Constraint Graph Visualization

Anthony Mak

Abstract—This paper presents some ideas for visualizing constraint graphs. The motivation for constraint graph visualization is for better understanding and debugging of CP (constraint programming) programs. (Constraint graphs also have the potential for visualizing logic programs such as Prolog programs.) Two important issues in providing good visualization for these types of programs are to find good visual metaphors for various program elements and to be able to handle large graphs. This paper will present some metaphors for visualizing the static and dynamic aspects of CP programs. It will also discuss some issues in handling large (constraint) graphs, such as how to use hierarchical clustering to improve the visualization.

Index Terms—Dynamic graph visualization, online hierarchical clustering, constraint programming.

1 Introduction

CP (Constraint Programming) visualization is about visualizing the execution of constraint programs. CP differs from other programming languages in that they do not specify a sequence of steps to execute but rather the conditions(constraints) under which a solution must hold. CP visualization allows users to better understand their constraint programs and for implementors to better understand their solvers. Visualization can be used for correctness debugging which is about checking whether one's program executes unexpectedly, and for performance debugging which is used for finding performance bottlenecks and optimize one's programs.

One common way to visualize the execution of a CP program is in form of a search tree in which a node denotes a value has been assigned to a variable, and a branch points to the variable assigned next by the solver chronologically. It has the benefit of giving an overall/historical view of the execution. Another way to represent a search tree is in form of a treemap. In contrast, an adjacency matrix or co-occurance matrix give a static view of a problem by displaying the variables and their current domains in a table. Both tree and matrix representations can be in 2D or 3D. In order to limit the size of a search tree or a matrix, a subtree can be represented by a single node with special color e.g. to represent a subtree with no solution, and use a number instead of the size of a column to represent domain size in a matrix [4]. Another common way to represent a CP problem is as a graph, and Muller mentioned four different views for visualizing a constraint graph: Propagator Graph View, Single Propagator Graph View, Variable Graph View and Single Variable Graph View [14]. There are also other visualization methods such as kaleidoscope, multiple attribute pareto and lattice visualization mentioned in [11].

There are other visualization techniques which may be useful for CP visualization. The fisheye (or hyperbolic) technique allows large amount of elements to be displayed in a small screen space while at the same time focus the user's attention to the centre of the graph which can be set to display the most important data at the time. Comic strips are consecutive frames that are viewed at the same time to facilitate user understanding of algorithmic steps [3]. Miraftabi attempted to use agents for increasing user friendliness in program visualization in his paper [13].

The next section gives a definition for a constraint graph. Section 3 gives a brief description of the G12 platform and describe the relationship of this work with it. Section 4 will give a brief survey of softwares for constraint programming visualization. Section 6 will talk about visual metaphors for representing static and dynamic aspects of a CP program. Section 7 will explore into issues and presents solutions in visualizing large graphs, such as, clustering. Then, we will give a brief description of the software we have built for visualizing constraint graphs in section 8.

• *Anthony Mak is with Australian National University and most of the works in this paper were done when he was a research programmer with National ICT Australia, E-mail: anthony.mak@iname.com.*

2 Constraint Graph

A constraint graph is a visualization of a constraint network. A constraint network(CN), CN = $\{$Variables, Domains, Constraints$\}$, consists of a set of variables and their associated domains, and a set of constraints [6]. Each constraint is a relation defined on a subset of the variables. A constraint graph can help better understand the underlying constraint problem by visualizing its structure, for example, to see the symmetries of a problem. A constraint graph can also help performance debugging by visualizing a CP solver's dynamics through animations on the graph.

The two main elements in a constraint graph visualization are variables and constraints, and each can either be represented as a node or an edge depending on the views (see Figure 2). A constraint graph can be used to visualize a logic problem, a constraint programming problem, a SAT (propositional satisfiability) problem, or any relationship between a set of entities (any relation you can imagine!).

3 The G12 Platform

G12 [10] is a software platform being developed for solving large scale industrial combinatorial optimisation problems using constraint programming. It will be used to enable industries to exploit resources more efficiently; to support more efficient management of complex private and public utilities such as transportation, communication, power and water; and to support optimal and justifiable strategic decision making and investment. The project is split into four threads: building an expressive high-level modelling language (Zinc), building more powerful solving capabilities through various kinds of solvers hybridizations, a richer control language mapping the problem model to the underlying solving capabilities(Cadmium), and a richer problem solving environment.

In order to enable a richer development environment, some visualization tools G12 may include are constraint graph, search tree, algorithm visualization and an algorithm comparison tool.

