The SkyBlue Constraint Solver

Michael Sannella

Technical Report 92-07-02
Dept. of Computer Science and Engineering
University of Washington
February 1993

Abstract

A constraint describes a relationship that should be maintained, for example that the equality $A + B = C$ holds between three variables, that a set of displayed objects are aligned, or that the elements in a data structure are consistent with a graphic display of this structure. Constraint solvers have been successfully applied to problems in computer graphics including geometric design and user interface construction. This paper presents the SkyBlue constraint solver, an efficient incremental algorithm that uses local propagation to maintain sets of required and preferential constraints. SkyBlue is a successor to the DeltaBlue algorithm, which was used as the constraint solver in the ThingLab II user interface development environment. DeltaBlue has two limitations: cycles of constraints are prohibited, and the procedures used to satisfy a constraint can only have a single output. SkyBlue relaxes these restrictions, allowing cycles of constraints to be constructed (although SkyBlue may not be able to satisfy all of the constraints in a cycle) and supporting multi-output methods. The SkyBlue algorithm has been incorporated into Multi-Garnet, an extended version of the Garnet user interface development system that supports multi-way constraints. Multi-Garnet has been used to build several user interfaces exploiting the features of SkyBlue that would have been difficult to build within Garnet. This paper describes the basic SkyBlue algorithm and outlines several techniques that significantly improve its performance for large constraint graphs. Performance measurements are presented demonstrating that SkyBlue is efficient enough to use in interactive user interfaces.

Keywords: constraints, constraint hierarchies, local propagation, user interfaces, interactive techniques.

Contents

1 Introduction

User interface toolkits can use constraint solvers to maintain consistency between application data and a display of that data, to maintain consistency among multiple views of data, and to maintain layout relationships among graphical objects. By giving the system responsibility for maintaining the various relationships in a user interface, the programmer is freed from the tedious and error-prone task of maintaining these relationships by hand, making it easier to develop and maintain complex graphical user interfaces. Many user interface development systems have provided integrated constraint solvers, including GROW [2], Garnet [14], Rendezvous [9], and ThingLab II [13]. References [3, 6] contain additional references to constraint-based systems.

One important class of constraint solvers accepts a set of mathematical equations between variables, and uses symbolic or numerical techniques to find variable values that satisfy the equations. These solvers have been used in many computer graphics applications, including surface modeling tools [20] and constraint-based graphic editors [8]. However, mathematical constraint solvers are limited to problems where the constraints can be expressed as mathematical equations.

At the other end of the complexity spectrum, many user interface systems include simple facilities for maintaining the connection between the user interface and the application program. For example, a user interface system may allow procedures to be designated as callback procedures, which are called when particular events occur. These can be used to update the application data structures when the user manipulates the interface. Systems with such facilities include GROW [2], Interviews [12], and the Smalltalk Model-View-Controller [11]. These facilities can be difficult to use if there are multiple callbacks that access the same data structures: the programmer may need to understand the internal details of the consistency mechanism to prevent undesirable interactions. This undermines a major advantage of constraint solvers, namely that they provide a declarative way to state relationships that should be maintained. Another limitation of many simple consistency mechanisms is that they only allow constraints to be satisfied in one direction. User interface applications often require information to flow in multiple directions. For example, a user interface may include multiple views of the same data structure that must be kept consistent as edits are made within any view.

The SkyBlue algorithm lies between these two extremes in terms of complexity and expressiveness. General constraints between variables (containing arbitrary data, not just numbers) are represented by short procedures (methods) that satisfy the constraints. A constraint may have multiple methods that allow it to be satisfied in multiple directions. When variable values are changed, local propagation is used to quickly resatisfy the constraints. SkyBlue *incrementally* resatisfies the set of constraints as new constraints are added and existing constraints are removed, which is particularly useful for applications where the constraint set may change during user interactions. For example, a drawing program based on SkyBlue could allow the user to add geometric constraints between graphic objects, and resatisfy these constraints immediately as the user moves elements of the drawing.

SkyBlue has several features that distinguish it from simple local propagation solvers, and that expand the range of user interface problems where it is applicable. First, SkyBlue uses strengths associated with constraints to control which solution is produced if all of the constraints cannot be satisfied, or if there is more than one possible solution. Second, SkyBlue allows cycles of constraints as well as constraint solving procedures with multiple outputs. Despite its additional power, SkyBlue is competitive in performance with simpler constraint maintenance systems. It has been integrated

into a user interface toolkit (Multi-Garnet) that has been used to develop interactive applications (Section 9.4 presents performance figures comparing Multi-Garnet to Garnet for one application).

1.1 Local Propagation

One of the simplest and most general ways to solve constraints is by *local propagation*. For this technique each constraint is represented by a set of *methods*, procedures that access some of the constraint's variables and calculate values for the remaining variables that satisfy the constraint. Once some variable values are known, a local propagation solver can execute methods to calculate values for other variables, then it can execute methods that use these variables to calculate values for further variables, and so forth until all of the constraints have been satisfied. For example, given the constraints $A + B = C$ and $C + D = E$, if the values of A, B, and E are known, the two constraints can be satisfied by executing the methods $C \leftarrow A + B$ and $D \leftarrow E - C$ in this order. A variety of local propagation algorithms have been developed that use different techniques to choose which methods should be executed [17, 18].

Local propagation solvers cannot solve all possible sets of constraints, such as sets of simultaneous equations. However, local propagation solvers have the advantage that they are very general, since a method can perform an arbitrary computation. In particular, such solvers can handle non-numeric constraints. For example, in a user interface system one might want to maintain the relationship between a string naming a font, and an object representing the font. One method for this constraint could scan file directories and read font definition files while creating a new font object that corresponds to a given font name string. In the other direction, another method could extract a field from the font object to update the font name string.

Local propagation also allows incrementally resatisfying a set of constraints when some of the variable values are changed. This can be very efficient if only a few of the variables need to be updated. For example, if the constraints $A + B = C$ and $C + D = E$ are satisfied by executing $C \leftarrow A + B$ and $D \leftarrow E - C$, and then the value of E is changed, only the second method needs to be evaluated again.

1.2 Constraint Hierarchies and DeltaBlue

An important issue when using constraint solvers in user interfaces is the behavior of the solver when the set of constraints is overconstrained (i.e., there is no solution that satisfies all of the constraints). It is not acceptable for the solver to signal an error and halt while controlling a user interface. A related problem is how to deal with underconstrained systems (i.e., there are multiple solutions). For example, given the constraint $A + B = C$, if the value of C is changed the constraint could be resatisfied by changing the value of A or B (or both). In a user interface, different alternatives may correspond to different visible behaviors of the interface, and the programmer may want to control which behavior is chosen. The *constraint hierarchy* theory presented in [3] provides a way to specify declaratively how these situations should be handled. A constraint hierarchy is a set of constraints, each labeled with a *strength*, indicating how important it is to satisfy each constraint. Given an overconstrained constraint hierarchy, a constraint solver may leave weaker constraints unsatisfied in order to satisfy stronger constraints. If a hierarchy is underconstrained, the user can add weak

constraints to control which solution is chosen.

Another important issue is efficiency. When using a constraint solver in a user interface, user interactions may cause variables to be changed and constraints to be added and removed, and the solver must quickly resatisfy the constraints. To support interactive performance in this situation, the constraint solver must use *incremental* algorithms that produce new solutions without examining all of the constraints in the network.

The DeltaBlue algorithm, an incremental algorithm for maintaining constraint hierarchies using local propagation, was developed to address these issues [6, 13, 16]. The ThingLab II user interface development environment was based on DeltaBlue, demonstrating its feasibility for constructing user interfaces [13]. However, DeltaBlue has two significant limitations: cycles in the graph of constraints and variables are prohibited (if a cycle is found, an error is signaled and the cycle is broken by removing a constraint) and constraint methods can only have one output variable.[1]

It is not always possible to solve cycles of constraints using local propagation. However, there are several reasons why it is desirable to allow constructing such constraint graphs. First, when constructing a constraint graph, cycles may be introduced by mistake. It may be appropriate to generate a warning in this situation, but the constraint solver should try to solve the rest of the constraints even if the cycle cannot be solved. Second, in some situations it may be possible to solve the subgraph containing the cycle by calling a more powerful solver.

DeltaBlue's requirement that methods have only one output variable restricts the types of constraints that can be represented. There are many situations where it would be convenient to create constraint methods with multiple outputs. For example, suppose the variables X and Y represent the Cartesian coordinates of a point, and the variables ρ and θ represent the polar coordinates of this same point. To keep these two representations consistent one would like to define a constraint with a two-output method $(X, Y) \leftarrow (\rho\cos\theta, \rho\sin\theta)$, and another two-output method in the other direction: $(\rho, \theta) \leftarrow (\sqrt{X^2 + Y^2}, \arctan(Y, X))$. It would be possible to create a set of four constraints, each with one single-output method ($X \leftarrow \rho\cos\theta$, etc.), but this set would contain cycles. Multi-output methods are also useful for accessing the elements of compound data structures. For example, one could unpack a compound CartesianPoint object into two variables using a constraint with methods $(X, Y) \leftarrow (Point.X, Point.Y)$ and $Point \leftarrow CreatePoint(X, Y)$.

1.3 SkyBlue

The SkyBlue algorithm was developed to remove the limitations of DeltaBlue by supporting cycles of constraints and allowing methods to have any number of outputs. Whereas DeltaBlue would signal an error and remove constraints when a cycle was detected, SkyBlue allows cycles to be constructed. SkyBlue cannot satisfy the constraints around a cycle, but it correctly maintains the non-cyclic constraints elsewhere in the graph. Future work will extend SkyBlue to call a specialized constraint solver to solve the constraints around a cycle, and continue using local propagation to satisfy the rest of the constraints.

SkyBlue is currently being used as the constraint solver in Multi-Garnet [15], a package that ex-

[1] Reference [7] presents another algorithm that solves constraint hierarchies, with roughly the same power and limitations as DeltaBlue.

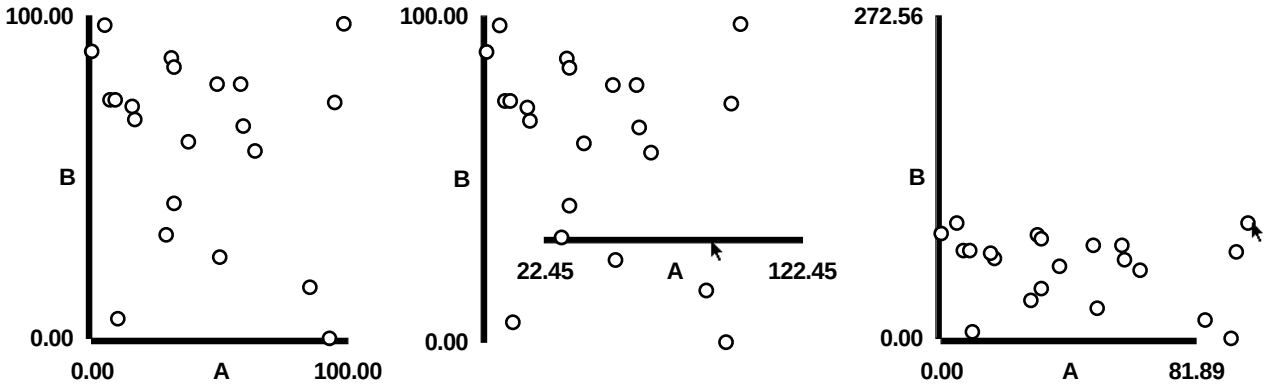


Figure 1: A scatterplot built using SkyBlue constraints: The initial scatterplot, the initial scatterplot after moving the X-axis, and the initial scatterplot after scaling the point cloud by moving a point.

tends the Garnet user interface construction system [14] with support for hierarchies of multi-way constraints. Figure 1 shows three views of a graphic user interface constructed in Multi-Garnet: a scatterplot displaying a set of points. SkyBlue constraints are used to specify relationships between the data values, the screen positions of the points, and the positions and range numbers of the X and Y-axes. As the scatterplot points and axes are moved with the mouse, SkyBlue maintains the constraints so that the graph continues to display the same data.

The scatterplot application exercises many of the features of SkyBlue. SkyBlue resatisfies the constraints quickly enough to allow continuous interaction (Section 9.4 gives some performance figures). Weak constraints are added during different interactions to specify variables that should not be changed during the interaction. Multi-way constraints allow any of the scatterplot points to be selected and moved, which changes the positions of the other points and reshapes or moves the point cloud. Finally, the scatterplot uses constraints with multi-output methods, such as a constraint with three two-output methods that maintains the relationship between the X-coordinates of the ends of the X-axis, the range values at the ends of the axis, and the scale and offset variables used to position points relative to the axis. It would be difficult to build this application in Garnet (which only supports one-way single-output constraints) without maintaining some of the relationships by mechanisms other than the constraint solver.

Multi-Garnet and SkyBlue implementations are available (contact the author for more information). SkyBlue is also currently being used as the constraint solver in an implementation of the Kaleidoscope language [5] and as an equation manipulation tool in the Pika simulation system [1].

2 Method Graphs

A SkyBlue constraint is represented by one or more *methods*. Each method is a procedure that reads the values of a subset of the constraint's variables (the method's *input variables*) and calculates values for the remaining variables (the method's *output variables*) that satisfy the constraint. For example, the constraint $A + B = C$ could be represented by three methods: $C \leftarrow A + B$, $A \leftarrow C - B$, and $B \leftarrow C - A$. If the value of A or B were changed, SkyBlue could maintain the constraint by executing

$C \leftarrow A + B$ to calculate a new value for C.

The meaning of a SkyBlue constraint is specified entirely by its methods. By definition a constraint is satisfied from the moment one of its methods is executed until one of its variables is changed. SkyBlue does not check that the methods of a constraint are defined consistently to implement a particular relation. Method procedures should be defined as functions without side-effects from the input variables to the output variables, so that SkyBlue may execute them at arbitrary times to satisfy the constraints.[2]

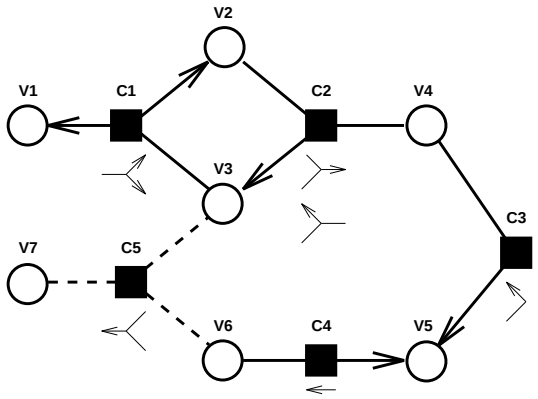


Figure 2: A method graph with an unenforced constraint ($C5$), a method conflict (at $V5$), and a directed cycle (between $C1$ and $C2$).

