```
delete group( A )
```

then the transition is:

$$\langle p,\ V \uplus \Delta_V,\ C \uplus \Delta_C,\ S \uplus N \rangle \overset{\textsf{delete group(A)}}{\longrightarrow_d} \langle p,\ V,\ C,\ S \rangle$$

where Δ_V, Δ_C, and N are defined as for the deletion of an object of type `point`. As well, $\forall (x + \gamma \leq y) \in C,\ x \notin \Delta_V$ and $y \notin \Delta_V$. These restrictions ensure that the transition is only performed if there are no other constraints on this group object and that there are no objects within the group. The constraints in Δ_C can be trivially removed from $C \uplus \Delta_C$ as they only transitively enforce that $p_{left} + 0 \leq p_{right}$ and $p_{top} + 0 \leq p_{bottom}$, which are already known to be true.

The only other possible object increment deals with objects of class LINE. As mentioned before, objects of class LINE are not discussed formally in this semantics. It must be noted, however, that an object must not be deleted if it has a `connects` constraint associated with it.

The insertion or deletion of a basic or an aliased constraint requires that the variables in the constraint already be present in V. If the increment is an insertion of a basic or an aliased constraint then the transition is:

$$\langle p,\ V,\ C,\ S \rangle \overset{\textsf{Constraint}}{\longrightarrow_i} \langle p,\ V,\ C \uplus \Delta_C,\ S' \rangle$$

where $consistent(\Delta_C,\ C)$ must be true, and $S' = solve\_insert(C, \Delta_C, S)$. If Constraint is a basic constraint, then the appropriate Δ_C is given in Table 3.13. Both X and Y in Table 3.13 are names of objects in the proper "dot notation" form for object names and both X and Y must be in V. If Constraint is an aliased constraint then $\Delta_C = \Delta_1 \uplus \Delta_2 \uplus \cdots \uplus \Delta_n$ where each Δ_i corresponds to the contribution from one of the n basic constraints which comprise the aliased constraint and each Δ_i is obtained from Table 3.13. The group processing of aliased constraints is deliberate.

If the increment is the deletion of a basic or an aliased constraint then the transition is:

$$\langle p,\ V,\ C \uplus \Delta_C,\ S \rangle \overset{\textsf{delete Constraint}}{\longrightarrow_d} \langle p,\ V,\ C,\ S' \rangle$$

where the values for Δ_C are determined exactly as they are for constraint insertions and $S' = solve\_delete(C \uplus \Delta_C, \Delta_C, S)$. Note that $C \uplus \Delta_C$ is consistent, and removing Δ_C cannot introduce any inconsistencies.

Table 3.13: Mapping input constraints to internal constraints.

Constraint	Δ_C or Δ_i
`constraint( left_of, [ left(X), left(Y)])`	$\{X_{left} + \delta \leq_C Y_{left}\}$
`constraint( left_of, [ left(X), right(Y)])`	$\{X_{left} + \delta \leq_C Y_{right}\}$
`constraint( left_of, [ right(X), left(Y)])`	$\{X_{right} + \delta \leq_C Y_{left}\}$
`constraint( left_of, [ right(X), right(Y)])`	$\{X_{right} + \delta \leq_C Y_{right}\}$
`constraint( above, [ top(X), top(Y)])`	$\{X_{top} + \delta \leq_C Y_{top}\}$
`constraint( above, [ top(X), bottom(Y)])`	$\{X_{top} + \delta \leq_C Y_{bottom}\}$
`constraint( above, [ bottom(X), top(Y)])`	$\{X_{bottom} + \delta \leq_C Y_{top}\}$
`constraint( above, [ bottom(X), bottom(Y)])`	$\{X_{bottom} + \delta \leq_C Y_{bottom}\}$
`constraint(` **h_d**`, [ D, left(X), left(Y)])`	$\{X_{left} + D \leq_C Y_{left}\}$
`constraint(` **h_d**`, [ D, left(X), right(Y)])`	$\{X_{left} + D \leq_C Y_{right}\}$
`constraint(` **h_d**`, [ D, right(X), left(Y)])`	$\{X_{right} + D \leq_C Y_{left}\}$
`constraint(` **h_d**`, [ D, right(X), right(Y)])`	$\{X_{right} + D \leq_C Y_{right}\}$
`constraint(` **v_d**`, [ D, top(X), top(Y)])`	$\{X_{top} + D \leq_C Y_{top}\}$
`constraint(` **v_d**`, [ D, top(X), bottom(Y)])`	$\{X_{top} + D \leq_C Y_{bottom}\}$
`constraint(` **v_d**`, [ D, bottom(X), top(Y)])`	$\{X_{bottom} + D \leq_C Y_{top}\}$
`constraint(` **v_d**`, [ D, bottom(X), bottom(Y)])`	$\{X_{bottom} + D \leq_C Y_{bottom}\}$

h_d = `horizontal_distance` **v_d** = `vertical_distance`

The behavior of the `connects` constraint must be noted. In particular, the `connects` constraint can only be applied to objects that already exist and an object of class LINE can only have one `connects` constraint associated with it.

The only attributes which affect the constraint system are `text` and `bitmap`. All other attributes are irrelevant to the constraint satisfaction problem of finding actual values for the variables which represent the objects' positions. Let w_{old} and h_{old} be the width and height, respectively, of the previous text string or bitmap for a `text` or `bitmap` object, respectively. Let w_{new} and h_{new} be the width and height, respectively, of the new text string or bitmap. The transition to handle this special situation first removes the old constraints associated with the previous width and height, and then adds the new constraints associated with the new width and height. Since the only difference between the old constraints and the new constraints is the attribute value, if the current set of constraints is consistent before the $\rightarrow_a$ transition, it will be consistent after the $\rightarrow_a$ transition.

If the increment is:

```
attribute( A, bitmap, Filename )
```

or

```
attribute( A, text, Text ),
```

then the transition is:

$$\langle p,\ V,\ C \uplus \Delta_{Old},\ S\rangle \overset{\mathsf{Attribute}}{\longrightarrow_a} \langle p,\ V,\ C \uplus \Delta_{New},\ S'\rangle$$

where

$$\begin{aligned}
\Delta_{Old} &= \{\mathsf{A}_{left} + w_{old} \leq_O \mathsf{A}_{right},\ \mathsf{A}_{top} + h_{old} \leq_O \mathsf{A}_{bottom}\},\\
\Delta_{New} &= \{\mathsf{A}_{left} + w_{new} \leq_O \mathsf{A}_{right},\ \mathsf{A}_{top} + h_{new} \leq_O \mathsf{A}_{bottom}\},\\
S_1 &= solve_delete(\ C \uplus \Delta_{Old},\ \Delta_{Old},\ S),\\
S' &= solve_insert(C,\ \Delta_{New},\ S_1),
\end{aligned}$$

and A is the canonical name of the object.

If none of the preceding transitions apply to the increment then the following transition is performed:

$$\langle p,\ V,\ C,\ S\rangle \overset{\mathsf{Increment}}{\longrightarrow_f} fail$$

where $fail$ is the special "failure" state. If the $fail$ state is reached then there is no solution and no further transitions[3].

3.10.3 The Solution Functions

The algorithm for function $S' = solve\_insert(C, \Delta_C, S)$ is shown in Figure 3.15. Note that the consistency of $C \uplus \Delta_C$ is checked before $solve\_insert()$ is called. Each constraint in the set Δ_C is added to C, and a new solution is found after each addition. Observe that when the function is finished the solution corresponds to the

[3] An implementation, however, may simply reject Increment (producing appropriate error messages) and allow the transitions to continue from the state $\langle p,\ V,\ C,\ S\rangle$.

set of constraints $C \uplus \Delta_C$. The function returns the solution after all constraints are processed. The function $S' = solve\_delete(C, \Delta_C, S)$ is similar to $solve\_insert()$ and is shown in Figure 3.16. The purpose of this function is to remove each constraint in Δ_C, one at a time, from C, and then return the new solution. The current set of constraints when the algorithm finishes is $C - \Delta_C$, and since C is consistent, $C - \Delta_C$ must also be consistent. The functions $incremental\_solve\_insert()$ and $incremental\_solve\_delete()$ are discussed in detail in Chapter 4.

$C_{now} = C$
$S_{now} = S$
`foreach` $c \in \Delta_C$ `do begin`
    $C_{prev} = C_{now}$
    $S_{prev} = S_{now}$
    $S_{now} = incremental\_solve\_insert(C_{prev},\ (c),\ S_{prev})$
    $C_{now} = C_{prev} \uplus \{c\}$
`end`
$S' = S_{now}$

Figure 3.15: The algorithm for $S' = solve\_insert(C,\ \Delta_C,\ S)$.

$C_{now} = C$
$S_{now} = S$
`foreach` $c \in \Delta_C$ `do begin`
    $C_{prev} = C_{now}$
    $S_{prev} = S_{now}$
    $S_{now} = incremental\_solve\_delete(C_{prev},\ (c),\ S_{prev})$
    $C_{now} = C_{prev} - \{c\}$
`end`
$S' = S_{now}$

Figure 3.16: The algorithm for $S' = solve\_delete(C,\ \Delta_C,\ S)$.

The operational semantics of the "batch" mode (See Section 3.7) is very similar to the semantics presented in the previous sections. The functions $solve\_insert()$ and $solve\_delete()$ in the "batch" mode do not update the solution, but rather simply return $S' = S$. The `solve` directive (See Section 3.7) is used to call the $solve\_batch()$

function. The following transition is added to the semantics to handle the `solve` directive:

$$\langle p,\ V,\ C,\ S\rangle \overset{\texttt{solve}}{\longrightarrow}_s \langle p,\ V,\ C,\ S'\rangle$$

where $S' = solve\_batch(C)$. Note that the consistency checks are still performed for each increment, so a solution for the current set of constraints C is always possible. The $solve\_batch()$ function is discussed in detail in Chapter 4.

3.10.4 Operational Semantics Example

This section gives a brief example of the states and transitions of the operational semantics of ICOLA-4 for a small segment of a picture description. Figure 3.17 shows part of a picture description where a group `g1` is being defined in the scope of the `root` object. Recall that δ is the default inter-object and intra-object distance.

```
⋮
group( g1 ).
      object( box_1, box ).
      object( box_2, box ).
      constraint( left_of, [ box_1, box_2 ]).
      object( box_3, box ).
      constraint( above, [ box_1, box_3 ]).
```

Figure 3.17: A portion of a picture description.

The state of the constraint system after Figure 3.17 is processed will be, in part, as shown in Figure 3.18.

If the next increment is

```
object( box_4, box )
```

then the transition for the insertion of an object of class RegBB will be performed. The state after the transition will be, in part, as shown in Figure 3.19.

$$\begin{aligned}
\langle\ & \texttt{root.g1},\ \{\ldots,\ \texttt{root.g1}_{left},\ \texttt{root.g1}_{right},\ \ldots\},\\
& \{\ldots,\ \texttt{root.g1}_{left} + 0 \leq_O \texttt{root.g1}_{right},\ \ldots\},\\
& \{\ldots,\ \texttt{root.g1}_{left} = 0,\ \texttt{root.g1}_{left_{low}} = 0,\\
& \qquad \texttt{root.g1}_{right} = \delta + \delta + \delta,\ \texttt{root.g1}_{right_{low}} = \delta + \delta + \delta,\\
& \qquad \texttt{root.g1.box_3}_{left} = 0,\ \texttt{root.g1.box_3}_{left_{low}} = 0,\\
& \qquad \texttt{root.g1.box_3}_{right} = \delta,\ \texttt{root.g1.box_3}_{right_{low}} = \delta,\ \ldots\}\rangle
\end{aligned}$$

Figure 3.18: The state of the constraint system after Figure 3.17.

$$\begin{aligned}
\langle\ & \texttt{root.g1},\ \{\ldots,\ \texttt{root.g1}_{left},\ \texttt{root.g1}_{right},\ \texttt{root.g1.box_4}_{left},\\
& \texttt{root.g1.box_4}_{right},\ \ldots\},\\
& \{\ldots,\ \texttt{root.g1}_{left} + 0 \leq_O \texttt{root.g1}_{right},\\
& \qquad \texttt{root.g1}_{left} + 0 \leq_O \texttt{root.g1.box_4}_{left},\\
& \qquad \texttt{root.g1.box_4}_{left} + 0 \leq_O \texttt{root.g1.box_4}_{right},\\
& \qquad \texttt{root.g1.box_4}_{right} + 0 \leq_O \texttt{root.g1}_{right},\ \ldots\},\\
& \{\ldots,\ \texttt{root.g1}_{left} = 0,\ \texttt{root.g1}_{left_{low}} = 0,\\
& \qquad \texttt{root.g1}_{right} = \delta + \delta + \delta,\ \texttt{root.g1}_{right_{low}} = \delta + \delta + \delta,\\
& \qquad \texttt{root.g1.box_3}_{left} = 0,\ \texttt{root.g1.box_3}_{left_{low}} = 0,\\
& \qquad \texttt{root.g1.box_3}_{right} = \delta,\ \texttt{root.g1.box_3}_{right_{low}} = \delta,\\
& \qquad \texttt{root.g1.box_4}_{left} = 0,\ \texttt{root.g1.box_4}_{left_{low}} = 0,\\
& \qquad \texttt{root.g1.box_4}_{right} = \delta,\ \texttt{root.g1.box_4}_{right_{low}} = \delta,\ \ldots\}\rangle
\end{aligned}$$

Figure 3.19: The current state after the addition of `box_4`.

Using this new state, an increment

```
constraint( left_of, [ right( box_3 ), right( box_4 )])
```

is converted by the pre-processor (See Section 3.9) to

```
constraint( left_of, [  right( root.g1.box_3 ),
                        right( root.g1.box_4 ) ]).
```

The constraint to be added to the current set of constraints is

$$\texttt{root.g1.box\_3}_{right} + \delta \leq_C \texttt{root.g1.box\_4}_{right},$$

where δ is the default distance. Before this constraint is added to the current set of constraints, the consistency check must be performed. If the consistency check is successful, then $solve\_insert()$ function is called, and the new state is, in part, as shown in Figure 3.20.

$$
\begin{array}{l}
\langle\ \texttt{root.g1},\ \{\ldots,\ \texttt{root.g1}_{left},\ \texttt{root.g1}_{right},\ \texttt{root.g1.box\_4}_{left},\\
\quad \texttt{root.g1.box\_4}_{right},\ \ldots\},\\
\quad \{\ldots,\ \texttt{root.g1}_{left} + 0 \leq_O \texttt{root.g1}_{right},\\
\qquad\quad \texttt{root.g1}_{left} + 0 \leq_O \texttt{root.g1.box\_4}_{left},\\
\qquad\quad \texttt{root.g1.box\_4}_{left} + 0 \leq_O \texttt{root.g1.box\_4}_{right},\\
\qquad\quad \texttt{root.g1.box\_4}_{right} + 0 \leq_O \texttt{root.g1}_{right},\\
\qquad\quad \texttt{root.g1.box\_3}_{right} + \delta \leq_C \texttt{root.g1.box\_4}_{right},\ \ldots\},\\
\quad \{\ldots,\ \texttt{root.g1}_{left} = 0,\ \texttt{root.g1}_{left_{low}} = 0,\\
\qquad\quad \texttt{root.g1}_{right} = \delta + \delta + \delta,\ \texttt{root.g1}_{right_{low}} = \delta + \delta + \delta,\\
\qquad\quad \texttt{root.g1.box\_3}_{left} = 0,\ \texttt{root.g1.box\_3}_{left_{low}} = 0,\\
\qquad\quad \texttt{root.g1.box\_3}_{right} = \delta,\ \texttt{root.g1.box\_3}_{right_{low}} = \delta,\\
\qquad\quad \texttt{root.g1.box4}_{left} = \delta,\ \texttt{root.g1.box\_4}_{left_{low}} = 0,\\
\qquad\quad \texttt{root.g1.box\_4}_{right} = \delta + \delta,\ \texttt{root.g1.box\_4}_{right_{low}} = \delta + \delta,\ \ldots\}\rangle
\end{array}
$$

Figure 3.20: The current state after the addition of the `left_of` constraint.

Note that the values of $\texttt{root.g1}_{right}$ and $\texttt{root.g1}_{right_{low}}$ do not change.

3.11 Chapter Summary

In this chapter PDI-4 is presented along with an operational semantics. Highlights of PDI-4 include support for both the incremental insertion and deletion of objects and constraints. As well, attributes are fully supported in ICOLA-4. The operational semantics formalize the steps required to convert PDI into graphics primitives. Chapter 4 presents the incremental algorithms which solve the system of constraints discussed in the operational semantics.

CHAPTER 4

ICOLA – THE ALGORITHM

The new incremental constraint satisfaction algorithm is the most important component of ICOLA-4. This algorithm uses incremental techniques for updating the internal data structures and for determining a new solution. A non-incremental or "batch" algorithm is also implemented in ICOLA-4. The primary purpose of the non-incremental algorithm is to provide a reference point against which the performance of the incremental algorithm can be compared. This chapter presents both the incremental and non-incremental algorithms, and briefly compares them with the incremental algorithm of ICOLA-3. A general overview of the algorithms in ICOLA-4 is given first, followed by a more detailed presentation, and then a discussion of the implementation issues.

4.1 Notation

The intentional "overloading" of certain notational constructs must be noted. For example, the "x" in "`foreach` $(x + \alpha \leq y) \in C$" is a meta-level variable named x. If the formulae $\texttt{root}_{left} + 0 \leq_O \texttt{root}_{right}$ matches $x + \alpha \leq y$ above, then x represents $root_{left}$. The "x" in "`if` $(x + \alpha \leq y)$" refers to the *value* of the variable represented by x. If $x + \alpha \leq y$ matches $\texttt{root}_{left} + 0 \leq_O \texttt{root}_{right}$, and the value of $\texttt{root}_{left}$ is 0, then x is the value of $\texttt{root}_{left}$ which is 0. In "$x = z + \gamma$", the variable represented by x is assigned the value of the variable represented by z, plus the value represented by γ. Hence if x is $\texttt{root}_{right}$, z is $\texttt{root}_{left}$, and γ is 10, then the operation being performed is the assignment of the value of $\texttt{root}_{left}$ plus 10 to $\texttt{root}_{right}$. The correct interpretation of such occurrences of variables will be either clear from the context or explicitly stated.

Let the function $is\_left(x)$ be true if variable x represents the left edge of an object. Similarly, the function $is\_top(x)$ is true if variable x represents the top edge of an object.

Let Q be a FIFO (First-In, First-Out) queue. Let the function $empty(Q)$ be true if the queue Q is empty. The procedure $add(Q, w)$ updates Q by adding some item w to the tail of queue Q. The procedure $delete(Q, w)$ removes item w from the head of queue Q.

Let G be a weighted directed graph where nodes in G correspond to variables in V, and for each $(p + \gamma \leq q) \in C$ there is an arc of weight γ from node p to node q in G. A variable y *depends* on x if there is a path from node x to node y, $x \neq y$, in G. This means that x must have its value determined before y.

4.2 General Overview

ICOLA-4 has both incremental and non-incremental constraint solution algorithms. The non-incremental ("batch") algorithm consists of a single $solve\_batch()$ function. $solve\_batch()$ is invoked each time a solution is requested using the `solve` directive (See Section 3.7). Each time it is called it re-solves the entire system of constraints and returns the new solution. The batch algorithm is used primarily for performance comparisons with the incremental algorithm.

The incremental algorithm finds a new solution after each increment. It consists of two functions: $incremental\_solve\_insert()$ and $incremental\_solve\_delete()$. The incremental insertion function, $incremental\_solve\_insert(C, (x+\alpha \leq y), S)$, returns a new solution, S', which is determined from the current set of constraints C, a new constraint $x + \alpha \leq y$ to be added to C, and S, the current solution to C. The incremental deletion function, $incremental\_solve\_delete(C, (x + \alpha \leq y), S)$, returns the new solution, S', which is determined from the current set of constraints C, the constraint $x + \alpha \leq y$ to be removed from C, and the current solution S. A brief comparison of ICOLA-4's incremental algorithm to other constraint satisfaction algorithms is given in Section 5.3. Quantitative performance results for the solution algorithms are given in Section 5.4.

Both the incremental and batch solution algorithms in ICOLA-4 rely on the fact that the set of constraints is consistent. The consistency check is easily performed using the technique presented in Section 3.10. A fundamental principle of the algorithms is that the value of variable q will not be used in any computations until value of each variable p, $(p + \omega \leq q) \in C'$, is known to be correct.

The non-incremental algorithm in ICOLA-4 is similar to the constraint algorithm in ICOLA-3. ICOLA-3's algorithm, however, is slightly more computationally expensive, and does not provide the consistent pictures like ICOLA-4. The incremental algorithm of ICOLA-4 is a significant improvement over the algorithm in ICOLA-3, both in the consistency of images produced and in quantitative performance results (See Section 5.4). The algorithms in both ICOLA-3 and ICOLA-4 update the internal data structures in a similar fashion, and rely on the fact that the system is consistent before determining a solution.

Severely under-constrained systems have an infinite number of individual solutions (a family of solutions) and, in general, the only finite representation of such a solution is a symbolic one. A symbolic solution is inadequate, however, for drawing an image corresponding to the picture description, as actual numeric values are required for the coordinates of the objects. While any one solution can be chosen from the infinite family of solutions, some solutions produce more aesthetically pleasing pictures than others. Rather than allowing the user to select a picture, the tendency toward the "more desirable" solutions is controlled through the use of heuristics. The two main heuristics of ICOLA-4 are:

1. An object's right edge will be as far left as possible. Similarly, an object's bottom edge will be as close as possible to the top of the layout.

2. Objects will tend to be as small as possible.

It is important to note that the first heuristic has priority over the second. Without this specific ordering of the heuristics, the heuristics would be unable to sufficiently restrict the constraints to produce a unique picture. Figure 4.2 and Figure 4.3 show two different pictures which could result from the picture description in Figure 4.1 if the second heuristic is applied to one object before another.

```
object( box_1, box ).
object( box_2, box ).
constraint( above, [ box_1, box_2 ]).
constraint( horizontal_distance,
            [ 100, left( box_1 ), right( box_2 )]).
constraint( horizontal_distance,
            [ 10, left( box_2 ), right( box_1 )]).
```

Figure 4.1: A potentially ambiguous picture description.

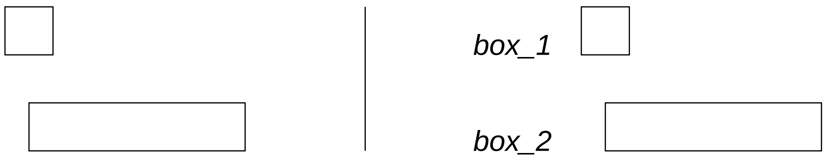


Figure 4.2: The resulting picture if `box_1` is made small first.

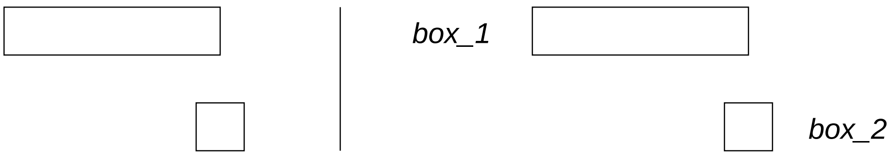


Figure 4.3: The resulting picture if `box_2` is made small first.

If the second heuristic has priority over the first, then the order in which the objects are chosen has an affect on the final picture. If the choice of which object to apply the heuristic to first is left indeterminate, however, then the figure generated as correct output by ICOLA would be indeterminate. While a trivial solution is to fix the order in which the objects are chosen (e.g. use the order in which they are defined in the picture description), such a restriction is not desirable. It is better to produce the same image regardless of the order in which the objects are specified.

A solution to this problem is to give the first heuristic priority over the second. Then there is only one set of values which will satisfy the first heuristic, and consequently only one set of values which will satisfy the second heuristic. The proof of this statement is found in Theorems 1 and 2 which are given later in this section. Figure 4.2 shows the picture generated by ICOLA-4 for the picture description in Figure 4.1.

The goal of both the incremental and non-incremental algorithms is to find a solution which satisfies the two heuristics. In terms of variables and constraints, this means that the values of variables A_{right} and A_{bottom} will be minimized, after which the values of A_{left} and A_{top} will be maximized. As mentioned in Section 3.9.1 the domain of each variable is the set of non-negative integers.

Let $\pi(y)$ be the value of variable y. To satisfy the first heuristic, the following must hold for each variable $y \in V$:

$$\pi(y) = max(\{\pi(x) + \alpha \mid (x + \alpha \leq y) \in C\}).$$

The above ensures that all variables are assigned the lowest possible values and guarantees that the values of variables representing right or bottom edges of objects will be as low as possible. To satisfy the second heuristic, the following must hold for each variable $x \in V$ which represents the left or top edge of an object:

$$\pi(x) = min(\{\pi(y) - \alpha \mid (x + \alpha \leq y) \in C\}).$$

The above ensures that variables which represent the left or top edge of an object are given the maximum possible value given the constraints and the current assignments to variables representing the right or bottom edges of objects. For example, for the constraints $(x + \alpha \leq p) \in C$ and $(x + \gamma \leq q) \in C$, the position of x is determined by the lesser of $p - \alpha$ and $q - \gamma$, as the lesser of the two is the maximum value x can be assigned and remain consistent with the constraints.

As Theorems 1 and 2 show, there is only one possible value for each variable which will satisfy the heuristics. The goal of "confluence" with respect to the addition and deletion of constraints has thus been achieved. This is true since the order in which constraints are added or removed is not important if the constraint solver finds the single correct solution. The goal of confluence has also been achieved with respect to the addition and deletion of objects. The only restrictions on the order of objects and constraints, therefore, is that objects be declared before they are referenced in a constraint, and that objects and constraints are inserted before they are deleted.