4 Existing Software

DPvis [7] is a software we found that has constraint graph visualization. It can display interaction graphs (constraint graphs) and search trees. It is also possible to do time travelling to display previous graphs by clicking on the corresponding node on the search tree. Also they cleverly use edge colors and shapes to represent information about constraints. However, it visualizes a solver's operations in steps not in real time. Also, it is used for visualizing SAT problems not CP problems therefore it can only visualize variables with domain of size 2. There are other well known tools for CP visualization even though they do not not have constraint graph visualization. **Eclipse** is a constraint programming platform that provides many visualization tools. With the toolkit, it can display a search tree and various ways of variable

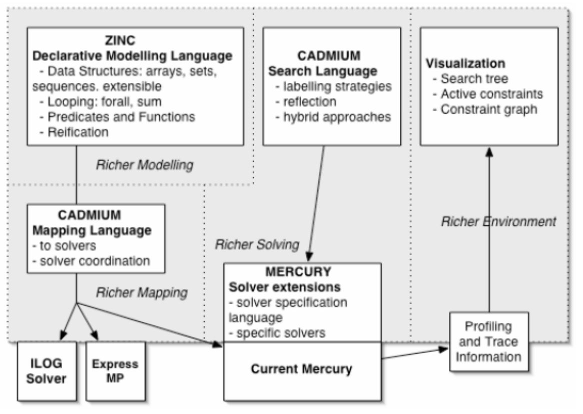


Fig. 1. The G12 Constraint Programming Platform

visualization in the form of gantt chart, matrix and desktop metaphors. **CLPGUI** [9] is developed by INRIA and has search tree visualization and finite domain variables views in 2D and 3D. Interestingly, it uses annotations to specify not only what parts of a program to visualize but also to set up the gui controls. **Oz Explorer** [19] is the visualization package that comes with the Mozart constraint programming platform. It has search tree visualization. What is unique about it is that this visualization tool can support user guided search, and it makes use of recomputation distance to trade space for time to reduce memory usage. The **VIFID/TRIFID** [5] tools have some novel ways of representing domain sizes and evolution of finite domain variables and on how to abstract variable values and constraints. **ILOG** has some visualization APIs however they are mainly graphic APIs for drawing general graphics not particularly for visualizing logic or constraint programs.

5 Design Criteria

We follow the following guidelines when developing our software:

1. Specific to constraint problem
2. Scalable
3. Intelligent
4. Generic
5. Variety of views/metaphors
6. Use state of art tools, methodology and framework
7. 2D

We decided to make the software specific to visualizing constraint problem. Although it is possible to create a general visualization platform, but it is better to do a few things well rather than a tool that can do many things but not so well. Secondly, it is very important for the tool to be highly scalable because often there are many great visualization tools out there but when swarmed with large amount of data their performances suffer to an unusable state. We want the visualization tool to be intelligent, for example by using machine learning techniques for clustering and layout, and agent to help navigation on the graph and GUI. Although this may sound ambitious, it is important because visualization is about revealing information to users but it does not help to swarm users with too much graphical information and users often do not know what they should look for when diagnosing a problem. AI techniques have the potential to adapt to users' needs and filter out all but relevant visual events. The software needs to be generic because we do not want it to be usable only with our CP solvers and it should be simple for the open communities to integrate with their systems. The software needs to provide a variety of views and visual metaphors because multiple metaphors of the same data reveal extra information about the data to the user and these extra views may allow the users to spot significant events and patterns in the problem. We decided to visualize a constraint graph as a 2D but not a 3D graph because although a 3D graph allows us to do rotation and other interesting manipulations in 3D, a 3D visualization is only necessary if there is a 3rd dimension in the data we are trying to visualize. Also, using 2D allows us to scale up our application to much larger datasets so more nodes can be visualized on the graph.

6 Graphic Metaphors for CP Elements

In order for the visualization to reveal useful information to the user, there need to be good mappings (metaphors) from what one is trying to visualize (the problem data) to how one is going to visualize (the visual elements). There are two aspects of a constraint problem that we would like to visualize: the static part which includes visualizing the structure of the problem and the types of the elements in a problem, and the dynamic part which is about visualizing how the problem is changed as it is being solved by a constraint solver.

6.1 Visualizing Structure

6.1.1 Views

Some common views are variable view, constraint view, bipartite view and hyperplane view.