To satisfy a set of constraints, SkyBlue chooses one method to execute from each constraint, known as the *selected method* of the constraint. The set of constraints and variables form an undirected *constraint graph* with edges between each constraint and its variables. The constraint graph, together with the selected methods, form a directed *method graph*. In this paper, method graphs are drawn with circles representing variables and squares representing constraints (Figure 2). Lines are drawn between each constraint and its variables. If a constraint has a selected method, arrows indicate the outputs of the selected method. If a constraint has no selected method, it is linked to its variables with dashed lines. Small diagrams beneath each constraint square indicate the unselected methods for the constraint (if any). These diagrams are particularly useful when a constraint doesn't have methods in all possible directions or has multi-output methods (such as $C1$).

The following terminology will be used in this paper. If a constraint has a selected method in the current method graph the constraint is *enforced*, otherwise it is *unenforced*. Assigning a method as the selected method of a constraint is known as *enforcing* the constraint. Assigning no method as the selected method of a constraint is known as *revoking* the constraint. A variable that is an output of a constraint's selected method is *determined* by that constraint. A variable that is not an output of any selected method is *undetermined*. Following the selected method's output arrows leads to *downstream* variables and constraints. Following the arrows in the reverse direction leads to *upstream* variables and constraints.

If a method graph contains two or more selected methods that output to the same variable, this is called a *method conflict*. In Figure 2, there is a method conflict between the selected methods of

[2]Sometimes, it may be useful to define SkyBlue methods that are not functions or that have side-effects. See Section 7 for more information.

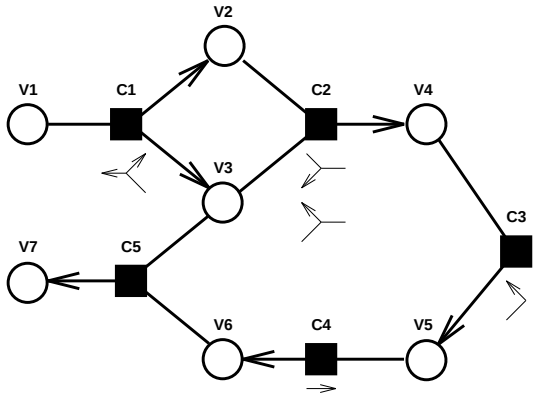


Figure 3: A method graph where all of the constraints can be satisfied.

$C3$ and $C4$. SkyBlue prohibits method conflicts because they prevent satisfying both constraints simultaneously. If we satisfy $C3$ by executing its selected method (setting $V5$), and then satisfy $C4$ by executing its selected method (again setting $V5$), then $C3$ might no longer be satisfied. If a method graph has no method conflicts and no directed cycles, then it can be used to satisfy the enforced constraints by executing the selected methods so any determined variable is set before it is read (i.e., executing the methods in topological order). For example, Figure 3 shows a method graph for the same constraints where all of the constraints can be satisfied by executing the selected methods for $C1$, $C2$, $C3$, $C4$, and $C5$, in this order. Note that the method graph specifies how to satisfy the enforced constraints, regardless of the particular values of the variables.

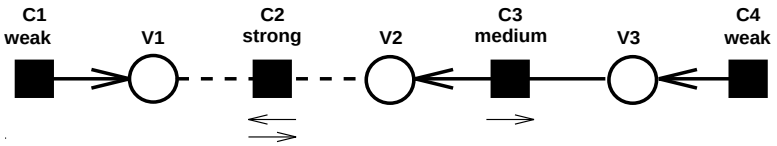
If a method graph contains directed cycles, such as the one between $C1$ and $C2$ in Figure 2, it is not possible to find a topological sort of the selected methods. SkyBlue may produce method graphs with cycles. Section 4.5 discusses how SkyBlue executes methods and handles cycles of methods.

3 Constraint Hierarchies

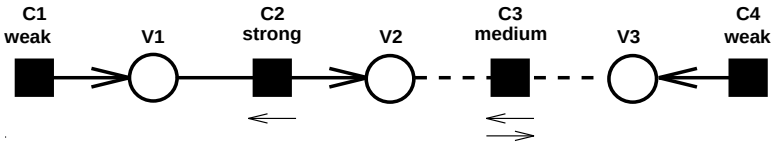
If there is no conflict-free method graph that enforces every constraint in a constraint graph, then the solver has to choose which constraints to enforce. Alternatively, if there are multiple possible method graphs for a given constraint graph, the solver will have to choose between them. The programmer can influence these choices by organizing the constraints into a constraint hierarchy.

A constraint hierarchy is a set of constraints where each constraint is labeled with one of an ordered list of *strengths*. In this paper, strengths will be written using the symbolic names `required`, `strong`, `medium`, `weak`, and `weakest`, in order from strongest to weakest. When solving a constraint hierarchy it is permitted to leave a weaker constraint unsatisfied, if this is necessary to satisfy a stronger constraint. If there are two or more conflicting constraints with the same strength, one is chosen to be satisfied arbitrarily.[3]

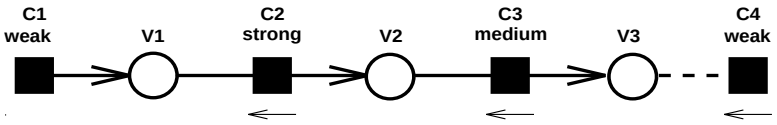
[3] Constraint hierarchies are usually defined with the additional condition that all `required` constraints must be satisfied. The SkyBlue algorithm does not enforce this condition. See Section 4.3 for more information.



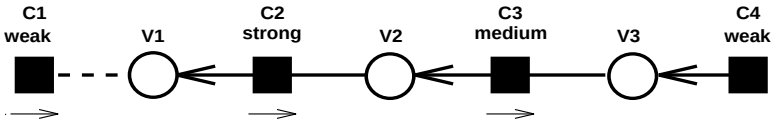
(a) A non-LGB method graph.



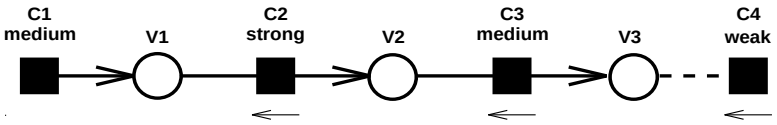
(b) Another non-LGB method graph.



(c) An LGB method graph.



(d) Another LGB method graph for the same constraints.



(e) This is the only LGB method graph if $C1$ has `medium` strength.

Figure 4: LGB and non-LGB Method Graphs.

The SkyBlue solver uses the constraint strengths to construct *locally-graph-better* (or *LGB*) method graphs [13]. A method graph is LGB if there are no method conflicts and there are no unenforced constraints that could be enforced by revoking one or more weaker constraints (and possibly changing the selected methods for other enforced constraints with the same or stronger strength).[4] For example, consider the method graph in Figure 4a. This graph is not LGB because the `strong` constraint $C2$ could be enforced by choosing the method that outputs to $V2$ and revoking the `medium` constraint $C3$, producing Figure 4b. Actually, this method graph is not LGB either since $C3$ could be enforced by revoking $C4$, producing Figure 4c. This method graph is LGB since there are no unenforced constraints that could be enforced by revoking a weaker constraint.

There may be multiple LGB method graphs for a given constraint graph. Figure 4d shows another LGB method graph which is neither better nor worse than Figure 4c. Given these constraints, SkyBlue would construct one of these two method graphs arbitrarily. The constraint strengths

[4] Reference [13] defines "locally-graph-better" such that directed cycles are prohibited. As used in this paper, LGB method graphs may include cycles.

could be modified to favor one alternative over the other. For example, if the strength of $C1$ was changed to `medium`, the only LGB method graph would be the one in Figure 4e. One way for the programmer to control the method graphs constructed is to add *stay* constraints that have a single null method with no inputs and a single output. A stay constraint specifies that its output variable should not be changed. A similar type of constraint is a *set* constraint, which sets its output to a constant value. Set constraints can be used to inject new variable values into a constraint graph. In Figure 4, $C1$ and $C4$ are stay or set constraints.

Reference [3] presents several different ways to define which variable values "best" satisfy a constraint hierarchy. The concept of read-only variables extends this theory to constraints that may not be able to set some of their variables, such as SkyBlue constraints without methods in all possible directions. For many constraint graphs, LGB method graphs compute "locally-predicate-better" solutions to the constraint hierarchy as defined in the theory. Reference [13] examines the relation between LGB method graphs and locally-predicate-better solutions.

4 The SkyBlue Algorithm

The SkyBlue constraint solver maintains the constraints in a constraint graph by constructing an LGB method graph and executing the selected methods in the method graph to satisfy the enforced constraints. Initially, the constraint graph and the corresponding LGB method graph are both empty. SkyBlue is invoked by calling two procedures, `add-constraint` to add a constraint to the constraint graph, and `remove-constraint` to remove a constraint. As constraints are added and removed, SkyBlue incrementally updates the LGB method graph and executes methods to resatisfy the enforced constraints.

The complete SkyBlue algorithm is rather complex so the explanation is divided into several sections. Sections 4.1 and 4.2 present an overview of `add-constraint` and `remove-constraint`. Section 4.4 describes how a constraint is enforced by constructing a method vine, the basic operation of the SkyBlue algorithm. Section 4.5 explains how SkyBlue executes the selected methods of the enforced constraints and how cycles are handled. The algorithm described in these sections produces correct results, but its performance suffers as the constraint graph becomes very large. Section 5 presents several techniques used in the complete algorithm that significantly improve the efficiency of SkyBlue for large constraint graphs.

These sections are meant to present a high-level view of SkyBlue, so they do not give all of the low-level details of the SkyBlue algorithm. More detailed information on SkyBlue is included along with complete pseudocode for the algorithm in Section 6.

4.1 Adding Constraints

When a new constraint is added to the constraint graph it may be possible to alter the method graph to enforce it by selecting a method for the constraint, switching the selected methods of enforced constraints with the same or stronger strength, and possibly revoking one or more weaker constraints. This process is known as constructing a *method vine* or *mvine* (this is described in Section 4.4). If constraints are revoked during this process, this may enable other unenforced constraints to be

enforced by constructing additional mvines. `add-constraint` adds a new constraint `cn` to the constraint graph by performing the following steps:

1. Add `cn` to the constraint graph (unenforced) and try to enforce `cn` by constructing an mvine. If it is not possible to construct such an mvine, leave `cn` enenforced and return from `add-constraint`. In this case, the method graph is unchanged (it is still LGB).
2. Repeatedly try to enforce all of the unenforced constraints in the constraint graph by constructing mvines until none of the remaining unenforced constraints can be enforced. Note that each time an unenforced constraint is successfully enforced, one or more weaker constraints may be revoked. These newly-unenforced constraints must be added to the set of unenforced constraints.
3. Execute all of the selected methods in the method graph to satisfy the constraints (see Section 4.5).

The second step must terminate because there are a finite number of constraints. Each time an unenforced constraint is enforced, one or more weaker constraints may be added to the set of unenforced constraints. These additional constraints may be enforcible, adding still weaker constraints to the set of unenforced constraints, but this process cannot go on indefinitely. Eventually the process will stop with a set of unenforcible constraints. When the second step terminates the method graph must be LGB. If it was not LGB then there would be an unenforced constraint that could be enforced by setting its selected method, switching the selected methods of constraints with the same or stronger strengths, and possibly revoking one or more weaker constraints. This is precisely the case when it is possible to construct an mvine. If no more mvines can be constructed then the method graph must be LGB.

As an example, suppose that `add-constraint` has just added $C2$ to the constraint graph and the current method graph is shown in Figure 4a. One way that an mvine could be constructed is by enforcing $C2$ with the method that outputs to $V2$ and revoking $C3$ (Figure 4b). Given this method graph, the second step would try constructing an mvine to enforce $C3$, possibly by revoking $C4$ (Figure 4c). At this point it is not possible to construct an mvine to enforce $C4$ so the second step terminates. This method graph is LGB. Alternatively, if the first mvine had been constructed by revoking $C1$ then the LGB method graph of Figure 4d would have been produced immediately and the second step would not have been able to enforce $C1$.

4.2 Removing Constraints

`remove-constraint` is very similar to `add-constraint`. When an enforced constraint is removed this may allow some unenforced constraints to be enforced, which leads to the same process of repeatedly constructing mvines as in `add-constraint`. `remove-constraint` removes a constraint `cn` from the constraint graph by performing the following steps:

1. If `cn` is currently unenforced, simply remove it from the constraint graph and return from `remove-constraint`. Removing an unenforced constraint cannot make any other constraints enforcible so the method graph is still LGB.

2. Repeatedly try enforcing all of the unenforced constraints in the constraint graph by constructing mvines (adding revoked constraints to the set of unenforced constraints) until none of the unenforced constraints can be enforced. As in `add-constraint` this step eventually terminates with an LGB method graph.

3. Execute all of the selected methods in the method graph to satisfy the constraints (see Section 4.5).

4.3 `required` Constraints

The strongest strength, `required`, is typically used for constraints that must *always* be enforced. This raises the issue of how to handle two `required` constraints that conflict (i.e., both constraints cannot be enforced without causing a method conflict). Some solvers handle this situation by automatically removing one of the conflicting `required` constraints. This is not a very satisfactory solution, because it removes control from the application over which constraints are in the constraint graph.

SkyBlue addresses this issue by treating the `required` strength the same as any other strength. The SkyBlue algorithm does *not* guarantee that all `required` constraints in the constraint graph are enforced. If two conflicting `required` constraints are added, one will be enforced, and the other will be left unsatisfied. Later, if the enforced conflicting constraint is removed, then the unsatisfied `required` constraint will be enforced.

If an application using SkyBlue requires that all `required` constraints must always be satisfied, then this application will have to do a little extra work to detect and handle conflicts between `required` constraints. Specifically, after `add-constraint` is called to add a `required` constraint, the application can detect whether the newly-added constraint is enforced, and call `remove-constraint` to remove this constraint if it is not enforced. This will guarantee that all `required` constraints in the constraint graph are enforced. Note that SkyBlue will never revoke a `required` constraint once it is enforced, except when the constraint is removed by an explicit call to `remove-constraint`. This is true because the only other time that constraints are revoked is when mvines are constructed. Since `required` is the strongest strength, and constructing an mvine can only revoke constraints weaker than the root constraint of the mvine, then `required` constraints cannot be revoked while constructing an mvine.

4.4 Constructing Method Vines

The SkyBlue algorithm is based on repeatedly trying to enforce an unenforced constraint by changing the selected methods of constraints with the same or stronger strength and possibly revoking one or more constraints with weaker strengths. There are many ways this could be implemented, including trying all possible assignments of selected methods without method conflicts. The technique used in SkyBlue, known as constructing a method vine (or mvine), uses a backtracking depth-first search.

An mvine is constructed by selecting a method for the constraint we are trying to enforce (the root constraint) and adding it to the mvine. If this method has a method conflict with the selected methods of other enforced constraints, we select new methods for these other constraints and add

them to the mvine. These new selected methods may conflict with yet other selected methods, and so on. This process extends through the method graph, which builds a "vine" of newly-chosen selected methods growing from the root constraint. This growth process may terminate in the following ways:

1. If a newly-selected method in the mvine outputs to variables that are not currently determined by any constraint, then this branch of the mvine is not extended any further.
2. If a newly-selected method in the mvine conflicts with a selected method whose constraint is weaker than the root constraint, then the weaker constraint is revoked, rather than attempting to find an alternative selected method for it. As a result, all of the methods in the mvine will belong to constraints with equal or stronger strengths than the root constraint.
3. If an alternative selected method is chosen for a constraint and there is a method conflict with another selected method in the mvine, then we cannot add this method to the mvine and must try another method. If all of the methods of this constraint conflict with other selected methods in the mvine, then the mvine construction process backtracks: previously-selected methods are removed from the mvine and the mvine is extended using other selected methods for these constraints. If no method can be chosen for the root constraint that allows a complete conflict-free mvine to be constructed, then the root constraint cannot be enforced.