Theorem 1 *There is only one set of unique values which satisfy the first heuristic.*

Proof: Let V be a set of variables. Let C be a set of constraints of the form $x + \alpha \leq y$ over the variables in V. Let G be a weighted, directed graph where nodes in G correspond to variables in V, and if $(p + \gamma \leq q) \in C$, then there is an arc of weight γ from node p to node q in G. Since G is acyclic, there exists at least one single-source shortest path between $\texttt{root}_{left}$ and any other node z_1 in the left-to-right ordering. There also exists at least one single-source shortest path between $\texttt{root}_{top}$ and any other node z_2 in the top-to-bottom ordering. The length of these shortest paths can be determined by Dijkstra's algorithm [Cor+92]. The properties of Dijkstra's algorithm guarantee that there is only one possible lowest value for each variable. □

Theorem 2 *There is only one set of unique values which satisfy the second heuristic after the first heuristic is applied.*

Proof: From Theorem 1 there is only one set of values which satisfy the first heuristic. Hence all variables which represent the right or bottom edges of objects must have a minimum value. If the variable x represents the left or top edge of an object then the value of x will be the maximum allowed by all formulae of the form $(x + \alpha \leq y) \in C$, where C is the set of constraints. This maximum value will be the minimum of the $y - \alpha$ values for each constraint $(x + \alpha \leq y) \in C$. Since the maximum value is unique, the value for the variable is unique. □

4.3 The Data Structures

There are a number of data structures used in the constraint solution algorithms.

The value of q_{count} represents the total number of variables which must be updated before variable q is updated. For example, when assigning to a variable q a value which satisfies the first heuristic, q_{count} is equal to the number of constraints $(p+\gamma \leq q) \in C$ where it is not known if p has the correct value. When satisfying the second heuristic, q_{count} is equal to the number of constraints $(q+\omega \leq r) \in C$ where it is not known if r has the correct value. The q_{count} values are used in both the incremental and non-incremental algorithms to ensure that the value of a variable q is not used to determine the value of any other variable until it is known that the value of q is correct. The value of each q_{count} must be 0 both before and after any solution algorithm is performed.

Related to q_{count} are the values of $q_{indegree}$ and $q_{outdegree}$. $q_{indegree}$ equals the number of constraints $(p+\gamma \leq q) \in C$, and $q_{outdegree}$ equals the number of constraints $(q+\omega \leq r) \in C$. The cost of maintaining $q_{indegree}$ and $q_{outdegree}$ is amortized by including their maintenance cost with the cost of inserting (or deleting) a constraint involving q. For example, when a constraint $a+\gamma \leq b$ is added to C, $a_{outdegree}$ is increased by one, and $b_{indegree}$ is increased by one. The $q_{indegree}$ and $q_{outdegree}$ values are only used in the non-incremental algorithm.

Recall that q_{low} is the lowest value that can be assigned to variable q, and that $q_{low} \leq q$. For variables A_{right} and A_{bottom} it always holds that $q = q_{low}$ because of the first heuristic. Recall as well that the solutions S and S' are of the form

$$\{x_1 = \gamma_1,\ x_{1_{low}} = \omega_1,\ x_2 = \gamma_2,\ x_{2_{low}} = \omega_2,\ \ldots,\ x_n = \gamma_n,\ x_{n_{low}} = \omega_n\}$$

where γ_i and ω_i are non-negative integers. When a variable x is assigned a new value, the corresponding equation in S' is also changed.

q_{marked}, $q_{changed}$, $q_{initialized}$, and q_{update} are boolean flags which indicate certain properties of variable q. If q_{marked} is $true$, then variable q is considered to be "marked". A value of $true$ for $q_{changed}$ indicates that the value of q has changed. $q_{initialized}$ is $true$ if q has been assigned an initial value in the second half of the non-incremental algorithm. q_{update} equals $true$ if q should be updated. The purpose of these flags will be clearer in the context of the algorithms. All of these flags must have a value of $false$ both before and after any solution algorithm is performed.

The queue $Queue$ is a FIFO queue used in the non-incremental algorithm. The FIFO queues $Queue$, $Changed$, $Update_1$, and $Update_2$ are used in the incremental algorithm. The size of these queues is theoretically infinite. Only variables are added to or deleted from the queues.

The space complexity of both the incremental and non-incremental algorithms is $O(v + c)$, where v is the number of variables and c is the number of constraints in the current system.

4.4 The Non-Incremental Algorithm

The non-incremental ("batch") algorithm of ICOLA-4 ignores all previous solutions and evaluates all constraints to determine the absolute position of every object. The function which realizes the non-incremental algorithm is called $solve\_batch()$, and is divided into two parts. The first part, shown in Figure 4.4, is responsible for assigning to each variable a value which satisfies the first heuristic. The second part, shown in Figure 4.5, is responsible for satisfying the second heuristic. The second part is discussed in detail later.

Figure 4.4 shows the first part of the non-incremental algorithm. Lines (1-4) are responsible for initializing the value of each variable to 0, and for initializing the z_{count} value for each $z \in V$. In line (3) z_{count} is assigned $z_{indegree}$, which means that the value of z_{count} is equal to the number of constraints $(p + \alpha \leq z) \in C$ for some p and α. z_{count} will be used in lines (5-17) to ensure that z is not used to compute the value of any other variable until the value of z is known. The time complexity of these first four lines is $O(v)$, where v is the number of variables.

Each constraint in C must be considered, and each variable in V potentially changed. To accomplish its goal, the algorithm uses a breadth-first traversal strategy, beginning with $\texttt{root}_{left}$ and $\texttt{root}_{top}$, and ending at $\texttt{root}_{right}$ and $\texttt{root}_{bottom}$.

```
( 1)  foreach z ∈ V do begin
( 2)          z = 0
( 3)          z_count = z_indegree
( 4)  end
( 5)  add(Queue, root_left)
( 6)  add(Queue, root_top)
( 7)  while (¬empty(Queue)) do begin
( 8)          delete(Queue, z)
( 9)          z_low = z
(10)          foreach (z + γ ≤ q) ∈ C do begin
(11)                  q_count = q_count − 1
(12)                  if (z + γ > q) then
(13)                          q = z + γ
(14)                  if (q_count == 0) then
(15)                          add(Queue, q)
(16)          end
(17)  end
```

Figure 4.4: The Non-Incremental Algorithm - Part 1.

Lines (8-17) are performed until the FIFO queue $Queue$ is empty. The next element z is removed from $Queue$ (line (8)) and z_{low}[1] is assigned the current value of z (line (9)). For that variable z, each constraint of the form $(z + \gamma \leq q) \in C$ is considered, q_{count} is decreased by one, and the value of q is updated (line (13)) if $z + \gamma$ is greater than the current value of q. If q_{count} is 0, then all equations of the form $(p + \omega \leq q) \in C$ have been considered, and q can now be processed. If q is ready to be processed it is added to the queue $Queue$ in line (15). Since the value of q was initialized to 0 in the first part of the batch algorithm and since it is only changed if $z + \gamma > q$ (for some z and γ), it must be the case that q is assigned the lowest possible value, which satisfies the first heuristic. Since the value q_{count} is equal to the number of constraints $(p + \omega \leq q) \in C$, q_{count} will not equal 0 until q_{count} is decreased by 1 for each $(p + \omega \leq q) \in C$ considered by the breadth-first strategy

[1] Observe that assigning z_{low} the current value of z is only required if the incremental algorithm is expected to find a new solution based on a solution produced by the non-incremental algorithm. If the algorithms are not used in such an interchangeable fashion, then z_{low} is not required by the non-incremental algorithm.

in this part. This means that q will not be added to $Queue$ until the values of all variables p are known, and each $(p+\omega \leq q) \in C$ is considered. Since each constraint $(p+\omega \leq q) \in C$ is also considered exactly once, the time complexity of part 1 is $O(v+c)$, where v is the number of variables and c is the number of constraints.

```
( 1)  foreach z ∈ V do
( 2)          z_count = z_outdegree
( 3)  add(Queue, root_right)
( 4)  add(Queue, root_bottom)
( 5)  while (¬empty(Queue)) do begin
( 6)          delete(Queue, z)
( 7)          z_initialized = false
( 8)          foreach (p + γ ≤ z) ∈ C do begin
( 9)                  p_count = p_count − 1
(10)                  if (is_left(z) ∨ is_top(z)) then begin
(11)                          if (¬p_initialized) then begin
(12)                                  p = z − γ
(13)                                  p_initialized = true
(14)                          end
(15)                          else if (z − γ < p) then
(16)                                  p = z − γ
(17)                  end
(18)                  if (p_count == 0) then
(19)                          add(Queue, p)
(20)          end
(21)  end
```

Figure 4.5: The Non-Incremental Algorithm - Part 2.

The second part of the batch algorithm, shown in Figure 4.5, is responsible for satisfying the second heuristic (that objects are as small as possible). In this part of the algorithm changes are made only to variables which represent the left and top edges of objects. Lines (1) and (2) initialize z_{count} to the value of $z_{outdegree}$ for each $z \in V$. Hence the value of z_{count} is equal to the number of constraints $(z+\alpha \leq q) \in C$. The time complexity of the first two lines is $O(v)$.

Lines (10-19) determine the highest possible value for variables which represent the left and top edges of objects. This satisfies the second heuristic. $p_{initialized}$ is used to ensure that p is given an initial value only once. The value of $p_{initialized}$ is set to $true$ in line (13), just after p is given a starting value, and then set to $false$ in line (7) once the value of p is correct and p is added to the queue. After it is initialized, the value of p is updated only when it can be made larger. p will only be increased exactly as far as the $(p + \omega \leq q) \in C$ constraints allow, which satisfies the second heuristic. Line (18) ensures that p is not added to $Queue$ until all $(p + \omega \leq q) \in C$ are considered. Again, each variable is added to $Queue$ exactly once in lines (3-21) and each $(p + \omega \leq q) \in C$ is considered exactly once in lines (3-21). Hence the time complexity of the second part of the batch algorithm is $O(v + c)$, and the overall complexity of the batch algorithm is $O(v + c)$, where v is the number of variables and c is the number of constraints.

4.4.1 The Non-Incremental Algorithm Example

This section examines the behavior of the non-incremental algorithm on the simple picture description found in Figure 4.6. Only the left-to-right layout will be shown for this example. The algorithm determines the top-to-bottom layout in a similar fashion. Figure 4.7 shows the formulae corresponding to the left-to-right spatial layout of the picture description in Figure 4.6. The image corresponding to Figure 4.6 is found in Figure 4.8.

```
object( a, box ).
object( b, box ).
constraint( above, [ a, b ]).
constraint( left_of, [ right( a ), right( b ) ]).
```

Figure 4.6: One box to the left of another.

Figure 4.9 shows the *count* values for each variable after lines (1-4) in part 1 of the non-incremental algorithm are performed. The variable $\texttt{root}_{left}$ is the first variable removed from queue $Queue$. Variables $\texttt{root.a}_{left}$, $\texttt{root.b}_{left}$ and $\texttt{root}_{right}$ all have their *count* values decreased by 1, and the value of each variable remains

$$\begin{array}{ll}
\texttt{root}_{left} + 0 \leq_O \texttt{root}_{right}, & \texttt{root}_{left} + 0 \leq_O \texttt{root.a}_{left}, \\
\texttt{root}_{left} + 0 \leq_O \texttt{root.b}_{left}, & \texttt{root.a}_{right} + 0 \leq_O \texttt{root}_{right}, \\
\texttt{root.b}_{right} + 0 \leq_O \texttt{root}_{right}, & \texttt{root.a}_{left} + \delta \leq_O \texttt{root.a}_{right}, \\
\texttt{root.b}_{left} + \delta \leq_O \texttt{root.b}_{right}, & \texttt{root.a}_{right} + \delta \leq_C \texttt{root.b}_{right}
\end{array}$$

Figure 4.7: The left-to-right formulae for one box to the left of another.

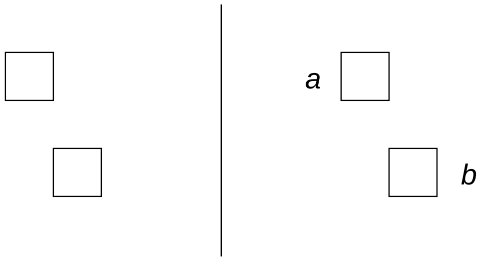


Figure 4.8: One box to the left of another.

at 0. $\texttt{root.a}_{left}$ and $\texttt{root.b}_{left}$ are both added to $Queue$ since their *count* values are 0. $\texttt{root.a}_{left}$ is removed from $Queue$ and the value of $\texttt{root.a}_{right}$ is updated to $0 + \delta = 0 + 20 = 20$. $\texttt{root.a}_{right_{count}}$ is decreased by 1, and since $\texttt{root.a}_{right_{count}}$ is 0, $\texttt{root.a}_{right}$ is added to $Queue$. Next, $\texttt{root.b}_{left}$ is removed from $Queue$. The value of $\texttt{root.b}_{right}$ is updated to 20, and $\texttt{root.b}_{right_{count}}$ is decreased by 1 to 1. $\texttt{root.a}_{right}$ is removed from $Queue$. Both $\texttt{root.b}_{right_{count}}$ and $\texttt{root}_{right_{count}}$ are decreased by 1. $\texttt{root.b}_{right}$ is changed to 40 and $\texttt{root}_{right}$ is changed to 20. Since $\texttt{root.b}_{right_{count}}$ is now 0, $\texttt{root.b}_{right}$ is now added to $Queue$. $\texttt{root.b}_{right}$ is then removed from $Queue$ and $\texttt{root}_{right_{count}}$ is decreased by 1 to 0. $\texttt{root}_{right}$ is changed to 40, and $\texttt{root}_{right}$ is added to $Queue$. $\texttt{root}_{right}$ is removed from $Queue$, at which point the first heuristic is satisfied.

$$\begin{array}{lll}
\texttt{root}_{left_{count}} = 0, & \texttt{root}_{right_{count}} = 3, & \texttt{root.a}_{left_{count}} = 1, \\
\texttt{root.a}_{right_{count}} = 1, & \texttt{root.b}_{left_{count}} = 1, & \texttt{root.b}_{right_{count}} = 2
\end{array}$$

Figure 4.9: The *count* values after lines (1-4) of part 1 of the non-incremental algorithm.

$$\begin{array}{llll}
\texttt{root}_{left_{low}} = 0, & \texttt{root}_{left} = 0, & \texttt{root}_{right_{low}} = 40, & \texttt{root}_{right} = 40, \\
\texttt{root.a}_{left_{low}} = 0, & \texttt{root.a}_{left} = 0, & \texttt{root.a}_{right_{low}} = 20, & \texttt{root.a}_{right} = 20, \\
\texttt{root.b}_{left_{low}} = 0, & \texttt{root.b}_{left} = 0, & \texttt{root.b}_{right_{low}} = 40, & \texttt{root.b}_{right} = 40
\end{array}$$

Figure 4.10: The variable assignments after part 1 ($\delta = 20$).

Figure 4.10 shows the variable assignments after part 1 of the batch algorithm is performed. Part 2 of the batch algorithm behaves in a very similar fashion to part 1. Figure 4.11 shows the *count* values after lines (1-2) of part 2 of the batch algorithm are performed. The algorithm in part 2 proceeds from right-to-left, potentially changing only the values of those variables which represent the left or top edges of an object. In this example, only $\texttt{root.b}_{left}$ is changed from a value of 0 after part 1, to a value of 20 after part 2. Figure 4.12 shows the final solution which corresponds to the image in Figure 4.8.

$$\begin{array}{lll}
\texttt{root}_{left_{count}} = 3, & \texttt{root}_{right_{count}} = 0, & \texttt{root.a}_{left_{count}} = 1, \\
\texttt{root.a}_{right_{count}} = 2, & \texttt{root.b}_{left_{count}} = 1, & \texttt{root.b}_{right_{count}} = 1
\end{array}$$

Figure 4.11: The *count* values after lines (1-2) of part 2 of the non-incremental algorithm.

$$\begin{array}{llll}
\texttt{root}_{left_{low}} = 0, & \texttt{root}_{left} = 0, & \texttt{root}_{right_{low}} = 40, & \texttt{root}_{right} = 40, \\
\texttt{root.a}_{left_{low}} = 0, & \texttt{root.a}_{left} = 0, & \texttt{root.a}_{right_{low}} = 20, & \texttt{root.a}_{right} = 20, \\
\texttt{root.b}_{left_{low}} = 0, & \texttt{root.b}_{left} = 20, & \texttt{root.b}_{right_{low}} = 40, & \texttt{root.b}_{right} = 40
\end{array}$$

Figure 4.12: The new solution ($\delta = 20$).

4.5 The Incremental Algorithm

The incremental constraint solver is the most important component of ICOLA-4. The algorithm for the incremental constraint solver is realized as two functions –

incremental_solve_insert() and *incremental_solve_delete*(). The first function, *incremental_solve_insert*(), is responsible for finding a new solution given the insertion of a new constraint. The second function, *incremental_solve_delete*(), is responsible for finding a new solution given the deletion of an existing constraint. Each function returns the new solution. Recall that the constraints are known to be consistent before either function is called.

4.5.1 Incremental Constraint Insertion

Incremental solution algorithms rely heavily on the previous solution when determining a new solution. The algorithms attempt to change only those portions of the previous solution which require updating. The solution for the picture description in Figure 4.13 is shown graphically as the image in Figure 4.14. The incremental addition of the constraint

```
constraint( left_of, [ left( c ), right( g ) ])
```

will result in the image seen in Figure 4.15, but only the positions of objects `g` and `h` should be re-computed by the incremental constraint solver. Determining which variables of the constraint system must change is the job of the incremental solution algorithm. In this section the *incremental_solve_insert*() function is explored. The example shown in Figure 4.14 will be used throughout this section. A more detailed discussion of this example is presented in Section 4.5.2.

Given $S' = incremental\_solve\_insert(C,\ (x + \alpha \leq y),\ S)$, the goal of the incremental insertion algorithm is to construct a new solution S' based on the previous set of constraints C, the previous solution S, and the new constraint $x + \alpha \leq y$. Figure 4.16 gives an overview of the algorithm.

The high-level descriptions of the five parts indicated in Figure 4.16 are as follows:

Part 1 Determines the set of variables which may have to change as a result of the constraint insertion.

```
object( a, box ).
object( b, box ).
constraint( left_of, [ a, b ]).
object( c, box ).
constraint( left_of, [ b, c ]).
object( d, box ).
constraint( left_of, [ c, d ]).
object( e, box ).
constraint( left_of, [ d, e ]).
object( f, box ).
object( g, box ).
constraint( left_of, [ f, g ]).
object( h, box ).
constraint( left_of, [ g, h ]).
constraint( left_of, [ h, e ]).
constraint( above, [ a, f ]).
constraint( above, [ a, g ]).
constraint( above, [ a, h ]).
```

Figure 4.13: Picture description for the incremental insertion example.

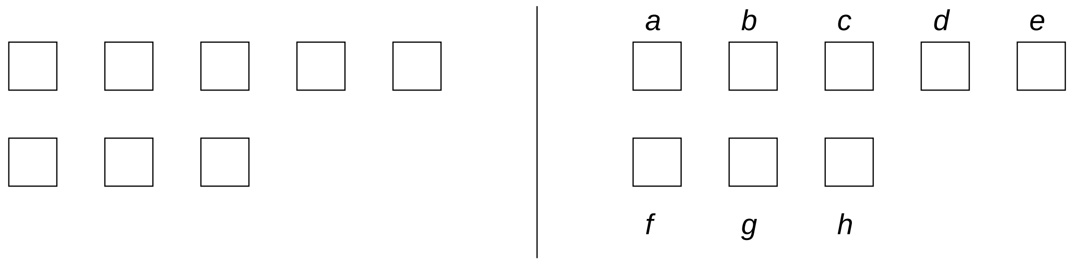


Figure 4.14: The image produced from the picture description in Figure 4.13.

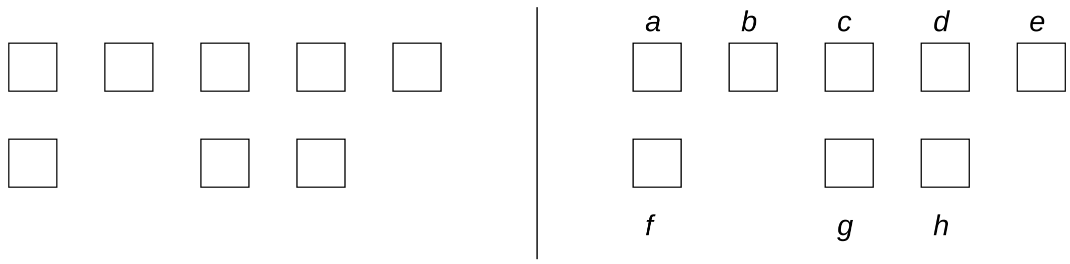


Figure 4.15: The image after the addition of the new constraint.

```
incremental_solve_insert(C, (x + α ≤ y), S) begin
        S' = S
        C' = C ⊎ {x + α ≤ y}
        if ((x + α > y) ∨ (x_low + α > y_low)) then begin
                Part 1
                Part 2
                Part 3
                Part 4
                Part 5
        end
        return S'
end
```

Figure 4.16: Overview of the incremental insertion algorithm.

Part 2 Updates, according to the rules of the first heuristic, the variables determined in part 1.

Part 3 Starts to determine the set of variables which may have to be changed to satisfy the second heuristic. Part 3 determines those variables directly affected by the updates in part 2.

Part 4 Finishes the work begun in part 3 by determining the variables indirectly affected by the updates in part 2. Part 4 also determines the variables to be used as the starting point for Part 5.

Part 5 Determines the correct values, according to the second heuristic, for the sets of variables determined in parts 3 and 4.

As will be seen in Section 4.5.3, parts 1, 3, and 5 are also used by the incremental deletion algorithm.

The first step of the incremental insertion algorithm is to determine if the addition of the constraint $x + \alpha \leq y$ to C will require a change to S', or if the constraint is already true. If $x + \alpha \leq y$ and $x_{low} + \alpha \leq y_{low}$ then no variables need to be changed and *incremental_solve_insert*() is finished. Otherwise, $x + \alpha > y$ or $x_{low} + \alpha > y_{low}$, and a new solution must be computed. If $x + \alpha > y$, then one of x or y must change.

If $x_{low} + \alpha > y_{low}$, then y_{low} (and y) must change. If $x_{low} + \alpha \leq y_{low}$, then y_{low} (and y) will remain the same, and x will have to change. x_{low} will never change given the constraint $x + \alpha \leq y$, as x_{low} is already as small as possible. Note that the consistency check (performed before the *incremental_solve_insert*() function is called) ensures that the above changes can be made to x, y, or y_{low} and that a solution can be found.

The following discussion focuses on the case where $x + \alpha > y$ or $x_{low} + \alpha > y_{low}$ and the new solution S' must be updated. The first priority of the update is to ensure that the first heuristic is satisfied. This is accomplished by parts 1 and 2 of the incremental insertion algorithm. Let $Queue$ be a FIFO queue. Part 1 of the incremental insertion algorithm is shown in Figure 4.17.

```
add(Queue, y)
while (¬empty(Queue)) do begin
        delete(Queue, z)
        foreach (z + γ ≤ q) ∈ C' do begin
                q_count = q_count + 1
                if (¬q_marked) then begin
                        add(Queue, q)
                        q_marked = true
                end
        end
end
```

Figure 4.17: The Incremental Insertion Algorithm - Part 1.

Given the insertion of the constraint $x + \alpha \leq y$, the goal of part 1 of the incremental insertion algorithm is to identify all those variables that could be affected by a change in the value of variable y. In the example in Figure 4.14, this corresponds to determining which objects or edges of objects must move if the position of the right edge of `g` changes. Since the constraints in Figure 4.13 state that object `g` is to the left of `h`, and that `h` is to the left of `e`, a change to the position of the right edge of `g` may affect the positions of the left and right edges of `h` and `e`. It is the goal of part 1 of the algorithm to identify that `h` and `e` may have to move to satisfy the constraint

```
constraint( left_of, [ left( c ), right( g ) ]).
```

If G is the weighted, directed graph of C', then the variables "marked" as requiring potential changes will be those corresponding to the nodes which *depend* on y (See Section 4.1). Figure 4.18 shows a portion of the left-to-right ordering in graph G. All variables $p_1, p_2, \ldots, p_n$, represented in G by nodes $p_1, p_2, \ldots, p_n$, may have to change if the value of y changes. If any of the p_i change, then any variables which depend on p_i may also have to change, and so on. Part 1 of the incremental insertion algorithm captures these relationships. For each variable q which may have to change, q_{count} is set equal to the number of constraints $(p + \omega \leq q) \in C'$ where p might have to change. The time complexity of part 1 is $O(v + c)$, where v is the number of variables and c is the number of constraints.

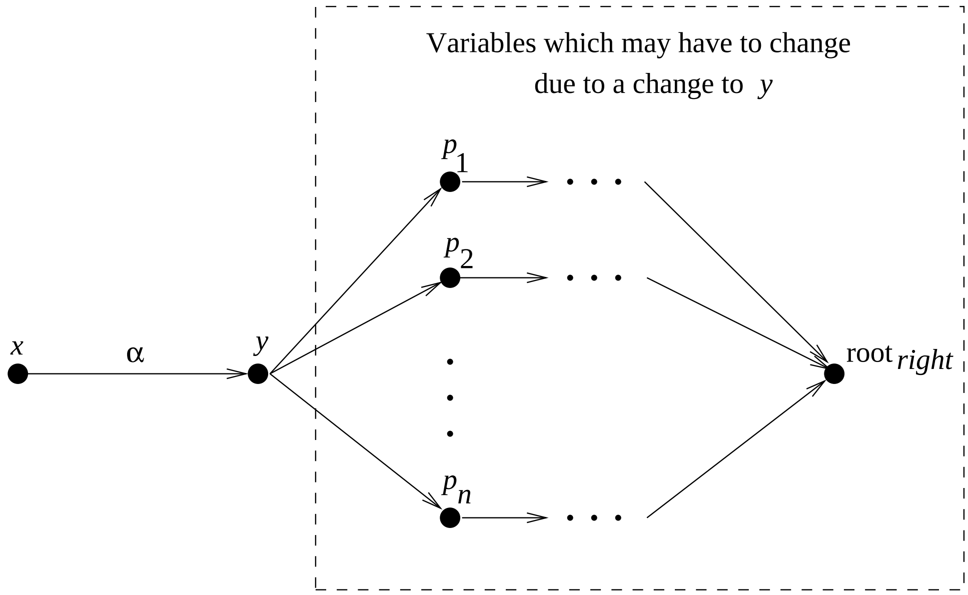


Figure 4.18: Nodes in G which depend on y.

