The SkyBlue Constraint Solver and Its Applications

Michael Sannella
Department of Computer Science
and Engineering, FR-35
University of Washington
Seattle, WA 98195
sannella@cs.washington.edu

Abstract

The SkyBlue constraint solver is 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. Like DeltaBlue, SkyBlue represents constraints between variables by sets of short procedures (methods) and incrementally resatisfies the set of constraints as individual constraints are added and removed. DeltaBlue has two significant limitations: cycles of constraints are prohibited, and constraint methods can only have a single output variable. 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. This paper presents the SkyBlue algorithm and discusses several applications that have been built using SkyBlue.

1 Introduction

The DeltaBlue algorithm is an incremental algorithm for maintaining sets of required and preferential constraints (constraint hierarchies) using local propagation [4, 5, 9]. The ThingLab II user interface development environment was based on DeltaBlue, demonstrating its feasibility for constructing user interfaces [5]. 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.

The SkyBlue algorithm was developed to remove these limitations. Even though it is not always possible to solve cycles of constraints using local propagation, SkyBlue allows constructing such cycles. SkyBlue cannot satisfy the constraints around a cycle, but it correctly maintains the non-cyclic constraints elsewhere in the graph. In some situations it may be possible to solve the subgraph containing the cycle by calling a more powerful solver. Future work will extend SkyBlue to call specialized constraint solvers to solve the constraints around a cycle, and continue using local propagation to satisfy the rest of the constraints.

SkyBlue also supports constraints with multi-output methods, which are useful in many situations. 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 $(\rho, \theta) \leftarrow (\sqrt{X^2 + Y^2}, \arctan(Y, X))$. 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 $Point \leftarrow CreatePoint(X, Y)$.

Support for multi-output methods introduces a performance issue. It has been proved that supporting multi-output methods is NP-complete [5]. In actual use, the worst-case time complexity has not been a problem. 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. Over the same types of constraint graphs that DeltaBlue can handle (no cycles, single-output methods), SkyBlue has been measured at about half the speed of DeltaBlue. In the future SkyBlue may be extended to detect when the constraint graph contains no cycles or multi-output methods and achieve the speed of DeltaBlue in this case.

SkyBlue is currently being used as the constraint solver in several applications (see Section 6). SkyBlue implementations are available from the author. Detailed information on SkyBlue, including complete pseudocode, is available in a technical report [7].

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.

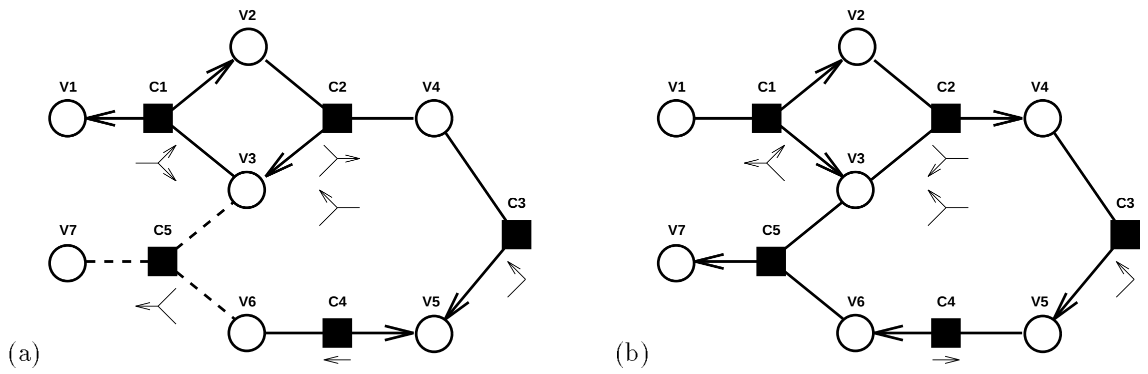


Figure 1: (a) A method graph with an unenforced constraint ($C5$), a method conflict (at $V5$), and a directed cycle (between $C1$ and $C2$). (b) Another method graph for the same constraints where all of the constraints can be satisfied.