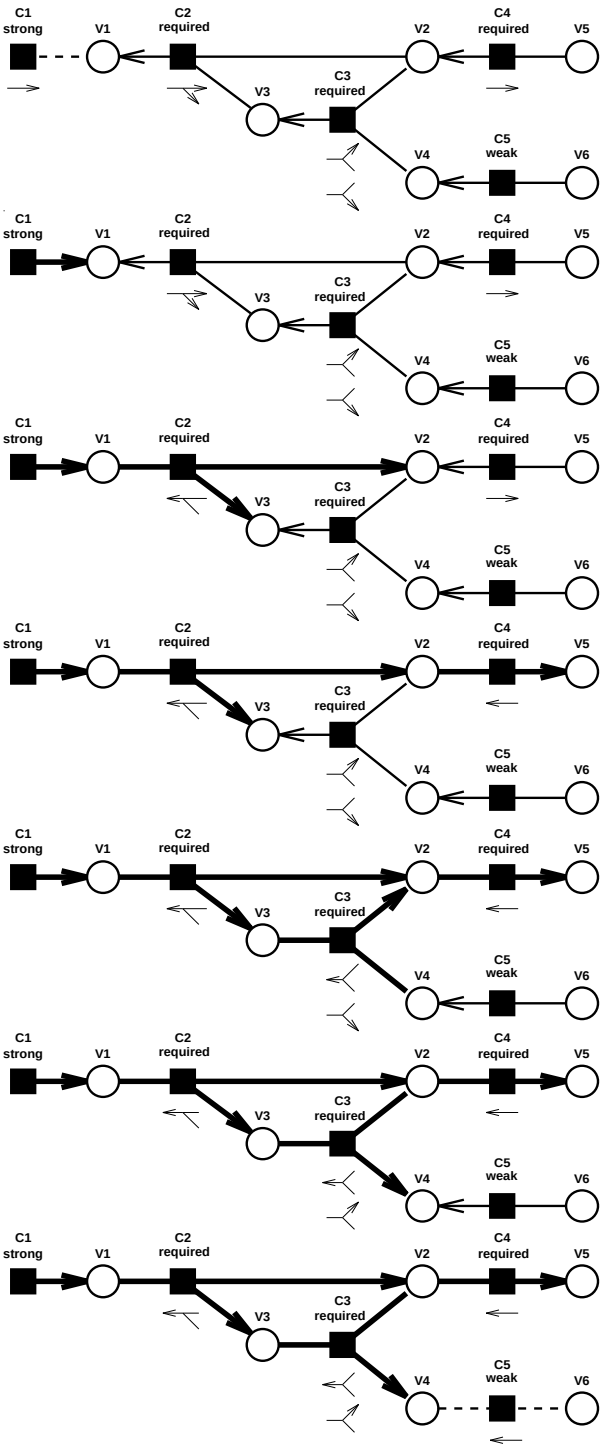
Figure 5 presents an example demonstrating the process of constructing an mvine.

A complete mvine is a connected subgraph of the method graph. If all of the constraint methods in the mvine have a single output, then an mvine will have the structure of a single stalk leading from the root constraint through a series of other constraints with changed selected methods. If there is a method with multiple outputs in the mvine, the mvine will divide into multiple branches with one branch for each output. The different branches cannot be extended independently since methods in one branch prevent choosing methods in another branch that output to the same variables. The backtracking search must take this into account by trying all possible combinations of selected methods for the constraints in the different branches (see Section 6.3 for pseudocode that implements this search).

An mvine is not necessarily a tree: separate branches may merge and it may contain directed cycles. The mvine construction process ensures that there are no method conflicts in an mvine, but it is possible for one mvine method to output to one of the *inputs* of another mvine method.

4.5 Executing Methods

Given a method graph without method conflicts or directed cycles, the enforced constraints can be satisfied by executing the selected methods in order so any determined variable is set before it is read (i.e., executing the methods in topological order). If there is a directed cycle, it is not possible to find a topological sort of the selected methods. In this case, SkyBlue sorts and executes only the selected methods upstream of cycles. Any methods in a cycle or downstream of a cycle are not executed and their output variables are marked to specify that their values do not necessarily satisfy the enforced constraints. If a cycle is later broken the methods in the cycle and downstream



Suppose we start with this method graph, and we want to enforce the `strong` constraint $C1$ by building an mvine.

First, $C1$'s selected method is set to its only method so it determines $V1$.

This causes a method conflict with $C2$ so we have to enforce $C2$ with its other method.

This causes method conflicts with $C3$ and $C4$. Suppose we process $C4$ first: we can simply switch its selected method so it determines $V5$. $V5$ is not determined by any other constraints so we don't have to extend this branch of the mvine.

Now, we have to process $C3$ by choosing another method. Suppose we try the method that determines $V2$. This is not permitted because it causes a method conflict with $C2$, which is already in the mvine.

Therefore, we have to backtrack and try another method for $C3$. Suppose we now try the method that determines $V4$ (causing a method conflict with $C5$).

Now we need to handle $C5$. Because it is weaker than $C1$ we don't have to find an alternative method but can simply revoke it, producing this final method graph.

Figure 5: Constructing an mvine. Methods in the mvine are drawn with thicker lines.

are executed correctly. SkyBlue will be extended in the future to call more powerful solvers to find values satisfying a cycle of constraints and then propagate these values downstream.

For more detailed information, see Section 6.6 for pseudocode used in SkyBlue to execute methods.

One particular optimization SkyBlue does not currently implement is detecting when a method calculates an output value that is the same as the current value. In this case, it may be possible to avoid executing downstream methods. Algorithms that implement this optimization are described in references [10, 19].

5 Performance Techniques

The SkyBlue algorithm as presented in Section 4 works correctly, but its performance suffers as the constraint graph becomes very large. This happens for two reasons: First, larger constraint graphs may contain greater numbers of unenforced constraints that SkyBlue has to try enforcing by constructing mvines. Second, each attempt to construct an mvine may involve searching through more enforced constraints. The following subsections describe techniques used in SkyBlue to improve its performance with larger constraint graphs. Section 9.2 presents performance figures demonstrating that these techniques improve SkyBlue's performance.

These techniques use two strategies. One strategy is to avoid mvine construction whenever possible. In many cases it is possible to determine that an unenforced constraint cannot be enforced without actually trying to construct an mvine. Rather than considering the set of all of the unenforced constraints in the constraint graph, a much smaller set of unenforced constraints can be collected and processed until none of them can be enforced. Sections 5.1 and 5.2 present techniques based on this strategy.

The second strategy is to reduce backtracking while constructing mvines by rejecting some of the alternative selected methods without trying to construct the rest of the mvine. Section 5.3 discusses a technique based on this strategy.

5.1 The Collection Strength Technique

When SkyBlue is started, the initial empty method graph is LGB. Every call to `add-constraint` or `remove-constraint` leaves an LGB method graph. Therefore, the current method graph must be LGB whenever `add-constraint` or `remove-constraint` is called. This fact can be used to avoid collecting and trying to enforce some of the unenforced constraints.

Whenever `add-constraint` is called to add a constraint `cn` to the constraint graph, it is impossible to enforce any unenforced constraints with the same or stronger strength than `cn`, other than `cn` itself. If it was possible to enforce any such constraint after `cn` was added, then it would have been possible to enforce it before `cn` was added and the previous method graph would not have been LGB.

Whenever `remove-constraint` is called to remove an enforced constraint `cn`, it is impossible to

enforce any constraints that are stronger than `cn`. If it was possible to enforce any stronger constraint after `cn` was removed, then it would have been possible to enforce it before `cn` was removed and the previous method graph would not have been LGB. Note that unlike `add-constraint`, removing a constraint may allow unenforced constraints with the same strength to be enforced, as well as weaker ones.

5.2 The Local Collection Technique

If the method graph is LGB and a constraint is added or removed from the constraint graph, any unenforced constraints in a subgraph unconnected to the added or removed constraint clearly cannot be enforced. It is possible to be more selective: Whenever `add-constraint` is called to add a constraint `cn` and an mvine is successfully constructed to enforce it, it is sufficient to collect unenforced constraints that constrain variables downstream in the method graph from all of the "redirected variables" whose determining constraint has changed. Whenever `remove-constraint` is called to remove a constraint `cn`, it is sufficient to collect unenforced constraints that constrain variables downstream from the variables previously determined by `cn`.

Whenever SkyBlue successfully constructs an mvine, additional unenforced constraints can be added to the set of collected unenforced constraints by scanning downstream from the redirected variables. As each of these constraints is processed it can be removed from the set. When the set is empty there are no more unenforced constraints that can be enforced.

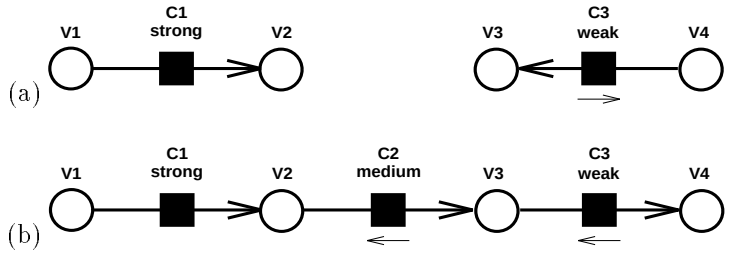


Figure 6: Method graphs before and after adding $C2$.

A similar technique can be used to reduce the number of methods executed. Rather than executing the selected methods of all enforced constraints in the constraint graph, it is only necessary to collect and execute the selected methods of newly-added constraints, and methods downstream of redirected variables. For example, suppose a constraint graph is satisfied using the method graph in Figure 6a. If $C2$ is added, producing the LGB method graph in Figure 6b, then only the selected methods for $C2$ and $C3$ would have to be executed (in that order). The selected method for $C1$ does not have to be re-executed.

5.3 Walkabout Strengths

An mvine is constructed by repeatedly choosing a new selected method for a constraint and then trying to extend the mvine from the outputs of the new selected method. It will be possible to complete the mvine below the outputs only if the mvine eventually encounters undetermined variables

or constraints weaker than the root constraint, and there are no method conflicts between different branches of the mvine. If SkyBlue could predict that one of these conditions was untrue then the selected method could be rejected immediately without trying to extend the mvine.

The DeltaBlue algorithm predicts whether a constraint can be enforced by using the concept of *walkabout strengths* [6]. A variable's walkabout strength is the strength of the weakest constraint that would have to be revoked to allow that variable to be determined by a new constraint. This could be the strength of the constraint that currently determines the variable or the strength of a weaker constraint elsewhere in the method graph that could be revoked after switching the selected methods of intermediate constraints. If the variable is not currently determined by any constraint then the walkabout strength is defined as `weakest`, which is a special strength weaker than any constraint. A variable will also have a walkabout strength of `weakest` if it can be left undetermined by switching selected methods without revoking any constraints.[5]

One important property of DeltaBlue's walkabout strengths is that they can be calculated using local information. The walkabout strength of a variable determined by a constraint can be calculated from the constraint's strength, its methods, and the walkabout strengths of the rest of the constraint's variables. If the method graph has no cycles (required for DeltaBlue), all of the variable walkabout strengths can be updated by setting the walkabout strengths of all undetermined variables to `weakest` and processing each enforced constraint in topological order to set the walkabout strengths of the determined variables.

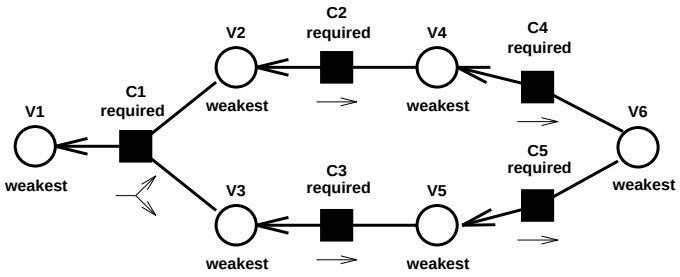


Figure 7: Method graph with a possible conflict.

There is a problem with using walkabout strengths in SkyBlue because methods may have multiple outputs. Consider the method graph of Figure 7. DeltaBlue would correctly calculate the walkabout strengths of $V2$–$V6$ to be `weakest`. But what about $V1$? The walkabout strengths of $V2$ and $V3$ imply that $V1$ should have a walkabout strength of `weakest`, since the alternative (multi-output) method can be chosen that outputs to $V2$ and $V3$, which both have `weakest` walkabout strengths. However, it is not possible for a method to set *both* $V2$ and $V3$ simultaneously, without revoking one of the `required` constraints. Simply switching methods would lead to a method conflict with both $C4$ and $C5$ determining $V6$. However, this cannot be detected without exploring the graph, which would remove one of the benefits of walkabout strengths (i.e., they can be calculated using local information).

In SkyBlue, the definition of walkabout strength is modified. A variable's walkabout strength is defined as a *lower bound* on the strength of the weakest constraint in the current method graph that would need to be revoked to allow the variable to be determined by a new constraint. SkyBlue

[5] Another interpretation of the `weakest` strength is that each variable has an implicit stay constraint with a strength of `weakest`, which specifies that the variable value doesn't change unless a stronger constraint determines it.

uses the modified definition of walkabout strengths to reject methods when constructing an mvine: if any of the outputs of a method have walkabout strengths equal to or stronger than the root constraint, then it is not possible to complete the mvine using this method. Walkabout strengths cannot eliminate all of the backtracking during mvine construction but it can reduce it considerably.

Whenever SkyBlue successfully constructs an mvine it modifies the method graph, so the walkabout strengths must be updated to correspond to the new method graph. This could be done by processing all of the enforced constraints in the constraint graph (in topological order) and recalculating the walkabout strengths of the determined variables. It is possible to apply the technique from Section 5.2 in this situation by processing only the enforced constraints downstream of the redetermined variables. See Section 6.4 for pseudocode implementing this.

```
compute-walkabout-strengths(cn: Constraint)
   current-outputs := cn.selected-method.outputs
   For all variables out-var in current-outputs do
      min-strength := cn.strength
      For all methods mt in cn.methods do
         If not(member(out-var, mt.outputs)) then
             ;; mt doesn't output to out-var, so it
             ;; is a possible alternative method.
            max-strength := max-out(mt, current-outputs)
            If weaker(max-strength, min-strength) then
               min-strength := max-strength
      out-var.walk-strength := min-strength

 ;; max-out returns the strongest walkabout strength among
 ;; mt's outputs, ignoring any variables in current-outputs.
max-out(mt: Method, current-outputs: List of Variables): Strength
   max-strength := *weakest-strength*
   For all variables var in mt.outputs do
      If not(member(var, current-outputs)) then
         If weaker(max-strength, var.walk-strength) then
            max-strength := var.walk-strength
   Return max-strength
```

Figure 8: Pseudocode for recomputing walkabout strengths.