Part 2 of the incremental insertion algorithm (Figure 4.19) ensures that the value of the variables which depend on y are as low as possible. In Figure 4.14, the positions of the right edges of objects `h` and `e` (which were identified in part 1 as potentially requiring changing) are considered. The expected result after part 2 is

```
if (x_low + α > y_low) then
        y_low = x_low + α
y_changed = true
add(Queue, y)
add(Changed, y)
while (¬empty(Queue)) do begin
        delete(Queue, z)
        z_marked = false
        if (z_changed) then
                z = z_low
        foreach (z + γ ≤ q) ∈ C' do begin
                q_count = q_count − 1
                if (z_changed) then
                        if (z_low + γ > q_low) then begin
                                q_low = z_low + γ
                                q_changed = true
                        end
                if (q_count == 0) then begin
                        add(Queue, q)
                        if (q_changed) then
                                add(Changed, q)
                end
        end
end
```

Figure 4.19: The Incremental Insertion Algorithm - Part 2.

the image shown in Figure 4.20. Note that this image is just a representation of the solution after part 2 – this image is never shown to the user. Also note that the position of the left edge of object g does not satisfy the second heuristic. This will be corrected in parts 3, 4, and 5.

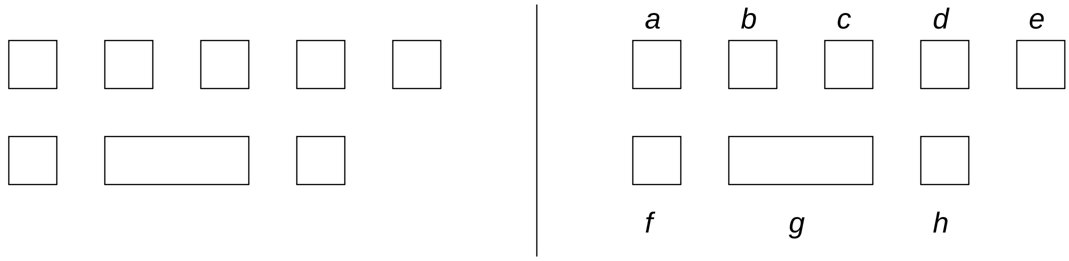


Figure 4.20: The image for the (internal) solution after part 2.

Recall that, in general, the constraint inserted is $x + \alpha \leq y$. If $x_{low} + \alpha > y_{low}$ (line (1)), then it is not sufficient to decrease the value of x or simply increase the value of y, as y_{low} must change. Note that if x represents the right or bottom edge of an object that the value of x cannot change – it must already be as low as possible. If x represents the left or top edge of an object, then the value of x in the previous solution was determined by each q, $(x + \omega \leq q) \in C'$.

Irrespective of whether y_{low} is assigned a new value, y is marked as changed, and added both to *Queue* and *Changed*. The FIFO queue *Changed* contains those variables which were marked in part 1, and which have their values changed in part 2. These variables are added to *Changed* in the order they are updated in part 2 and form the starting point for determining which variables need to be changed to satisfy the second heuristic. As each variable z is removed from *Queue* the "marked" flag for z is cleared, which is only to ensure that the flag is ready for use in part 3. If $z_{changed}$ is $true$, then z is assigned the value of z_{low} in line (10). This ensures that z has the lowest possible value and that the first heuristic is satisfied. Note that the value of z may be increased in part 5 of the algorithm. For each z removed from *Queue*, constraints of the form $(z + \gamma \leq q) \in C'$ are considered in line (11). If the value of z has changed, then an attempt is made to update q. If the value of z has not changed, then there is no need to re-compute the value of q with respect to z. This is due to the fact that the constraint $(z + \gamma \leq q) \in C'$ is already true from the previous solution. Note that the new constraint added to C, $x + \alpha \leq y$, is never considered in part 2, as y is initially added to the queue and x can never be reached if y is the starting point and C' is consistent.

If $z_{low} + \gamma > q_{low}$ (line (14)) then q_{low} is updated, and marked as changed. Note that the value of q_{low} is always increased – it is never decreased. This means that q_{low} always has the lowest possible value with respect to the constraints, and that q_{low} satisfies the first heuristic. Since z is assigned z_{low} if z_{low} has changed, z also satisfies the first heuristic. If every constraint $(p + \omega \leq q) \in C'$ has been considered, where p was marked in part 1, then q is added to *Queue*. As well, if the value of q_{low} has changed, then q is added to the *Changed* queue. The time complexity of part 2 is also $O(v + c)$, as each variable is added to $Queue$ at most once and added to $Changed$ at most once, and each constraint is considered at most once.

Theorem 3 *Given the set of variables V, the previous set of constraints C, the previous solution S, and the insertion of the constraint $(x + \alpha \leq y)$, after parts 1 and 2 are complete the first heuristic is satisfied, and the value of p_{low} in S' is correct for each $p \in V$.*

Proof: Let $C' = C \uplus \{x + \alpha \leq y\}$. For each $p \in V$:

1. If p_{low} is potentially affected by a change to y_{low}, then p_{low} will be marked in part 1. In part 2, p_{low} will be changed, as necessary, if any r_{low}, $(r+\gamma \leq p) \in C'$, changes. If no such r_{low} value changes, then it is unnecessary to change p_{low} as the value of p_{low} is already correct (from the previous solution S).

2. If the value of p_{low} is not potentially affected by a change to y_{low} then y and p are independent, or y would be potentially affected by a change to p.

If the value of a variable q does not equal the value q_{low} after part 2, then the following must be true:

1. A change to y would never cause the value of q to change. That is, either y and q are independent, or y depends on q.

2. The value of q_{low} was not updated (which means it was already as low as possible, and hence already correct), and q must represent the left or top edge of an object.

Since the weighted, directed graph G induced by V and C' is acyclic, the algorithm must terminate. □

In the example shown in Figure 4.20, the left edge of object g will be identified as potentially not satisfying the second heuristic. The goal of parts 3 and 4 is to identify the edges of objects which may be positioned incorrectly. Note that only variables which represent the left or top edge of an object need to be considered (line

```
( 1)  while (¬empty(Changed)) do begin
( 2)        delete(Changed, z)
( 3)        if (is_left(z) ∨ is_top(z)) then
( 4)              add(Update_1, z)
( 5)        else z_changed = false
( 6)        foreach (p + γ ≤ z) ∈ C' do
( 7)              if (is_left(p) ∨ is_top(p)) then begin
( 8)                    if (is_left(z) ∨ is_top(z)) then
( 9)                          p_count = p_count + 1
(10)                    if (¬p_changed ∧ ¬p_marked) then begin
(11)                          add(Update_2, p)
(12)                          p_marked = true
(13)                    end
(14)              end
(15)  end
```

Figure 4.21: The Incremental Insertion Algorithm - Part 3.

(7)). Since satisfying the second heuristic involves increasing the values of variables, it is constraints of the form $(p + \gamma \leq z) \in C'$ (line (6)) which are important when determining the new value for p. Hence p_{count} will be increased by 1 (line (9)) for each constraint of the form $(p + \gamma \leq z) \in C'$ where the value of z is not known to be correct (i.e. has the potential to be updated).

Figure 4.21 shows part 3 of the incremental insertion algorithm. Part 3 simply determines two other queues of variables, $Update_1$ and $Update_2$. Figure 4.22 shows a portion of the left-to-right layout of graph G. $Update_1$ contains the variables from $Changed$ which represent the left edges of objects. $Update_2$ contains the variables corresponding to nodes $r_1, r_2, \ldots, r_n$. From Figure 4.20, the right edge of object `g`, and the left and right edges of object `h` will be present in queue $Changed$. $Update_1$ will contain the variable representing the left edge of object `h`. Among other variables (See Section 4.5.2), $Update_2$ will contain the variable representing the left edge of object `g`, as the position of the left edge of `g` needs to be changed.

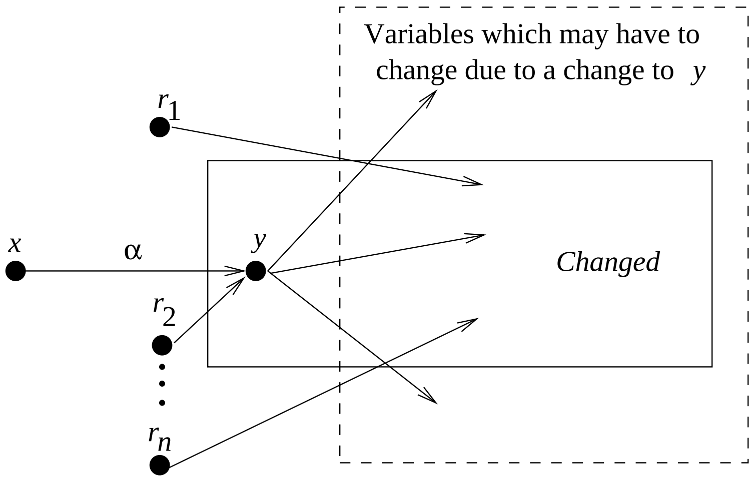


Figure 4.22: Determining the contents of $Update_2$.

The queues $Update_1$ and $Update_2$ in part 3 are defined as:

$$Update_1 = \{z|\ z \in Changed \wedge (is\_left(z) \vee is\_top(z))\}$$
$$Update_2 = \{p|\ (p + \gamma \leq z) \in C' \wedge z \in Changed \wedge (is\_left(p) \vee is\_top(p))\}$$

$Update_1$ contains variables which represent the left or top edges of an object and that may have to change as a result of the changes made to variables in part 2. The queue $Update_2$ contains those variables p, $(p + \gamma \leq z) \in C'$, where p represents the left or top edge of an object and $z_{changed}$ is $true$. Part 3 begins the construction of $Update_1$ and $Update_2$ with the variables in $Changed$. Those variables in $Changed$ which represent the left or top edge of an object are immediately added to the queue $Update_1$. In line (6) each constraint of the form $(p + \gamma \leq z) \in C'$ is considered for each variable z removed from $Changed$. If $is\_left(p)$ or $is\_top(p)$ is true, then the change to z may cause a change to the value of p, and hence p is added to $Update_2$. p is also marked in line (12) to ensure it is not added to the queue $Update_2$ more than once. The time complexity of part 3 is also $O(v + c)$, as there are at most v variables in $Changed$, each variable is considered at most once, and each of the c constraints is considered at most once.

Part 4 is shown in Figure 4.23. Lines (1-12) obtain the remaining variables from V which represent the left or top edge of an object, and which may change due to the changes to the variables added to $Changed$ in part 2. Each variable z removed

(1) `while` $(\neg empty(Update_2))$ `do begin`
(2) $\quad delete(Update_2, z)$
(3) $\quad add(Update_1, z)$
(4) $\quad$ `foreach` $(p + \gamma \leq z) \in C'$ `do`
(5) $\qquad$ `if` $(is\_left(p) \vee is\_top(p))$ `do begin`
(6) $\qquad\quad p_{count} = p_{count} + 1$
(7) $\qquad\quad$ `if` $(\neg p_{changed} \wedge \neg p_{marked})$ `then begin`
(8) $\qquad\qquad add(Update_2, p)$
(9) $\qquad\qquad p_{marked} = true$
(10) $\qquad\quad$ `end`
(11) $\qquad$ `end`
(12) `end`
(13) `while` $(\neg empty(Update_1))$ `do begin`
(14) $\quad delete(Update_1, z)$
(15) $\quad$ `if` $(z_{count} == 0)$ `then`
(16) $\qquad add(Queue, z)$
(17) `end`

Figure 4.23: The Incremental Insertion Algorithm - Part 4.

from $Update_2$ is added to $Update_1$. A change to the value of z may result in the need to change p, $(p+\gamma \leq z) \in C'$, where $is\_left(p)$ or $is\_top(p)$ is true. Lines (4-11) obtain these variables p provided they are not already in $Update_1$ or $Update_2$. Line (6) also ensures that for the constraints $(p + \omega \leq q) \in C'$ that q has the correct value before the value of q is used to update p. The time complexity of lines (1-12) is $O(v+c)$, as each variable is removed from $Update_2$ only once, and each constraint is considered at most once.

Lines (13-17) of part 4 add the variables z from $Update_1$ to queue $Queue$ where all q in each $(z + \gamma \leq q) \in C'$ are already known (i.e. where each z_{count} equals 0). Since the time complexity of lines (13-17) is $O(v)$, the total time complexity of part 4 is $O(v + c)$, where v is the number of variables and c is the number of constraints.

Part 5 of the incremental insertion algorithm satisfies the second heuristic, which is to make objects as small as possible. This is achieved by increasing the value of variables of the form A_{left} and A_{top} as much as the constraints and the first heuristic will allow. The effect of part 5 on the image in Figure 4.20 is to move the left edge of object g to the right, and hence make g as small as possible. Recall that in parts 3 and 4 the left edge of g was identified as potentially requiring a change. Part 5 corrects the position of the left edge of g, and the final result is the image in Figure 4.24.

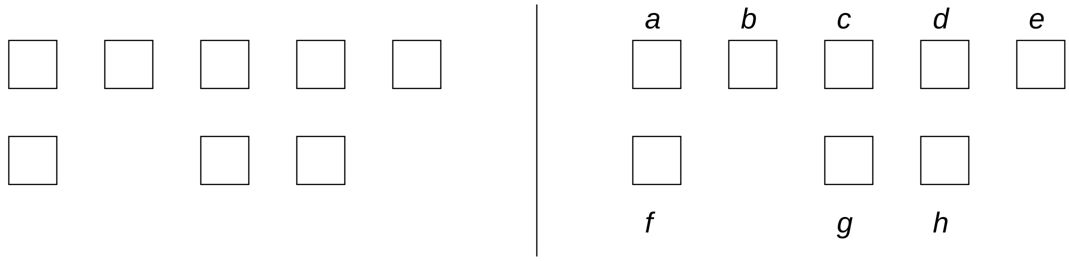


Figure 4.24: The final image of the incremental insertion example.

```
while (¬empty(Queue)) do begin
        delete(Queue, z)
        z_changed = false
        z_marked = false
        z_update = false
        if (is_left(z)) then
                z = root_right
        else if (is_top(z)) then
                z = root_bottom
        foreach (z + γ ≤ q) ∈ C' do
                if (q − γ < z) then
                        z = q − γ
        foreach (p + ω ≤ z) ∈ C' do
                if (is_left(p) ∨ is_top(p)) then begin
                        p_count = p_count − 1
                        if (p_count == 0) then
                                add(Queue, p)
                end
end
```

Figure 4.25: The Incremental Insertion Algorithm - Part 5.

Figure 4.25 shows part 5 of the incremental insertion algorithm. The queue $Queue$ contains at least one variable, since if y is A_{left} or A_{top} then y is in $Queue$, or if y is A_{right} or A_{bottom} then A_{left} or A_{top}, respectively, is added to $Queue$ in part 3.

After variable z is removed from $Queue$, the flags $z_{changed}$, z_{marked}, and z_{update} are all set to $false$ to ensure that these flags are $false$ after the algorithm completes. In lines (6-9) z is assigned an initial value. The value of z is immediately corrected in lines (10-12), where, for each constraint $(z + \gamma \leq q) \in C'$, z is assigned the value $q - \gamma$ if $q - \gamma < z$. Lines (13-18) are responsible for ensuring that all p are updated properly, where $is\_left(p)$ or $is\_top(p)$ is true. Lines (16-17) ensure that each variable p is added to $Queue$ if all variables r, $(p + \omega \leq r) \in C'$, that needed to change have been updated. Each variable p is considered exactly once. For each p, each constraint $(p + \gamma \leq q) \in C'$ is considered exactly once, and each constraint $(r + \omega \leq p) \in C'$ is considered exactly once. Hence the time complexity of part 5 is $O(v + c)$ where v is the number of variables and c is the number of constraints.

Since the time complexity of each component is $O(v + c)$, the total complexity of the incremental insertion algorithm is $O(v + c)$. Note, however, that the number of variables and constraints considered is typically only a small fraction of the total variables and constraints. As seen in Section 5.4, the real performance of the incremental insertion algorithm depends on the properties of the picture description, and the performance can be quite good.

Theorem 4 *After parts 3, 4, and 5 are performed the second heuristic is satisfied, and the value for each $p \in V$ is correct in S'.*

Proof: Parts 3 and 4 only add q to the queue $Queue$ if $is\_left(q)$ or $is\_top(q)$ is true, and if the value of q may be affected by a change to a variable added to the queue $Changed$ in part 2. In part 5, q is assigned an initial value that is the highest possible value – $\texttt{root}_{right}$ or $\texttt{root}_{bottom}$. The value of q is then decreased such that for each constraint $(z+\gamma \leq q) \in C'$, z is assigned $q-\gamma$ if $q-\gamma < z$ and if the value of q is known to be correct. This ensures that z is assigned the highest possible value with respect to the constraints and the first heuristic, and hence that the value of z is correct. If the value of some variable z does not change, then z was not added to $Changed$ in part 2, and hence must be correct from the previous solution S.

Since the graph G induced by V and C is acyclic, the algorithm must terminate.
□

4.5.2 The Incremental Insertion Example

It can be quite difficult to appreciate the incremental performance of the incremental insertion algorithm without seeing its behavior on an example picture description. Unfortunately, even small picture descriptions have large sets of constraints and variables. A picture description which contains only two or three objects does not allow the full incremental nature of the algorithm to be shown. The following example was chosen so that the incremental behavior of the algorithm can be clearly shown in a single example.

The picture description used in this example is found in Figure 4.26, and the corresponding image is seen in Figure 4.27. Note that this discussion of the incremental insertion algorithm will focus only on the left-to-right layout of the objects, and that the top-to-bottom ordering is ignored. Figure 4.28 shows the formulae representing the left-to-right relationships in the picture description in Figure 4.26. Figure 4.29 shows the graph representation of the set of constraints and variables. In all graph representations in this section the dashed arcs have a weight of 0, and the solid arcs have a weight of 20 (the value of the default distance, δ, is 20). The solution to the constraints is seen in Figure 4.30. Figure 4.31 shows the solution in the graph representation. For each node z, the value of the variable is indicated in brackets ("[]"), and the low value of the variable (z_{low}) is shown outside of the brackets. This solution is the starting point for this example of the behavior of incremental insertion algorithm.

The essence of this example is the addition of the constraint

```
constraint( left_of, [ left( c ), right( g ) ]),
```

which is equivalent to adding the formula $\texttt{root.c}_{left} + \delta \leq_C \texttt{root.g}_{right}$ to the current set of internal constraints. Figure 4.32 shows the graph representation of the constraints after the addition of the new constraint. As the graph shows, some of

```
object( a, box ).
object( b, box ).
constraint( left_of, [ a, b ]).
object( c, box ).
constraint( left_of, [ b, c ]).
object( d, box ).
constraint( left_of, [ c, d ]).
object( e, box ).
constraint( left_of, [ d, e ]).
object( f, box ).
object( g, box ).
constraint( left_of, [ f, g ]).
object( h, box ).
constraint( left_of, [ g, h ]).
constraint( left_of, [ h, e ]).
constraint( above, [ a, f ]).
constraint( above, [ a, g ]).
constraint( above, [ a, h ]).
```

Figure 4.26: Picture description for the incremental insertion example.

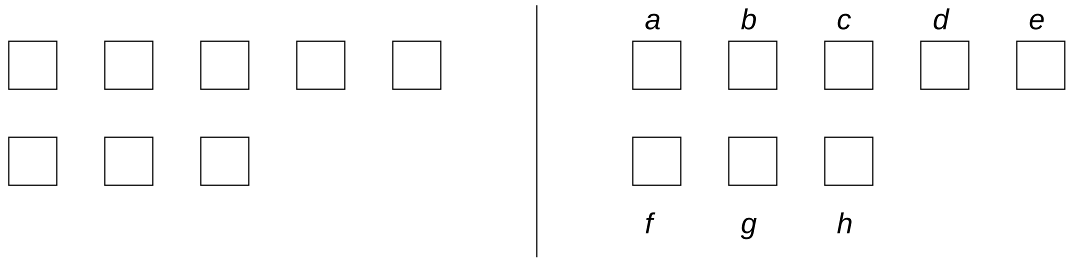


Figure 4.27: The image produced by the picture description in Figure 4.26.

the values for the variables are incorrect, and must be updated by the constraint solver.

Recall that the consistency of the new constraint with the old set of constraints is already assured before *incremental_solve_insert*() is called. This is verified by Figure 4.32, which shows that the graph of the left-to-right constraints is acyclic, and hence that the constraints are consistent.

The first step performed by the incremental insertion algorithm is to add the constraint to the current set of constraints, and then determine if this new constraint

$$
\begin{array}{ll}
\mathtt{root}_{left} + 0 \leq_O \mathtt{root}_{right}, & \mathtt{root}_{left} + 0 \leq_O \mathtt{root.a}_{left}, \\
\mathtt{root}_{left} + 0 \leq_O \mathtt{root.b}_{left}, & \mathtt{root}_{left} + 0 \leq_O \mathtt{root.c}_{left}, \\
\mathtt{root}_{left} + 0 \leq_O \mathtt{root.d}_{left}, & \mathtt{root}_{left} + 0 \leq_O \mathtt{root.e}_{left}, \\
\mathtt{root}_{left} + 0 \leq_O \mathtt{root.f}_{left}, & \mathtt{root}_{left} + 0 \leq_O \mathtt{root.g}_{left}, \\
\mathtt{root}_{left} + 0 \leq_O \mathtt{root.h}_{left}, & \mathtt{root.a}_{right} + 0 \leq_O \mathtt{root}_{right}, \\
\mathtt{root.b}_{right} + 0 \leq_O \mathtt{root}_{right}, & \mathtt{root.c}_{right} + 0 \leq_O \mathtt{root}_{right}, \\
\mathtt{root.d}_{right} + 0 \leq_O \mathtt{root}_{right}, & \mathtt{root.e}_{right} + 0 \leq_O \mathtt{root}_{right}, \\
\mathtt{root.f}_{right} + 0 \leq_O \mathtt{root}_{right}, & \mathtt{root.g}_{right} + 0 \leq_O \mathtt{root}_{right}, \\
\mathtt{root.h}_{right} + 0 \leq_O \mathtt{root}_{right}, & \mathtt{root.a}_{left} + \delta \leq_O \mathtt{root.a}_{right}, \\
\mathtt{root.b}_{left} + \delta \leq_O \mathtt{root.b}_{right}, & \mathtt{root.c}_{left} + \delta \leq_O \mathtt{root.c}_{right}, \\
\mathtt{root.d}_{left} + \delta \leq_O \mathtt{root.d}_{right}, & \mathtt{root.e}_{left} + \delta \leq_O \mathtt{root.e}_{right}, \\
\mathtt{root.f}_{left} + \delta \leq_O \mathtt{root.f}_{right}, & \mathtt{root.g}_{left} + \delta \leq_O \mathtt{root.g}_{right}, \\
\mathtt{root.h}_{left} + \delta \leq_O \mathtt{root.h}_{right}, & \mathtt{root.a}_{right} + \delta \leq_C \mathtt{root.b}_{left}, \\
\mathtt{root.b}_{right} + \delta \leq_C \mathtt{root.c}_{left}, & \mathtt{root.c}_{right} + \delta \leq_C \mathtt{root.d}_{left}, \\
\mathtt{root.d}_{right} + \delta \leq_C \mathtt{root.e}_{left}, & \mathtt{root.f}_{right} + \delta \leq_C \mathtt{root.g}_{left}, \\
\mathtt{root.g}_{right} + \delta \leq_C \mathtt{root.h}_{left}, & \mathtt{root.h}_{right} + \delta \leq_C \mathtt{root.e}_{left}
\end{array}
$$

Figure 4.28: The left-to-right formulae.

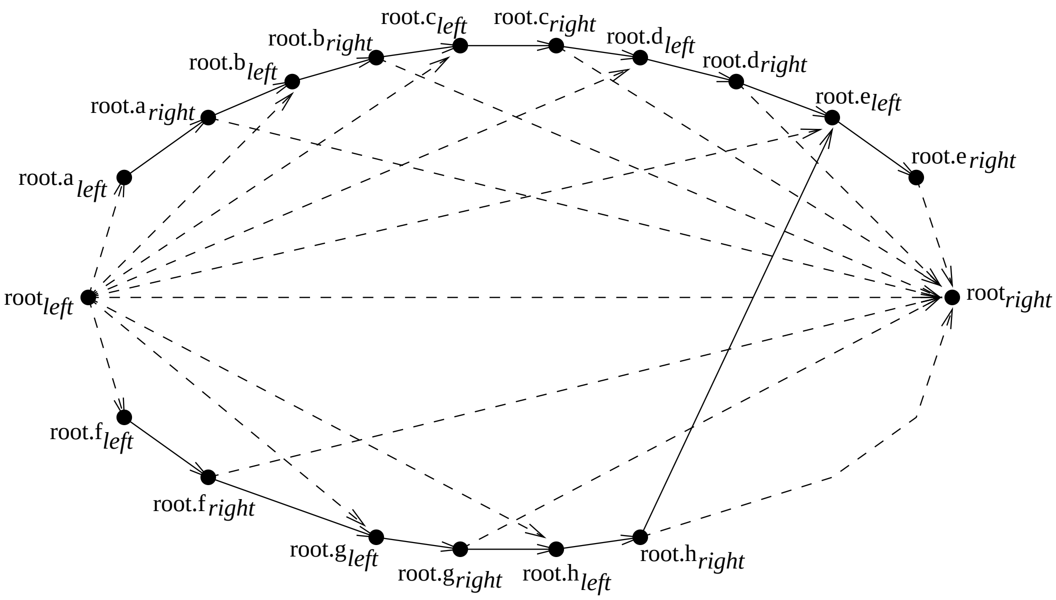


Figure 4.29: The graph representation of the left-to-right formulae.

may cause a change to the current solution. Since $\delta = 20$, $\mathtt{root.c}_{left} = 80$, and $\mathtt{root.g}_{right} = 60$, it is obvious that $\mathtt{root.c}_{left} + \delta \leq_C \mathtt{root.g}_{right}$ is not true, and that the value of $\mathtt{root.c}_{left}$ or $\mathtt{root.g}_{right}$ or both may have to change. The value of

$\texttt{root}_{left_{low}} = 0,$	$\texttt{root}_{left} = 0,$	$\texttt{root}_{right_{low}} = 180,$	$\texttt{root}_{right} = 180,$
$\texttt{root.a}_{left_{low}} = 0,$	$\texttt{root.a}_{left} = 0,$	$\texttt{root.a}_{right_{low}} = 20,$	$\texttt{root.a}_{right} = 20,$
$\texttt{root.b}_{left_{low}} = 40,$	$\texttt{root.b}_{left} = 40,$	$\texttt{root.b}_{right_{low}} = 60,$	$\texttt{root.b}_{right} = 60,$
$\texttt{root.c}_{left_{low}} = 80,$	$\texttt{root.c}_{left} = 80,$	$\texttt{root.c}_{right_{low}} = 100,$	$\texttt{root.c}_{right} = 100,$
$\texttt{root.d}_{left_{low}} = 120,$	$\texttt{root.d}_{left} = 120,$	$\texttt{root.d}_{right_{low}} = 140,$	$\texttt{root.e}_{right} = 140,$
$\texttt{root.e}_{left_{low}} = 160,$	$\texttt{root.e}_{left} = 160,$	$\texttt{root.e}_{right_{low}} = 180,$	$\texttt{root.e}_{right} = 180,$
$\texttt{root.f}_{left_{low}} = 0,$	$\texttt{root.f}_{left} = 0,$	$\texttt{root.f}_{right_{low}} = 20,$	$\texttt{root.f}_{right} = 20,$
$\texttt{root.g}_{left_{low}} = 40,$	$\texttt{root.g}_{left} = 40,$	$\texttt{root.g}_{right_{low}} = 60,$	$\texttt{root.g}_{right} = 60,$
$\texttt{root.h}_{left_{low}} = 80,$	$\texttt{root.h}_{left} = 80,$	$\texttt{root.h}_{right_{low}} = 100,$	$\texttt{root.h}_{right} = 100$

Figure 4.30: The current solution for the incremental insertion example ($\delta = 20$).