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 1). 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 a method graph the constraint is *enforced* in that method graph, 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 a *method conflict.* In Figure 1a, there is a method conflict between the selected methods of $C3$ 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 1b 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. 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 1a, 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 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.

3 Constraint Hierarchies

An important property of any constraint solver is how it behaves when the set of constraints is overconstrained (i.e., there is no solution that satisfies all of the constraints) or underconstrained (i.e., there are multiple solutions). If the solver is maintaining constraints within a user interface application, it is not acceptable to handle these situations by signaling an error or halting. The *constraint hierarchy* theory presented in [2] provides a way to specify declaratively how a solver should behave in these situations. A constraint hierarchy is a set of constraints, each labeled with a *strength*, indicating how important it is to satisfy each constraint.[1] 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 solver can choose any solution. The user can control which solution is chosen by adding weak *stay* constraints to specify variables whose value should not be changed.

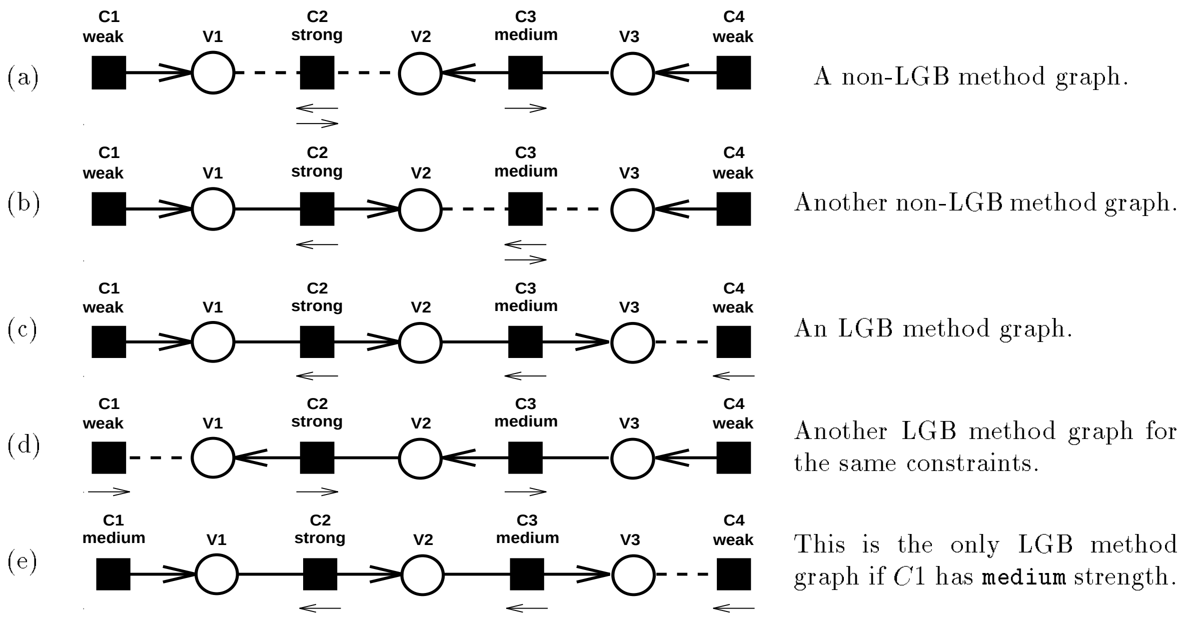


Figure 2: LGB and non-LGB Method Graphs.

The SkyBlue solver uses the constraint strengths to construct *locally-graph-better* (or *LGB*) method graphs [5]. 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

[1] In this paper, strengths will be written using the symbolic names `required`, `strong`, `medium`, and `weak`, in order from strongest to weakest.

methods for other enforced constraints with the same or stronger strength).[2] For example, consider the method graph in Figure 2a. 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 2b. Actually, this method graph is not LGB either since $C3$ could be enforced by revoking $C4$, producing Figure 2c. This method graph is LGB since the only unenforced constraint ($C4$) cannot be enforced by revoking a weaker constraint.