SkyBlue uses the modified definition of walkabout strengths to simplify the processing of cycles. If there are directed cycles in the method graph, it is not possible to calculate the walkabout strengths for all inputs of the selected methods in the cycle before processing their constraints. This could be handled by examining all of the constraints in the cycle, but this would require non-local computation. Instead, SkyBlue breaks the cycle by choosing a selected method in the cycle and calculating the walkabout strengths of its outputs as if all of its input variables in the cycle had walkabout strengths of `weakest`. This is guaranteed to be a correct lower bound. This simplifies the updating of walkabout strengths at the cost of increasing the search when constructing an mvine, because the walkabout strengths in a cycle and downstream may be weaker than necessary.

Figure 8 shows pseudocode for setting the walkabout strengths of the outputs of an enforced constraint's selected method, once the walkabout strengths of the method's inputs are known. For each of these output variables, `compute-walkabout-strengths` examines the alternative methods of the constraint that don't output to it (and thus could be used to enforce this constraint if another constraint determined this variable). For each of these methods the strongest walkabout strength among the method's outputs is found (by calling `max-out`). When examining the outputs of an alternative method some of the outputs may be ignored: if an alternative method output is already being determined by the constraint then no additional constraints would need to be revoked for the constraint to determine this variable.

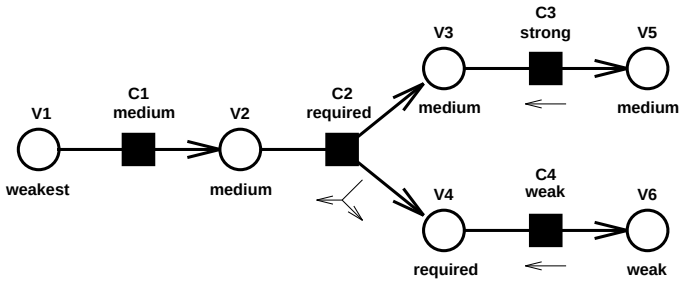


Figure 9: Method graph with an multi-output method.

The walkabout strengths in Figure 9 could be calculated by setting $V1$'s walkabout strength to `weakest` and then calling `compute-walkabout-strengths` to process $C1$, $C2$, $C3$, and $C4$ in that order. Note that $C1$ is only a one-way constraint: if it had another method that set $V1$, then the walkabout strength of $V2$ would be `weakest`. Note also that the walkabout strength of $V4$ is `required` because the unselected method for $C2$ also outputs to this variable.

6 SkyBlue Pseudocode

This section presents pseudocode for the SkyBlue algorithm, derived from the original Common Lisp implementation. It is detailed enough to be used to implement SkyBlue in a variety of languages. It has been used to develop implementations in Smalltalk and C++. Contact the Author for the availability of implementations of SkyBlue in various languages.

6.1 SkyBlue Data Structures

This section presents the data structures used in the pseudocode. Record fields are accessed using "dot" notation. For example, the `mark` field of a constraint `cn` is referenced by `cn.mark`, and set by `cn.mark :=` *value*.

6.1.1 Variables

field name	*type*	*description*
`value`	Any	the value of this variable
`constraints`	List of Constraints	all constraints that reference this variable
`determined-by`	Constraint	the constraint that determines this variable, or `nil`
`walk-strength`	Strength	the walkabout strength of this variable
`mark`	Mark	this variable's mark
`valid`	Boolean	true if this variable value is valid

One variable record is used to represent each constrained variable. In addition to its value, a variable record has a list of all the constraints in the constraint graph that refer to it (`constraints`) and a pointer to the enforced constraint that determines its value in the current method graph (`determined-by`). If no constraint determines the variable's value, the `determined-by` field is set to `nil`. The `walk-strength` field specifies the walkabout strength of the variable (as defined in Section 5.3). The variable `mark` field is used when building method vines (see Section 6.3). The `valid` field is used when executing methods to satisfy constraints. This is `false` if this variable is in a cycle or downstream of a cycle, otherwise it is `true` (see Section 6.6).

Variables are initialized at creation time as if they were determined by a virtual stay constraint with a strength of `weakest`.

```
create-variable(initial-value: Any): Variable
   var := new Variable
   var.value := initial-value
   var.constraints := {}
   var.determined-by := nil
   var.walk-strength := *weakest-strength*
   var.mark := nil
   var.valid := true
   Return var
```

Note that the `constraints` field only includes those constraints that have been added to the constraint graph. Each time a constraint is added or removed from the constraint graph, the `constraints` fields of its variables are modified.

6.1.2 Constraints

field name	*type*	*description*
`strength`	Strength	this constraint's level in the constraint hierarchy
`variables`	List of Variables	the variables constrained by this constraint
`methods`	List of Methods	the possible methods for enforcing this constraint
`selected-method`	Method	the method used to enforce this constraint, or `nil`
`mark`	Mark	this constraint's mark

A constraint record represents a constraint with the strength specified by `strength` that constrains the variables in `variables`. The `mark` field is used throughout the SkyBlue algorithm to mark

constraints while traversing the constraint graph. The `methods` field contains the alternative methods that could be executed to satisfy the constraint. The `selected-method` field records which of these methods is used to enforce the constraint in the current method graph. If the constraint is not enforced, the `selected-method` field is set to `nil`. The `enforced` function is used to test if a constraint is enforced:

```
enforced(cn: Constraint): Boolean
   If cn.selected-method ≠ nil then
      Return true
   Else
      Return false
```

Constraints are initialized at creation time by specifying their strength, variables, and methods:

```
create-constraint(str: Strength,
                  vars: List of Variables,
                  mts: List of Methods): Constraint
   cn := new Constraint
   cn.strength := str
   cn.variables := vars
   cn.methods := mts
   cn.selected-method := nil
   cn.mark := nil
   Return cn
```

6.1.3 Methods

field name	*type*	*description*
`code`	Procedure	a procedure that calculates the method outputs
`inputs`	List of Variables	the input variables of this method
`outputs`	List of Variables	the output variables of this method

A method record represents one of the possible ways to enforce a constraint. A method has an enforcement procedure (`code`) and fields containing the constrained variables that it reads (`inputs`) and writes (`outputs`).

Methods are initialized at creation time by specifying their `code` procedure, inputs, and outputs:

```
create-method(proc: Procedure,
              input-vars: List of Variables,
              output-vars: List of Variables): Methods
   mt := new Method
   mt.code := proc
   mt.inputs := input-vars
   mt.outputs := output-vars
   Return mt
```

The `inputs` and `outputs` lists must not contain any duplicates. These two fields must also partition

the variables of the constraint. This means that for any method `mt` or a constraint `cn`, all variables of the constraint must be included in the `inputs` or `outputs` lists (i.e, `mt.inputs` $\cup$ `mt.outputs` = `cn.variables`), and no variable can be in both the `inputs` or `outputs` lists (i.e., `mt.inputs` $\cap$ `mt.outputs` = $\emptyset$).

In the pseudocode below, a method is executed by calling `execute-method`. This procedure should call the method's `code` procedure to read the values of the input variables and set the output variable values such that the constraint is satisfied. The exact details of how the `code` procedure is invoked depend on the implementation language. One possibility would be to call the `code` procedure passing its method record as a single argument, and let the procedure explicitly extract the input and output variables from the method record.

6.1.4 Strengths

Strengths are data objects representing the different constraint hierarchy levels. These have been represented as Lisp atoms or integers in different implementations of SkyBlue. The implementation must provide some way for the programmer to access particular strength objects when creating constraints. The SkyBlue pseudocode references the following function and variable:

`weaker(s1: Strength, s2: Strength)` is a function that returns `true` if strength `s1` is strictly weaker than strength `s2`, `false` otherwise. Strengths must be totally ordered: for any two strengths `s1` and `s2`, either `weaker(s1,s2)` or `weaker(s2,s1)` or `s1=s2` must be true.

`*weakest-strength*` is a global variable containing the weakest strength. This strength is used as the strength of undetermined variables, which implicitly have a stay constraint with the weakest strength. This should not be used as the strength of any constraint created by the programmer.

SkyBlue assumes that the list of strengths is fixed, and does not support changing or reordering the list of strengths while maintaining the constraint graph. An implementation could allow the user to initialize the list of strengths to an arbitrary list of strength objects, before any constraints are added to the constraint graph. In this case, `*weakest-strength*` must be set to the weakest constraint strength.

In examples in this paper, the strength levels are referred to by the symbolic names `required`, `strong`, `medium`, and `weak`, and `weakest`, in order from strongest to weakest.

6.1.5 Marks

Both variable and constraint records contain a `mark` field. The `mark` field is used throughout the SkyBlue pseudocode in procedures that construct method vines, traverse the constraint graph structure, and topologically sort enforced constraints in the method graph.

The function `new-mark()` is a primitive function that returns a new and unique mark value. Current implementations of SkyBlue use monotonically increasing 32-bit integers and assume that the counter will never overflow.

In the pseudocode, `nil` is also used as a mark value to un-mark variables and constraints. Any value that would never be returned by `new-mark` could also be used.

6.1.6 Lists and Stacks

The SkyBlue pseudocode uses various types of collections of objects. A `List` is an ordered list which may contain duplicates. A `Stack` is a normal first-in-first-out stack. The order of the elements is maintained and objects may appear in the stack multiple times. The functions `push(obj: Any, st: Stack)` and `pop(st: Stack)` are used to add an element on the top of the stack, and remove (and return) the top element of the stack. Elements are added to a list using phrases such as "`Add cn to var.constraints`". If the position (at the beginning or end) of the new element in the list is not specified, then it doesn't matter.

Both lists and stacks allow iterating through all of the elements without changing the collection. The function `member(obj: Any, lst: List or Stack of Any)` returns `true` if the object is present in the list, `false` otherwise. All operations that add or remove elements to lists or stacks are destructive operations. For example, a list can be passed to a subprocedure which adds some more elements to the list, and these new elements can later be accessed by the calling procedure.

6.2 SkyBlue Entry Points: `add-constraint` and `remove-constraint`

`add-constraint` and `remove-constraint` are the entry points for SkyBlue. At any moment there is a set of constraints considered to be in the constraint graph. `add-constraint` adds a constraint to the constraint graph. `remove-constraint` removes a constraint from the constraint graph. `add-constraint` must not be called on a constraint that is already in the constraint graph, and `remove-constraint` must not be called on a constraint that is not in the constraint graph.

`add-constraint` adds the constraint `cn` to the constraint graph, updates the LGB method graph, and executes the selected methods in the method graph. `add-constraint` first initializes the fields of the constraint and updates the `constraints` fields of its variables. Then, it calls `update-method-graph` to update the LGB method graph to include the unenforced constraints in the list `unenforced-cns` (initially just `cn`), accumulating any redetermined variables and newly-added constraints in `exec-roots`. Finally, `exec-from-roots` is called to execute the selected methods downstream of these roots, to satisfy the constraints in the method graph.

It is possible that `add-constraint` will not be able to enforce the constraint `cn`, if there are stronger constraints in the graph. In this case, `update-method-graph` will not change the method graph, and `exec-roots` will be empty (and `exec-from-roots` will do nothing).

```
add-constraint(cn: Constraint)
   unenforced-cns := new empty List of Constraints
   exec-roots := new empty List of Any
    ;; initialize cn's fields, and register cn with its variables
   cn.selected-method := nil
   cn.mark := nil
   For all variables var in cn.variables do
      Add cn to var.constraints
    ;; add cn to method graph (if possible), collecting exec-roots.
   Add cn to unenforced-cns
   update-method-graph(unenforced-cns, exec-roots)
    ;; satisfy method graph constraints by executing methods.
   exec-from-roots(exec-roots)
```

`remove-constraint` removes the constraint `cn` from the constraint graph, updates the LGB method graph, and executes the selected methods in the method graph. `remove-constraint` first unregisters `cn` by removing it from the lists of constraints maintained by its variables. If `cn` is not enforced, then nothing else needs to be done. However, if `cn` is enforced, then it may be possible to enforce other constraints when it is removed. In this case, `remove-constraint` sets the `determined-by` and `walk-strength` fields of `cn`'s output variables to indicate that these variables are determined by an implicit `weakest` stay constraint, and calls `propagate-walk-strength` to propagate these new walkabout strengths downstream.

Next, `remove-constraint` calls `collect-unenforced` to collect any unenforced constraints downstream of `cn`'s old outputs that may be enforcible now that `cn` is removed. As discussed in Section 5.1, the only unenforced constraints that could possibly be enforced when `cn` is removed are those with the same strength, or weaker. The last two arguments to `collect-unenforced` specify that only constraints with these strengths should be collected.

Once the unenforced constraints are collected, `update-method-graph` is called to try to include them in the LGB method graph, accumulating any redetermined variables and newly-added constraints in `exec-roots`. Finally, `exec-from-roots` is called to execute constraint methods downstream of these roots, to satisfy the constraints in the method graph.

```
remove-constraint(cn: Constraint)
    ;; unregister cn from variables
   For all variables v in cn.variables do
      Remove cn from v.constraints
   If enforced(cn) then
      unenforced-cns := new empty List of Constraints
      exec-roots := new empty List of Any
      cn-old-outputs := cn.selected-method.outputs
       ;; unenforce cn, and undetermine cn's output vars
      cn.selected-method := nil
      For all variables var in cn-old-outputs do
         var.determined-by := nil
         var.walk-strength := *weakest-strength*
       ;; save newly-undetermined variables as possible exec-roots
      Add all variables in cn-old-outputs to exec-roots
       ;; propagate walkabout strength from undetermined vars
      propagate-walk-strength(cn-old-outputs)
       ;; collect unenforced constraints downstream of undetermined
       ;; variables with the same or weaker strength than cn
      collect-unenforced(unenforced-cns, cn-old-outputs, cn.strength, true)
       ;; update method graph to include the unenforced constraints
      update-method-graph(unenforced-cns, exec-roots)
       ;; satisfy method graph constraints by executing methods.
      exec-from-roots(exec-roots)
```

`update-method-graph` updates the current method graph to be locally-graph-better, by trying to enforce the constraints in `unenforced-cns`. Each constraint `cn` in `unenforced-cns` is processed by calling `build-mvine` (Section 6.3) to try to build an mvine that enforced `cn`. When `build-mvine` succeeds (returning `true`), it adds any *redetermined variables* (variables whose `determined-by` field has been changed) to `redetermined-vars`. In this case, we call `propagate-walk-strength` to update the walkabout strengths downstream of `cn` and the redetermined variables and call `collect-unenforced` to add any additional unenforced constraints downstream of the redetermined variables to `unenforced-cns`. Finally, `update-method-graph` updates `exec-roots` by adding any constraints that were successfully enforced, as well as any redetermined variables that are now undetermined. This loop iterates, removing constraints from `unenforced-cns` (and adding constraints to `unenforced-cns` in `collect-unenforced`) until `unenforced-cns` is empty. Section 4.1 discusses why this process must terminate. Note: As a heuristic, we always try removing and enforcing the strongest unenforced constraint in `unenforced-cns`.