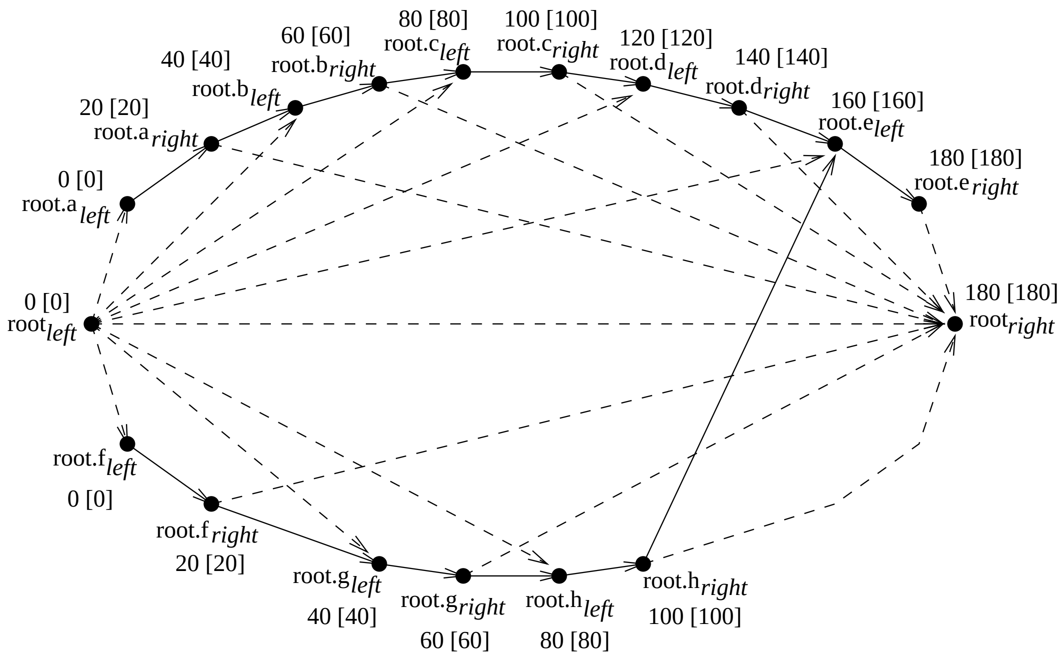


Figure 4.31: The graph representation of the current (left-to-right) solution.

$\texttt{root.c}_{left}$ cannot change since $\texttt{root.c}_{left}$ is already as low as possible. The value of $\texttt{root.g}_{right}$, therefore, must change. That change, in turn, may cause $\texttt{root.h}_{left}$ and other variables to change. The next step is to determine the set of variables which may have to change as a result of the constraint insertion.

In part 1 of the algorithm, the value of q_{count} is determined for each $q \in V$. The variables added to $Queue$, in order, are $\texttt{root.g}_{right}$, $\texttt{root.h}_{left}$, $\texttt{root}_{right}$, $\texttt{root.h}_{right}$, $\texttt{root.e}_{left}$, and $\texttt{root.e}_{right}$. These are the variables which may change as a re-

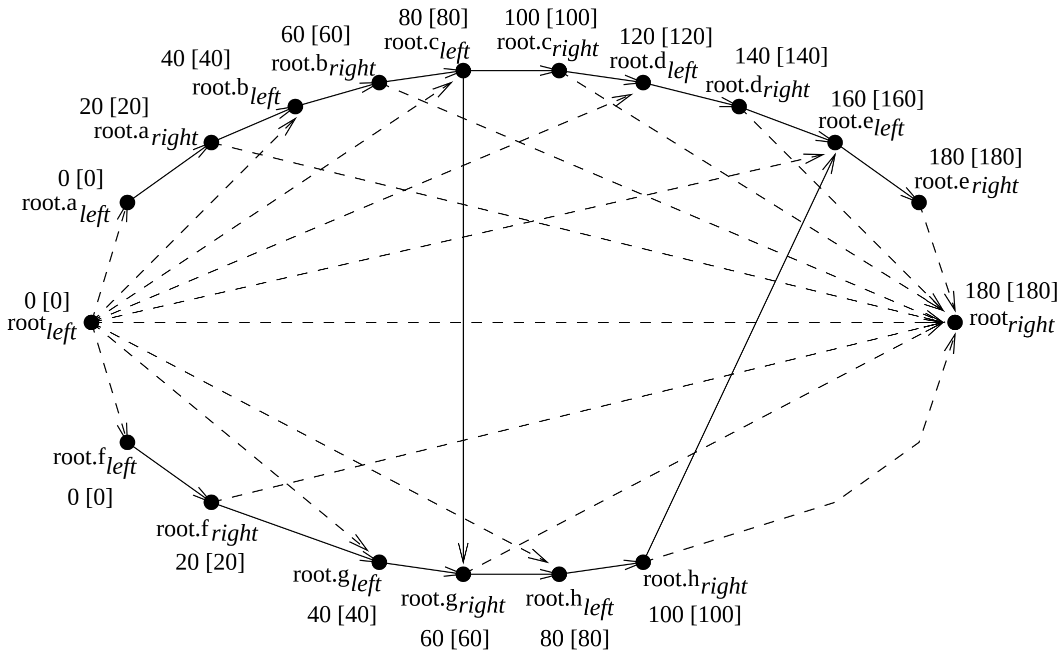


Figure 4.32: The graph representation after the addition of the new constraint.

sult of the constraint insertion. (Note that $\texttt{root}_{right}$ could have been added before $\texttt{root.h}_{left}$.) Figure 4.33 shows all of the non-zero *count* values after part 1 is performed.

$$\texttt{root}_{right_{count}} = 3, \quad \texttt{root.h}_{left_{count}} = 1, \quad \texttt{root.h}_{right_{count}} = 1,$$
$$\texttt{root.e}_{left_{count}} = 1, \quad \texttt{root.e}_{right_{count}} = 1$$

Figure 4.33: All the non-zero *count* values.

Once the variables which may have to change are identified, the next step is to update their values, according to the rules of the first heuristic. In part 2 of the incremental insertion algorithm, variables $\texttt{root.g}_{right}$, $\texttt{root.h}_{left}$, and $\texttt{root.h}_{right}$ are assigned new values and placed in queue *Changed* in that order. Figure 4.34 shows the graph representation of the solution, including the new values for the variables placed in *Changed*, after parts 1 and 2 are complete. Note that only variables added to *Queue* in part 1 are considered in part 2. Note also that the values of the variables $\texttt{root.e}_{left}$, $\texttt{root.e}_{right}$, and $\texttt{root}_{right}$ remain unchanged, even though they were in *Queue* in part 1. They remain unchanged because the value assigned to $\texttt{root.h}_{left}$

was not sufficient to require a change $\texttt{root.e}_{left}$. Since $\texttt{root.e}_{left}$ did not change, there is no reason to change $\texttt{root.e}_{right}$. As well, the value of $\texttt{root}_{right}$ was already sufficiently large enough to satisfy the constraints. Figure 4.34 shows that once part 2 is complete, the first heuristic is satisfied. It is important to observe that only the variables $\texttt{root.c}_{left}$, $\texttt{root.g}_{right}$, $\texttt{root.h}_{left}$, $\texttt{root.h}_{right}$, $\texttt{root.e}_{left}$, $\texttt{root.e}_{right}$, and $\texttt{root}_{right}$ are considered in parts 1 and 2 of the incremental insertion algorithm. None of the other 12 variables are considered. It is also important to realize that if object `root.d` was a complex object with n other objects strictly contained within it (i.e. the contained objects are not involved in any constraint with any object outside of `root.d`), that none of those n objects would be considered nor updated in this example.

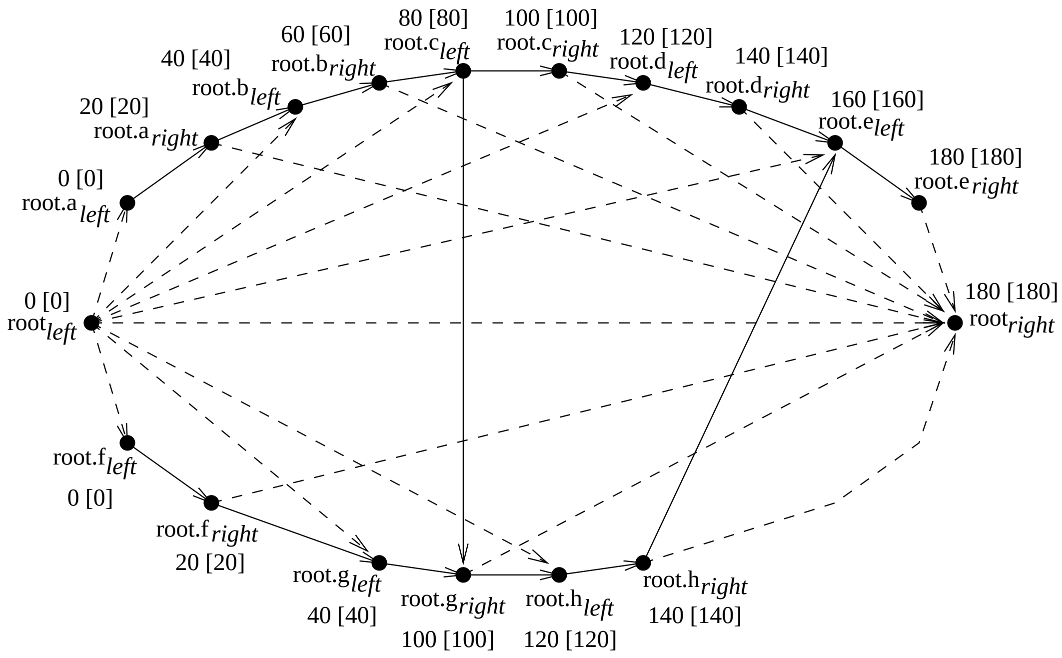


Figure 4.34: The graph representation of the solution after parts 1 and 2.

The next step is to determine which variables may have to change to satisfy the second heuristic. All these variables represent left edges of objects. The first variables considered are those changed in part 2. The only variable in Figure 4.34 which fits this criteria is $\texttt{root.h}_{left}$, and in part 3 of the algorithm the queue $Update_1$ is determined to contain only $\texttt{root.h}_{left}$. The next variables considered are those which may be directly affected by the changes in part 2. In Figure 4.34, these variables are $\texttt{root.c}_{left}$, $\texttt{root.g}_{left}$, and $\texttt{root}_{left}$. Part 3 adds these variables, in that order, to the queue $Update_2$. Recall that variables in $Update_1$ may need to be changed to satisfy the second heuristic, and that $Update_2$ is used to temporarily store variables during the construction of $Update_1$. Part 4 of the algorithm completes the construction of $Update_1$. $Update_1$ then contains, in order, $\texttt{root.h}_{left}$, $\texttt{root.c}_{left}$, $\texttt{root.g}_{left}$, and $\texttt{root}_{left}$. The contents of queue $Queue$ are then determined to be, in order, $\texttt{root.h}_{left}$, $\texttt{root.c}_{left}$ and $\texttt{root.g}_{left}$. The values of some or all the variables in $Queue$ may have to change to satisfy the second heuristic. As is seen in Figure 4.34, only the value of $\texttt{root.g}_{left}$ requires changing, and that updating will be done in part 5.

Part 5 updates the values of variables in $Update_1$ with respect to the second heuristic. This updating begins with the variables in $Queue$. Both $\texttt{root.h}_{left}$ and $\texttt{root.c}_{left}$ remain at their old values, but the value of $\texttt{root.g}_{left}$ is changed from 40 to 80. Figure 4.35 shows the final solution using the graph representation. The new values for the variables are shown in Figure 4.36. Any variable not shown in Figure 4.36 did not have its value changed. The image corresponding to the solution is found in Figure 4.37.

As can be seen in this example, a great portion of the variables and constraints need not be considered by the incremental insertion algorithm. The actual number of variables and constraints considered for any given picture description varies with the picture description. The relationship between the properties of a picture description and performance of the algorithm is discussed in Section 5.4.

4.5.3 Incremental Constraint Deletion

The removal of a constraint never introduces any inconsistencies and always results in a less (or equally) specific constraint system. After a constraint deletion, therefore, the new solution could simply be the previous solution. Such a solution, however, would not necessarily satisfy the two heuristics of ICOLA-4. Figure 4.39 shows

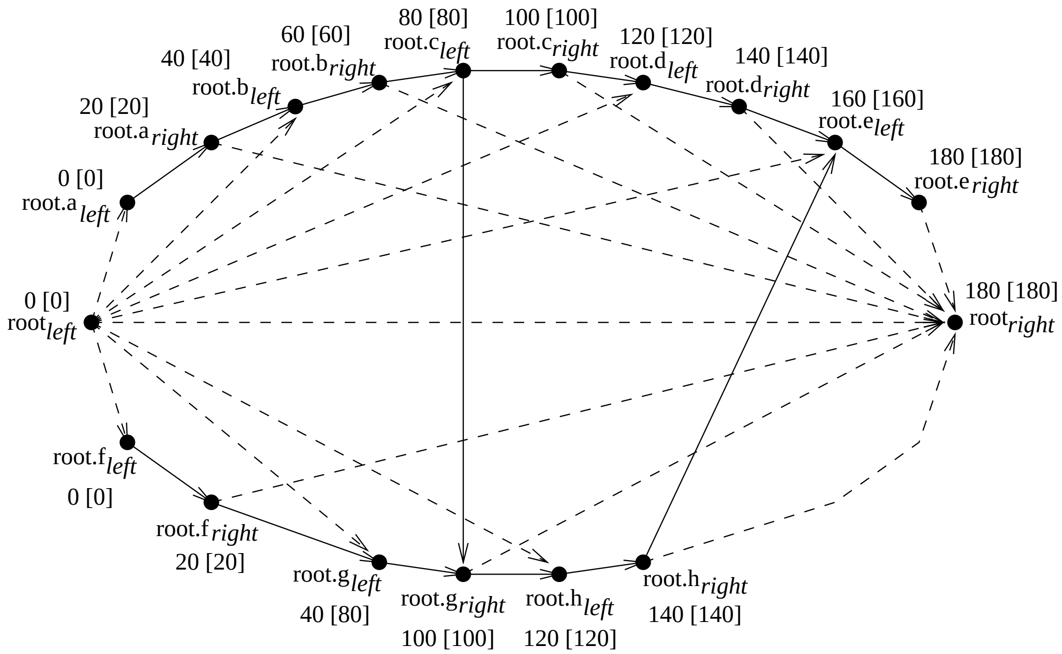


Figure 4.35: The graph representation of the final (left-to-right) solution.

$$\texttt{root.g}_{left_{low}} = 40, \quad \texttt{root.g}_{left} = 80, \quad \texttt{root.g}_{right_{low}} = 100, \quad \texttt{root.g}_{right} = 100,$$
$$\texttt{root.h}_{left_{low}} = 120, \quad \texttt{root.h}_{left} = 120, \quad \texttt{root.h}_{right_{low}} = 140, \quad \texttt{root.h}_{right} = 140$$

Figure 4.36: The solution after part 5 ($\delta = 20$).

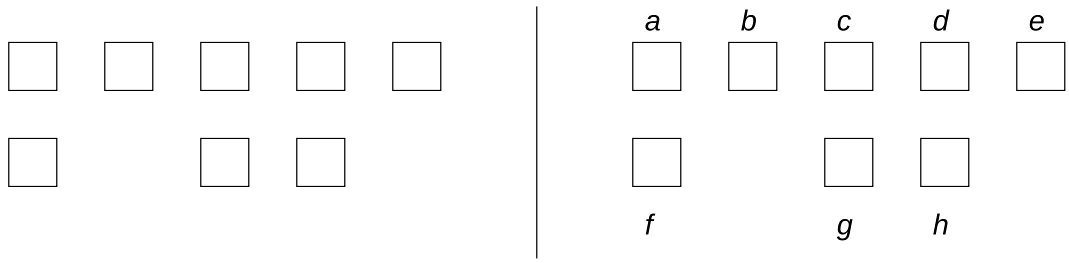


Figure 4.37: The new image.

the image produced by ICOLA-4 for the picture description in Figure 4.38. The increment

```
delete constraint( left_of, [ left( c ), right( g ) ])
```

is also satisfied by the image in Figure 4.39, but the heuristics are not (objects `g` and `h` should be further to the left). The image which satisfies the heuristics after the deletion is shown in Figure 4.40. This example is used throughout the following discussion. More details are given in Section 4.5.4.

```
object( a, box ).
object( b, box ).
constraint( left_of, [ a, b ]).
object( c, box ).
constraint( left_of, [ b, c ]).
object( d, box ).
constraint( left_of, [ c, d ]).
object( e, box ).
constraint( left_of, [ d, e ]).
object( f, box ).
object( g, box ).
constraint( left_of, [ f, g ]).
object( h, box ).
constraint( left_of, [ g, h ]).
constraint( left_of, [ h, e ]).
constraint( left_of, [ left(c), right(g) ]).
constraint( above, [ a, f ]).
constraint( above, [ a, g ]).
constraint( above, [ a, h ]).
```

Figure 4.38: Picture description for the incremental deletion example.

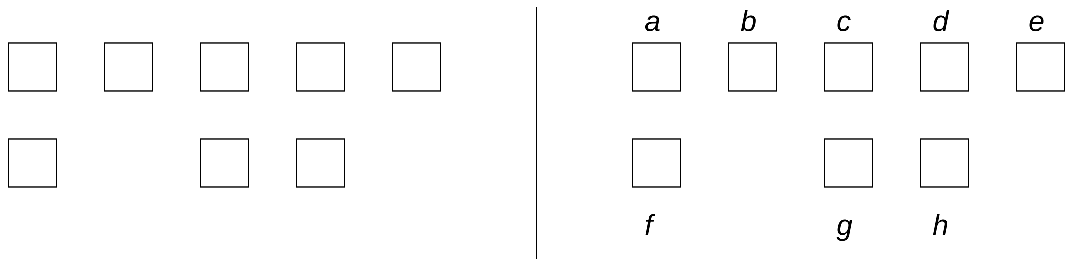


Figure 4.39: The image produced from the picture description in Figure 4.38.

The purpose of the incremental deletion algorithm is to determine which variables of the constraint system must change, and then update those variables to the correct values. Given $S' = incremental\_solve\_delete(C,\ (x + \alpha \leq y),\ S)$, the incremental constraint deletion algorithm must construct a new solution S' based on the new set of constraints $C - \{x + \alpha \leq y\}$ and the previous solution S. An overview of the

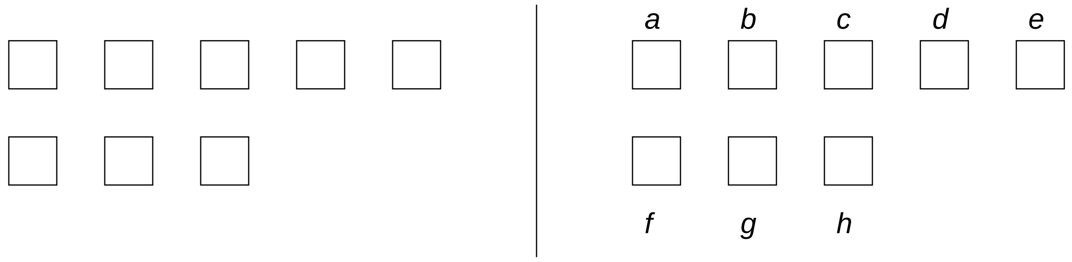


Figure 4.40: The image after the removal of the constraint.

incremental deletion algorithm is given in Figure 4.41. The high-level description of each of the parts is exactly the same as that for the incremental insertion algorithm (See Section 4.5.1). Parts 1 and 2 are concerned with satisfying the first heuristic. Parts 3, 4, and 5 are concerned with satisfying the second heuristic. Parts 1, 3, and 5 of the incremental deletion algorithm are exactly the same as the corresponding parts of the incremental insertion algorithm. Part 2 of the deletion algorithm is significantly different from part 2 of the insertion algorithm as it is concerned with decreasing the values of variables instead of increasing the values. Part 4 of the incremental deletion algorithm is very similar to part 4 of the incremental insertion algorithm. Because the removal of the constraint $x + \alpha \leq y$ may isolate x from y, the incremental deletion algorithm needs to ensure that x is updated properly. This is accomplished in the first two lines of part 4, and is the only difference between part 4 of the incremental deletion algorithm and part 4 of the incremental insertion algorithm.

The first step is to determine if removing the constraint $x + \alpha \leq y$ from C will affect the values of any variables in the current solution. If $C - \{x + \alpha \leq y\}$ still implies $x + \alpha < y$ and $x_{low} + \alpha < y_{low}$, then removing the constraint will not change the current solution, and the new solution S' will simply be the current solution S. The case $x + \alpha > y$ or $x_{low} + \alpha > y_{low}$ cannot occur, as that would mean the constraints $x + \alpha \leq y$ and $x_{low} + \alpha \leq y_{low}$ were not satisfied in S. Hence the only situations of interest are if $x + \alpha = y$ or $x_{low} + \alpha = y_{low}$, in which cases the removal of the constraint may require an update to the solution. Note that the computation of the new solution is only to satisfy the heuristics – after the deletion the previous

```
incremental_solve_delete(C, (x + α ≤ y), S) begin
    S' = S
    C' = C − {x + α ≤ y}
    if ((x + α = y) ∨ (x_low + α = y_low)) then begin
        Part 1
        Part 2
        Part 3
        Part 4
        Part 5
    end
    return S'
end
```

Figure 4.41: Overview of the incremental deletion algorithm.

solution will still satisfy all the constraints.

As in the constraint insertion algorithm, the first step in the constraint deletion algorithm is to ensure that the first heuristic is satisfied. Figure 4.17 in Section 4.5.1 shows part 1 of the incremental deletion algorithm and Figure 4.42 shows part 2.

The algorithm begins as the incremental insertion algorithm does – by determining which variables may have to change if the value of y changes a result of the removal of the constraint $x + \alpha \leq y$. In Figure 4.39, the constraint removal means that the position of the right edge of object `g` may change. Hence the variables of concern in Part 1 are those which represent the left and right edges of objects `h` and `e`. Part 2 of the algorithm ensures that the value of each variable satisfies the first heuristic.

In part 2, the *propagate* flag is cleared as each variable z is removed from *Queue*. The *propagate* flag indicates whether the value of z changes, and if the change should be propagated to variables q, $(z + \omega \leq q) \in C'$, and onward. If z must be updated then lines (8-19) are performed. In lines (8-11) *new_value* is assigned the correct value for z. Since *new_value* is initially as low as possible, and since its value is only changed if $p_{low} + \gamma > new\_value$ for some constraint $(p + \gamma \leq z) \in C'$, it must be the case that *new_value* in line (12) is the lowest possible value that z can obtain. If

```
( 1)  add(Queue, y)
( 2)  y_update = true
( 3)  while (¬empty(Queue)) do begin
( 4)          delete(Queue, z)
( 5)          z_marked = false
( 6)          propagate = false
( 7)          if (z_update) then begin
( 8)                  new_value = 0
( 9)                  foreach (p + γ ≤ z) ∈ C' do
(10)                          if (p_low + γ > new_value) then
(11)                                  new_value = p_low + γ
(12)                  if (z_low > new_value) then begin
(13)                          z_low = new_value
(14)                          z = z_low
(15)                          propagate = true
(16)                          add(Changed, z)
(17)                  end else
(18)                          z_update = false
(19)          end
(20)          foreach (z + ω ≤ q) ∈ C' do begin
(21)                  q_count = q_count − 1
(22)                  if (q_count == 0) then
(23)                          add(Queue, q)
(24)                  if (propagate) then
(25)                          q_update = true
(26)          end
(27)  end
```

Figure 4.42: The Incremental Deletion Algorithm - Part 2.

$z_{low} > new\_value$, then the current value of z_{low} is too high, and both z and z_{low} must be reduced to new_value. Since z is now changed, the change should be propagated to variables q, $(z + \omega \leq q) \in C'$. Hence $propagate$ is set to $true$ and z is added to the $Changed$ queue. Recall that $Changed$ contains those variables whose values are updated. If $z_{low} \leq new\_value$, then no changes are necessary to z or z_{low} to satisfy the first heuristic since it must be true that $z_{low} = new\_value$ ($z_{low} < new\_value$ would mean the solution S contained a violation of the first heuristic). The fact that z is not updated is noted by setting z_{update} to $false$ in line (18). Lines (20-26) are responsible for ensuring that each variable q, $(z+\omega \leq q) \in C'$, is added to queue $Queue$ if all p, $(p+\phi \leq q) \in C'$, are known to be correct. As well, if z has changed (i.e. $propagate$ is $true$) then the value of q should also be updated, and hence q_{update} is set to $true$.