There may be multiple LGB method graphs for a given constraint graph. Figure 2d shows another LGB method graph which is neither better nor worse than Figure 2c. Given these constraints, SkyBlue would construct one of these two method graphs arbitrarily. The constraint strengths 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 2e. 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 2, $C1$ and $C4$ are stay or set constraints.

Reference [2] 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 (defined in Reference [2]). Reference [5] 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 presentation of SkyBlue is divided into several sections. Sections 4.1 and 4.2 present an overview of `add-constraint` and `remove-constraint`. Section 4.3 describes how a constraint is enforced by constructing a method vine, the core of the SkyBlue algorithm. 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. More detailed information on SkyBlue, including complete pseudocode for the algorithm, is available in a technical report [7].

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* (described in Section 4.3). `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

[2]Reference [5] defines "locally-graph-better" such that directed cycles are prohibited. In this paper, the definition of LGB is modified so LGB method graphs may include directed cycles.

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 the selected methods in the method graph to satisfy the enforced constraints (as described in Section 2).

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, since no more mvines can be constructed.

As an example, suppose that `add-constraint` has just added $C2$ to the constraint graph and the current method graph is shown in Figure 2a. One way that an mvine could be constructed is by enforcing $C2$ with the method that outputs to $V2$ and revoking $C3$ (Figure 2b). Given this method graph, the second step would try constructing an mvine to enforce $C3$, possibly by revoking $C4$ (Figure 2c). 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 2d 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. `remove-constraint` removes a constraint `cn` from the constraint graph by performing the following steps:

1. If `cn` is currently unenforced, 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 the selected methods in the method graph to satisfy the enforced constraints (as described in Section 2).

4.3 Constructing Method Vines

The SkyBlue algorithm is based on attempting to enforce an unenforced constraint by changing the selected methods of constraints with the same or stronger strength and/or 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). If this method has a method conflict with the selected methods of other enforced constraints, we select new methods for these other constraints. These new selected methods may conflict with yet other selected methods, and so on. This process extends through the method graph, building 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 3 presents an example demonstrating the process of constructing an mvine. A complete mvine is a connected subgraph of the method graph. An mvine is not necessarily a tree: separate branches may merge and it may contain directed cycles. 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 different branches cannot 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.

5 Performance Techniques

The SkyBlue algorithm described in Section 4 works correctly, but its performance suffers as the constraint graph becomes very large. This happens because larger constraint graphs may contain greater numbers of unenforced constraints that SkyBlue has to try enforcing, and 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.

5.1 The Collection Strength Technique

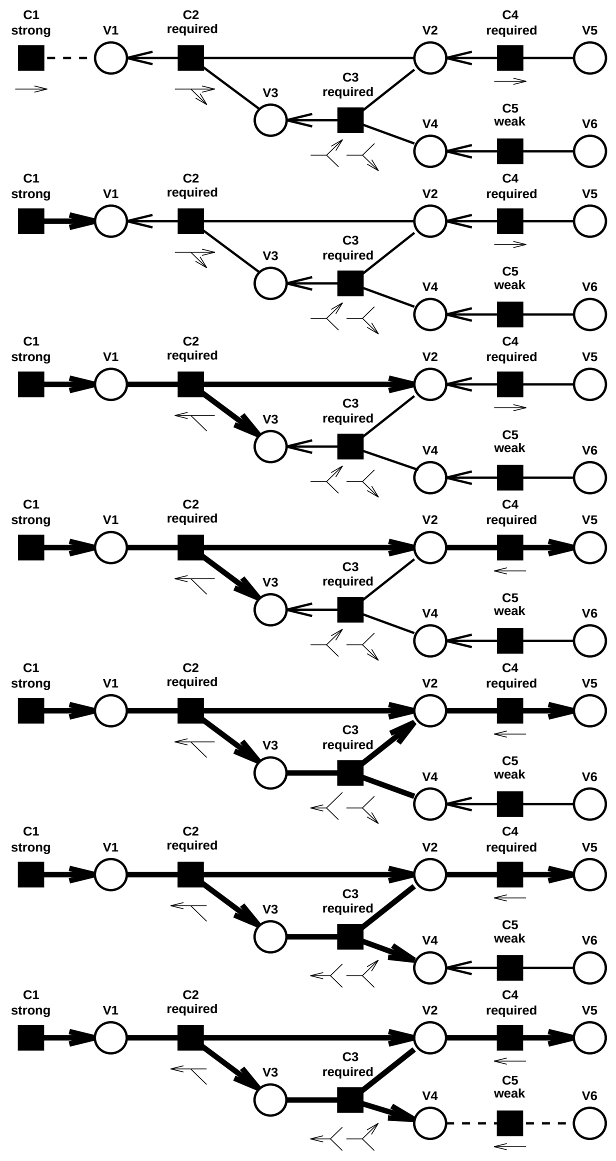
The initial SkyBlue method graph is empty and LGB. Every call to `add-constraint` or `remove-constraint` leaves an LGB method graph. Therefore, the current method graph must be LGB whenever `add-constraint` or `remove-constraint` is called. This fact can be used to avoid collecting and trying to enforce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