Section 5.1 discusses the "collection strength" performance technique, which justifies ignoring any unenforced constraints with the same strength or stronger than a newly-added constraint, because they cannot possibly be enforced. `update-method-graph` uses an extended version of this technique: when collecting unenforced constraints after an mvine is successfully constructed, it is only necessary to collect unenforced downstream constraints that are weaker than the root constraint of the mvine. Note that the root constraint may be weaker than the original constraint being added or removed by `add-constraint` or `remove-constraint`, so this allows discarding more constraints. The last two arguments to `collect-unenforced` specify that only constraints weaker than `cn` should be collected.

```
update-method-graph(unenforced-cns: List of Constraints,
                    exec-roots: List of Any)
   While unenforced-cns ≠ {} do
       ;; remove the strongest constraint from unenforced-cns
      cn := the strongest constraint from unenforced-cns
      Remove all constraints equal to cn from unenforced-cns
       ;; try building mvine, collecting redetermined vars
      redetermined-vars := new empty List of Variables
      ok := build-mvine(cn, redetermined-vars)
      If ok then
          ;; we found an mvine: propagate walkabout strengths
          ;; down the mvine, and from redetermined vars
         propagate-walk-strength( { cn and all variables in redetermined-vars } )
          ;; collect any cns strictly weaker than cn that may now be enforcible
         collect-unenforced(unenforced-cns, redetermined-vars, cn.strength, false)
          ;; add newly-added cn and undetermined vars to exec-roots
         Add cn to exec-roots
         For all variables var in redetermined-vars do
            If var.determined-by = nil then
               Add var to exec-roots
```

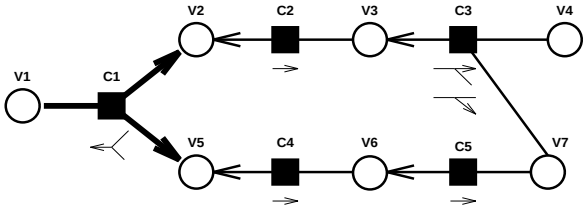
6.3 Building Method Vines: `build-mvine`

`build-mvine` tries to enforce `cn` by constructing a "method vine" (or mvine) starting with `cn` (as defined in Section 4.4). `build-mvine` tries to find a way to enforce the unenforced constraint `cn`, by changing the selected methods of stronger constraints, and revoking weaker constraints in the network. If it is successful, then it modifies the method graph and returns `true`. If `build-mvine` cannot find a way to construct an mvine, then it returns `false` and the method graph is not changed.

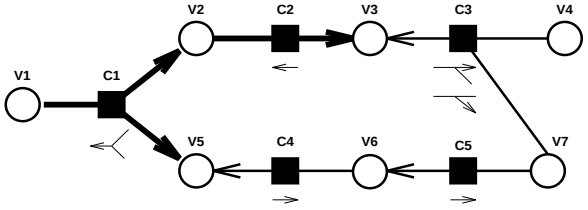
If `build-mvine` successfully enforces `cn`, it may change some of the selected methods of enforced constraints. Whenever the selected method of a constraint is changed, the `determined-by` fields of the variables are changed appropriately, and any *redetermined* variables are added to `redetermined-vars`, which can be used later when collecting unenforced constraints. If `build-mvine` cannot enforce `cn`, nothing is added to `redetermined-vars`.

`build-mvine` performs a backtracking search when assigning new selected methods to constraints. This can get complicated if a method in the mvine has multiple outputs, because each output may cause another branch in the mvine, and it may be necessary to try all possible combinations of methods in the different branches to find one without conflict. `build-mvine` handles this by using `mvine-stack`, a stack of constraints that need to have their selected methods changed. Figures 10 and 11 show how `mvine-stack` is modified during backtracking.

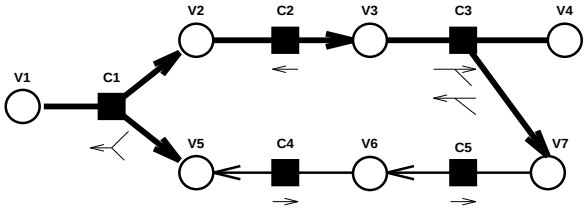
Rather than actually changing the `selected-method` fields of constraints during the search, `build-mvine` marks the constraints in the mvine, along with the variables that will be determined by these constraints. Once a complete mvine is found, the constraint `selected-method` fields and the variable `determined-by` fields are changed. At this same time, any redetermined variables are added to `redetermined-vars`. If no complete mvine is found, then the root constraint is left unenforced



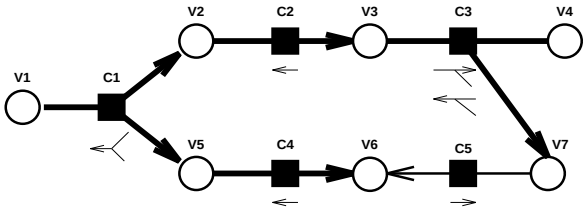
`mvine-stack` $= \{C2, C4\}$. We have just selected a method for $C1$, added it to the mvine, and added the conflicting constraints $C2$ and $C4$ to the stack (with $C2$ on the top of the stack).



`mvine-stack` $= \{C3, C4\}$. We have popped $C2$ from the top of the stack, selected a new method for it, and pushed the conflicting constraint $C3$ on the stack.



`mvine-stack` $= \{C4\}$. We have processed $C3$.



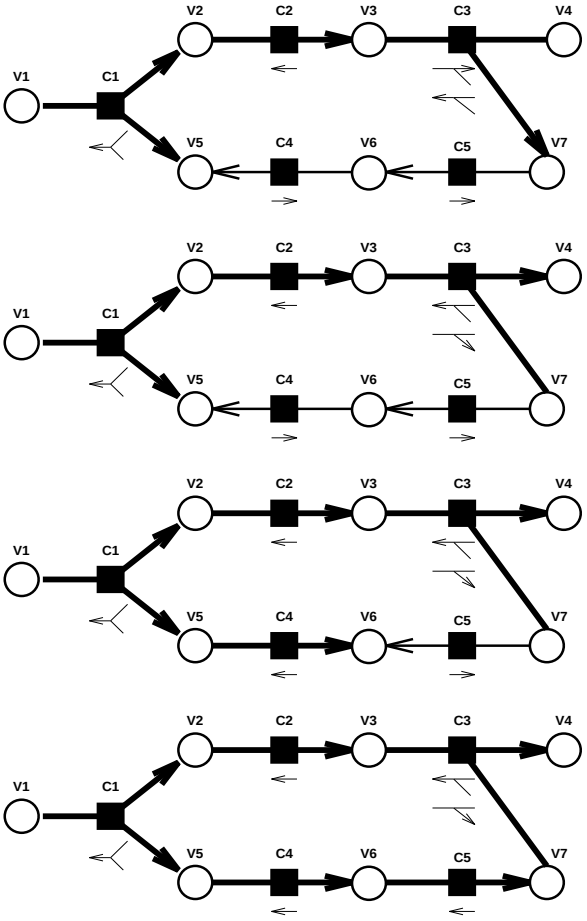
`mvine-stack` $= \{C5\}$. We have processed $C4$. At this point, we try to process $C5$, and discover that it has no methods that do not conflict with methods in the mvine. So we backtrack and restore the stack to $\{C4\}$.

Figure 10: Using `mvine-stack` while constructing an mvine with backtracking. All constraints are `required`. Methods in the mvine are drawn with thicker lines. (Continued in next figure)

and the method graph is not changed.

The procedure `build-mvine` allocates a new mark to use during the search, initializes `mvine-stack`, and then calls `mvine-enforce-cn` to start building the mvine starting with `cn` as a branch. `mvine-grow`, `mvine-revoke-cn`, and `mvine-enforce-cn` are three mutually recursive procedures that perform the backtracking search. All of these procedures (along with `build-mvine`) return `true` if a complete mvine was found, `false` otherwise.

```
build-mvine(cn: Constraint,
            redetermined-vars: List of Variables): Boolean
   mvine-stack := new empty stack of Constraints
   done-mark := new-mark()
    ;; try to build an mvine, starting by enforcing the root cn
   Return mvine-enforce-cn(cn, cn.strength, done-mark,
                           mvine-stack, redetermined-vars)
```



`mvine-stack` = $\{C4\}$. $C4$ has no other methods to try, so we backtrack and restore the stack to $\{C3, C4\}$.

`mvine-stack` = $\{C4\}$. We have processed $C3$ again, and found another method to try.

`mvine-stack` = $\{C5\}$. We have processed $C4$.

`mvine-stack` = $\{\}$. We have processed $C5$ successfully, without adding any more conflicting constraints to the stack. The stack is empty, so we have finished extending the mvine.

Figure 11: Using `mvine-stack` while constructing an mvine with backtracking. (continued from previous figure)

`mvine-grow` pops the next constraint `cn` from `mvine-stack`, and determines whether it should be enforced (by selecting another method for the constraint, and adding it to the mvine), or simply revoked. `mvine-enforce-cn` or `mvine-revoke-cn` is called to handle `cn`, and process the rest of the constraints in `mvine-stack`. If these functions return `false`, then we backtrack by restoring `cn` to `mvine-stack` and returning `false`.

It is possible that a given constraint may appear multiple times in `mvine-stack`, because it was added multiple times when processing different constraints. Whenever a constraint is processed in `mvine-enforce-cn` or `mvine-revoke-cn`, its `mark` field is set to `done-mark`. `mvine-grow` detects when the next constraint in `mvine-stack` is marked, and calls itself to process the rest of the constraints.

```
mvine-grow(root-strength: Strength,
           done-mark: Mark,
           mvine-stack: Stack of Constraints,
           redetermined-vars: List of Variables): Boolean
   If mvine-stack = {} then
       ;; no more constraints to process: we have found a complete mvine!
      Return true
   Else
      cn := Pop(mvine-stack)
      If cn.mark = done-mark then
          ;; cn has already been marked: process other constraints
         ok := mvine-grow(root-strength, done-mark,
                          mvine-stack, redetermined-vars)
      ElseIf weaker(cn.strength, root-strength) then
          ;; cn is weaker than the root constraint: revoke it
         ok := mvine-revoke-cn(cn, root-strength, done-mark,
                               mvine-stack, redetermined-vars)
      Else
          ;; try to find an alternative method for cn
         ok := mvine-enforce-cn(cn, root-strength, done-mark,
                                mvine-stack, redetermined-vars)
       ;; if backtracking, restore the popped constraint to mvine-stack
      If not(ok) then push(cn, mvine-stack)
      Return ok
```

`mvine-revoke-cn` revokes the constraint `cn` by marking the constraint, and calling `mvine-grow` to handle the rest of the constraints in `mvine-stack`. If a complete mvine is not found, then we backtrack by unmarking the constraint. If a complete mvine is found, then we reset the `determined-by` and `walk-strength` fields of any of the old method's outputs that are unmarked. If they are marked, then this means that they are determined by a method in the mvine, and they will be reset when this method is processed in `mvine-enforce-cn`. Whenever any variable `determined-by` field is changed, the variable is added to `redetermined-vars`. Finally, we set `cn.selected-method` to `nil` to revoke the constraint.

```
mvine-revoke-cn(cn: Constraint,
                root-strength: Strength,
                done-mark: Mark,
                mvine-stack: Stack of Constraints,
                redetermined-vars: List of Variables): Boolean
    ;; mark this cn.  we will process it by revoking it.
   cn.mark := done-mark
    ;; try building rest of mvine
   ok := mvine-grow(root-strength, done-mark,
                    mvine-stack, redetermined-vars)
   If ok then
       ;; we found the entire mvine!
       ;; undetermine unmarked outputs, and add to redetermined-vars
      For all variables var in cn.selected-method.outputs do
         If var.mark ≠ done-mark then
            var.determined-by := nil
            var.walk-strength := *weakest-strength*
            Add var to redetermined-vars
       ;; set selected-method for this cn
      cn.selected-method := nil
      Return true
   Else
       ;; no mvine found: we are backtracking.
      cn.mark := nil
      Return false
```

`mvine-enforce-cn` tries to enforce the constraint `cn` by marking it, and examining each of the methods that this constraint might be enforced with (as determined by `possible-method`). It tries enforcing `cn` with each method, by marking the output variables of the method, and calling `mvine-grow` to construct the rest of the mvine. When we choose a method, we add any constraints that determine the method's outputs to `mvine-stack`, to be processed (either revoked or given an alternative selected method) when extending the mvine.

If a consistent mvine is found, then we reset the `determined-by` and `walk-strength` fields of any of the old method's inputs that are unmarked, just as in `mvine-revoke-cn`. If the constraint didn't have a selected-method, then `cn` must be the unenforced root, so we don't have to do this. Finally, the `selected-method` field is set to the method chosen, and the `determined-by` fields of the method outputs are updated. Whenever any variable `determined-by` field is changed, the variable is added to `redetermined-vars`.

If no consistent mvine is found, then we unmark the method outputs, restore `mvine-stack` to its initial state (by popping the constraints that we added), and try the next method. If we have run out of methods to try, then we unmark the constraint and backtrack.

```
mvine-enforce-cn(cn: Constraint,
                 root-strength: Strength,
                 done-mark: Mark,
                 mvine-stack: Stack of Constraints,
                 redetermined-vars: List of Variables): Boolean
   ;; mark this constraint: we will try making it a branch
   cn.mark := done-mark
   ;; try each possible method: return if one allows building mvine
   For all methods mt in cn.methods do
      If possible-method(mt, cn, root-strength, done-mark) then
          ;; collect other constraints to process in mvine
         next-cns := { all constraints that determine a variable in mt.outputs }
         For all constraints new-cn in next-cns do push(new-cn, mvine-stack)
          ;; let's try to build the mvine using this method: mark the output vars
         For all variables var in mt.outputs do var.mark := done-mark
          ;; try building rest of mvine
         ok := mvine-grow(root-strength, done-mark,
                          mvine-stack, redetermined-vars)
         If ok then
             ;; we found the entire mvine!
             ;; If cn.selected-method=nil, cn is the unenforced root of the mvine.
             ;; Otherwise, undetermine unmarked outputs of old method,
             ;; and add them to redetermined-vars.
            If cn.selected-method ≠ nil then
               For all variables var in cn.selected-method.outputs do
                  If var.mark ≠ done-mark then
                     var.determined-by := nil
                     var.walk-strength := *weakest-strength*
                     Add var to redetermined-vars
             ;; set the selected method for cn, redetermine the outputs of
             ;; new method, and add the outputs to redetermined-vars.
            cn.selected-method := mt
            For all variables var in mt.outputs do
               var.determined-by := cn
               Add var to redetermined-vars
            Return true
         Else
             ;; no mvine found: try next method.
             ;; undo current method choice by unmarking method outputs.
            For all variables var in mt.outputs do var.mark := nil
             ;; pop constraints we added to mvine-stack
            For all constraints new-cn in next-cns do Pop(mvine-stack)
   ;; no more methods to try: unmark cn and backtrack
   cn.mark := nil
   Return false
```

`possible-method` returns `true` if the method `mt` is a method that could be used for enforcing `cn`. For a method to be a possible candidate, it must be true that (1) it doesn't output to any marked variables (set by methods already in the mvine), and (2) all the walkabout strengths of the method

outputs are weaker than the mvine root constraint's strength (ignoring outputs of `cn`'s current selected method, if any).