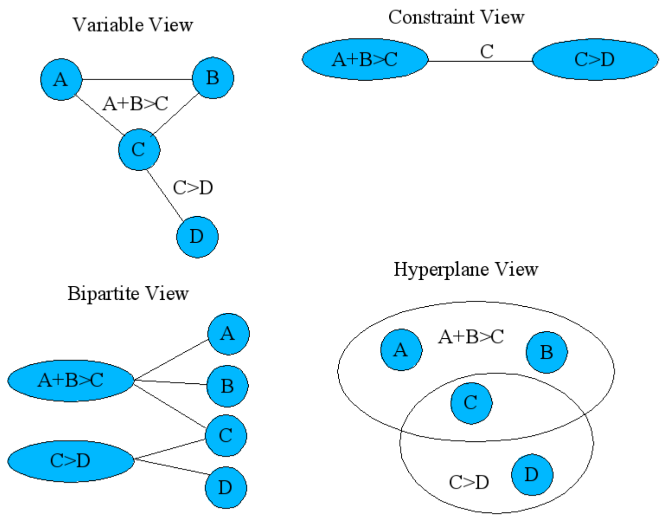


Fig. 2. Different ways of displaying a constraint graph

In variable view, a variable appears as a node on the graph and an edge joins two nodes if there exists a constraint between them. In constraint view, a constraint appears as a node, and an edge joins two nodes if the constraints they represent have a common variable. In bipartite view, both variables and constraints appear as nodes and a variable node is joined with a constraint node if the constraint contains the variable. Note there can never be edges between two variable nodes or two constraint nodes. In hyperplane view, a variable appears as a node, and variables that are common to a constraint will be covered by a plane. The other elements in a constraint graph are the domains of each of the variables. The dynamic aspects of a constraint graph are due to propagation, backtracking, variable satisfaction and domain reduction inside a solver which causes the graph to change.

6.1.2 Visualizing variables and constraints

The two main elements in a constraint graph are nodes and edges. Nodes can have different sizes and colors, and edges can have different length, thickness and colors. Thus we can use them to represent different aspects of variables and constraints and here is one way to do it:

- Node Size: It can be used to represent variables' domain sizes in which a larger node represents a variable with larger domain size.
- Node Color: It can be used to represent whatever property or type a variable (or constraint) has.

- Edge Thickness: It can be used to represent how constrained two variables are. For example, the more constraints there are between two variables, the thicker is the edge between them.
- Edge Color: It can be used to represent the type of a constraint.

6.2 Visualizing Change

6.2.1 Assignment, Domain Change and Backtrack

As a problem is being solved by a solver, the corresponding constraint graph which represents the problem can also be changed. A variable can be assigned or satisfied. The domain of a variable can be reduced due to propagating constraints. And also a solver can backtrack which can lead to parts of the problem become unsolved again. The most straightforward presentations of these dynamics are to remove a node (and the connecting edges) when the variable it represents is satisfied, reduces the size of a node when the variable domain size reduces, and to add the corresponding nodes and edges back into the graph and redraw when the solver backtracks.

6.2.2 Propagation

We think the best way to visualize constraint propagation is to use animation. As a result of propagation, variables can become satisfied which will result in the corresponding nodes being deleted from the graph, and in other cases the domains of variables can be changed which will result in the corresponding nodes' sizes being changed. Therefore, in order to visualize this propagation effect using animation, a node could ”blink” before it is removed or its size changed.

7 Visualizing Large Graphs

In general, graph layout is a NP-complete problem and has exponential complexity. For online visualization where the graph can change, it can become very computationally expensive if updates of the visualization is required often. Also memory requirement can be a problem (although to a lesser extend) because each node may require additional bookkeeping information. When the number of elements on screen is more than one thousand, there may be problem fitting all elements on the screen. And when a lot of nodes are displayed simultaneously it will become very difficult to comprehend the graph because of a lot of node overlappings and edge crossings. Therefore, the key issues seem to be using an efficient layout algorithm and reducing the number of elements on screen if we want to display large graph effectively. In [16] Novák gives a good description of the various aspects of visualizing a large graph.

7.1 Navigation

For large graphs, navigation can be problematic since there can be more nodes than could be fitted into a screen. The common ways to handle this problem are to make use of scrollbars, zooming or an overview pane. An overview pane is a scaled down version of the whole graph, and the user can jump to a particular position on the main graph by clicking on a corresponding location on the overview pane.