Each variable q is added to $Queue$ at most once, since q_{count} must be 0 before q is added, and 1 is added to q_{count} in part 1 only once for each constraint. Each constraint is potentially considered once in line (9), and once in line (20), but a maximum twice for each variable. Hence the time complexity of part 2 of the incremental deletion algorithm is also $O(v+c)$, where v is the number of variables, and c is the number of constraints.

Theorem 5 *The first heuristic is satisfied after parts 1 and 2 of the incremental deletion algorithm are performed.*

Proof: In part 1, q_{count} is increased by 1 for each constraint $(p+\gamma \leq q) \in C'$ where the value of p may have to change. If q_{count} is 0, then q_{low} does not have to change as a result of the deletion. In part 2, a variable q is changed if it was "marked" in part 1, and even then only once q_{count} is 0. Beginning at 0, the new_value for q_{low} is only increased for each $(p+\gamma \leq q) \in C'$ if $p_{low}+\gamma > new\_value$, and in which case new_value is set to $p_{low}+\gamma$. Hence the value of q_{low} is the lowest possible value, which satisfies the first heuristic. □

Part 3 of the incremental deletion algorithm is seen in Figure 4.21 in Section 4.5.1. Part 4 of the algorithm is shown in Figure 4.43. Recall that the constraint removed from C is $x+\alpha \leq y$. The first two lines of part 4 ensure that if x is not marked as changed, then x is added to $Update_2$ if $is\_left(x)$ or $is\_top(x)$ is true. Recall as well that $Update_2$ contains the variables p where $is\_left(p)$ or $is\_top(p)$ is true, and the value of p may have to change to satisfy the second heuristic. The remainder of part 4 is identical to part 4 of the incremental insertion algorithm shown in Figure 4.23 in Section 4.5.1. As in the incremental insertion algorithm, parts 3 and 4 determine which variables may need to be changed to satisfy the second heuristic. For the example in Figure 4.39, parts 3 and 4 will determine that the left edge of object g, among other objects, may need to be updated. Part 5 performs the update, and the final result is the image in Figure 4.44. Part 5 of the incremental deletion algorithm is exactly the same as part 5 of the incremental insertion algorithm. Part 5 is shown in Figure 4.25 in Section 4.5.1. Note that $Queue$ contains at least one variable in part 5.

```
if (¬x_marked ∧ (is_left(x) ∨ is_top(x))) then
        add(Update_2, x)
while (¬empty(Update_2)) do begin
        delete(Update_2, z)
        add(Update_1, z)
        foreach (p + γ ≤ z) ∈ C' do
                if (is_left(p) ∨ is_top(p)) do begin
                        p_count = p_count + 1
                        if (¬p_changed ∧ ¬p_marked) then begin
                                add(Update_2, p)
                                p_marked = true
                        end
                end
end
while (¬empty(Update_1)) do begin
        delete(Update_1, z)
        if (z_count == 0) then
                add(Queue, z)
end
```

Figure 4.43: The Incremental Deletion Algorithm - Part 4.

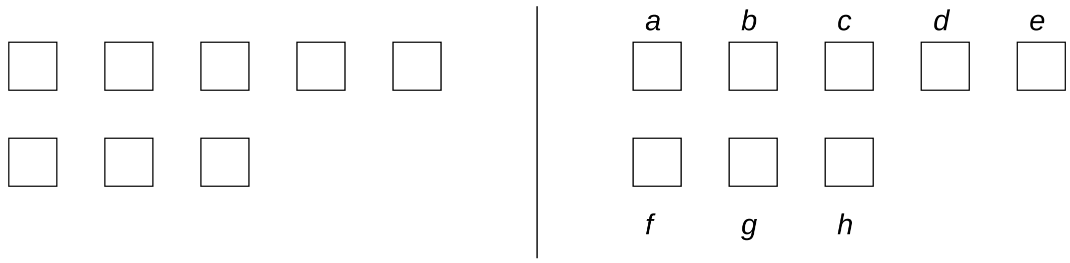


Figure 4.44: The final image of the incremental deletion example.

Theorem 6 *After parts 3, 4, and 5 of the incremental deletion algorithm are performed, the second heuristic is satisfied, and the value for each $p \in V$ is correct in S'.*

Proof: The proof is exactly the same as the proof for Theorem 4. (The additional two lines in part 4 of the incremental deletion algorithm do not change the proof.) $\square$

Since the time complexities of parts 1, 3, 4, and 5 from the incremental insertion algorithm is $O(v+c)$, the total time complexity of the incremental deletion algorithm is $O(v+c)$, where v is the number of variables and c is the number of constraints.

4.5.4 The Incremental Deletion Example

The example shown in this section begins with the set of constraints and the solution at the end of Section 4.5.2. The current solution is shown in Figure 4.45 and its graph representation in Figure 4.46. In all graph representations in this section the dashed arcs have a weight of 0, and the solid arcs have a weight of 20 (the default distance, δ, is 20). For each node z in the graph representation, the value of the variable is indicated in brackets ("[]"), and the low value of the variable (z_{low}) is shown outside of the brackets.

$\texttt{root}_{left_{low}} = 0$,	$\texttt{root}_{left} = 0$,	$\texttt{root}_{right_{low}} = 180$,	$\texttt{root}_{right} = 180$,
$\texttt{root.a}_{left_{low}} = 0$,	$\texttt{root.a}_{left} = 0$,	$\texttt{root.a}_{right_{low}} = 20$,	$\texttt{root.a}_{right} = 20$,
$\texttt{root.b}_{left_{low}} = 40$,	$\texttt{root.b}_{left} = 40$,	$\texttt{root.b}_{right_{low}} = 60$,	$\texttt{root.b}_{right} = 60$,
$\texttt{root.c}_{left_{low}} = 80$,	$\texttt{root.c}_{left} = 80$,	$\texttt{root.c}_{right_{low}} = 100$,	$\texttt{root.c}_{right} = 100$,
$\texttt{root.d}_{left_{low}} = 120$,	$\texttt{root.d}_{left} = 120$,	$\texttt{root.d}_{right_{low}} = 140$,	$\texttt{root.e}_{right} = 140$,
$\texttt{root.e}_{left_{low}} = 160$,	$\texttt{root.e}_{left} = 160$,	$\texttt{root.e}_{right_{low}} = 180$,	$\texttt{root.e}_{right} = 180$,
$\texttt{root.f}_{left_{low}} = 0$,	$\texttt{root.f}_{left} = 0$,	$\texttt{root.f}_{right_{low}} = 20$,	$\texttt{root.f}_{right} = 20$,
$\texttt{root.g}_{left_{low}} = 40$,	$\texttt{root.g}_{left} = 80$,	$\texttt{root.g}_{right_{low}} = 100$,	$\texttt{root.g}_{right} = 100$,
$\texttt{root.h}_{left_{low}} = 120$,	$\texttt{root.h}_{left} = 120$,	$\texttt{root.h}_{right_{low}} = 140$,	$\texttt{root.h}_{right} = 140$

Figure 4.45: The current solution for the incremental deletion example ($\delta = 20$).

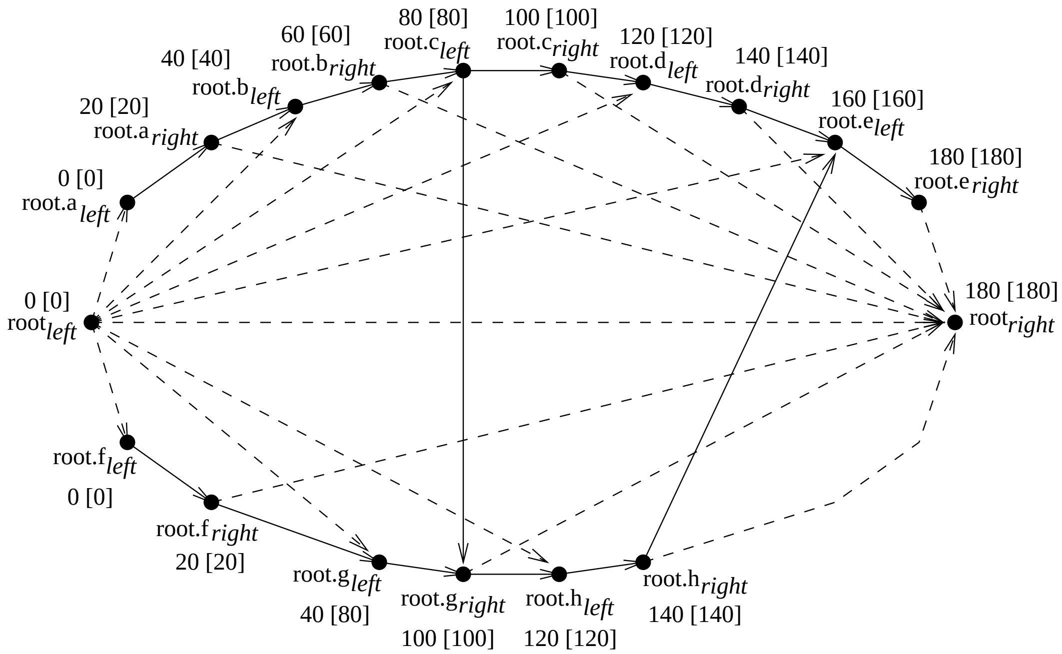


Figure 4.46: The graph representation of the current solution.

In this example, the increment of interest is

```
delete constraint( left_of, [ left( c ), right( g ) ])
```

which is equivalent to removing the formula $\texttt{root.c}_{left} + \delta \leq_C \texttt{root.g}_{right}$ from the current set of internal constraints. The removal of this constraint must leave the solution and set of constraints exactly the same as the solution and set of constraints at the beginning of the incremental insertion example in Section 4.5.2.

The first step performed by the incremental deletion algorithm is to remove this constraint from the current set of constraints, and then determine if the removal of this constraint may cause a change to the current solution. Figure 4.47 shows the graph representation after the constraint is removed.

Since $\delta = 20$, $\texttt{root.c}_{left} = 80$, $\texttt{root.g}_{right} = 100$ and $\texttt{root.c}_{left} + \delta = \texttt{root.g}_{right}$, the removal of the constraint may cause a change to the solution. As is seen in Figure 4.47, the positions of objects `root.g` and `root.h` must be changed.

Since the solution may have to change, the next step is for part 1 to determine which variables may change. Part 1 of the incremental deletion algorithm behaves exactly the same as part 1 of the incremental insertion algorithm. In part 1, the

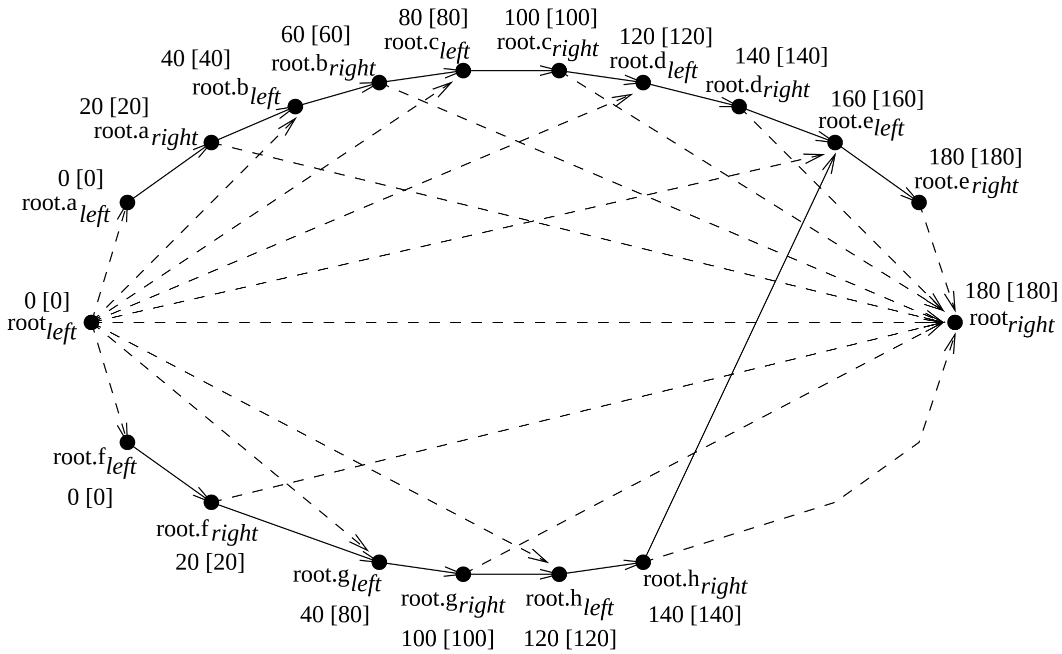


Figure 4.47: The graph representation after the deletion of the constraint.

value of q_{count} is determined for each $q \in V$. The variables added to $Queue$, in order, are $\texttt{root.g}_{right}$, $\texttt{root.h}_{left}$, $\texttt{root}_{right}$, $\texttt{root.h}_{right}$, $\texttt{root.e}_{left}$, and $\texttt{root.e}_{right}$. Figure 4.48 shows the non-zero *count* values after part 1 is performed.

$$\begin{array}{lll} \texttt{root}_{right_{count}} = 3, & \texttt{root.h}_{left_{count}} = 1, & \texttt{root.h}_{right_{count}} = 1, \\ \texttt{root.e}_{left_{count}} = 1, & \texttt{root.e}_{right_{count}} = 1 & \end{array}$$

Figure 4.48: The non-zero *count* values after part 1 of the incremental deletion algorithm.

Once the variables which may have to change are identified in part 1, the next step is for part 2 to update those variables according to the second heuristic. In part 2 of the incremental deletion algorithm, variables $\texttt{root.g}_{right}$, $\texttt{root.h}_{left}$, and $\texttt{root.h}_{right}$ are assigned new values and placed, in that order, in queue $Changed$. The new values are shown in Figure 4.49. As this figure indicates, the value of $\texttt{root.g}_{left}$ is incorrect. This value will be corrected by parts 3, 4, and 5 of the incremental deletion algorithm. Observe that, as in the incremental insertion example, very few of the variables are considered or changed in parts 1 and 2. As well, if `root.d` was a complex object strictly containing a number of other objects, then none of those objects would be considered nor updated in this example.

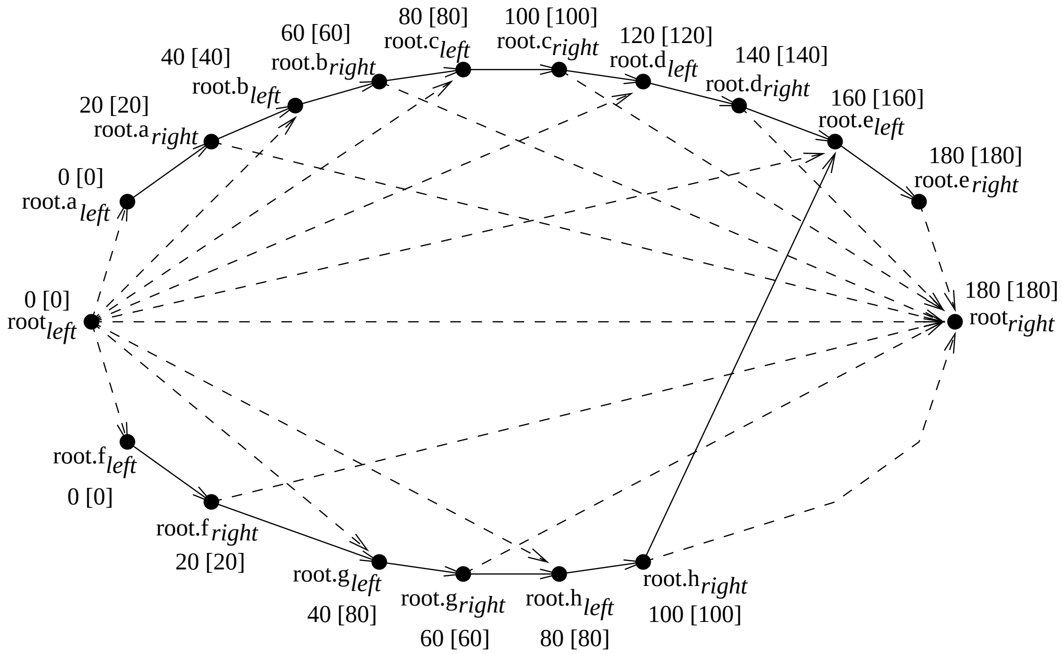


Figure 4.49: The graph representation after parts 1 and 2 of the incremental deletion example.

The next step is to determine which variables may have to change to satisfy the second heuristic. Part 3 first considers only the variables changed in part 2. Of these variables, only root.h$_{left}$ may need to be changed, and hence only root.h$_{left}$ is added to $Update_1$. Next, root.c$_{left}$, root.g$_{left}$, and root.r$_{left}$ are added, in order, to $Update_2$ since these variables may be affected by the changes to the variables made in part 2. Recall that $Update_2$ is used to temporarily store variables during the construction of $Update_1$. To ensure that root.c$_{left}$ is updated properly, that variable is added to $Update_2$ in part 4. Part 4 then completes the construction of $Update_1$. $Update_1$ contains, in order, root.h$_{left}$, root.c$_{left}$, root.g$_{left}$, and root$_{left}$. The latter part of part 4 then adds the variables root.h$_{left}$, root.c$_{left}$ and root.g$_{left}$, in that order, to $Queue$. The variables in $Queue$ form the starting point for updating all the variables in $Update_1$. These variables are updated in part 5.

Part 5 determines the correct values for all the variables in $Update_1$, starting with the values in $Queue$. As shown in Figure 4.32, the only variable which needs to be changed is $\texttt{root.g}_{left}$. The value of that variable is changed from 80 to 40. After part 5 is performed, the solution is as shown in Figure 4.50, and the image produced is as shown in Figure 4.27. The graph representation of the solution is given in Figure 4.51.

$$\begin{array}{llll} \texttt{root.g}_{left_{low}} = 40, & \texttt{root.g}_{left} = 40, & \texttt{root.g}_{right_{low}} = 60, & \texttt{root.g}_{right} = 60, \\ \texttt{root.h}_{left_{low}} = 80, & \texttt{root.h}_{left} = 80, & \texttt{root.h}_{right_{low}} = 100, & \texttt{root.h}_{right} = 100 \end{array}$$

Figure 4.50: The solution after part 5 of the deletion algorithm.

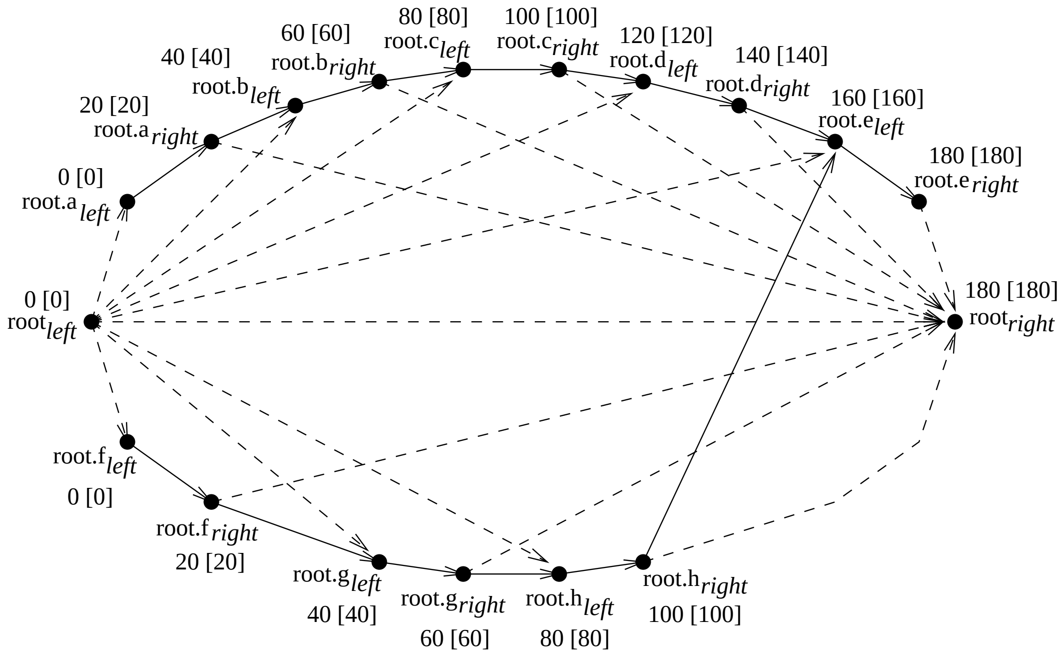


Figure 4.51: The graph representation of the final solution.

As shown in this example, the incremental deletion algorithm is able to avoid re-computing the values of a large portion of the variables. This behavior is typical of most picture descriptions. While the actual number of variables and constraints considered will vary according to the properties of the picture description, the results shown in Section 5.4 indicate that this avoidance of computation can significantly enhance the performance of the incremental algorithms.

4.6 Implementation

In ICOLA-4 the variables and constraints are encoded internally as graphs. One graph represents the layout of objects along the horizontal axis and the other graph, the layout of objects along the vertical axis. The variables correspond to nodes in the graph, and the constraints correspond to arcs in the graph. As each new object or constraint is added, nodes and/or arcs are added to the graphs. The absolute positions of the objects are computed by the algorithms discussed in the previous sections.

While the maximum x and y coordinate values for each object are theoretically infinite, the current implementation sets an upper bound of $2^{32} - 1$. For a very large picture (e.g. one involving coordinates of this magnitude) the physical memory required by the implementation would typically be the limiting factor long before such an actual coordinate was reached.

ICOLA-4 allows a picture description to be solved in non-incremental ("batch") mode, incremental mode, or an arbitrary combination of the two. In order to facilitate this functionality, the implementation must ensure that the current solution is correct when switching from batch mode to incremental mode. This is easily accomplished by calling the batch solver when the `mode_increment` (See Section 3.7) directive is encountered in a picture description.

ICOLA can currently handle 10,000 new objects[2], and a virtually unlimited number of constraints (limited by the memory capacity of the computer). The limit of 10,000 objects is arbitrarily set, and easily extended.

4.6.1 Implementation Data Structures

The variables and constraints in the constraint system are represented internally as a weighted, directed graph G where nodes in G correspond to variables in V,

[2] In the current implementation, internal object identifiers from deleted objects are not re-used, as the 10,000 object limit is rarely reached. Internal object identifiers could easily be re-used in a future implementation.

and if $(p + \gamma \leq q) \in C$, then there is an arc of weight γ from node p to node q in G. Although objects of class LINE are represented by nodes in the graph, their positions are calculated differently than objects of class BB. Section 4.6.2 discusses how objects of class LINE are placed.

In the pre- and post-processors, dictionaries (in Prolog) maintain the current set of symbolic object labels or identifiers, and a unique numeric identifier for each object. The numeric identifiers are assigned by the pre-processor and used internally by the constraint system. These identifiers are not available for use in the picture descriptions. Internal symbolic names for each object are comprised of the object's symbolic label and a symbolic representation of that object's scope. The dictionaries also contain all the attributes for each object.

It should be noted that ICOLA-3 used similar data structures. The incremental algorithms used on those data structures, however, are substantially different in ICOLA-4.

4.6.2 Positioning Lines

After the objects of class BB are positioned via the incremental algorithms, the final step is to position objects of class LINE. Since objects of class LINE are added after all other objects are place, the positions of the bounding boxes are already known and only a few simple calculations are required to determine the exact location of the `line` or `arc` object. Given the constraint

```
constraint( connects, [ A, B, C ]),
```

the position of A, an object of class LINE, is between the closest edges of the bounding boxes of objects B and C. If one of B or C is completely above the other, then the nearest top and bottom edges (one from each object) bound A. If one of B or C is completely to the left of the other, then the nearest left and right edges (one from each object) bound A. Otherwise A is drawn between the midpoints of the nearest edges of the bounding boxes of B and C. Figure 4.52 shows the possible line connections between two objects in a variety of positions and Figure 4.53 gives the algorithm to position an object of class LINE A from objects B to C. Since objects of class LINE are positioned after all other objects, other line positioning algorithms could easily replace the current algorithm.

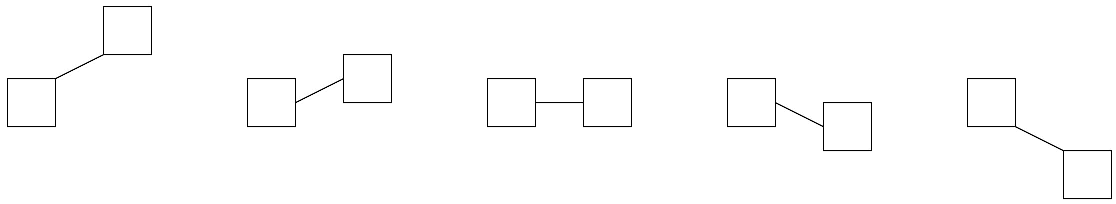


Figure 4.52: Possible object orientations and `line` positions.

if $\mathsf{B}_{bottom} \leq \mathsf{C}_{top}$ then begin
  $\mathsf{A}_{top} = \mathsf{B}_{bottom}$
  $\mathsf{A}_{bottom} = \mathsf{C}_{top}$
end else if $\mathsf{B}_{top} \geq \mathsf{C}_{bottom}$ then begin
  $\mathsf{A}_{top} = \mathsf{B}_{top}$
  $\mathsf{A}_{bottom} = \mathsf{C}_{bottom}$
end else begin
  $\mathsf{A}_{top} = (\mathsf{B}_{top} + \mathsf{B}_{bottom})/2$
  $\mathsf{A}_{bottom} = (\mathsf{C}_{top} + \mathsf{C}_{bottom})/2$
end

if $\mathsf{B}_{right} \leq \mathsf{C}_{left}$ then begin
  $\mathsf{A}_{left} = \mathsf{B}_{right}$
  $\mathsf{A}_{right} = \mathsf{C}_{left}$
end else if $\mathsf{B}_{left} \geq \mathsf{C}_{right}$ then begin
  $\mathsf{A}_{left} = \mathsf{B}_{left}$
  $\mathsf{A}_{right} = \mathsf{C}_{right}$
end else begin
  $\mathsf{A}_{left} = (\mathsf{B}_{left} + \mathsf{B}_{right})/2$
  $\mathsf{A}_{right} = (\mathsf{C}_{left} + \mathsf{C}_{right})/2$
end

Figure 4.53: The algorithm for positioning an object A of class LINE.

The time complexity of the algorithm for positioning n objects of class LINE is $O(n)$.