Whenever `add-constraint` adds a constraint `cn`, it is impossible for it 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` removes an enforced constraint `cn`, it is impossible to enforce any unenforced 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 "redetermined variables" whose determining constraint



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.

We have to process $C3$ by choosing another method. Suppose we try the one 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 3: Constructing an mvine. Methods in the mvine are drawn with thicker lines.

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 newly-redetermined variables. As each of these constraints is processed (it is enforced, or it is determined that it cannot be enforced) it can be removed from the set. When the set is empty there are no more unenforced constraints that can be enforced.

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-enforced constraints, and methods downstream of redetermined variables.

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 this method. It will be possible to complete the mvine below these 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* [4]. 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 other 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.[3]

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.

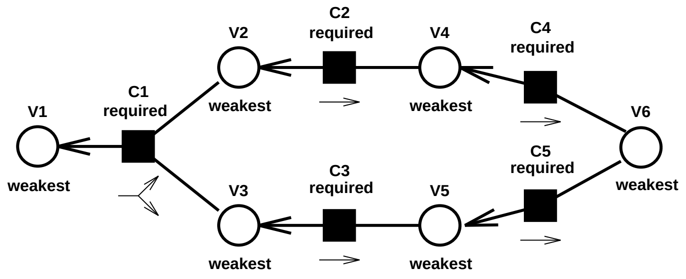


Figure 4: Method graph with a possible conflict.

There is a problem with using walkabout strengths in SkyBlue because methods may have multiple outputs. Consider the method graph of Figure 4. 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 for $C1$ 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 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. The use of 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 is done by processing all of the

[3] 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.

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.

SkyBlue uses the modified definition of walkabout strengths to simplify the processing of cycles. If the method graph contains directed cycles, it is not possible find a topological sort for the constraints. The walkabout strengths for variables in the cycle could be calculated by examining all of the constraints in the cycle, but this would require non-local computation. Instead, SkyBlue chooses a selected method in the cycle and calculates 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.

5.4 Comparing Performance Techniques

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 performance techniques described above.

local	*walkabout*	*time*	*number mvines*		*number*
collection	*strengths*	*(seconds)*	*attempted*	*constructed*	*backtracks*
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 5: Measurements collected while executing a sequence of 10000 constraint operations in SkyBlue with different performance techniques enabled.

Figure 5 shows the timing results when executing the sequence with four different configurations of the SkyBlue algorithm. 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 cases. 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. 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.

6 SkyBlue Applications

SkyBlue is currently being used as the constraint solver in Multi-Garnet [8], a package that extends the Garnet user interface construction system [6] with support for hierarchies of multi-way constraints. SkyBlue is also currently being used as the constraint solver in an implementation of the Kaleidoscope language [3] and as an equation manipulation tool in the Pika simulation system [1].

6.1 Multi-Garnet

Garnet is a widely-used user interface toolkit built on Common Lisp and X windows [6]. However, Garnet only supports one-way constraints, all of which must be required (no hierarchies). The Multi-Garnet package uses the SkyBlue solver to add support for multi-way constraints and constraint hierarchies to Garnet [8].