The second condition allows us to use the walkabout strengths of the variables to exclude methods that we know we cannot use without revoking a constraint of the same strength or stronger than the mvine root constraint somewhere down the mvine. When examining a potential method, we don't have to worry about the walkabout strengths of variables that `cn` currently determines, since we can switch to a new method that outputs to these variables without revoking any constraints.

```
possible-method(mt: Method, cn: Constraint,
                root-strength: Strength,
                done-mark: Mark): Boolean
   For all variables var in mt.outputs do
       ;; if an output variable is marked, we can't use this method.
      If var.mark = done-mark then
         Return false
       ;; if an output variable's walkabout strength is too strong,
       ;; and it is not currently determined by the constraint,
       ;; then we can't use this method.
      If not(weaker(var.walk-strength, root-strength)) then
         If cn.selected-method = nil then
            Return false
         If not(member(var, cn.selected-method.outputs)) then
            Return false
   Return true
```

6.4 Propagating Walkabout Strengths: `propagate-walk-strength`

`propagate-walk-strength` recalculates the walkabout strengths of all variables downstream of the variables and constraints specified by `roots`. First, `pplan-add` (Section 6.7) is called to create an ordered pplan `walk-pplan` containing the enforced constraints downstream of the roots, and marking all of the constraints with `prop-mark`. Then, `propagate-walk-strength` examines each constraint in order, recalculating the walkabout strengths of the constraint's output variables, and unmarking the constraint.

If none of the "immediate upstream constraints" of a constraint (the constraints determining the input variables of the constraint) are marked with `prop-mark`, then we know that all of the constraint's inputs have correct walkabout strength, and we can simply calculate the walkabout strengths for the output variables.

On the other hand, if any of the immediate upstream constraints are marked with `prop-mark`, this indicates that there is a cycle of constraints in `walk-pplan`. In this case, rather than attempting to analyze the cycle to find the "real" walkabout strengths, we set the walkabout constraints of any input variables determined by marked constraints to the weakest strength, and calculate the output variable walkabout strengths from these values. Since the walkabout strength is only guaranteed to be a lower bound, it is always safe to use the weakest strength.

```
propagate-walk-strength(roots: List of Any)
   prop-mark := new-mark()
    ;; make pplan to propagate downstream from roots
   walk-pplan := new empty Stack of Constraints
   pplan-add(walk-pplan, roots, prop-mark)
    ;; scan through pplan
   While walk-pplan ≠ {} do
      cn := pop(walk-pplan)
      If any-immediate-upstream-cns-marked(cn, prop-mark) then
          ;; Some of cn's upstream constraints have not been processed:
          ;; there must be a cycle.  Set any inputs determined by
          ;; unprocessed constraints to the weakest walkabout strength
         For all variables var in cn.selected-method.inputs do
            If var.determined-by ≠ nil then
               If var.determined-by.mark = prop-mark then
                  var.walk-strength := *weakest-strength*
       ;; compute walkabout strengths for output variables ofcn, and mark it done
      compute-walkabout-strengths(cn)
      cn.mark := nil
```

`any-immediate-upstream-cns-marked` returns `true` if any of the constraints determining the inputs of `cn` are marked with the given mark.

```
any-immediate-upstream-cns-marked(cn: Constraint, mark: Mark): Boolean
   For all variables var in cn.selected-method.inputs do
      If var.determined-by ≠ nil then
         If var.determined-by.mark = mark then
            Return true
   Return false
```

`compute-walkabout-strengths` calculates the walkabout strength of the variables which are currently determined by the constraint `cn`, as described in Section 5.3. A variable's walkabout strength is defined as a *lower bound* on the strength of the weakest constraint in the current method graph that would need to be revoked to allow the variable to be determined by a new constraint. Initially, the candidate strength `min-strength` is set to the strength of `cn`, since one possibility would be to simply revoke `cn`. Then, all of the methods of `cn` that do not output to `var` are examined, to find any which `cn` could switch to while revoking a weaker constraint. For each such method, the maximum walkabout strength of the method outputs is found, ignoring any variables that `cn` already determines (since no additional constraints would have to be revoked for `cn` to determine these variables). If this maximum walkabout strength is weaker than `min-strength`, then it appears that we could switch `cn` to use this method while only revoking one or more constraints with this weaker strength. The walkabout strength of `out-var` is set to the minimum of `cn.strength` and the "maximum method output walkabout" strengths.

Note that the output variables of a multi-output method may have different walkabout strengths, since there may be different sets of candidate methods that don't output to each variable.

`compute-walkabout-strengths` correctly handles the case where one method's output variables are a subset of another method's output variables. In particular, it returns `*weakest-strength*` if

there is a method whose outputs are a subset of the selected method. Normally, one wouldn't define such a constraint, but it is possible to get into this situation in Multi-Garnet [15] when using indirect variable paths.

```
compute-walkabout-strengths(cn: Constraint)
   current-outputs := cn.selected-method.outputs
   For all variables out-var in current-outputs do
      min-strength := cn.strength
      For all methods mt in cn.methods do
         If not(member(out-var, mt.outputs)) then
             ;; mt doesn't output to out-var, so it
             ;; is a possible alternative method.
            max-strength := max-out(mt, current-outputs)
            If weaker(max-strength, min-strength) then
               min-strength := max-strength
      out-var.walk-strength := min-strength

 ;; max-out returns the strongest walkabout strength among
 ;; mt's outputs, ignoring any variables in current-outputs.
max-out(mt: Method, current-outputs: List of Variables): Strength
   max-strength := *weakest-strength*
   For all variables var in mt.outputs do
      If not(member(var, current-outputs)) then
         If weaker(max-strength, var.walk-strength) then
            max-strength := var.walk-strength
   Return max-strength
```

6.5 Collecting Unenforced Constraints: `collect-unenforced`

`collect-unenforced` examines all constraints that access the variables in `vars` (and variables downstream from these variables). Any of these constraints that are unenforced and weaker than `collection-strength` are added to `unenforced-cns`. If `collect-equal-strength` is `true`, `collect-unenforced` also collects unenforced constraints with the same strength as `collection-strength`.

`collect-unenforced` itself simply allocates a new mark, and calls `collect-unenforced-mark` on each of the variables to do the actual work. `collect-unenforced-mark` examines the constraints in `var.constraints`, and calls itself recursively on the downstream variables (the outputs of each enforced constraint using `var` as an input). As each enforced constraint is traced through, or each unenforced constraint is examined, it is marked with `done-mark`, so it won't be processed again. This also prevents an infinite loop if there is a cycle of constraints.

Note: `collect-unenforced` could be written to call `pplan-add` (Section 6.7) to construct an ordered list of the downstream constraints, rather than performing the recursion in `collect-unenforced-mark`. However, this isn't worth the additional storage and time cost, since it doesn't matter what order the unenforced constraints are collected.

```
collect-unenforced(unenforced-cns: List of Constraints,
                   vars: List of Variables,
                   collection-strength: Strength,
                   collect-equal-strength: Boolean)
  done-mark := new-mark()
  For all variables var in vars do
     collect-unenforced-mark(unenforced-cns, var, collection-strength,
                             collect-equal-strength, done-mark)

collect-unenforced-mark(unenforced-cns: List of Constraints,
                        var: Variable,
                        collection-strength: Strength,
                        collect-equal-strength: Boolean,
                        done-mark: Mark)
   For all constraints cn in var.constraints do
      If cn ≠ var.determined-by and cn.mark ≠ done-mark then
          ;; cn uses var, and we haven't processed it yet: process it now
         cn.mark := done-mark
         If enforced(cn) then
             ;; cn is an enforced constraint that consumes var:
             ;; collect constraints downstream of cn's outputs.
            For all variables out-var in cn.selected-method.outputs do
               collect-unenforced-mark(unenforced-cns, out-var, collection-strength,
                                       collect-equal-strength, done-mark)
         ElseIf weaker(cn.strength, collection-strength) or
                  ( collect-equal-strength and (cn.strength = collection-strength) )
                then
             ;; cn is an unenforced cn that is weak enough to collect.
            Add cn to unenforced-cns
```

6.6 Executing Methods: `exec-from-roots`

`exec-from-roots` takes a list of redetermined variables and newly-added constraints, and executes methods for the constraints, and downstream constraints, to update the variable values to satisfy all of the enforced constraints.

If there are cycles, it may not be possible to satisfy all of the enforced constraints just by executing methods. The current version of SkyBlue doesn't even try. If a cycle is found, none of the methods are executed for the constraints in the cycle and downstream. If the cycle is later broken, then the methods are executed correctly. Future enhancements to SkyBlue may call more powerful solvers to find values satisfying a cycle of constraints, and then propagate these values downstream.

Method execution is controlled by the `valid` variable field. This field should be `false` whenever a variable is in a method graph cycle, or downstream of one, otherwise it is `true`. If all of a constraint method's inputs are valid (i.e., their `valid` fields are `true`), then it is acceptable to execute the method, and validate all of the method outputs. Otherwise, the method should not be executed, and all of its outputs should be invalidated.

Maintaining the `valid` fields makes `exec-from-roots` a little complicated. Given an acyclic constraint graph, it is only necessary to execute methods downstream from newly-enforced constraints. However, if there are cycles, and downstream variables are invalidated, then we also need to handle the case where a variable is invalid, and then becomes undetermined (due to a constraint being removed, or constraint methods being changed). By definition, any undetermined variable should be valid, so we must validate it, and try evaluating downstream constraint methods that use it. This is implemented in `update-method-graph` by adding all variables that became undetermined when the mvine is built to `exec-roots` along with the newly-enforced constraints. If any of these variables are still undetermined and invalid, then `exec-from-roots` can validate them and execute methods downstream. Note that if a newly-undetermined variable is already valid, we don't necessarily have to execute downstream methods.

`exec-from-roots` works by constructing a pplan (Section 6.7), an ordered list of the constraints whose selected methods need to be executed. This pplan contains all of the constraints downstream of the constraints in `exec-roots`, as well as those downstream of the invalid undetermined variables in `exec-roots` (which we now validate). `exec-from-roots` takes advantage of the fact that `pplan-add` can be called multiple times to add more constraints to the pplan.

Next, `exec-from-roots` scans through the constraints in `exec-pplan`. If all of the upstream constraints that determine the constraint's inputs have been processed (as determined by calling `any-immediate-upstream-cns-marked`, defined in Section 6.4), then we call `execute-propagate-valid` to try executing the method. This will execute the method if all of its inputs are valid, validating or invalidating the outputs appropriately. If some of the upstream constraints have not been processed, there must be a cycle, so `exec-from-cycle` is called to handle it. This may process (and unmark) other constraints in `exec-roots`, so we have to check that each constraint is marked before we process it.

```
exec-from-roots(exec-roots: List of Any)
   prop-mark := new-mark()
   exec-pplan := new empty stack of Constraints
    ;; make pplan to execute selected methods of newly-enforced
    ;; constraints in exec-roots and downstream constraints.
   For all constraints cn in exec-roots do
      pplan-add(exec-pplan, cn, prop-mark)
    ;; also execute selected methods downstream of undetermined
    ;; variables being changed from valid=false to valid=true.
   For all variables var in exec-roots do
      If (var.determined-by = nil) and not(var.valid) then
         pplan-add(exec-pplan, var, prop-mark)
         var.valid := true
    ;; scan through pplan
   While exec-pplan ≠ {} do
      cn := pop(exec-pplan)
      If cn.mark ≠ prop-mark then
          ;; this cn has already been processed: do nothing
      ElseIf any-immediate-upstream-cns-marked(cn, prop-mark) then
          ;; Some of this cn's upstream cns have not been processed;
          ;; there must be a cycle.  Handle cycle: possibly processing
          ;; and unmarking other cns
         exec-from-cycle(cn, prop-mark)
      Else
          ;; All of this cn's upstream cns have been processed:
          ;; unmark it and try to execute its method
         cn.mark := nil
         execute-propagate-valid(cn)
```

`execute-propagate-valid` executes the selected method of `cn` if all of its inputs are valid, and validates or invalidates the output variables. Executing the method will retrieve the values of the input variables, and set the output variables to values that satisfy the constraint.

```
execute-propagate-valid(cn: Constraint)
   inputs-valid := true
   For all variables var in cn.selected-method.inputs do
      If not(var.valid) then inputs-valid := false
   If inputs-valid then
      execute-method(cn.selected-method)
   For all variables var in cn.selected-method.outputs do
      var.valid := inputs-valid
```

Currently, `exec-from-cycle` handles a cycle including `cn` by invalidating the output variables of all constraints in the cycle (and downstream of it). All of these constraints are unmarked, to indicate that they have been processed (which also stops the recursion). In future versions of SkyBlue, `exec-from-cycle` could be replaced with a procedure which analyzes the cycle, and calls a more powerful solver to find acceptable values, and validates the variables in the cycle.

```
exec-from-cycle(cn: Constraint, prop-mark: Mark)
   If cn.mark = prop-mark then
      cn.mark := nil
      For variable var in cn.selected-method.outputs do
         var.valid := false
         For all constraints consuming-cn in var.constraints do
            If (consuming-cn ≠ cn) and enforced(consuming-cn) then
               exec-from-cycle(consuming-cn, prop-mark)
```

6.7 Constructing Propagation Plans: `pplan-add`

At several points in the SkyBlue algorithm, it is useful to topologically sort the enforced constraints in the method graph downstream from a given set of variables and constraints. `pplan-add` collects and sorts these constraints using a depth-first search.

`pplan-add` is used to construct a *propagation plan* (or *pplan*), a stack containing all of the enforced constraints in the method graph downstream of one or more constraints and variables. The constraints in a pplan are ordered such that any constraint in the pplan is higher on the stack than all of its downstream constraints (if there are no cycles). Therefore, by scanning through the constraints in a pplan, one can examine the constraints in a top-down fashion without worrying that a constraint will be processed before its upstream constraints.

If there are cycles downstream in the method graph, the pplan will contain constraints whose upstream constraints appear later in the pplan. It is easy to detect such cycles when processing the pplan constraints. When `pplan-add` constructs a pplan, it marks all of the collected constraints with the mark `done-mark` (this mark is used to terminate infinite loops, and guarantee that each constraint is only collected once). If each constraint is unmarked as it is processed, a cycle can be detected by checking whether all of a constraint's immediate upstream constraints are unmarked. If this is not true, then this constraint must be in a cycle, and the rest of the constraints in the cycle must appear later in the pplan. This type of scanning is done in `exec-from-roots` and `propagate-walk-strength`, which include code to handle any cycles which are detected.

`pplan-add` takes a stack `pplan` to collect the constraints, a root object `obj` to trace downstream from, and a mark `done-mark`. If `obj` is an enforced constraint that has not been collected yet (its mark is not `done-mark`), then it is marked and collected, and all of the constraints downstream of its outputs are collected. Actually, `obj` is collected *after* all of the downstream constraints are collected (pushed on the stack), so that the constraint will appear on the stack *before* its downstream constraints. If `obj` is a variable, then all of the enforced constraints that use it are collected. If `obj` is a list, all of the constraints downstream of its elements are collected.