7.2 Filtering

There are many motivations to perform filtering on a graph. When there are too many elements on screen, it will be computationally expensive and thus takes a long time to redraw a graph. Of equal importance, it will cause much edge crossings and reduces human comprehension of a graph. Another reason for filtering maybe simply the user wants to focus on one type of nodes on the graph. For example, a graph can be filtered based on the attribute value of a node, such as, the subproblem that the variable it represents belong to. Or to hide some of the edges base on some criteria, for example, do not display edges that represent global constraints because they connect all the nodes. Another approach is not to display the whole problem initially but to grow the graph. This mean to show only the parts of the problem that has been processed (by the CP solver).

7.3 Abstraction

Graph abstraction can reduce the number of nodes on screen but also allow the user to view a problem at different levels of detail. For example, there can be an abtraction level where each node in the constraint graph represents an element at the CP modelling language level, and another abstraction where each node represents a variable in a solver as the original high level CP representation is grounded to a lower solver specific problem representation in order for the solver to be able to understand the problem.

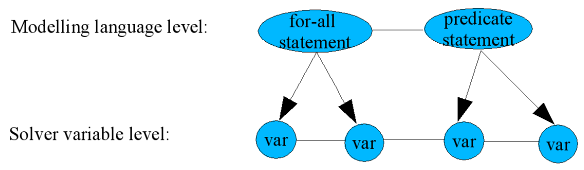


Fig. 3. Different levels of graph abstraction

7.4 Incremental Update

For a dynamic graph where nodes and edges can be added and deleted over time, it is necessary to update the visualization periodically. However, calculating the layout for the whole graph every time will be computationally expensive. In this case it is better to use a layout algorithm which supports incremental update. In particular force-directed layout algorithms are suitable for this purpose. (Although we found edge crossing is problematic in force-directed algorithms and may explore other layout algorithms in the future.) For online visualization, new graph data can come continuously and we found it is best to freeze the thread for layout each time before a batch input of graph data for performance reason.

7.4.1 Force-Directed Layout

Force-directed layouts work by using repulsion and attraction forces. Nodes that are highly connected are attracted to each other. And nodes exert repulsion forces on each other when they are close. The forces affect the layout incrementally such that at each timestep each node moves a little in a direction depending upon the sum of the attractive and repulsive forces exerted on it from its neighbours. Because of this incremental aspect and also the fact that when new nodes are added to a graph the positions of the existing nodes do not have to be recalculated, force-directed layout algorithms are suitable for online visualization. By default new nodes are added in random positions in force-directed algorithms, however this is not suitable for online visualization since new nodes can be added continuously. It will take a long time for new nodes to travel to their optimal/stable positions and also the large displacements as they move to their optimal positions will cause much visual disruptions. In order to remove visual disruptions, the idea is to position each variable or constraint node close to their neighbours initially. Algorithm 1 and 2 show how to do this in the variable view and constraint view.

Algorithm 1 Initial position for new (variable) node v

```
Require: At least one constraint that contains the variable $v$
Ensure: $(x, y)$ position for new node
  if no constraint exists which contains $v$ then
    wait until at least 1 such constraint is added to graph
  end if
  for all constraint $ci$ which contains $v$ do
    for all variable $vi$ in $ci$ do
      $sumx \Leftarrow sumx + vi.x\_position$
      $sumy \Leftarrow sumy + vi.y\_position$
      Increment $num\_of\_nodes$ by 1
    end for
  end for
  return $(sumx/num\_of\_nodes, sumy/num\_of\_nodes)$
```

Algorithm 2 Initial position for new (constraint) node *c*

```
Require: No precondition
Ensure: $(x, y)$ position for new node
  for all variable $vi$ in $c$ do
    for all constraint $ci$ which contains $vi$ do
      $sumx \Leftarrow sumx + ci.x\_position$
      $sumy \Leftarrow sumy + ci.y\_position$
      Increment $num\_of\_nodes$ by 1
    end for
  end for
  if $num\_of\_nodes > 0$ then
    return $(sumx/num\_of\_nodes, sumy/num\_of\_nodes)$
  else
    return random position {no neighbouring node}
  end if
```

7.5 Clustering

There can be 2 types of clustering: placing nodes that are similar (according to some measures) in close euclidean distance; replacing each set of nodes that are similar or close with a single node on the graph. The first type can be used to reveal more structures in the graph. The second type has the benefits of reducing the number of nodes on the screen thus increasing human comprehension of the graph and also reduces cpu time for rendering. A requirement for the clustering to be used in our tool is that the clusters need to be updated dynamically because the underlying graph can change over time.