In the current implementation, objects of type `arc` are approximated using a series of line segments. This is done since the graphics package does not directly support drawing arcs. A more exact implementation of arcs would be both computationally expensive, time consuming, and a superfluous effort since many other graphics packages have arc drawing capabilities. Better arcs can be obtained by using a graphics package with support for them.

4.7 Other Implementation Details

While the constraint engine, written in C, is the heart of the system, both the pre- and post-processors perform important roles in ICOLA (See Figure 3.14). The preprocessor, written in Prolog, parses the input picture description statements, performs name lookup and dictionary operations, applies defaults, handles attribute changes, and translates the picture description into the internal form used by the constraint solver. The post-processor, also written in Prolog, converts the resulting set of object coordinates into calls to the display predicates of the graphics engine (GM), taking into account the attributes of the objects. The pre- and post-processors of the implementation correspond directly to the pre- and post-processors described in Section 3.9.

Figure C.1 in Appendix C shows the user-interface of ICOLA-4. The user-interface has such features as buttons, scrollable images, and pop-up windows (for error messages and user-input). While the scrollable main viewing area may only show a portion of the entire image at any time, an "aerial view" always contains the entire image. Also provided is a scrollable text window which contains the constraints defining the current image. Picture descriptions can be either entered interactively by the user, or read from an input file.

4.8 Chapter Summary

In this chapter the details of the incremental and non-incremental algorithms in ICOLA-4 are presented along with examples of their behavior. The algorithms are found to have a time complexity of $O(v + c)$, where v is the number of variables and c is the number of constraints being considered. In practice, however, only a small number of variables and constraints are examined for a typical example. The quantitative performance results for the algorithms are given in Section 5.4.

CHAPTER 5

ICOLA – DISCUSSION AND RESULTS

This chapter briefly compares PDI-4 and the constraint engine in ICOLA-4 to other constraint-based graphical layout languages and engines. Quantitative performance results are presented and limitations of the implementation are discussed.

5.1 ICOLA's Picture Specification Language

Using Chang's classifications (See Section 2.3), PDI is a visual information processing language with linear constructs (i.e. constructing picture descriptions in PDI does not require a special graphical editor). As noted in Section 2.3, Helm and Marriott [HeMa90] have four requirements they believe are necessary for any picture specification language. While PDI does not directly conform to all of their requirements, it must be remembered that PDI is low-level, and that computer programs, not humans, are expected to generate the picture descriptions.

1. *It must be expressive enough to encode the relationships between the pictures and their meaning.* The PDI language is both very simple and quite capable of describing complex pictures. The "meaning" of a picture is captured in the relational constraints between the objects. As Helm et al. [Hel+90] also note, it is these relative relationships that are important, not the absolute position, size, nor orientation of the objects.

2. *It must allow high-level abstractions similar to those people use for describing pictures.* While PDI does not support constructs like "next_to", it does support a simple set of constraints which are sufficient for describing pictures. The level of abstraction is high enough that users need not be concerned about actual coordinate values – only relative relationships need to be known. The `complex` object also provides an aggregation abstraction. There is no ability, however, to define new object types.

3. *It should have a formal, declarative semantics.* The operational semantics are presented in Section 3.10. A declarative semantics for PDI, however, is beyond the scope of this thesis. Fortunately, the PDI language is declarative, which should make the development of a declarative semantics easier.

4. *The generation and recognition algorithms should be efficiently implementable.* As discussed in Chapter 4 the generation of pictures can be efficiently implemented. There are no provisions in ICOLA for recognizing pictures as picture recognition capabilities are not required by the Wand framework.

It should be noted that PDI is theoretically capable (implementation permitting) of producing any arbitrary bitmap image. This is possible using the `point` object, the `horizontal_distance` and `vertical_distance` constraints, and the `colour` attribute. These four language constructs are sufficient to describe any picture, so there is no theoretical limit to the class of images that can be drawn.

5.2 ICOLA Versus Other Layout Systems

In this section ICOLA-4 is briefly compared to a number of other layout systems. In particular, emphasis is placed on the relevant similarities and differences between ICOLA-4 and the other systems.

A key difference between most graph (tree) layout algorithms and ICOLA-4 is that ICOLA-4 must be concerned with the left-to-right and top-to-bottom ordering of the objects. Graph drawing algorithms are typically free to position nodes of a graph as required to obtain a "good" layout (e.g. one which minimizes edge crossings). A graph drawing algorithm, for example, can typically invert the ordering of nodes from top to bottom, and still have a meaningful and correct representation of the graph. ICOLA-4, however, is more constrained, as it is intended for applications where relative positioning is required.

ICOLA-4 is also not concerned with such problems as wire routing in VLSI layout. Roach's system [Roac84] (See Section 2.5.4), for example, is designed for creating VLSI layouts in which routing is a very important consideration. Routing is a complex problem which ICOLA is not designed to handle. A routing algorithm has, however, been successfully added to a special version of ICOLA-4 to allow it to be used as the graphics engine for a software engineering diagram layout tool called ADL (Automatic Diagram Layout) [Mort95]. The ADL tool generates picture descriptions which adhere to a specific data-flow diagram layout methodology. Automatic software diagram layout [Mort95, Pro+91] presents special problems, such as determining the correct length and spacing for data flow arcs. ICOLA-4 is only concerned with constructing an image which is consistent with the specified objects and constraints. To support the ADL research a new object called `polyline` was added to the special version of ICOLA-4. The `polyline` object is of class LINE and is a series of line segments specified as a series of points. A number of other features, such as special labels for the `line` and `polyline` objects, were also added. These features are not critical components of ICOLA-4 with respect to this research, and hence are not discussed elsewhere in this thesis.

Roach's system [Roac84] also requires that any two variables (variables represent component (object) positions) must have either an explicit or an implicit relationship. ICOLA-4 does not have such a restriction. Objects in ICOLA-4 must only have an explicit or implicit relationship with the group object that contains them. It is unclear whether Roach's system is incremental.

The PDI language constructs in ICOLA are similar to those used in COOL, the COnstraint-based Object Oriented Layout system of Kamada and Kawai [Kama89, KaKa91]. Both systems have `above`, `below`, `left_of`, and `right_of` constraints.

While COOL requires the reference points to lie on a geometric curve at equal intervals, ICOLA requires that the reference points of the objects are on horizontal and vertical lines.

The restrictions on ICOLA shown in this section are only restrictions if ICOLA is expected to do something other than that for which it is designed. ICOLA's primary application is in the production of pictures for logic program debugging environments. Preliminary investigations indicate that ICOLA's algorithms and PDI language will serve this purpose quite well.

5.3 ICOLA's Constraint Satisfaction Algorithm

As indicated in Section 2.4, there are many different issues involved in constraint satisfaction, and many different approaches to solving a system of constraints. The success of other special purpose solvers (e.g. DeltaBlue [Fre+90] and SkyBlue [Sann93]) showed that constraint-based graphical layouts could be quickly computed by stand-alone systems. As mentioned in Section 2.4.5, Dubé and Yap [DuYa93] claim that general-purpose logic programming constraint solvers which incorporate geometric constraints would be typically too inefficient to be practical. It is also believed that the speed lost by using a constraint-based logic programming language like CHIP [Din+88] or CLP($\Re$) [JaMi87] would far outweigh the extra abilities afforded by these languages. Preliminary results [Hoff95] indicate that ICOLA-4 is much faster than at least one general-purpose constraint satisfaction tool, the finite domain library of ECL^{i}PS^e [Bri+95], for solving the picture descriptions handled by ICOLA-4.

The linear constraint technology in the system of Helm et al. [Hel+92] is similar in a number of aspects to the internal constraints in ICOLA-4. As discussed in Section 2.5.4, their system also provides for efficient incremental addition and deletion of constraints. Both systems can determine the satisfiability of the set of constraints, but where ICOLA-4 rejects the most recently added constraint as the cause of the inconsistency, the system of Helm et al. can identify all of the constraints which contribute to the inconsistency. Their system also determines a set of canonical constraints before finding a solution.

The DeltaBlue algorithm in ThingLab II [Mal+89] is similar to the incremental algorithm of ICOLA-4 in a number of respects. Both are incremental, use graph structures to represent the constraints, and rely on local propagation to achieve good performance. DeltaBlue, however, allows the specification of more general constraints, including equalities. The solution algorithm may, however, encounter situations where multiple equally good solutions exist. In ICOLA-4 there is only a single solution defined by the heuristics (Section 4.2), and ICOLA-4 immediately determines that solution.

The constraint satisfaction technique used in ICOLA-4 is an *add-and-synthesize* approach (See Section 2.4.3). Where incremental search techniques rely on techniques like dependency-directed backtracking to avoid re-computation of failing goals, ICOLA-4's algorithm ensures the value of a variable is correct before it is used in further computations.

5.4 ICOLA's Quantitative Performance

This section discuss the performance aspects of ICOLA-4, beginning with a brief description of the methodology used, the characteristics of the test cases, and then the results of the performance analysis.

5.4.1 Methodology

From a user's perspective, the quantitative performance of ICOLA-3 or ICOLA-4 is based on the responsiveness of the tool. This responsiveness, in turn, is based on the efficiency of the user-interface, the pre- and post-processors, and the constraint engine. Of these components, the constraint engine is the most important, and hence is the primary subject of the performance analysis.

Using the profiling capabilities of the C compiler it is found that 90% of the CPU time used by ICOLA-4 is being used by the components written in Prolog. These components are the user-interface and the pre- and post-processors. The 10% used by the constraint engine is the focus of these performance results. This 10% includes the time to update the internal sets of variables and constraints, and to perform the consistency check.

To measure only the time required by the constraint solver it is necessary to avoid measuring the overhead introduced by the Prolog components. The approach used in ICOLA-3 is to measure the time spent in the constraint engine for each individual increment. The sum of these times gives the total time for the picture description. Unfortunately, the resolution of the clock on the test machine is only 16 milliseconds, which is typically greater than the time for the constraint solvers to process an individual increment. For ICOLA-3, this means that many increments report a solution time of 0 milliseconds when the actual time is between 0 and 16 milliseconds. This underestimates the actual time to find a solution and slightly overstates the performance of ICOLA-3.

For ICOLA-4, special code in the implementation allows the constraint solver to be run without the Prolog components. Picture descriptions are converted from their normal form into a special "condensed" form which can be processed by ICOLA-4 without the Prolog components. By measuring the time required to process the "condensed" form, the overhead of the Prolog interface is eliminated. As well, the poor clock resolution has a smaller impact on the measurements, as there is only a single timing done for the entire picture description. The performance of ICOLA-4, however, is slightly understated, as the time measurements include the time to read the special "condensed" picture description from a file.

All performance timings for ICOLA-3 and ICOLA-4 were made on a lightly-loaded SparcStation 10. The timings shown for ICOLA-3 and ICOLA-4 are the average of 10 separate runs of each test. Other quantitative values, such as the number of constraints or variables considered, are not influenced by the load on the machine nor the resolution of the clock.

5.4.2 The Test Cases

The example picture descriptions used for testing the performance of the constraint solver varied in size, complexity, and characteristics. The test cases were chosen to examine the behavior of the incremental solver of ICOLA-4 with respect to:

1. degenerate or pathological picture descriptions.

2. large, complex picture descriptions with little consideration paid to the contents of the image.

3. typical logic programming layout examples.

The first set of test cases is biased towards the non-incremental solver – they are designed to make the incremental solver do as much work as possible. The remaining test cases are not biased towards either solver.

There are three sets of test cases, comprised of 23 test cases in total, which correspond to the above three classifications. Test cases 1 through 12 are the "degenerate examples". In particular, tests 1 through 6 are referred to as the "degenerate insertion examples", and tests 7 through 12 are referred to as the "degenerate deletion examples". The second set of test cases, which includes tests 13 through 15, are referred to as the "large picture descriptions". The third set of test cases, consisting of tests 16 through 23, are referred to as the "typical logic programming examples". The specific properties of these classifications are now discussed in detail.

The first set of test cases are the "degenerate examples" (tests 1 through 12). The "degenerate insertion examples" (tests 1 through 6), contain pathological sequences of simple object and constraint insertions designed explicitly to perform very poorly with ICOLA-4's incremental algorithm. A small sample of the picture description is shown in Figure 5.1. Each new object is constrained so that all other previously inserted objects must be moved to the right of and below the new object. This means, of course, that the position of all other objects in the layout must be recomputed for every new constraint. Figure 5.2 shows an annotated sequence of the images produced as each new object and constraints are encountered. Note that un-annotated versions of the pictures in this sequence are not shown. The number of objects in the degenerate insertion examples range from 101 to 8001, and the number of constraints from 198 to 15998.

```
object( a0, box ).
object( a1, box ).
constraint( left_of, [ left( a1 ), left( a0 ) ]).
constraint( above, [ top( a1 ), top( a0 ) ]).
object( a2, box ).
constraint( left_of, [ left( a2 ), left( a1 ) ]).
constraint( above, [ top( a2 ), top( a1 ) ]).
object( a3, box ).
constraint( left_of, [ left( a3 ), left( a2 ) ]).
constraint( above, [ top( a3 ), top( a2 ) ]).
object( a4, box ).
constraint( left_of, [ left( a4 ), left( a3 ) ]).
constraint( above, [ top( a4 ), top( a3 ) ]).
object( a5, box ).
constraint( left_of, [ left( a5 ), left( a4 ) ]).
constraint( above, [ top( a5 ), top( a4 ) ]).
⋮
```

Figure 5.1: Part of a degenerate picture description.

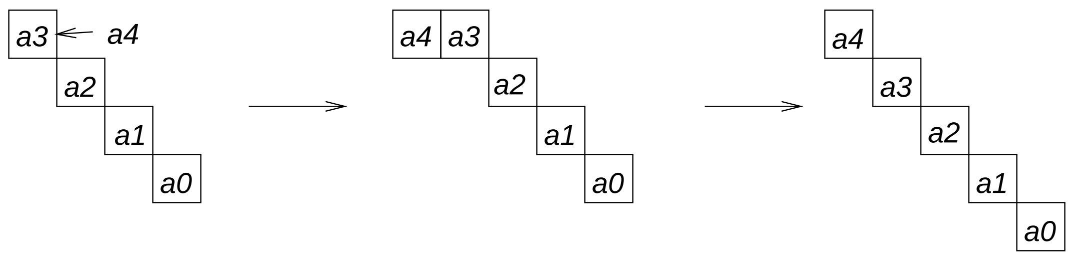


Figure 5.2: Sequence of images produced by the degenerate picture description.

The "degenerate deletion examples", (tests 7 through 12) are the same as the "degenerate insertion examples" (tests 1 through 6) with the exception that they also delete (in a degenerate fashion) the previously inserted objects and constraints. Figure 5.3 contains the beginning and the end of test case 7. All of the degenerate deletion examples follow this pattern. The sequence shown in Figure 5.2 is reversed as constraints are deleted. The differences between the individual test cases in the degenerate deletion examples is only in the number of objects and constraints added and then removed. The number of objects and constraints added and then removed range from 101 objects and 198 constraints in the smallest test case to 8001 objects and 15998 constraints in the largest.

```
object( a0, box ).
object( a1, box ).
constraint( left_of, [ left( a1 ), left( a0 ) ]).
constraint( above, [ top( a1 ), top( a0 ) ]).
object( a2, box ).
constraint( left_of, [ left( a2 ), left( a1 ) ]).
constraint( above, [ top( a2 ), top( a1 ) ]).
object( a3, box ).
constraint( left_of, [ left( a3 ), left( a2 ) ]).
constraint( above, [ top( a3 ), top( a2 ) ]).
⋮
delete constraint( above, [ top( a3 ), top( a2 ) ]).
delete constraint( left_of, [ left( a3 ), left( a2 ) ]).
delete object( a3, box ).
delete constraint( above, [ top( a2 ), top( a1 ) ]).
delete constraint( left_of, [ left( a2 ), left( a1 ) ]).
delete object( a2, box ).
delete constraint( above, [ top( a1 ), top( a0 ) ]).
delete constraint( left_of, [ left( a1 ), left( a0 ) ]).
delete object( a1, box ).
delete object( a0, box ).
```

Figure 5.3: A degenerate picture description with deletions.

The "large picture descriptions" (tests 13, 14, and 15) exercise nearly all components of ICOLA-4, including its line drawing capabilities. These are large collections of assorted picture descriptions initially used during the development of ICOLA-4.

The "typical logic programming examples" (tests 16 through 23) comprise the third set of test cases. The typical logic programming examples are binary trees of depth 3 through 10. The number of nodes in the binary trees range from 15 to 2047. These trees consist of nodes, which comprise a box with a text label inside, but no branches (i.e. no `line` objects are in the picture description). The branches were not included as the position of `line` objects are not computed by the constraint solution algorithms. The `line` objects would occur very frequently in these picture descriptions and adding them would distort the measurement of the performance of the constraint solution algorithms. Figure 5.4 shows the basic component of the tree, and Figure 5.5 shows the resulting picture from test 16.

```
⋮
group( a1 ).
object( a0, box ).
object( a3label, text ).
constraint( inside, [ a3label, a0 ]).
attribute( a3label, text, "a0" ).
object( a1, box ).
object( a1label, text ).
constraint( inside, [ a1label, a1 ]).
attribute( a1label, text, "a1" ).
object( a2, box ).
object( a2label, text ).
constraint( inside, [ a2label, a2 ]).
attribute( a2label, text, "a2" ).
constraint( left_of, [ a1, a0 ]).
constraint( left_of, [ a0, a2 ]).
constraint( above, [ a0, a1 ]).
constraint( above, [ a0, a2 ]).
end_group( a1 ).
⋮
```

Figure 5.4: Part of a picture description of a simple tree-like structure.

Table 5.1 shows various quantitative properties of the 23 test cases. For each test case, Table 5.1 shows the number of objects and constraints inserted or deleted. It also shows the highest number of internal variables and internal constraint formulae present for that test case. These maximum values are important as for some test cases, like those in the degenerate deletion examples (tests 7 through 12), the number of internal variables and constraint formulae decreases from these maximum values as objects and constraints are removed from the system. By the end of each of the

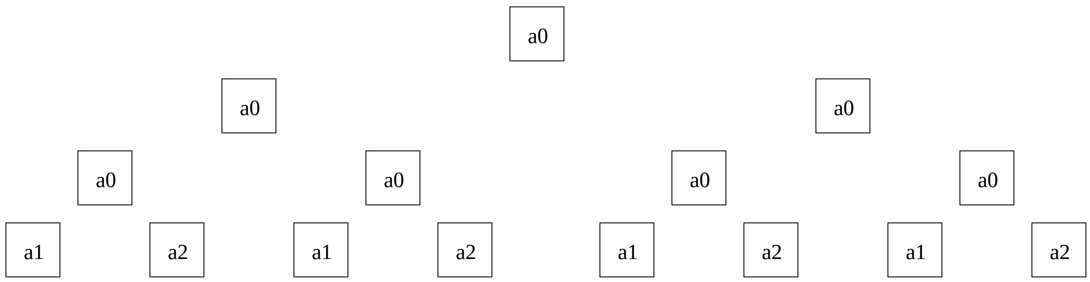


Figure 5.5: The tree structure for test case 16.

Table 5.1: Properties of the test picture descriptions.

Test	Number of Objects Inserted/ Deleted	Maximum Number of Variables (Internal)	Number of Constraints Inserted/ Deleted	Maximum Number of Constraints (Internal)	Number of Increments	Number of Solves
1	101	404	198	800	298	398
2	201	804	398	1600	598	798
3	501	2004	998	4000	1498	1998
4	1001	4004	1998	8000	2998	3998
5	2001	8004	3998	16000	5998	7998
6	8001	32004	15998	64000	23998	31998
7	101	404	396	800	596	796
8	201	804	796	1600	1196	1596
9	501	2004	1996	4000	2996	3996
10	1001	4004	3996	8000	5996	7996
11	2001	8004	7996	16000	11996	15996
12	8001	32004	31996	64000	47996	63995
13	365	1460	528	2822	1261	1372
14	1461	5844	2120	11314	5073	5496
15	5845	23380	8488	45282	20316	21992
16	37	148	43	306	100	178
17	77	308	91	642	212	370
18	157	628	187	1314	436	754
19	317	1268	379	2658	884	1522
20	637	2548	763	5346	1780	3058
21	1277	5108	1531	10722	3572	6130
22	2557	10228	3067	21474	7156	12274
23	5117	20468	6139	42978	14324	24562

degenerate deletion examples, all inserted objects and constraints are completely removed. The table also shows the number of increments in each test case, and the number of calls to the $incremental\_solve\_insert()$ and $incremental\_solve\_delete()$ functions. The number of objects in the 23 test cases range from 37 to 8001, and the number of constraints range from 43 to 31996. The number of calls to the $incremental\_solve\_insert()$ or $incremental\_solve\_delete()$ functions range from 178 to 63995. Internally, at the formulae level, the number of variables range from 148 to 32004, and the number of constraint formulae range from 306 to 64000. These test cases represent a wide range of potential picture descriptions. The performance of ICOLA-4 on these descriptions gives a good indication of ICOLA-4's performance on any picture description.

5.4.3 The Performance Results

The raw data for Figures 5.6 and 5.7 is found in Table B.1 in Appendix B. Figure 5.6 shows the performance of ICOLA-4 compared to ICOLA-3 for the degenerate insertion examples (test cases 1 through 6). In this comparison, a new solution is computed after each increment. As the figure indicates, both the the incremental and non-incremental algorithms in ICOLA-4 are faster than the constraint algorithm in ICOLA-3.

Figure 5.7 focuses only on tests 1 through 5 of the degenerate insertion examples. This figure shows that for smaller picture descriptions that the performance of ICOLA-4 is also better than the performance of ICOLA-3. Hence ICOLA-4 should be preferred over ICOLA-3 for all picture descriptions. None of the other test cases could be performed with ICOLA-3 as they contain language constructs which are not supported in ICOLA-3.

Table B.2 in Appendix B shows the raw performance of ICOLA-4 for each of the 23 test cases. Figures 5.8, 5.9, 5.10, and 5.11 show the performance of the batch algorithm invoked after each increment, the incremental algorithm invoked after each increment, the batch algorithm invoked once at the end of the picture

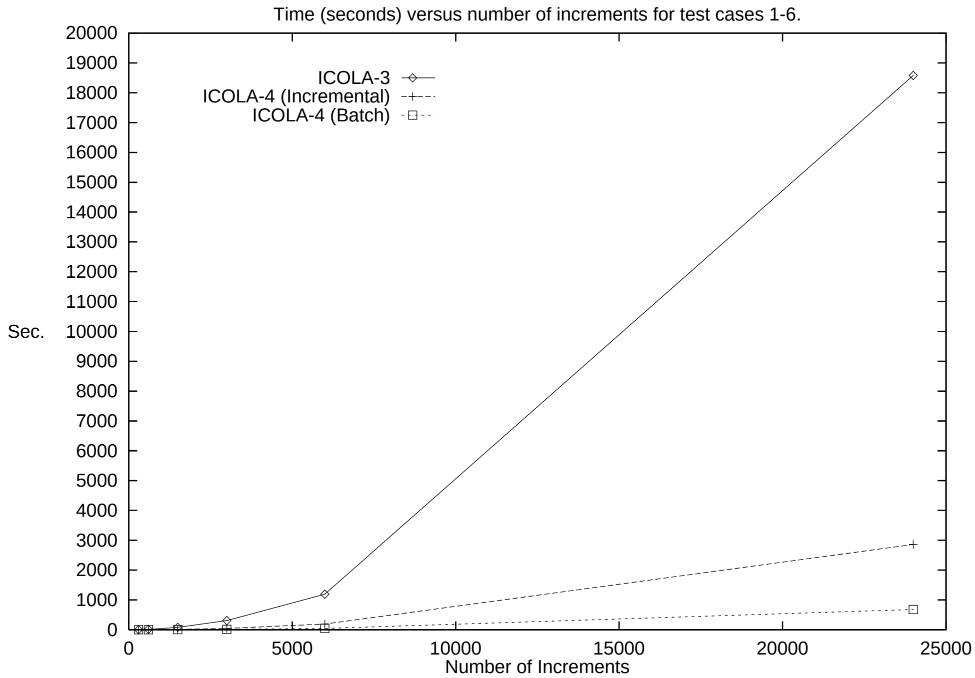


Figure 5.6: The performance of ICOLA-3 and ICOLA-4 on degenerate insertion examples.

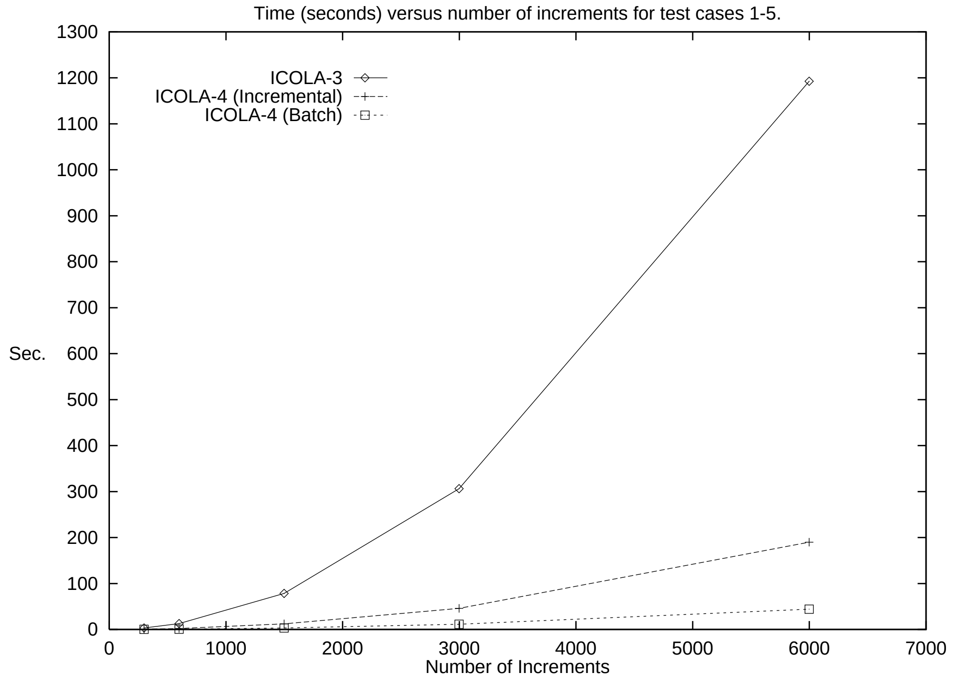


Figure 5.7: The performance of ICOLA-3 and ICOLA-4 on selected degenerate insertion examples.

description, and the time to read and process the picture description without solving the constraints.

