Graph Algorithm Animation with Grrr

Peter J. Rodgers and Natalia Vidal

Computing Laboratory, University of Kent, UK
{P.J.Rodgers, N.Vidal}@ukc.ac.uk

Abstract. We discuss geometric positioning, highlighting of visited nodes and user defined highlighting that form the algorithm animation facilities in the Grrr graph rewriting programming language. The main purpose of animation was initially for the debugging and profiling of Grrr code, but recently it has been extended for the purpose of teaching algorithms to undergraduate students. The animation is restricted to graph based algorithms such as graph drawing, list manipulation or more traditional graph theory. The visual nature of the Grrr system allows much animation to be gained for free, with no extra user effort beyond the coding of the algorithm, but we also discuss user defined animations, where custom algorithm visualisations can be explicitly defined for teaching and demonstration purposes.

1 Introduction

Grrr is a visual graph rewriting programming language [16,17]. It is general purpose, allowing the implementation of complex graph algorithms and has a visual view of graphs. We believe these factors make it a good system in which to code graph algorithm animation. Much of the work described here was initially designed as debugging tools for the initial Spider language and later Grrr language, but their wider applicability has encouraged us to extend the ideas and develop more general animation techniques.

We describe three algorithm animation techniques in this paper. The first technique is that of user defined emphasis, which has always been in our system in a limited fashion, as the programmer can use built in transformations to highlight chosen nodes and subgraphs of the host graph. Second, tools for animation have resulted from the recent graph drawing variation, Grrr, which allows nodes to be positioned at a geometric point. The movement of the node on the screen can be followed for better understanding of the progress of the algorithm. Third, we have recently implemented the automatic highlighting of subgraphs that have been matched in the host graph. This means that sections of the host graph that have been visited are shown.

The animation of node movement and the highlighting of matched subgraphs come for 'free', in that they can be used without any user input except specifying that animation is required. The user highlighting is for custom animation and requires a programmer to ensure that the correct section of the host graph is highlighted during the progress of the algorithm. The node movement animation can also be used to produce custom animation by the programmer specifying the preferred location of nodes for best comprehension. The three techniques given here can be combined as desired.

We do not claim any originality for our animation methods, but to our knowledge this is the first time a graph rewriting language has been associated with algorithm animation. We believe our system is very suited to animation because of its visual emphasis, and because the design has resulted in semantics which are useful for animation. The visual nature of the programs in Grrr means there is no 'impedance mismatch' that might occur when defining a textual program for visual execution.

Grrr allows the execution of rewriting to be viewed on the screen as it happens. This step view of rewriting is an important part of the animation process as it allows highlights and movement to be shown as the algorithm progresses. The user can also step through a program manually, taking their own time to observe the execution.

Another feature of Grrr that helps with algorithm animation is the ability to hide subsections of the host graph so that only the data structures that are being manipulated or which are relevant to the user can be seen, and so the housekeeping underneath can be hidden to avoid confusion.

Algorithm animation in Grrr has two main roles: firstly, the original intended role was to aid the debugging of programs written in Grrr, so that graph match highlighting can indicate where the rewrites are operating, or showing node movement can indicate the way nodes are manipulated in graph drawing. The second role is that of an educational nature to visualise algorithms in order to teach them, the standard motivation behind algorithm animation systems.

The algorithm animation in Grrr is entirely restricted to graph highlighting and movement, whereas many dedicated animation systems have facilities for more abstract representation, using extra graphics and shading to aid visualisation [2,3,12,20]. This type of animation is not easy to define with the graph rewriting described in this paper. However, we note that several systems allow similar types of animation to the graph oriented approach provided in Grrr, e.g. [6,9,10,13], so we feel we are justified in restricting our system. We must note that most animation systems are primarily designed for teaching algorithms, but studies have thrown doubt over the usefulness of algorithm animation as a learning tool [8,19].

2 Programming with Graph Rewrites

Grrr is a graph rewriting programming language. It computes by rewriting a host graph according to user defined transformations. A key advantage to this approach is the combination of computational completeness and visual view of both the graph being rewritten and the transformations that rewrite the graph. This combination, along with features such as serial rewriting and serial trigger initiation make Grrr a potentially useful system for inherently visual, but complex tasks such as graph drawing and algorithm animation.