7.5.1 Generic Clustering using a Machine Learning Approach

The original motivation for using clustering in constraint graph visualization was to reduce the number of nodes on screen since a CP problem can contain a huge number of variables and constraints which could be hard for a human to comprehend. The idea is to group the nodes into different subsets by some criteria using some form of machine learning algorithm and then replace each subset with a single node on the graph. We have explored the k-mean and hierarchical clustering algorithms, and the later seems to be more suitable since it can better utilize the structural information in a constraint graph. In the basic hierarchical clustering algorithm, all nodes are set as singleton clusters initially. At each step, the two closest clusters are joined into a cluster which is represented by a subtree having the two clusters as its children. This process repeats until all the clusters are joined to form a single tree. The root represents a cluster that contains all the nodes whereas the leaf nodes represent all the nodes as individual unclustered nodes. In between the two levels, we can get different granularities of clusters. However, the basic algorithm is not incremental since it assumes all the nodes are already present when performing the clustering. Also, it is not an online method since it cannot update the clusters when subsequently nodes are deleted in a later stage in time.

For our purpose, we need an incremental/online clustering algorithm because re-applying the hierarchical clustering algorithm to the whole graph whenever a change occurs would be very expensive computationally. In the paper [2], the algorithm can cluster incrementally without recalculating the whole dendrogram each time a node is added. When a node is added to the dendrogram, the parts of the dendrogram that are affected will be cut and the naive hierarchical clustering algorithm will be applied until all the nodes are connected as a single tree again(we call this process Repair). Two clusters can be affected because sometimes after adding a node, the distance between them can become shorter or longer. Although the paper does not mention node deletion, we believe it can be treated in a similar way by first removing the node from the dendrogram and investigating which parts of it are affected and then repair the dendrogram as mentioned previously. The main bottleneck of hierarchical clustering algorithms is finding the closest clusters, Conga Line [8] is an efficient method to solve this problem. We will use the Basic Conga Line algorithm instead of the FastPair method, because although FastPair is faster in experimental results it does not merge its data structures as the number grows and this may not be suitable for dynamic graphs where nodes can be deleted continuously.

The distance function we use assumes an infinite/undefined value if there is no edge between 2 vertices, otherwise it is given in the following formula where *s* is the strength of the edge and *c1* and *c2* are the connectivities (in-degree and out-degree) of the two vertices. This formula has been provided by Olivier Buffet.

$$d = \frac{1}{s \times (c1 + c2)}$$

Algorithm 3 An Online Hierarchical Clustering Algorithm for Graph Clustering

```
Require: Graph data
Ensure: A dendrogram (contains clustering information) being maintained dynamically
  loop
    (Add or delete a node)
    if initial clustering has not been performed then
      if $NoOfNodes > N$ then
        $PerformInitialClustering$
      end if
    else
      if node added then
        $NoOfClusters += 1$
      end if
      Update Conga Line data structure
      Mark the node as added or deleted
      if $NoOfClusters > N$ then
        for all nodes $vi$ in the deleted list do
          Remove $vi$ from dendrogram
          Repair (see 7.5.1 )
        end for
        for all nodes $vj$ in the added list do
          Add node $vj$ to dendrogram (using Conga Line to find closest cluster to $vj$)
          Repair (see 7.5.1 )
        end for
        (Update clusters visually using dendrogram)
      end if
    end if
  end loop
  PerformInitialClustering:
  - Initialize Conga Line data structure
  - Apply basic hierarchical clustering (using Conga line to find closest clusters at each step and using equation A for distance function)
  - (Update clusters visually using dendrogram)
```

7.5.2 Using Information inside the Problem

Very similar for the purpose of abstraction, we can utilize the information about which array, predicate, function or quantifier a variable belongs to in a problem for enhancing the clustering. For example, we can place the variables which are in the same array in the original (Zinc) problem close to each other in the visualization, or to replace all the nodes that represent those variables by a single node in the graph (see Figure 3). We plan to explore this area in the future by finding out what are the best mappings from constructs in the modelling language level to a cluster.

8 IMPLEMENTATION

One of the earliest design decisions was whether to use 2D or 3D to visualize a constraint graph. We developed a prototype in 3D because we thought as computers become more powerful CPU bottleneck will not be a problem, and the visualization in 3D does look aesthetically pleasing. However, in the end we decided to use 2D because it allows us to scale up the application to visualize more elements simultaneously and 3D is only meaningful if there are three dimensions in the data to visualize. We use a Java visualization API called *Jung* because it is open source and supports incremental update. Incremental update means the visualization does not redraw the whole graph every time it is modified. Because we want to dynamically visualize data from a solver in real time, it is important to update incrementally for efficiency. In the current version of our software, we use a simple force directed layout because force directed layout algorithms are suitable for handling dynamically changing data. Also for efficiency, the viewer uses a threaded design and uses different threads for different tasks such as rendering and command parsing.