`pplan-add` can be called multiple times to construct a pplan from multiple roots, as long as no constraint marks are changed between calls. Consider: If `pplan-add` is called to add a constraint that is downstream of previously-added constraints, then it will already be marked, and the call to `pplan-add` will do nothing. If it is upstream of all of the constraints in the pplan, calling `pplan-add` will simply add the upstream constraints on top of the stack.

```
pplan-add(pplan: stack of Constraints, obj: Any, done-mark: Mark)
   If obj is a constraint then
      If enforced(obj) and obj.mark ≠ done-mark then
          ;; process unmarked, enforced constraint by marking it, collecting
          ;; downstream constraints, and pushing it on top of the pplan stack.
         obj.mark := done-mark
         For all variables var in obj.selected-method.outputs do
            pplan-add(pplan, var, done-mark)
         push(obj, pplan)
   ElseIf obj is a variable then
       ;; process variable by collecting downstream constraints starting
       ;; with the constraints directly consuming the variable.
      For all constraints cn in obj.constraints do
         If cn ≠ var.determined-by then
            pplan-add(pplan, cn, done-mark)
   ElseIf obj is a list then
      For all elements elt in obj do
         pplan-add(pplan, elt, done-mark)
```

7 Input and Output Constraints

SkyBlue methods are defined as side-effect-free functions that read the values of the method input variables and compute values for the method output variables that satisfy the constraints (Section 2). Sometimes, however, it is useful to define constraints that don't strictly follow this definition.

An *input constraint* is a constraint that has a single method with *no* input variables and one or more output variables. The values for the output variables are calculated by accessing information external to the constraint graph. One use for such a constraint would be to read input devices. For example, suppose that a programmer implementing an interactive graphics application wants to repeatedly set a constrained variable to the position of the mouse and resatisfy the constraints. This could be implemented by repeatedly adding and removing constraints that set the variable to different mouse positions. A simpler alternative (which may also be more efficient) is to define an input constraint whose single method reads the current position of the mouse, and sets its output variable to that value. When this constraint is added, the method graph would be changed, and the mouse constraint method would be executed once (if the mouse constraint was not overridden by a stronger constraint). To handle further new mouse positions, the application using SkyBlue must explicitly execute the mouse constraint's method (if it is enforced) as well as any downstream enforced methods. This can be done by calling `exec-from-roots` (Section 6.6), passing a list of the input constraints to be re-executed.

An *output constraint* is a constraint that has a single method with one or more input variables and *no* output variables. When the method is executed, it can access the values of the input variables, and change information external to the constraint graph (causing a side-effect). For example, an output constraint could be used to read some constrained variables, and update a graphic display. The output constraint method would be executed once when the constraint is added (it is always possible to enforce an output constraint, regardless of its strength), and re-executed whenever any

of the upstream methods are executed.[6]

It is important that input and output constraint methods do not access the constraints and variables in their constraint graph, other than their declared input or output variables. In particular, they should not call `add-constraint` or `remove-constraint`.

It is possible to construct methods with both input and output variables that also read external values and cause side-effects, but they would be more difficult to use than "pure" input and output constraints. In particular, it would be more difficult to control when they are executed: an input constraint with input variables would not be executed if any of its input variables were marked invalid, and would be re-executed whenever any of its input variables were changed. On the other hand, an output constraint with output variables might not be enforced, if its outputs were set by stronger constraints.

8 Extracting and Executing Plans

Sometimes when using input constraints it is possible to improve performance by extracting a *plan*, an object containing a sorted list of the downstream methods. If the method graph doesn't change, it could be quickly executed when the input constraint needs to be reexecuted. This section describes how to extract plans, execute them, and automatically invalidate them. The invalidation technique requires making some small changes to the SkyBlue pseudocode of Section 6. This section should be considered an optional extension to the SkyBlue algorithm.

Consider a constraint-based graphic editor where the user drags the mouse to move an object, and the constraints should be maintained while the object is moved. This could be implemented with an input constraint that is repeatedly executed (by an explicit call to `exec-from-roots`) every time the mouse is moved to set a variable to the position of the mouse, and propagate this value through a network of enforced constraints. Suppose that the rest of the network is not being changed while the mouse is being moved. In this case SkyBlue will repeatedly topologically order the same set of enforced constraints downstream of the input constraint, and execute them. This repeated sorting can be avoided by saving the sorted list of constraints, known as a *plan*, and simply executing these constraint's selected methods in order. It is also possible to detect cycles in the plan, and avoid cycle detection while executing the plan constraints.

An important issue that arises when using plans is that a plan may be *invalidated*. A plan is derived from a particular method graph. If the method graph is changed by adding or removing a constraint, then the plan no longer accurately represents the sequence of constraint methods that should be executed to satisfy the constraints. Actually, all constraint operations may not invalidate a particular plan: if the method graph is changed in a way that doesn't effect the constraints in the plan, than it would still be valid. A technique is described below for automatically invalidating a plan when it becomes invalid. It is not perfect (it may invalidate a plan unnecessarily), but it will never indicate a plan is valid when it isn't.

[6] Note that an output constraint method will not be executed if any of its input variables is in or downstream of a cycle (and thus is marked invalid, see Section 6.6).

field name	*type*	*description*
`valid`	Boolean	true if this plan is valid
`root-cns`	List of Constraints	constraints used to construct this plan
`good-cns`	List of Constraints	ordered list of enforced constraints in plan
`bad-cns`	List of Constraints	enforced constraints in and downstream of cycles

A SkyBlue plan is represented by a plan record. The `valid` field is `true` if this plan is still valid, otherwise `false`. `root-cns` contains the list of constraints passed to `extract-plan` (below) when constructing the plan. `good-cns` is the ordered list of enforced constraints whose selected methods should be executed (in order) when the plan is executed. `bad-cns` is the list of enforced constraints downstream from the root constraints that are in cycles or downstream of cycles. When the plan is executed, we can simply leave these constraint unexecuted, without doing cycle detection. They are saved in the plan object as part of plan invalidation.

`extract-plan` constructs a plan containing a subset of the enforced constraints in the current method graph. The subset contains all enforced constraints in `root-cns`, and all enforced constraints downstream of these roots. The constraints in the plan are ordered, so that executing the selected methods of these constraints sequence re-satisfies all constraints enforced by the method graph.

`extract-plan` is similar to `exec-from-roots`, except that it collects the constraints whose selected methods should be executed, rather than actually executing them. A pplan is constructed sorting the constraints downstream of `root-cns` (any unenforced constraints in `root-cns` are ignored by `pplan-add`), and constraints are popped off the pplan and collected on the list `good-cns`. If a cycle is found, all of the constraints in the cycle, and downstream of the cycle, are saved in the list `bad-cns`. After all of the constraints have been saved on one of these two lists, a plan object is constructed, with separate fields to hold the the good, bad, and root constraints, and a valid plan is created by calling `create-valid-plan`.

```
extract-plan(root-cns: List of Constraints): Plan
   good-cns := new empty List of Constraints
   bad-cns := new empty List of Constraints
   prop-mark := new-mark()
    ;; make pplan rooted with enforced constraints.
   pplan := new empty stack of Constraints
   pplan-add(pplan, root-cns, prop-mark)
    ;; scan through pplan
   While pplan ≠ {} do
      cn := pop(pplan)
      If cn.mark ≠ prop-mark then
          ;; this cn has already been processed: do nothing
      ElseIf any-immediate-upstream-cns-marked(cn, prop-mark) then
          ;; Some of this cn's upstream cns have not been processed;
          ;; there must be a cycle: add cn to invalid constraint list
          Add cn to bad-cns
          ;; note: do *not* unmark cn: all constraints in the cycle,
          ;; and downstream of it, will thus be added to the bad list.
      Else
          ;; All of cn's upstream cns have been processed: unmark and
          ;; add to *end* of good-cns (after the upstream cns).
          cn.mark := nil
          Add cn to end of good-cns
   Return create-valid-plan(root-cns, good-cns, bad-cns)
```

`execute-plan` executes the plan by trying to execute each constraint's selected methods, in order. `execute-propagate-valid` (Section 6.6) is used to execute the methods conditional on the `valid` fields of its input variables, because cycles may be introduced (or removed) upstream from the plan constraints, changing whether the plan constraint variables are valid (and whether the plan constraints should be executed). No matter what happens upstream, however, the `bad-cns` constraints will not be executed, and their output variables will remain invalidated (they must have been invalidated when the cycle was initially constructed, before the plan was extracted).

```
execute-plan(plan: Plan)
   If plan.valid then
      For all constraints cn in plan.good-cns do
         execute-propagate-valid(cn)
   Else
      Error "trying to execute invalid plan"
```

The SkyBlue plan invalidation technique is based on checking every time that a constraint has its `selected-method` field changed, and invalidating all plans could be invalidated by that change. There are two ways that a plan can become invalid: (1) if one of the constraints in the plan (in `root-cns` or `good-cns` or `bad-cns`) has its `selected-method` field changed, or (2) if some other constraint has its `selected-method` field changed to a new method that *inputs* from a variable that is determined by a constraint in the plan. In the first case, the old plan containing the constraint may be invalid because the order of constraints may have changed, or constraints may need to be removed (if the changed constraint was revoked). In the second case, the plans containing the upstream constraint are invalid since they should include the changed constraint.

This technique is implemented by making two changes to the SkyBlue algorithm: (1) Add an additional field `valid-plans` to each constraint. This field will contain a list of all valid plans that include that constraint (in the plan's `root-cns` or `good-cns` or `bad-cns` lists). Plans are added to the `valid-plans` field by `create-valid-plan`. (2) Every time that SkyBlue changes the `selected-method` field of any constraint, `invalidate-plans-on-setting-method` must be called.[7]

`create-valid-plan` creates a new plan with the specified constraint lists, and its `valid` field `true`. Then it adds this plan to the `valid-plans` fields of all of the constraints in the plan.

```
create-valid-plan(root-cns: List of Constraints,
                  good-cns: List of Constraints,
                  bad-cns: List of Constraints): Plan
   plan := new Plan
   plan.valid := true
   plan.root-cns := root-cns
   plan.good-cns := good-cns
   plan.bad-cns := bad-cns
   For all constraints cn in plan.root-cns or plan.good-cns or plan.bad-cns do
      Add plan to cn.valid-plans
   Return plan
```

`invalidate-plans-on-setting-method` invalidates any plans including the constraint whose selected method is being changed, as well as any plans containing constraints that are immediately upstream of this constraint's new selected method.

```
invalidate-plans-on-setting-method (cn: Constraint, new-mt: Method)
    ;; we will invalidate any plans including this constraint
   invalidate-constraint-plans(cn.valid-plans)
    ;; we will also invalidate plans including constraints that
    ;; determine the inputs of this new method.
   If new-mt ≠ nil then
      For all variables var in new-mt.inputs do
         If var.determined-by ≠ nil then
            invalidate-constraint-plans(var.determined-by)
```

`invalidate-constraint-plans` invalidates the plans in the `valid-plans` field of constraint `invalid-cn`, and removes these plans from *all* the `valid-plans` fields of the constraints in these plans. Small point: the field invalid-cn.valid-plans is not changed until after the loop is finished, in case a destructive change to this list might confuse the loop.

[7] We don't have to worry about whether `invalidate-plans-on-setting-method` is called before or after SkyBlue changes the `determined-by` fields of the method inputs, since either (1) the `determined-by` field of an input is not changed, or (2) the new constraint determining the var will have its `selected-method` field changed, and its plans will be invalidated.

```
invalidate-constraint-plans(invalid-cn: Constraint)
   For all plans plan in invalid-cn.valid-plans do
      plan.valid := false
      For all constraints cn in plan.good-cns or plan.bad-cns or plan.root-cns do
         If cn ≠ invalid-cn then
            Remove plan from cn.valid-plans
   invalid-cn.valid-plans := {}
```

9 Performance Measurements

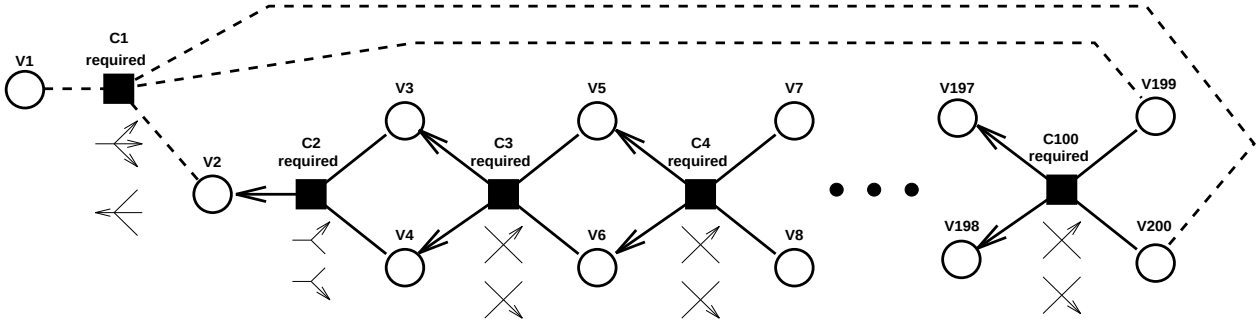
SkyBlue was developed to extend the range of user interface applications that could be constructed using local propagation constraints. In order to build real applications using SkyBlue, it must be fast enough to resatisfy constraints during user interactions. We are still investigating the performance characteristics of SkyBlue and trying to improve its performance. This section discusses the theoretical performance of SkyBlue and presents timings collected by measuring benchmarks and real user interfaces.[8]

9.1 Time Complexity

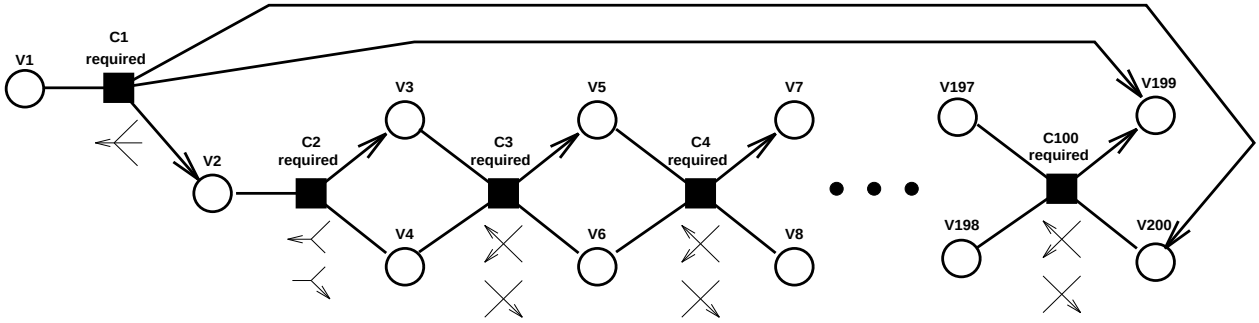
The worst-case time complexity of the DeltaBlue algorithm for adding or removing a constraint from the constraint graph is $O(MN)$, where N is the total number of constraints in the graph and M is the maximum number of methods in a constraint [13]. M is bounded by a small constant in most constraint graphs so this reduces to $O(N)$. Over the same types of constraint graphs that DeltaBlue can handle (no cycles, single-output methods), SkyBlue has the same worst-case behavior. However, if constraints have multi-output methods the performance can be much worse. Reference [13] proved that finding an LGB method graph for a constraint graph with multi-output methods is NP-complete. In particular, it is possible to construct cases where the worst case time is $O(M^N)$ for a constraint graph with N constraints that have M multi-output methods each (see Figure 12 for an example).