Previous graph rewriting languages include GOOD [15], Progres [18], Dactl /MONSTR [7,1] and Δ-grammar programming [11], each of which has a unique interpretation of programming with graph rewrites. These graph rewriting languages vary in several important aspects: the type of host graph that is to be rewritten may be any graph, or it may be restricted by disallowing duplicate nodes or arcs, or indeed may have some underlying hierarchical structure; the graph may be rewritten in a serial or parallel manner; the transformations may be initiated in a number of ways; the transformations may be applied in serial or parallel; and there are alternative ways that the user can specify the transformations. Typically, the systems have general programming features, but are aimed at specific applications.

Grrr is a development of the Spider graph rewriting programming language. Spider is a prototype system for database programming. Modified, it forms the basis of Grrr, a general purpose programming language, which we are using to explore the notions of visual graph drawing. Graph drawing has been seen in graph rewriting systems previously [4,21]. Our current project is attempting to demonstrate that programming a wide range of graph drawing algorithms is feasible in a graph rewriting visual language. To achieve this we are in the process of producing hierarchical, force directed and planar graph drawing algorithms in Grrr. We note that Grrr is still under development and future changes both to the semantics and implementation are likely.

Grrr features serial trigger initiation in a two graph rewrite specification method, the difference between the LHS and RHS in a rewrite indicate the changes to be made to the host graph. The rewrites are contained in transformations. When a transformation is called the LHS graphs are tested against the host graph in the top down method until one matches, that is they are tested in order of presentation in the transformation. We use this approach rather than alternatives such as 'best fit' (i.e. classifying LHS by how specific they are) because of its success in analogous textual rule based systems such as logic and functional languages. There is also the problem of interpreting best fit in a graph based system.

The transformations are called by trigger nodes (shown with a rectangular shape) in the host graph, and only one trigger is initiated at a time. This is achieved by a newest first execution order for the triggers in the graph. Only one LHS graph is matched at a time, and the rewriting occurs in a serial manner using a deterministic subgraph matching strategy that relies on the nodes and arcs in the graph having an internal ordering. The serial nature of Grrr aids algorithm animation as a parallel rewriting or trigger initiation strategy could hide that the progress of the algorithm.

The data graph (that is the part of the host graph that holds application data, usually shown with round nodes) can be distinguished from the part of the graph that holds associated information (that is, information derived from the data graph and information concerning execution, usually shown with oval nodes). A node type specified in a rewrite will only match with that node type in the host graph.

Grrr allows arbitrary graphs to rewritten. To avoid ambiguity when deleting or adding primitives, duplicate labels which appear in the LHS or RHS must be identified by the user. The identifier is an integer superscripted to the node label.

Current modifications to the rewriting process include attractor nodes, negatives, once only nodes and single match rewrites. For example, Fig. 2 shows a transformation with a single match rewrite, indicated by a shaded background. The LHS of this rewrite will match once and only once when the associated trigger node is called. After matching, the single match rewrite will be ignored when further calls of the particular trigger node are made.

Fig. 2 also shows the use of negative primitives in LHS graphs. Here, the second rewrite contains a negative node and arc, indicated by the primitives having thick outlines (not to be confused with the highlighting of nodes in the host graph). For this LHS to match, the positive part of the graph must match, and there must be no corresponding match of all the LHS including the negatives.

Fig. 1 illustrates the use of attractor nodes, with the RHS of the second rewrite having the attractor node 'Minus', indicated by a shaded background. Attractor nodes pick up any dangling arcs after a rewrite has been performed. Normally such dangling arcs are deleted from the host graph.

Not shown in the examples is the use of once only nodes in LHS graphs. Here a node can be specified to be a once only node, and such a node will match no more than once with each corresponding node in the host graph. This allows for simple iteration through a graph.

To perform mathematical calculations and to express geometric operations in Grrr, there are many built in transformations. Many of the built ins are atomic, however others have been added for efficiency reasons.

Often the progress of Grrr programs is expressed in terms of number of steps. Each step is an execution of a trigger node, and can be considered much like the execution of a single instruction in a traditional textual programming language.

3 Illustrations of Use

Here we give all or part of three programs to illustrate the varied nature of the animation in Grrr. The transformations that make up the programs contain highlights in order to distinguish between nodes with different semantic meanings, which should not be confused with the highlights shown in the host graphs, which are purely for animation purposes.

There are three ways of including animation in Grrr programs: by indicating via a menu option that the matched part of the host graph should be highlighted, by indicating via a menu option that any node movement should be animated, and by adding a built in trigger to highlight a chosen node.