Figure 5 shows the Constraint Graph Viewer visualizing the hgen8 problem created by Edward Hirsch in Variable View dynamically . This problem has been chosen because it was the smallest unsolved problem in the 2002 SAT competition. Because the G12 platform is still under development, we tested our concepts by integrating the Dew_Satz [1] SAT solver with the viewer by modifying the solver to output the commands required by the viewer. Figure 6 and figure 7 show the viewer in Constraint View and Bipartite View respectively. It is possible to change between these views by clicking the View button. Node size can represent domain size or change frequency (such as how often the domains has been changed or backtrack frequency). In these examples, larger node sizes represent more backtracking happened on those variables.

8.1 On/Offline Visualizations

Graph visualization can have 3 types depending on the responsiveness of the visualization. In our tool, we will use online visualization and perhaps in the future allow interactivity as well.

- Offline Mode: The graph is only visualized after all the data has been computed.
- Online Mode: Nodes and edges are added incrementally.
- Interactive Online Mode: It is the same as Online Mode but interactions and modifications via the visualization will also change the states of the corresponding program.

8.2 Architecture

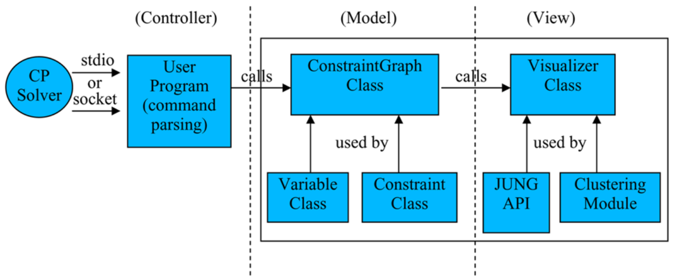


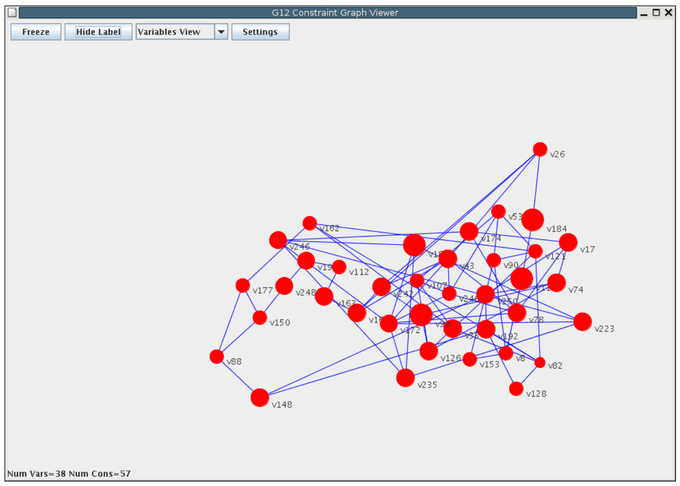
Fig. 4. Architecture of Constraint Graph Viewer

We decided to adopt a MVC(model-view-controller) design in the beginning and the API contains two main classes: ConstraintGraph(model) and Visualizer(view). The idea is to hide complexity from the application programmer, who wants to use the API to visualizer a constraint graph in their own programs, by requiring the programmer only to modify a ConstraintGraph object and then a Visualizer object that is linked to it will automatically perform the layouts and renderings. The controller component in our case is simply a main program written by the application programmer who is responsible for parsing input commands and subsequently initiating the ConstraintGraph and Visualizer objects. A MVC design will also facilitate the case when we choose to have multiple models and views in the future, for example, a tree view, a graph view and data models in the format of a graph or some other formats. Figure 4 shows the main classes in the API.