In actual use, the worst-case time complexity has not been a problem for several reasons. First, the constraint graphs that lead to $O(M^N)$ performance are rather contrived. We have not seen similar graphs occur in real applications. Second, we believe (and are trying to prove) that the exponential behavior only increases as a function of the number of constraints with multi-output methods, regardless of the number of other constraints. Although constraints with multi-output methods are very useful (if not indispensable), the overriding majority of the constraints used in real applications have single-output methods. Finally, both DeltaBlue and SkyBlue typically change only a small subgraph of the constraint graph when a constraint is added or removed so the actual performance is usually sub-linear in the number of constraints.

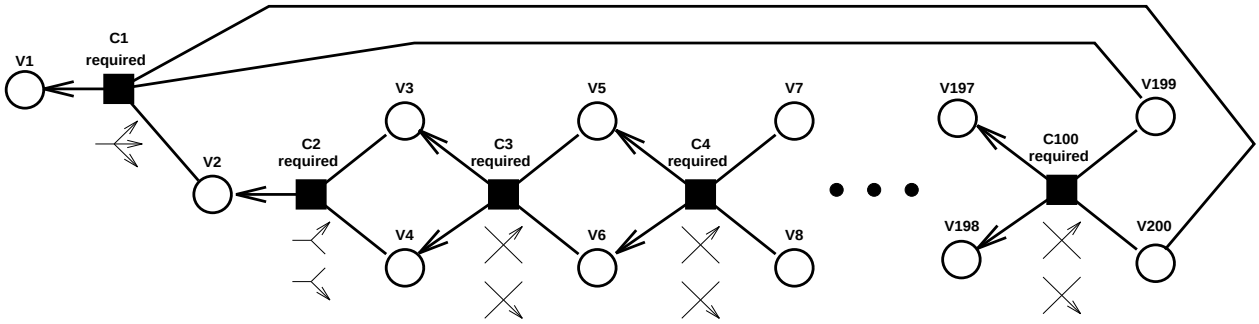
[8] Garnet is implemented in Common Lisp and Multi-Garnet is built on top of Garnet. All figures reported here were measured on a lightly loaded Sun Microsystems SPARCstation IPX using Allegro Common Lisp version 4.1, Garnet version 2.1, and Multi-Garnet version 2.2.



Suppose we have just called `add-constraint` to add the constraint $C1$ to the constraint graph. Initially, $C1$ is unenforced in the above method graph. Now, `update-method-graph` will try to enforce it, by constructing a method vine.



Suppose that we start building the mvine by selecting the method of $c1$ that outputs to $V2$, $V199$, and $V200$. Furthermore, suppose that we next try to extend the mvine through $C2$, $C3$, etc. No matter which selected methods are chosen for $C2$ through $C100$, a method conflict will be discovered between $C100$ and $C2$, when trying to select a method for $C100$. The above method graph shows one possible choice leading to a method conflict. Given this situation, `build-mvine` will backtrack, trying another selected method for $C100$, leading to a conflict, then backtrack to $C99$, etc.



`build-mvine` will try 2^{99} possible combinations of selected methods to $C2$–$C100$ before it tries the other method for $C1$, which leads to the LGB method graph above.

Figure 12: An example where adding a constraint could take worst-case exponential time.

9.2 Comparing Performance Techniques

SkyBlue has significant new capabilities absent from other solvers, such as support for multi-output methods. Therefore, it is not possible to create a benchmark that exercises all of the features of SkyBlue and directly compare it to other algorithms. For regression testing and performance tuning, a random sequence of 10000 calls to `add-constraint` and `remove-constraint` was generated and saved. Using this sequence it is possible to measure how the performance of SkyBlue is improved by the techniques described in Section 5.

local collection	*walkabout strengths*	*time (seconds)*	*number mvines*		*number backtracks*
			attempted	*constructed*	
on	on	25.3	24222	5840	12748
on	off	47.2	24222	5840	196012
off	on	44.1	77354	5826	36262
off	off	125.5	77354	5826	571534

Figure 13: Timings collected for executing a sequence of 10000 constraint operations in SkyBlue with different performance techniques enabled.

Figure 13 shows the timing results when executing the sequence with four different configurations of the SkyBlue algorithm. The first two columns specify whether two techniques were enabled or disabled. The *local collection* column specifies whether the technique from Section 5.2 was used to collect constraints local to the added or removed constraint when enforcing and executing methods, versus processing all of the constraints in the constraint graph. The *walkabout strengths* column specifies whether variable walkabout strengths were used to improve mvine searches and updated whenever an mvine was constructed, as described in Section 5.3. The times in the third column show that SkyBlue is most efficient with both techniques enabled, about half the speed with either technique disabled, and exceedingly slow with neither of the techniques enabled. The collection strength technique of Section 5.1 was enabled in all four configurations. Disabling this technique did not change the times as much as the other two techniques.

These timings are explained by the remaining three columns, which record the number of times SkyBlue attempted to enforce a constraint by constructing an mvine, the number of times the mvine was successfully constructed, and the number of times backtracking occurred while trying to construct an mvine. Notice that the walkabout strengths technique did not affect the number of mvines attempted or successfully constructed. However, it did have a major effect on the amount of backtracking. The local collection technique decreases the number of mvines attempted without any significant change in the number successfully constructed. These timings can be interpreted as follows: the local collection technique saves time by reducing the number of attempts to construct mvines and the number of constraint methods executed. The walkabout strength technique saves time by reducing the amount of backtracking when constructing an mvine. This particularly reduces backtracking (and time) when there are numerous unsuccessful mvines, such as when the local collection technique is disabled.

9.3 Comparing DeltaBlue and SkyBlue

To compare the performance of DeltaBlue and SkyBlue, a sequence of 10000 random calls to `add-constraint` and `remove-constraint` was generated subject to the restriction that all methods had a single output and there were no cycles in the constraint graph. DeltaBlue took 3.1 seconds to execute the sequence, versus 5.8 seconds for SkyBlue. This indicates that SkyBlue is about half the speed of DeltaBlue, given constraint graphs that DeltaBlue can handle. The reason that DeltaBlue is faster is because the assumption that there are no multi-output methods in the method graph allows it to ignore many unenforced constraints that SkyBlue tries to enforce. SkyBlue also spends extra time trying to detect and handle cycles. It may be possible to modify SkyBlue to detect when the constraint graph contains no cycles or multi-output methods and achieve the speed of DeltaBlue in this case.

9.4 Comparing Garnet and Multi-Garnet

Figure 14 shows timings collected by measuring several interactive graphics applications written in Garnet and Multi-Garnet. This allows comparing the performance of Garnet's constraint system with the SkyBlue solver used in Multi-Garnet. The applications were measured by moving an element on the display 500 times and measuring the total time used to resatisfy the constraints (moving other constrained objects) and update the display 500 times. The first four columns show the total measured times for the benchmarks (500 moves) as well as the update time for each move. The other columns divide the total times into separate elements. The *graphics redisplay* column shows the time to update the display using Garnet's redisplay algorithm. The *method execution* column shows the time spent executing methods to satisfy the constraints. The *Garnet overhead* and *Multi-Garnet overhead* columns list the remaining time spent doing Garnet and Multi-Garnet constraint operations, calculated by measuring the total time executing the benchmark with the two systems and subtracting the *graphics redisplay* and *method execution* figures.

benchmark	*Garnet*		*Multi-Garnet*		*graphics*	*method*	*Garnet*	*Multi-Garnet*
	total	*once*	*total*	*once*	*redisplay*	*execution*	*overhead*	*overhead*
Demo-Manyobjs	19.8	0.04	21.2	0.04	15.9	2.5	1.4	2.8
Move-Axis	45.0	0.09	44.0	0.09	35.6	4.3	5.1	4.1
Scale-Points	130.5	0.26	146.7	0.29	68.6	52.5	9.4	25.6

Figure 14: User interface benchmark timings using Garnet and Multi-Garnet (seconds).

The Demo-Manyobjs benchmark is a simple interactive program displaying a chain of boxes connected by lines. The lines are positioned between the boxes using constraints. As a box is moved the constraints are resatisfied and the connecting lines are repositioned. All the constraints in both the Garnet and Multi-Garnet versions are one-directional. In this case SkyBlue has about twice the overhead of Garnet.

The Move-Axis and Scale-Points benchmarks measured two interactions with the scatterplot of Figure 1. The Move-Axis benchmark measures the time to move the X-axis 500 times (the second plot) and the Scale-Points benchmark measures the time to scale the point-cloud by moving a point

500 times (the third plot). The Multi-Garnet version of this benchmark uses multi-way constraints with multi-output methods, which are not supported in Garnet. In order to compare the performance with Garnet, a simplified version was implemented in Garnet using only one-way constraints. The Garnet version only supports a small part of the functionality of the Multi-Garnet version. For example, the Garnet version only allows one particular point could be moved to scale the point cloud, whereas any point could be moved in the Multi-Garnet version. The measured Multi-Garnet overhead is one to two times that of Garnet, which is actually a significant accomplishment for Multi-Garnet. It offers much more than Garnet without significantly worse performance.

The last four columns demonstrate that the total time is dominated by the graphics redisplay and method execution time. These two elements would be the same no matter how the application was implemented, whether a constraint solver was used or the relationships were maintained explicitly (integrating the method procedures into an imperative program). This is typical in interactive systems using local propagation and suggests that constraint solvers such as SkyBlue can be used to construct interactive systems without adversely impacting performance.

The most important question when examining the performance of an interactive system is the actual redisplay rate that the user experiences when using the system. The Demo-Manyobjs and Move-Axes benchmarks have times of under 0.1 seconds per redisplay for both the Garnet and Multi-Garnet versions, which is quite acceptable. For the Scale-Points benchmark, the Garnet version uses 0.26 seconds per redisplay and the Multi-Garnet version uses 0.29 seconds per redisplay. This is not great but it is usable. Undoubtedly it could be improved significantly by reimplementing the whole system in a language like C, but even this Lisp implementation is fast enough to show that the SkyBlue constraint solver provides a practical tool for user interface construction.

10 Future Work

This paper has presented the SkyBlue algorithm, a constraint solver that uses local propagation to efficiently maintain constraint hierarchies. Future work will investigate improving SkyBlue's speed by calculating more accurate walkabout strengths in cycles, implementing heuristics to reduce backtracking when constructing mvines, and adapting techniques from DeltaBlue in cases where there are no multi-output methods or cycles in the method graph. We also intend to investigate various extensions to SkyBlue:

Cycle Solvers We intend to integrate SkyBlue with more powerful solvers to solve cycles of constraints. For example, if `eval-from-roots` discovers that all of the constraints in a cycle represent linear equations (using extra information associated with the constraints), then it can pass the constraints to an equation solver to find values for the variables in the cycle and then continue local propagation downstream from the cycle.

Constraint Compilers SkyBlue could be extended with a *constraint compiler* to compile a subgraph of the constraint graph (possibly including cycles) into a single complex constraint with methods to handle the different propagation directions [4]. This could be used to avoid repeated calls to a more powerful solver and as an encapsulation mechanism. SkyBlue provides a good base for developing a constraint compiler since compiled constraints typically have methods with multiple outputs.

Unsolvable Constraints We intend to extend the ability of SkyBlue to handle constraints that it cannot solve. Currently, it detects when the constraint graph includes cycles and marks variables whose values may not satisfy the enforced constraints. Sometimes it is not possible to know that constraints cannot be solved until the constraint methods are executed. For example, a constraint method may cause a divide-by-zero error when given certain inputs. It may not always be possible to satisfy all of the constraints in a constraint graph, but it is important for the constraint solver to keep track of which constraints are satisfied.

Acknowledgements

Thanks to Alan Borning, Ralph Hill, and Brad Vander Zanden for many useful discussions and comments on this paper. This work was supported in part by the National Science Foundation under Grants IRI-9102938 and CCR-9107395.

References

[1] Franz G. Amador, Adam Finkelstein, and Daniel S. Weld. Real-Time Self-Explanatory Simulation. (submitted to AAAI 93).

[2] Paul Barth. An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[3] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[4] Bjorn Freeman-Benson. A Module Compiler for ThingLab II. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 389–396, New Orleans, October 1989. ACM.

[5] Bjorn Freeman-Benson and Alan Borning. The Design and Implementation of Kaleidoscope'90, A Constraint Imperative Programming Language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 174–180, April 1992.

[6] Bjorn Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[7] Michel Gangnet and Burton Rosenberg. Constraint Programming and Graph Algorithms. In *Second International Symposium on Artificial Intelligence and Mathematics*, January 1992.

[8] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Third Eurographics Workshop on Object-oriented graphics*, Champery, Switzerland, October 1992. Also will be published in *Advances in Object-Oriented Graphics II*, Springer-Verlag, 1993.

[9] Ralph D. Hill. Languages for the Construction of Multi-User Multi-Media Synchronous (MUMMS) Applications. In Brad Myers, editor, *Languages for Developing User Interfaces*, pages 125–143. Jones and Bartlett, Boston, 1992.

[10] Scott E. Hudson. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Transactions on Programming Languages and Systems*, 13(3):315–341, July 1991.

[11] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk*, volume II. Prentice Hall, Englewood Cliffs, NJ, 1991.

[12] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing User Interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[13] John Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as Department of Computer Science and Engineering Technical Report 91-08-12.

[14] Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer*, 23(11):71–85, November 1990.

[15] Michael Sannella and Alan Borning. Multi-Garnet: Integrating Multi-Way Constraints with Garnet. Technical Report 92-07-01, Department of Computer Science and Engineering, University of Washington, September 1992.

[16] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience*, 1992. In press.

[17] D. Serrano and D. Gossard. Constraint Management in Conceptual Design. In D. Sriram and R. Adey, editors, *Knowledge Based Expert Systems in Engineering: Planning and Design*, pages 211–224. Computational Mechanics, 1987.

[18] Brad Vander Zanden. A Domain-Independent Algorithm for Incrementally Satisfying Multi-Way Constraints. Technical Report CS-92-160, Computer Science Department, University of Tennessee, July 1992.

[19] Brad Vander Zanden, Brad Myers, Dario Guise, and Pedro Szekely. The Importance of Pointer Variables in Constraint Models. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 155–164, Hilton Head, South Carolina, November 1991.

[20] William Welch and Andrew Witkin. Variational Surface Modeling. In *Proceedings of ACM SIGGRAPH'92*, pages 157–166, Chicago, Illinois, July 1992. Also in *Computer Graphics* 26(2), July 1992.