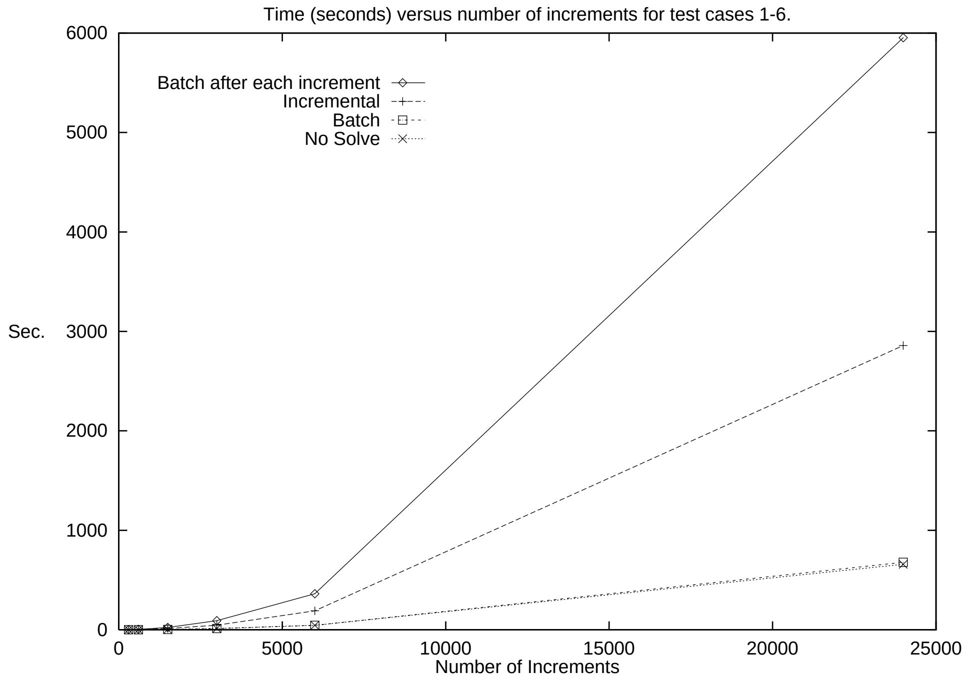


Figure 5.8: The performance of ICOLA-4 on the degenerate insertion examples.

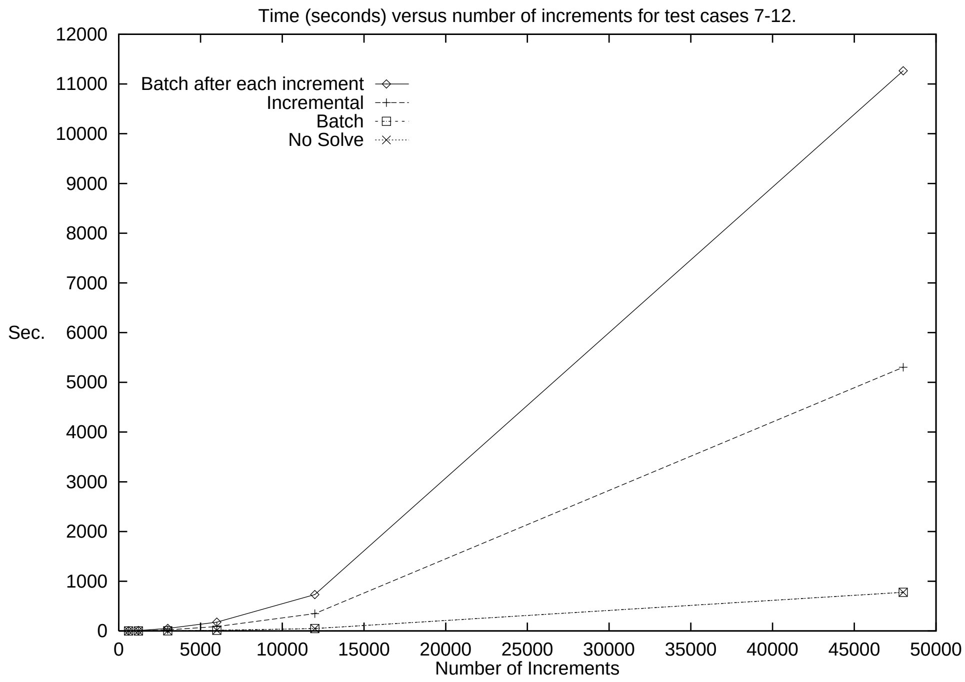


Figure 5.9: The performance of ICOLA-4 on the degenerate deletion examples.

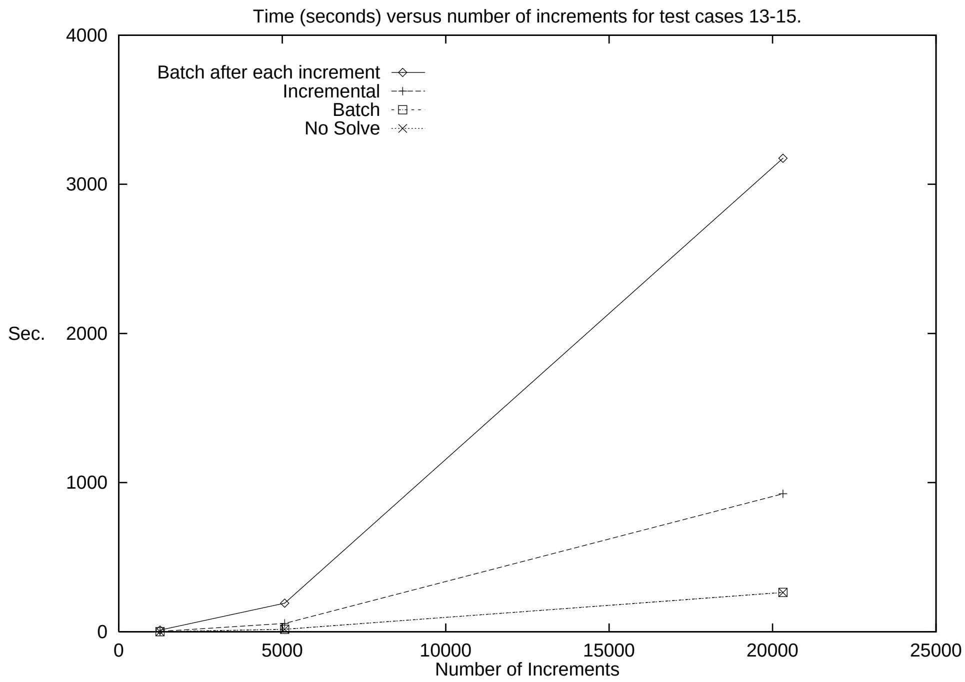


Figure 5.10: The performance of ICOLA-4 on the large picture descriptions.

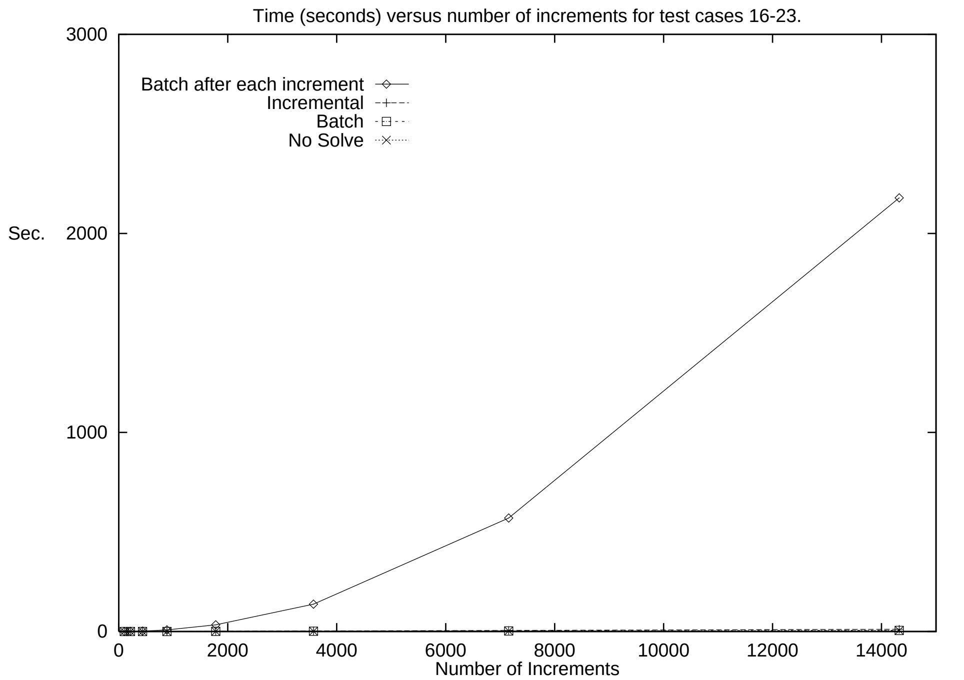


Figure 5.11: The performance on the typical logic programming examples.

As Figures 5.8, 5.9, 5.10, and 5.11 indicate, if the only image required for a picture description is the final image, then the non-incremental (batch) algorithm

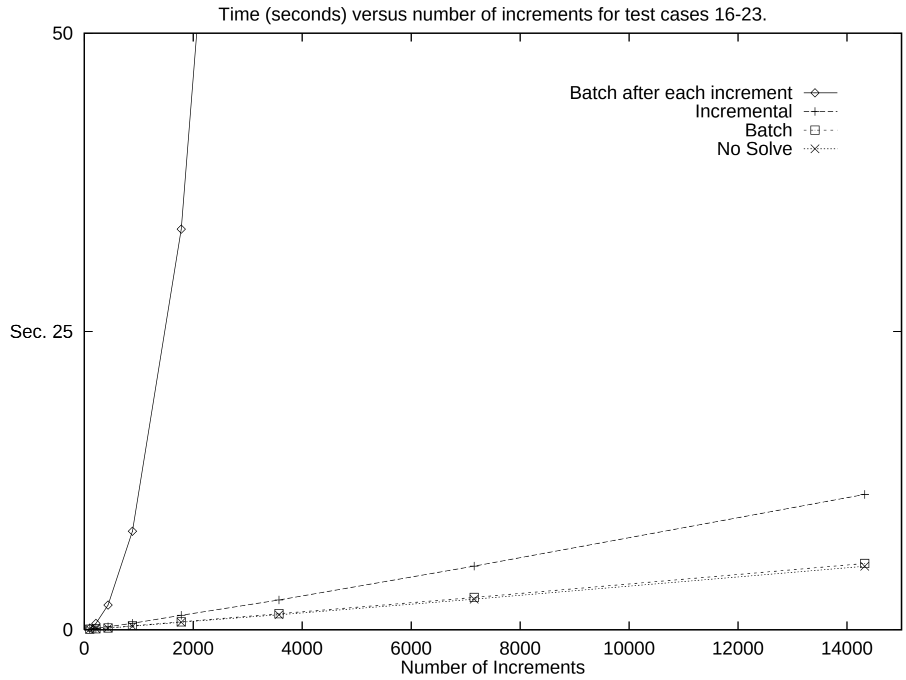


Figure 5.12: Emphasizing the performance of the incremental algorithm on the typical logic programming examples.

is the algorithm of choice. In all the test cases the single invocation of the batch algorithm performed better than the incremental algorithm. The performance of the batch algorithm is very close to the time required to process the picture description without finding any solution. This indicates that the batch algorithm is very efficient in finding a single solution as the time required to find that solution is negligible compared to the total time.

If a dynamic image is desired, with an updated picture after each increment, then the incremental algorithm is the best algorithm. As is expected, invoking the batch algorithm for each increment is much more expensive than using the incremental algorithm. It should be noted, as seen in Figures 5.8 and 5.9, that the incremental algorithm performs only twice as good as the batch algorithm for find a solution after each increment. This indicates that the overall performance of the two algorithms on degenerate picture descriptions is nearly identical, as the incremental algorithm only re-solves either the horizontal or the vertical positioning for each increment while the non-incremental algorithm solves both. These results are expected given the degenerate nature of the picture descriptions.

As shown in Figure 5.10, as the number of objects and constraints increases in the large picture descriptions, the performance of the incremental algorithm becomes much better than the non-incremental (batch) algorithm at finding a solution after each increment. For the typical logic programming examples, however, the performance results are even better. Figure 5.11 clearly shows that the incremental algorithm performs significantly better than the non-incremental algorithm when finding a solution after each increment.

Figure 5.12 more closely examines the incremental and batch results for ICOLA-4 on the typical logic programming examples. As is seen in this figure, the incremental algorithm is only twice as slow as the single call to the batch solver, and yet the incremental algorithm offers an updated solution after each increment. This figure clearly indicates that performance of the incremental algorithm can be quite good if the picture description is not degenerate in nature. The results in Figure 5.12 is typical of the performance results seen for ICOLA-4 for most picture descriptions.

The real-time system response of ICOLA-4 to the user is also very good. As seen in Table B.2 in Appendix B for test case 12 (the largest degenerate deletion example test case), the 47996 increments in the picture description file are solved in 5302618.7 milliseconds (ms), which is 110.5 ms per increment. For test case 23 (the largest typical logic programming example test case), the 14324 increments in the picture description file are solved in 11344.0 ms, which is under 1 ms per increment. The only delay perceived by users is in displaying large images to the screen (after all coordinate values are determined), which is caused by the poor performance of the graphics facilities used by ICOLA-4 and is not a problem with the constraint solver. This performance is fast enough to make ICOLA-4 suitable for “on-line” or interactive user applications.

It should be noted that the timings presented here are approximate and tend to understate the true performance of the solution algorithms. In test case 22, for example, the total time to process each increment and perform a consistency check without finding a solution is 2572.8 ms (See Table B.2 in Appendix B). This means that the non-incremental solution algorithm really took only (2715.9 - 2572.8) = 143.1 ms to find a solution once the set of internal variables and internal constraints were known. At the same time, the incremental algorithm took only (5342.0 - 2572.8) = 2769.2 ms to make 12274 calls to the incremental solution function, an average of 0.2256 ms per solution. Further research is needed determine a full set of metrics for evaluating the incremental constraint solver in ICOLA-4 as the metrics used in this section understate the performance of ICOLA-4.

5.5 Limitations of ICOLA-4

There are still a number of deficiencies in ICOLA-4.

The current graphics system is not efficient. The implementation of the Graphics Manager library routines in SICStus Prolog [SICS94] has a memory leak. The problem occurs when objects are repeatedly drawn to the screen and then erased. While SICStus Prolog's GM was the best graphics library available at the time the implementation began, it is now highly recommended that different graphics support be found for future versions of ICOLA.

The `arc` object is not supported by the graphics libraries. The memory leak in the graphics libraries makes it not worth the effort to code an arc-drawing algorithm, especially when such facilities are already available in many other graphics packages. Instead, a simplistic approximation algorithm which draws line-segments is implemented.

In the current implementation, if the position of any object changes, then the positions of all objects of class LINE are re-computed. The reason for this approach is that determining which objects of class LINE need to be changed is as computationally expensive as simply re-computing all of them.

The heuristics used in ICOLA-4 make it quite difficult for ICOLA-4 to conform to the Principle of Least Astonishment. Since the addition of a simple `left_of` constraint can move an object from the left side of the image to the right side of the image, it is trivial for a single `left_of` constraint to place a group of objects far away from its previous location. The difficulty with the Principle of Least Astonishment arises from the fact that there is only one solution to the constraints which satisfies the heuristics (See Section 4.2), and there are no provisions for "intermediate solutions" (e.g. solutions that would animate the movement of the objects from one location to another). To combat this problem, a picture description could provide a set of constraints which animates the movement of an object from one position to the other. This animation could achieve the desired effect of the Principle of Least Astonishment while leaving ICOLA to simply solve constraints.

There are some inherent difficulties with the `aligned` and `touching` constraints. For example, the constraint

```
constraint( aligned, [ left( box_a ), right( box_b ) ])
```

in the picture description in Figure 5.13 may produce a different picture than the constraint

```
constraint( aligned, [ right( box_b ), left( box_a ) ])
```

in Figure 5.14. While the picture descriptions are otherwise exactly the same, Figure 5.13 produces the image in Figure 5.15, and Figure 5.14 produces Figure 5.16.

```
object( box_a, box ).
object( box_b, box ).
constraint( above, [ box_a, box_b ]).
constraint( aligned, [ right( box_a ), left( box_b ) ]).
```

Figure 5.13: A simple picture description containing the `aligned` constraint.

```
object( box_a, box ).
object( box_b, box ).
constraint( above, [ box_a, box_b ]).
constraint( aligned, [ left( box_b ), right( box_a ) ]).
```

Figure 5.14: Another simple picture description containing an `aligned` constraint.

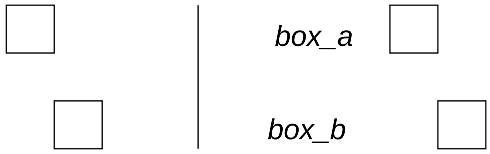


Figure 5.15: Image showing the expected behavior of the `aligned` constraint.

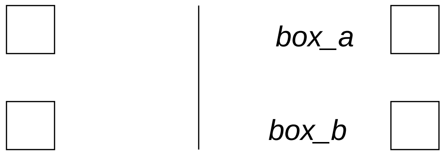


Figure 5.16: Image showing the unexpected behavior of the `aligned` constraint.

The reason for this behavior is that the `aligned` and `touching` constraints are converted to `horizontal_distance` and `vertical_distance` constraints. These latter constraints specify only minimum distances, and also imply a `left_of` or `above` relationship, respectively. The constraint

```
constraint( aligned, [ left( box_a ), right( box_b ) ])
```

is converted to

```
constraint( horizontal_distance,
            [ 0, left( box_a ), right( box_b ) ]),
```

while the constraint

```
constraint( aligned, [ right( box_b ), left( box_a ) ])
```

is converted to

```
constraint( horizontal_distance,
            [ 0, right( box_b ), left( box_a ) ]),
```

which is a completely different constraint. The respective `horizontal_distance` constraints are satisfied, however, in the images in Figures 5.15 and 5.16. The `aligned` and `touching` constraints are performing correctly under their current definitions as aliased constraints, but do not always perform consistently with the usual interpretation of "aligned" or "touching". Future revisions of ICOLA may address this problem.

If two objects overlap there is no guarantee as to which will occlude the other. The current behavior is for objects to be drawn in the order they are defined, and hence if objects overlap then the objects defined later will occlude objects defined earlier.

If an `oval` is placed inside a `box`, it may be the case that the corners of the `box` lie outside the edge of the `oval`. This is due to the fact that the placement of both objects is with respect to their bounding boxes, which is quite different from the actual rendering of the object in the case of the `oval`.

Future implementations of ICOLA may address all of the above limitations.

5.6 Chapter Summary

In this chapter the properties of ICOLA's picture specification language are compared to general visual languages. ICOLA-4 is briefly compared to other layout systems and the incremental constraint algorithm is compared to some of the constraint algorithms and techniques used in other systems. The performance of the incremental algorithm of ICOLA-4 is shown to be much better than the non-incremental algorithm in ICOLA-4. It is also much better than the constraint algorithm in ICOLA-3, and hence is also significantly better than all previous versions of ICOLA. The performance is also better than at least one general-purpose constraint satisfaction tool [Hoff95]. While ICOLA-4 has some limitations, the performance results indicate that ICOLA-4 is very capable of efficiently finding a solution to a large set of objects and constraints. The performance is also fast enough that it is suitable for "on-line" or interactive user-applications.

CHAPTER 6

CONCLUSION

ICOLA-4 is the new graphics engine for the Wand system. As shown in Chapter 3, the picture description language for ICOLA-4, PDI-4, is simple and yet capable of representing complex pictures using only relative positioning constraints.

Chapter 4 presented the non-incremental (batch) and incremental constraint solution algorithms of ICOLA-4. ICOLA-4 is capable of efficiently handling both the incremental insertion and the incremental deletion of objects and constraints. It is shown that for a highly under-constrained set of constraints that there is only one possible solution which satisfies the two heuristics of ICOLA-4. It is also shown that ICOLA-4 finds that solution.

Chapter 5 examined the quantitative performance aspects of ICOLA-4. For incremental changes to an image, the incremental algorithm of ICOLA-4 is shown to be superior to both the non-incremental algorithm of ICOLA-4, and to the constraint algorithm of ICOLA-3. If only the final image from a picture description is desired, then the non-incremental algorithm of ICOLA-4 is superior. ICOLA-4 is also faster than an existing, general-purpose constraint satisfaction tool [Hoff95].

ICOLA-4's primary concern is in the production of pictures for a logic program visualization tool. ICOLA-4's limitations occur primarily when it is expected to do something other than that for which it is designed.

6.1 Contributions

The relative positioning capabilities and the basic object classes of PDI and ICOLA provide a simple, declarative means of describing complex pictures. PDI-4 and ICOLA-4, however, add additional features and functionality not present in previous versions of PDI nor ICOLA.

PDI-4 now supports all language constructs initially specified for PDI but not supported by PDI-3 nor implemented in ICOLA-3. The addition of a local scoping scheme to PDI-4 allows easy replication of sub-parts of pictures as local object names must only be unique at the same scope. As well, the addition of attributes facilitates easy specification of other visual properties of the objects. The language is also powerful enough to describe any arbitrary bitmap image (See Section 5.1).

The primary contribution of this work is the new incremental constraint solver, which supports both the incremental insertion and the incremental deletion of objects and constraints. The modified behavior of the `horizontal_distance` and `vertical_distance` constraints allowed the realization of a simpler incremental constraint solver which produces more consistent images than ICOLA-3. One of the main design goals of the constraint solvers in ICOLA-4 is confluence – that constraints, objects, and attributes can be added in any order, and that the resulting picture will be the same. This goal has been achieved. As well, the image produced after a constraint or object deletion is exactly the same as if the constraint or object had never been inserted in the first place.

ICOLA-4 is practical, as the constraint solver is very fast and finds consistent solutions to picture specifications which are highly under-constrained. Incremental solutions are typically computed by the implementation in less than 1 millisecond. The implementation also allows a picture description to be solved in "batch" mode, incremental mode, or an arbitrary combination of the two. This allows greater flexibility in solving picture descriptions, and facilitates such functionality as grouping objects and associated constraints together, and re-drawing the image only after the entire group is solved.

ICOLA-4 is extendable, as demonstrated by the use of a slightly modified version ICOLA-4 in the Automatic Diagram Layout research [Mort95] (See Section 5.2). As well, parts 1 and 2 of the incremental algorithms could be used to find incremental solutions to general sets of constraints from other applications. In particular, the algorithms could be used for determining (in the domain of non-negative integers) the minimum values for variables x and y in inequalities of the form $x+\alpha \leq y$, where $\alpha \geq 0$. ICOLA-4 could also be used to draw images corresponding to temporal intervals (See Section 2.5.4) since ICOLA-4 supports all the relationships in Allen's 13-valued interval algebra [Alle83].

The operational semantics (See Section 3.10) and the incremental algorithms (See Chapter 4) provide a solid foundation from which other implementations of ICOLA-4 can be built. ICOLA-4 joins other incremental systems in showing that incremental systems can be much faster than their non-incremental counterparts. The usage of ICOLA in the Wand framework awaits only the development of a tool to map the configuration specification (See Section 2.6.2) to the picture description constructs.

6.2 Future Research

There are a number of issues related to ICOLA and PDI which should be investigated as future work.

1. User-defined object types – Allowing the user to define new object types and instances of those objects would enhance the ease with which more complicated objects could be expressed. Interesting issues include whether these user-defined object types should be changeable once an instance of the object is created.

2. User-defined constraints – User-defined constraints could behave as aliased constraints, and could be composed of the current constraints in PDI and other user-defined constraints. This would allow a very compact representation of

complicated relationships between objects. As well, all constraints except the `connects` constraint are between only two objects (or two edges of objects). User-defined constraints could potentially be applied to a set of objects rather than just a pair of objects. An interesting issue is whether these user-defined constraints should be modifiable once the constraint is applied.

3. Additional attributes – Potential new attributes include arbitrary rotation for all objects and separate inter-object and intra-object distances. The addition of user-defined attributes could also be explored.

4. Replacing `horizontal_distance` and `vertical_distance` constraints – While the current behavior of these constraints is more relative than in previous versions of PDI, these constraints could still be completely replaced by such constraints as `near_to` or `far_from`. The exact default distance for these new constraints could be specified via attributes like the current default distance attribute.

5. User changeable inter-object and intra-object distances – In the current system, the inter-object and intra-object distances are equivalent. Making these two distances independent would allow extra flexibility in the specification of pictures.

6. Improvement of triangle handling – The `triangle` object is very basic and could be extended to support other types of triangles such as scalene.

7. Incremental computation of line positions – As noted in Section 5.5, the current implementation re-computes the position of all objects of class LINE if an increment changes the position of any object. A better method of handling this re-computation would be to only re-compute the position of those objects of class LINE whose endpoints change.

8. Extended graphics facilities – In the current system, the graphics engine re-draws the entire picture after each increment. A system based on fast back-

trackable graphics or a transition-oriented drawing approach could increase the efficiency of the rendering of the image by re-drawing only those areas of the picture which have changed.

9. Three-dimensional images – The layout algorithm in ICOLA-4 has a very direct and obvious extension from two dimensions to three dimensions. Constraints such as `in_front_of` and `behind` in the third dimension could represent the same relationships as `left_of` and `right_of` in the horizontal dimension. $\texttt{root}_{front}$ and $\texttt{root}_{back}$ could represent the front and back of the object. Current objects like `oval` could be extended to `spheroid`, and `triangle` to `pyramid`. The `box` object could also be extended in a natural way to the third dimension. Interesting problems include specifying a perspective from which to view the objects, and determining which lines and surfaces are hidden from the given viewing point.

10. Development of performance metrics for ICOLA-4 and other incremental constraint solvers – As mentioned in Section 5.4.3, the timing values presented are approximate. In order to determine the true speed of the algorithm, additional work could be done on separating the overhead of inserting the objects and constraints from the cost of finding a solution. Initial investigation shows that there are many issues involved which are non-trivial in nature. In particular, one of the most difficult problems is how to properly amortize the costs associated with computing the new set of constraints after an increment.