8.3 Command Syntax

The constraint graph viewer takes input commands as plain text either from the standard input or from a network socket. It is important to note that before a constraint is added to the graph all the variables it refers to should already have been added, otherwise the command for adding that constraint will be ignored. To visualize a graph, apply a mix sequence of commands to add variables and constraints. For the ADD VAR command, it is optional to provide a description for the variable(var_desc) and the domain size in integer. For the ADD CON command, it is optional to provide the constraint's description and id and when an id is not provide the constraint will automatically be assigned as "CONX" where X starts from 0. The SLEEP command will tell the system not to process new commands for n seconds (new commands will be queued temporarily). Because the viewer is designed for online visualization, commands for adding and deleting variables and constraints can be sent to it continuously and it will update the rendering with the new data periodically. The visualization can update anywhere in between a sequence of add and delete statements, however, for some problems this would cause the viewer to visualize the problem in an inconsistent state. In such cases, one can encapsulate the ADD and DEL commands between a pair of BATCHBEGIN and BATCHEND commands such that the graph data object will still be updated but the visualization will only be updated after the BATCHEND command. It is possible to have space inside an id or description by enclosing it with a pair of double quotes(").

- Add a variable: ADD VAR *var_id var_desc (domain_size)*
- Delete a variable: DEL VAR *var_id*
- Add a constraint: ADD CON *con_id con_desc* [*var_id1 var_id2* ...]
- Delete a constraint: DEL CON *con_id*
- Change domain size: CHG DOMAIN *var_id (integer)*
- Sleep (and do not update graph with new data): SLEEP *seconds*
- Batch update: BATCHBEGIN, BATCHEND



Num Vars=38 Num Cons=57

Fig. 5. Constraint Graph Viewer (Variable View)

9 Agent

When a CP solver is solving a large and complex problem, it can often take a long time to solve and can generate enormous amounts of data. Because of these two reasons, it is often difficult for the person solving a problem to comprehend what are the important, relevant and unusual events. Also because of the large size of data, navigating through the graph may be difficult. Agent technology has a definite possibility to help solving these problems by filtering out irrelevant data and help with navigation on the GUI. As a CP solver is running there are many low level events occurring that can cause the graph to change and users

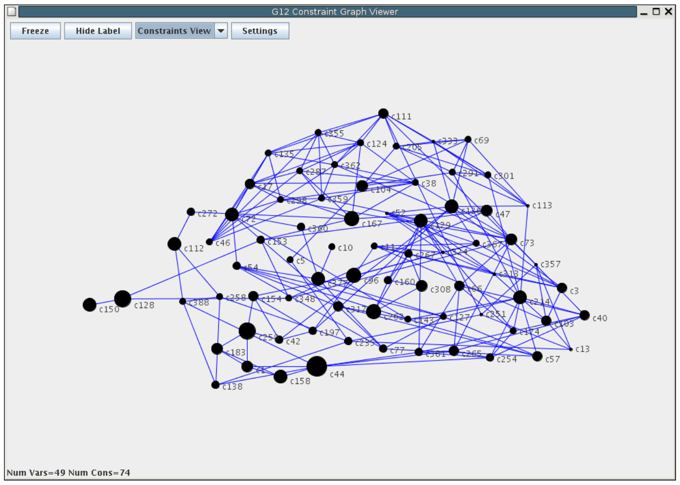


Fig. 6. Constraint View

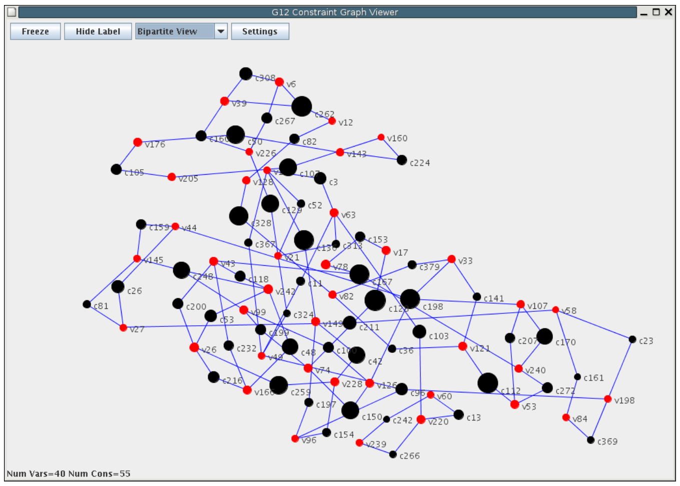


Fig. 7. Bipartite View

can perform high level actions to change the view of the graph in order to find what interests them. One idea is for the agent (see Figure 8) to automatically filter out irrelevant information and to learn the mapping between low level events and high level actions, so that these actions may be done automatically in order to maximize the amount of useful information about the problem revealed to the user.