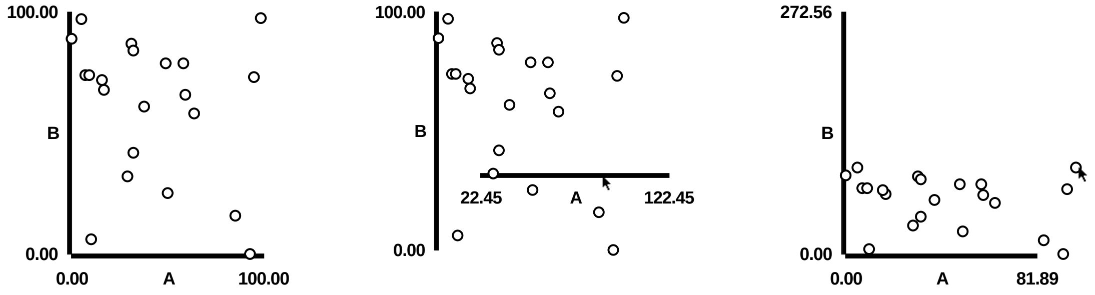


Figure 6: Three views of a scatterplot built within Multi-Garnet: The initial scatterplot, the initial scatterplot after moving the X-axis, and the initial scatterplot after scaling the point cloud by moving a point.

Figure 6 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 the relationship between the screen position of each point, the corresponding data value, 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 exploits many of the features of SkyBlue. SkyBlue resatisfies the constraints quickly enough to allow continuous interaction. The different interactions (move axis, move point cloud, scale point cloud, etc.) are defined by adding weak stay constraints to specify variables that should not be changed. Multi-way constraints allow any of the scatterplot points to be selected and moved, changing the axes data. This changes the positions of the other points, reshaping or moving 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 numbers displayed 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 without maintaining some of the relationships using other mechanisms in addition to the Garnet constraint solver.

6.2 The Pika Simulation System

SkyBlue is being used as an equation manipulation tool in a version of the Pika simulation system [1]. Pika constructs simulations in domains such as electronics or thermodynamics by collecting algebraic and differential equations representing relationships between object attributes. For example, in a simulation of an electronic circuit, one equation would relate the voltage across and the current through a particular resister. Pika processes these equations and passes them to a numerical integrator that calculates how the object attributes change over time.

Pika uses SkyBlue to manipulate the collected equations. Each equation is expressed as a SkyBlue constraint with one method for each possible output variable. SkyBlue chooses one method from each constraint so that no two constraints select the same output variable, and topologically orders the selected methods. Pika uses the ordered list of selected methods to set up the numerical integrator. Note that Pika does not use SkyBlue to maintain the constraints (equations) directly, but rather uses it to process the equations for the numerical integrator, which will maintain the equations during the simulation.

During equation processing, Pika takes advantage of SkyBlue's support for constraint hierarchies to influence the methods selected. There may be many possible ways to directionalize a given set of equations, leaving different sets of variables constant. Within the simulation, it may be preferable to keep some variables constant over others. This is represented by adding weak stay constraints to variables that should remain constant. SkyBlue will choose an equation ordering that leaves these variables constant, if possible.

Pika uses SkyBlue's facilities for incrementally adding and removing constraints to update the sorted list of equation methods as equations are added and removed. This may occur while the simulation is executing. For example, when the temperature of a container of water increases and it starts to boil, a different set of equations describing its behavior is activated.

Often there are cycles in the sets of selected methods produced by SkyBlue. Currently, Pika handles these cycles by extracting the equations in the cycle, and passing them to an symbolic mathematics system which tries to transform them to a non-cyclic set of equations. Pika replaces the cycle of equations by the reduced equations, SkyBlue (incrementally) updates the constraint graph, and Pika processes the new ordered list of selected methods.

Acknowledgements

Thanks to Alan Borning, Ralph Hill, and Brad Vander Zanden for useful discussions and comments on earlier versions of 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. In *Proceedings of the National Conference on Artificial Intelligence*, 1993. To appear.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] Michael Sannella. The SkyBlue Constraint Solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993.

[8] 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.

[9] 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.