11. Additional user interaction – User interaction is currently limited to adding or removing objects and constraints or to changing attributes via a text-based interface. Allowing a user to manipulate the graphical objects via a "point-and-click" interface would introduce many interesting problems, including determining what the proper behavior of the tool is if a user attempts to move an object to a location where it violates the current constraints.

12. Development of a formal, declarative semantics for PDI – The behavior of ICOLA-4 and PDI-4 is currently specified by an operational semantics, but there are no declarative semantics for PDI. The declarative semantics will be complicated by object and constraint deletions, and by object attributes.

6.3 Summary

The primary contribution of this work is the new incremental constraint solver which is capable of rapidly finding consistent solutions to highly under-constrained declarative picture descriptions. This work is not only practical, but important to the development of visual debugging and monitoring tools for the logic programming community. This research is also important for its development of a declarative graphics language which provides a simple means of describing complex pictures.

REFERENCES

[Aït+94] Aït-Kaci, H., Dumant, B., Meyer, R., Podelski, A., and Van Roy, P. *The Wild LIFE Handbook (pre-publication edition)*, DIGITAL, Paris Research Laboratory, March, 1994.

[AlKa90] Ali, K. A. M. and Karlsson, R. The Muse Or-Parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming* (October 29-November 1, Austin, Texas), 1990, pp. 757-776.

[Alle83] Allen, J. F. Maintaining Knowledge about Temporal Intervals, *Communications of the ACM 26*, 11 (November 1983), 832-843.

[Born81] Borning, A. H. The Programming Language Aspects of ThingLab, A Constraint Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems 3*, 4 (October 1981), 353-387.

[Bor+87] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M. Constraint hierarchies. In *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (October 4-8, Orlando, Florida), 1987, pp. 48-60.

[Bri+95] Brisset, P., et al. ECL^{i}PS^e *(ECRC Common Logic Programming System) Extensions User Manual*, Release 3.5, ECRC, West Germany, February, 1995.

[Car+93] Carro, M., Gómez, L., and Hermenegildo, M. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *Proceedings, ICLP'93*, (June 21-25, Budapest, Hungary), 1993, pp. 184-200.

[Chan87] Chang, S.-K. Visual Languages: A Tutorial and Survey, *IEEE Software 4*, 1 (January 1987), 29-39.

[Coh+92] Cohen, R. F., Di Battista, G., Tamassia, R., Tollis, I. G., Bertolazzi, P. A Framework for Dynamic Graph Drawing (Extended Abstract). In *ACM Proceedings of the Eighth Annual Symposium on Computational Geometry* (June 10-12, Berlin, Germany), 1992, pp. 261-270.

[Cor+92] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1992.

[CoCh90] Costagliola, G. and Chang, S.-K. DR PARSERS: a generalization of LR parsers. In *Proceedings of the 1990 IEEE Workshop on Visual Languages* (October 4-6, Skokie, Illinois), 1990, pp. 174-180.

[Cos+91] Costagliola, G., Tomita, M., and Chang, S.-K. A Generalized Parser for 2-D Languages. In *Proceedings of the 1991 IEEE Workshop on Visual Languages* (October 8-11, Kobe, Japan), 1991, pp. 98-104.

[Cri+90] Crimi, C., Guercio, A., Nota, G., Pacini, G., Tortora, G., and Tucci, M. Relational Grammars for Modeling Multi-dimensional Structures. In *Proceedings of the 1990 IEEE Workshop on Visual Languages* (October 4-6, Skokie, Illinois), 1990, pp. 138-173.

[Cruz93] Cruz, I. F. Using a Visual Constraint Language for Data Display Specification. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, (April 28-30, Newport, Rhode Island), 1993, pp. 24-35.

[Davi87] Davis, E. Constraint Propagation with Interval Labels, *Artificial Intelligence 32*, 3 (July 1987), 281-331.

[Bat+93] Di Battista, G., Eades, P., Tamassia, R., and Tolis, I. G. Algorithms for Drawing Graphs: an Annotated Bibliography, Technical Report, Brown University, November, 1993.

[Din+88] Dincbas, M., van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (December, Tokyo, Japan), 1988, pp. 693-702.

[DiLu87] Disz, T. and Lusk, E. A graphical tool for observing the behavior of parallel logic programs. In *Proceedings, 1987 Symposium on Logic Programming*, (August 31-September 4, San Francisco, California), 1987, pp. 46-53.

[DoHé93] Donikian, S. and Hégron, G. Constraint Management in a Declarative Design Method for 3D Scene Sketch Modeling. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, (April 28-30, Newport, Rhode Island), 1993, pp. 36-45.

[DuYa93] Dubé, T. and Yap, C.-K. The Geometry in Constraint Logic Programs. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, (April 28-30, Newport, Rhode Island), 1993, pp. 46-52.

[EiBr88] Eisenstadt, A. M. and Brayshaw, A. M. The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming, *Journal of Logic Programming 5*, 4, (1988), 277-342.

[Fer+91] Ferrucci, F., Pacini, G., Tortora, G., Tucci, M., and Vitiello, G. Efficient Parsing of Multidimensional Structures. In *Proceedings of the 1991 IEEE Workshop on Visual Languages* (October 8-11, Kobe, Japan), 1991, pp. 105-110.

[Fre+90] Freeman-Benson, B. N., Maloney, J., and Borning, A. An incremental constraint solver, *Communications of the ACM, 33*, 1 (January 1990), 54-63.

[Glei93] Gleicher, M. Practical Issues in Graphical Constraints. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, (April 28-30, Newport, Rhode Island), 1993, pp. 100-108.

[HaMö90] Haarslev, V. and Möller, R. A Declarative Formalism for Specifying Graphical Layout. In *Proceedings of the 1990 IEEE Workshop on Visual Languages* (October 4-6, Skokie, Illinois), 1990, pp. 54-59.

[HeMa90] Helm, R. and Marriott, K. Declarative Specification of Visual Languages. Technical Report RC 15813(#70304) 5/31/90, IBM Research Division, T. J. Watson Research Center.

[Hel+92] Helm, R., Huynh, T., Lassez, C., and Marriott, K. A linear constraint technology for interactive graphics systems, Technical Report RC 16913 (#74966). Also appears in *Proceedings of "Graphics Interface '92"*, (May, Vancouver, Canada), 1992.

[Hel+93] Helm, R., Huynh, T., Marriott, K., Vlissides, J. An Object-Oriented Architecture for Constraint-Based Graphical Editing. *Advances in Object-Oriented Graphics II*, Springer Verlag, 1993.

[Hel+90] Helm, R., Marriott, K., and Odersky, M. Building Visual Language Parsers. Technical Report RC 16200(#71795) 10/5/90, IBM Research Division, T. J. Watson Research Centre. Also appears in *Proceedings of Computer Human Interaction (CHI) '91* (April, New Orleans, Louisiana), 1991.

[HiJu91] Hirakawa, M. and Jungert, E. An Image Database System Facilitating Icon-Driven Spatial Information Definition and Retrieval. In *Proceedings of the 1991 IEEE Workshop on Visual Languages* (October 8-11, Kobe, Japan), 1991, pp. 192-198.

[Hoff95] Hoffman, R. H. A Comparison of Efficiency of Specialized versus General Constraint Solvers for a Picture Description Language, Unpublished report, CMPT 400 Project, University of Saskatchewan, April, 1995.

[Holt90] Holt, C. M. viz: A Visual Language Based on Functions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages* (October 4-6, Skokie, Illinois), 1990, pp. 221-226.

[JaMa94] Jaffar, J. and Maher, S. Constraint Logic Programming: A Survey, *Journal of Logic Programming 19-20*, (May–July 1994), 503-581.

[JaMi87] Jaffar, J. and Michaylov, S. Methodology and implementation of a CLP system. In *Proceedings of the Fourth International Conference on Logic Programming*, (May 25-27, Melbourne, Australia), 1987, pp. 196-218.

[JoHa91] Johnson, C. W. and Harrison, M. D. Declarative Graphics and Dynamic Interaction. In *EUROGRAPHICS '91*, 1991, pp. 195-207.

[Kama89] Kamada, T. *Visualizing Abstract Objects and Relations – A Constraint-Based Approach*, World Scientific, Singapore, 1989.

[KaKa91] Kamada, T. and Kawai, S. A General Framework for Visualizing Abstract Objects and Relations, *ACM Transactions on Graphics 10*, 1 (January 1991), 1-39.

[Kegi90] *KEGI User Manual*, Release 2.1, ECRC, West Germany, September, 1990.

[Knut84] Knuth, D. E. *The TEXbook*, Addison-Wesley, Reading, Massachusetts, 1984.

[KoMy91] Kojima K. and Myers, B. A. Parsing Graphic Function Sequences. In *Proceedings of the 1991 IEEE Workshop on Visual Languages* (October 8-11, Kobe, Japan), 1991, pp. 111-117.

[KuOs93a] Kusalik, A. J. and Oster, G. M. Towards a Generalized Graphical Interface for Logic-Programming Development. TR 93-2, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, February, 1993.

[KuOs93b] Kusalik, A. J. and Oster, G. M. Towards Generalized Visualization Support for Logic Programming. In *Proceedings of the Tenth International Conference on Logic Programming*, (June 21-24, Budapest, Hungary), 1993, p. 844.

[KuOs93c] Kusalik, A. J. and Oster, G. M. A Constraint-Based Declarative Display Description Language for Visualization. In *Proceedings of the 5th Workshop on Logic Programming Environments*, (October 29-30, Vancouver, B.C., Canada), 1993, pp. 59-67.

[Ladk87] Ladkin, F. Visual Grammars for Visual Languages. In *Proceedings of the 6th National Conference on Artificial Intelligence, AAAI–'87* (July 13-17, Seattle, Washington), 1987, pp. 683-688.

[Lele88] Leler, W. *Constraint Programming Languages – Their Specifications and Generation*, Addison-Wesley, Reading, Massachusetts, 1988.

[MaSt89] Maher, M. J., Stuckey, P. J. Expanding query power in constraint logic programming languages. In *Proceedings of the North American Conference on Logic Programming*, (October 16-20, Cleveland, Ohio), 1989, pp. 20-36.

[MaFi90] Mahling, D. E. and Fisher, D. L. The Cognitive Engineering of Visual Languages. In *Proceedings of the 1990 IEEE Workshop on Visual Languages* (October 4-6, Skokie, Illinois), 1990, pp. 22-28.

[Mal+89] Maloney, J. H., Borning, A., and Freeman-Benson, B. N. Constraint technology for user-interface construction in ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (October 1-6, New Orleans, Louisiana), 1989, pp. 381-388.

[Men+93] Menezes, F., Barahona, P., and Codognet, P. An Incremental Hierarchical Constraint Solver. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming* (April 28-30, Newport, Rhode Island), 1993, pp. 201-210.

[Meye93] Meyer, B. Pictures Depicting Pictures - On the Specification of Visual Languages by Visual Grammars, Informatik Berichte NR. 132, FernUniversität, February, 1993.

[MiFa90] Mittal, S. and Falkenhainer, B. Dynamic Constraint Satisfaction Problems. In *Proceedings of the 8th National Conference on Artificial Intelligence, AAAI–'90* (July 29–August 3, Boston, Massachusetts), 1990, pp. 25-32.

[Mort95] Morton, R. D. *Automatic Diagram Layout Within Metaview*, M.Sc. Thesis, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, June, 1995.

[Nye90] Nye, A. *Xlib programming manual: for Version 11 of the X Window System*, O'Reilly & Associates, Inc., Sebastopol, California, 1990.

[Oste92] Oster, G. Constraint satisfaction techniques and algorithms, Unpublished report, CMPT 417 Project, University of Saskatchewan, April, 1992.

[Oste93] Oster, G. M. Incremental Addition and Deletion of Constraints in Constraint Satisfaction Systems. In *Proceedings of the Fifth Annual Graduate Symposium on Computational Science* (April 12, Saskatoon, Saskatchewan, Canada), 1993, pp. 285-294.

[OsKu94] Oster, G. M. and Kusalik, A. J. ICOLA Technical Report. Unpublished technical report (Draft). Department of Computational Science, University of Saskatchewan, 1994.

[PeWa80] Pereira, F. C. N. and Warren, D. H. D. Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artificial Intelligence, 13* (1980), 231-278.

[PeKu94] Prestwich, S. D., and Kusalik, A. J. High-Level Visualization of Parallel Logic Program Execution. In *Proceedings of Parallel Architectures and Languages Europe, PARLE'94* (July 4-8, Athens, Greece), 1994, pp. 180-191.

[Pro+91] Protsko, L. B., Sorenson, P. G., Tremblay, J. P., and Schaefer, D. A. Towards the Automatic Generation of Software Diagrams, *IEEE Transactions on Software Engineering 17*, 1 (January 1991), 10-21.

[Quin91] *Quintus X Window Interfaces*, Quintus Corporation, Palo Alto, California, 1991.

[Roac84] Roach, J. A. The Rectangle Placement Language. In *Proceedings of the ACM IEEE 21st Design Automation Conference* (June 25-27, Albequerque, New Mexico), 1984, pp. 405-411.

[Rose79] Rosenfeld, A. *Picture Languages – Formal Models for Picture Recognition*, Academic Press, New York, New York, 1979.

[Row+87] Rowe, L. A., Davis, M., Messinger, E., Meyer, C., Spirakis, C., and Tuan, A. A Browser for Directed Graphs, *Software – Practice and Experience 17*, 1 (January 1987), 61-76.

[Sann93] Sannella, M. The SkyBlue Constraint Solver and Its Applications. In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, (April 28-30, Newport, Rhode Island), 1993, pp. 268-278.

[San+93] Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. Multi–way versus One–way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm, *Software – Practice and Experience 23*, 5 (May 1993), 529-566.

[Seid81] Seidel, R. A new method for solving Constraint Satisfaction Problems. In *Proceedings of the Seventh IJCAI* (Vancouver, British Columbia, Canada), 1981, pp. 338-342.

[SICS94] *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, KISTA, Sweden, April, 1994.

[Stuc91a] Stuckey, P. J. Incremental linear constraints solving and detection of implicit equalities, *ORSA Journal of Computing, 3*, 4 (Fall 1991), 269-274.

[Stuc91b] Stuckey, P. J. A Simplex-based algorithm for incremental linear arithmetic constraint solving and detection of implicit equalities, Technical Report, Department of Computer Science, University of Melbourne, Parkville 3052, Australia, 1991.

[Suth63] Sutherland, I. E. Sketchpad: A man-machine graphical communication system. In *AFIPS Conference Proceedings, Spring Joint Computer Conference*, 1963, pp. 2–19. Also appears in *Visual Programming Environments: Paradigms and Systems*, Los Alamitos, California, 1990, pp. 198-215.

[Tick92] Tick, E. Visualizing Parallel Logic Programs with VISTA. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (June 1-5, Tokyo, Japan), 1992, pp. 934-942.

[vanH90] van Hentenryck, P. Incremental constraint satisfaction in logic programming. In *Proceedings of the Seventh International Conference on Logic Programming*, (June 18-20, Jerusalem, Israel), 1990, pp. 189-202.

[vaLe91] van Hentenryck, P. and Le Provost, T. Incremental search in constraint logic programming, *New Generation Computing 9*, (1991), 257-275.

[Wilk91] Wilk, M. R. Equate: An object-oriented constraint solver. In *Proceedings of the 1991 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (October 6-11, Phoenix, Arizona), 1991, pp. 286-298.

[WiWe90] Wittenburg, K. and Weitzman, L. Visual Grammars and Incremental Parsing for Interface Languages. In *Proceedings of the 1990 IEEE Workshop on Visual Languages* (October 4-6, Skokie, Illinois), 1990, pp. 111-118.

[YuCh90] Yu, B. and Chang, S.-K. A Fuzzy Visual Language Compiler. In *Proceedings of the 1990 IEEE Workshop on Visual Languages* (October 4-6, Skokie, Illinois), 1990, pp. 162-167.

APPENDIX A

PDI-4 LISTINGS

The PDI-4 listing for Figure 3.12 is as follows:

```
reset.
object( goal, box ).
object( subgoal_1, box ).
constraint( below, [ subgoal_1, goal ]).
object( line_1, line ).
constraint( connects, [ line_1, goal, subgoal_1 ]).
attribute( line_1, style, bold ).
object( subgoal_4, box ).
constraint( below, [ subgoal_4, subgoal_1 ]).
object( line_4, line ).
constraint( connects, [ line_4, subgoal_1, subgoal_4 ]).
attribute( line_4, style, bold ).
object( subgoal_5, box ).
constraint( below, [ subgoal_5, subgoal_1 ]).
constraint( left_of, [ subgoal_1, subgoal_5 ]).
constraint( left_of, [ subgoal_4, subgoal_1 ]).
object( line_5, line ).
constraint( connects, [ line_5, subgoal_1, subgoal_5 ]).
attribute( line_4, style, regular ).
attribute( line_5, style, bold ).
object( subgoal_2, box ).
constraint( below, [ subgoal_2, goal ]).
constraint( left_of, [ subgoal_1, subgoal_2 ]).
constraint( aligned, [ right( subgoal_2 ), right( goal ) ]).
constraint( left_of, [ subgoal_5, subgoal_2 ]).
```

```
object( line_2, line ).
constraint( connects, [ line_2, goal, subgoal_2 ]).
attribute( line_5, style, regular ).
attribute( line_1, style, regular ).
attribute( line_2, style, bold ).
attribute( line_2, style, dashed ).
object( subgoal_3, box ).
constraint( below, [ subgoal_3, goal ]).
constraint( left_of, [ subgoal_2, subgoal_3 ]).
object( line_3, line ).
constraint( connects, [ line_3, goal, subgoal_3 ]).
attribute( line_3, style, bold ).
object( subgoal_6, box ).
constraint( left_of, [ subgoal_2, subgoal_6 ]).
constraint( below, [ subgoal_6, subgoal_3 ]).
constraint( left_of, [ subgoal_6, subgoal_3 ]).
object( line_6, line ).
constraint( connects, [ line_6, subgoal_3, subgoal_6 ]).
attribute( line_6, style, bold ).
```

The picture description to generate Figure 3.13 is:

```
reset.
object( box_1, box ).
object( triangle_1, triangle ).
constraint( above, [ box_1, triangle_1 ]).
attribute( triangle_1, rotation, 180 ).
attribute( triangle_1, filled, true ).
object( line_1, line ).
constraint( connects, [ line_1, box_1, triangle_1 ]).
attribute( line_1, style, dashed ).
attribute( line_1, arrowhead, head ).
object( box_2, box ).
constraint( below, [ box_2, triangle_1 ]).
object( happy, bitmap ).
constraint( inside, [ happy, box_2 ]).
attribute( happy, bitmap, "happyface.bm").
constraint( left_of, [ left( box_2 ), left( triangle_1 )]).
constraint( left_of, [ left( box_2 ), left( box_1 )]).
object( line_2, line ).
constraint( connects, [ line_2, triangle_1, box_2 ]).
attribute( line_2, style, bold ).
attribute( line_2, arrowhead, head ).
object( oval_1, oval ).
constraint( left_of, [ box_1, oval_1 ]).
object( line_3, line ).
constraint( connects, [ line_3, box_1, oval_1 ]).
attribute( line_3, arrowhead, tail ).
object( oval_1_text, text ).
constraint( inside, [ oval_1_text, oval_1 ]).
attribute( oval_1_text, text, "$$").
constraint( below, [ top( box_1 ), top( oval_1 )]).
```

APPENDIX B

RAW PERFORMANCE RESULTS

This appendix contains the raw performance results for the performance tests described in Chapter 5. Table B.1 shows the times, in milliseconds, for the degenerate insertion examples (tests 1 through 6) run on the constraint algorithm of ICOLA-3, and the incremental and non-incremental algorithms of ICOLA-4. In this comparison, a new solution is computed after each increment.

Table B.1: Performance comparison between ICOLA-3 and ICOLA-4.

Test	Number Objects	Number of Constraints	ICOLA-3 Time (ms)	ICOLA-4 Incremental Time (ms)	ICOLA-4 Batch Time (ms)
1	101	198	3105.4	555.9	912.6
2	201	398	12859.2	1960.3	3669.5
3	501	998	78300.2	12295.0	23073.6
4	1001	1998	306310.8	45918.0	91167.6
5	2001	3998	1192683.7	189846.1	362371.5
6	8001	15998	18580147.0	2857641.0	5953463.0

Table B.2 shows the raw performance of ICOLA-4 for the 23 test cases. Each timing value includes the total time to convert all the increments involved to the internal form and check their consistency. The second column shows the performance of ICOLA-4 when no solutions are found. The third column shows the performance of the batch algorithm of ICOLA-4 when a single, non-incremental solve of the entire picture description is performed. The performance of the incremental algorithm, where a new solution is found after each increment, is shown in the fourth column. The fifth column shows the performance of ICOLA-4 if the non-incremental algorithm is used to find a new solution after each increment.

Table B.2: ICOLA-4's raw performance results.

Test	No Solve Time (ms)	Batch (Solve at end) Time (ms)	Incremental Time (ms)	Batch (Solve each inc.) Time (ms)
1	191.0	202.4	555.9	912.6
2	526.6	562.2	1960.3	3669.5
3	3039.4	3056.2	12295.0	23073.6
4	11412.4	11440.8	45918.0	91167.6
5	44088.5	44553.5	189846.1	362371.5
6	657786.2	677709.1	2857641.0	5953463.0
7	283.5	283.5	1020.9	1650.9
8	757.7	757.7	3725.2	6756.1
9	3639.3	3639.3	22383.6	51992.3
10	12878.1	12878.1	88659.2	175820.3
11	49361.6	49361.6	347470.6	730982.0
12	778406.8	778406.8	5302618.7	11265154.8
13	1054.3	1065.9	3094.4	11554.0
14	16632.0	16882.1	56490.0	192296.8
15	263582.6	264194.0	925337.5	3174928.5
16	49.1	52.5	60.9	142.7
17	86.0	89.4	121.1	534.3
18	144.2	159.3	259.2	2077.7
19	299.2	331.3	544.4	8262.6
20	631.0	667.7	1217.6	33571.4
21	1261.0	1354.3	2499.5	137372.4
22	2572.8	2715.9	5342.0	570250.1
23	5330.9	5564.2	11344.0	2178829.6

APPENDIX C

ICOLA - THE INTERFACE

Figure C.1 (next page) shows the user-interface of ICOLA-4. In this figure, ICOLA-4 is incrementally determining the image for the picture description corresponding to Figure 3.12.

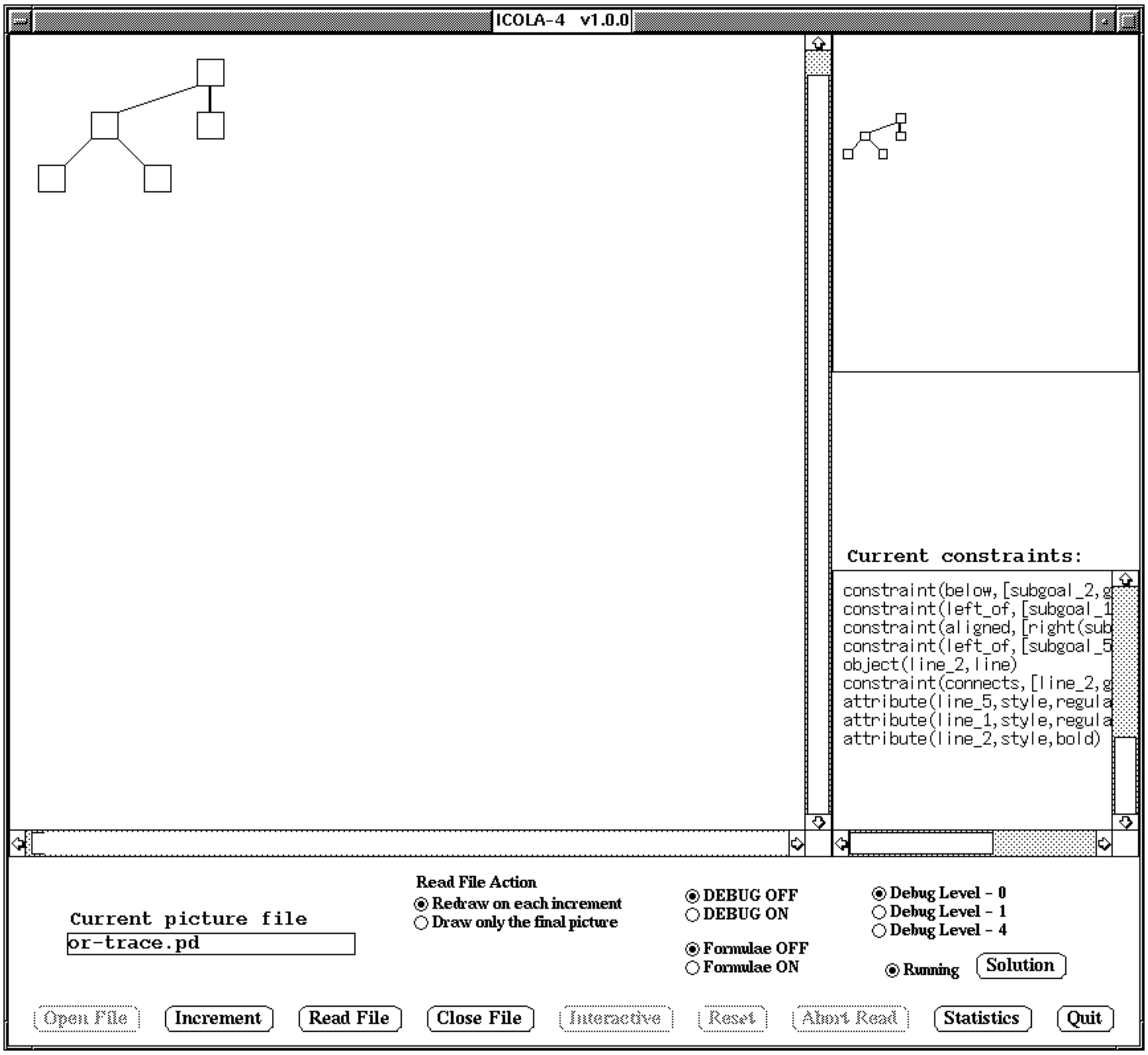


Figure C.1: The user-interface of ICOLA-4, version 1.0.0.