10 Future Directions

Having explored some basic ideas for graph visualization for constraint programming, we would like to do more experimentation and perhaps implement the following features. One idea is to use information inside a CP problem to improve on the layout of a graph such that the graph can reveal more useful information to the user. Currently, we use a simple force directed layout algorithm (spring layout), but there may be more modern algorithms that can both improve on the rendering speed and the aesthetics of the graph such as the conjugate gradient method. In our first attempt, we designed a data mining method for hierarchical clustering which we plan to incorporate in next version of our tool. We also wish to explore other machine learning methods for clustering and for user interfacing since they may generate interesting results. Few visualization packages make use of agent technologies, thus it could be interesting to implement an agent as described in the previous section to see what kinds of improvements it can bring in terms of user-friendliness and revealing more relevant and less irrelevant information to the users. We may also explore interactive online visulization if such needs arise in the future.

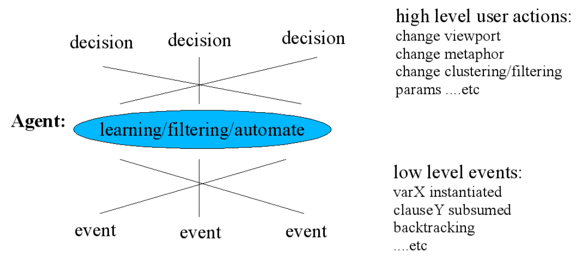


Fig. 8. Using agent to assist visualization

11 Conclusion

In this paper, we have explored some graphical metaphors for visualizing a constraint graph and also about using clustering to enhance the visualization. A tool was produced which implements the basic features mentioned in this paper. Although this paper and the tool is centered on visualizing CP programs, the ideas can equally be applicable on visualizing logic programs, SAT programs and any relationship about anything imaginable.

Acknowledgements

I wish to thank Olivier Buffet for his advice on how to use machine learning for graph clustering and providing the initial code for hierarchical clustering. I would also like to thank Andrew Slater for providing continuous advice on software design and the G12 platform. Anbulagan helped me to integrate the visualization tool to his SAT solver Dew_Satz and Mark Brown provided help for an initial integration into the G12 platform. I would also like to thank Tim Dwyer for giving me a lot of advice on various layout algorithms and graphical metaphors.

References

[1] Anbulagan and J. Slaney. Lookahead saturation with restriction for sat. In *CP*, pages 727–731, 2005.

[2] A. E. Arnaud Ribert and Y. Lecourtier. An incremental hierarchical clustering. *Vision Interface*, 1999.

[3] H. Biermann and R. Cole. Comic strips for algorithm visualization. Technical Report TR1999-778, 16, 1999.

[4] M. Carro and M. Hermenegildo. Some design issues in the visualization of constraint logic program execution. Technical Report CLIP1/97.1, 1997.

[5] M. Carro and M. V. Hermenegildo. Tools for constraint visualisation: The VIFID/TRIFID tool. In *Analysis and Visualization Tools for Constraint Programming*, pages 253–272, 2000.

[6] R. Dechter. Constraint Networks. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992.

[7] E.-M. Dieringer and C. Sinz. DPvis - a tool to visualize the structure of SAT instances. *SAT 2005 : international conference on theory and applications of satisfiability testing*, 2005.

[8] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.

[9] R. C. Francois Fages, Sylvain Soliman. CLPGUI: a generic graphical user interface for constraint logic programming, 2004.

[10] G12 Official Website - http://www.g12.cs.mu.oz.au/.

[11] D. Lalanne and P. Pu. Interactive problem solving via algorithm visualization, 2000.

[12] M. S. Marshall, I. Herman, and G. Melançon. An object-oriented design for graph visualization. *Software - Practice and Experience*, 31(8):739–756, 2001.

[13] R. Miraftabi. Intelligent agents in program visualizations: A case study with seal, 2001.

[14] T. Müller. Practical investigation of constraints with graph views. *Proceedings of the International Workshop on Implementation of Declarative Languages*, 1999.

[15] P. Mutzel and P. Eades. Graphs in software visualization - introduction. In *Software Visualization*, pages 285–294, 2001.

[16] O. Novák. Visualization of large graphs. Master's thesis, Czech Technical University in Prague, 2002.

[17] G. Roessling. *ANIMAL-FARM: An Extensible Framework for Algorithm Visualization*. PhD thesis.

[18] R. Sablowski and A. Frick. Automatic graph clustering. In *Proc. Graph Drawing, GD*, number 1190, pages 396–400, Berlin, Germany, 18–20 1996. Springer-Verlag.

[19] C. Schulte. Oz Explorer: A visual constraint programming tool. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, 1997. The MIT Press: Cambridge, MA, USA.