The automatic highlighting of matched subgraphs is a useful tool in debugging Grrr programs as it indicates that the desired part of the host graph has been matched by a LHS graph. However, it can also be used for other sorts of animation by indicating which part of the host graph has been visited as shown in the shortest path example, Section 3.2. When nodes and arcs are highlighted, the line thickness increase and the colour changes from black to purple. There is an element of arbitrariness to this, and the specification of highlights can be changed.

Animation of node movement is a result of recent work in adding geometric triggers to Grrr, and as with highlighting matched subgraphs it is a feature that can be used for either debugging or within a custom animation. When producing graph drawing algorithms, such as the force directed algorithm given in Section 3.1, it is very useful to observe the process of the algorithm for evaluating the success of the approach and confirming the correctness of the implementation. However, in terms of animation, the bubble sorting example given in Section 3.3 shows how geometric operations can be added to a purely graph theoretic algorithm in order to clarify the approach. In terms of visualisation, nodes that are moved are shown changing position on the screen.

The notion of adding triggers that change the appearance of nodes is entirely custom animation directed. The built in triggers include those to simply highlight the nodes. However there are more flexible commands to change the colour of nodes, and clear all highlights in the host graph.

3.1 Force Directed Graph Drawing

The animation of graph drawing algorithms requires no extra work by the user. The movement of nodes from one position to another can be shown by selecting a menu option. The fine tuning of algorithms is made easier because the immediate effect of altering parameters, or other changes to the drawing process can be seen. Also, the way poor drawings occur can be observed, so allowing changes to cope with situations such as subgraphs getting in to local minima or rogue nodes being misplaced.

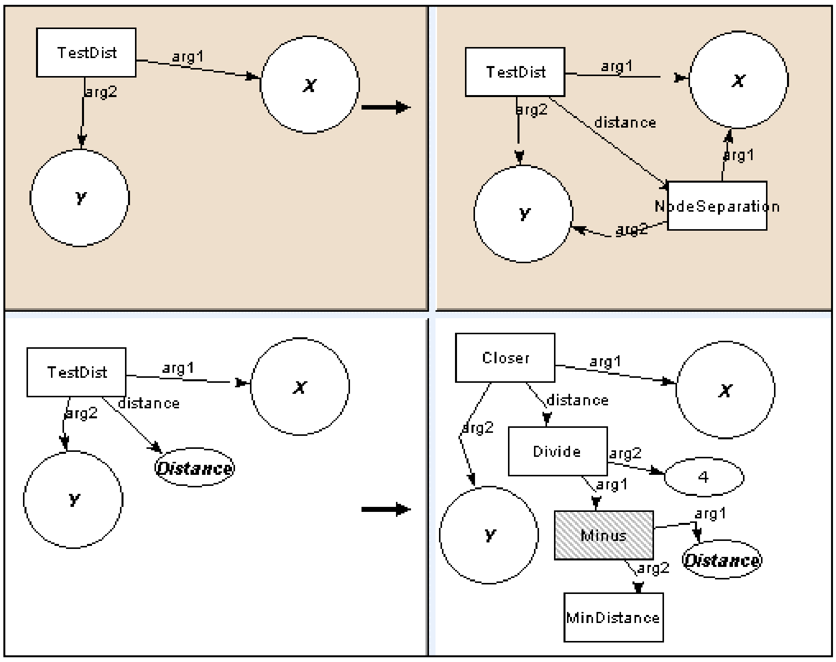

Fig. 1. The transformation 'TestDistance'. This brings connected nodes closer together. The distance they are moved together is greater when the nodes start further apart. The built in 'Closer' transformation moves both argument nodes an identical distance towards each other. The distance is calculated from the distance they are apart (found using the 'NodeSeparation' built in trigger) and the number of iterations that has taken place. The calculations are performed by the 'Divide' and 'Minus' built ins. The user defined transformation 'MinDistance' simply returns a constant number 90 in this implementation which can be used by 'Minus' which will in turn return a number that can then be used by 'Divide'.

As an example, we show part of a force directed graph drawing algorithm, it is difficult to show the actual animation in a research paper, but we hope that it is clear that changing various aspects of this algorithm are quite easy, even when the algorithm has been partially executed. The parameters for node movement can be altered both in the transformation definition and in the host graph. The function used for deriving the amount of node movement can also be altered from the very simple calculation given here into a more complex formula that may have a beneficial effect on layout.

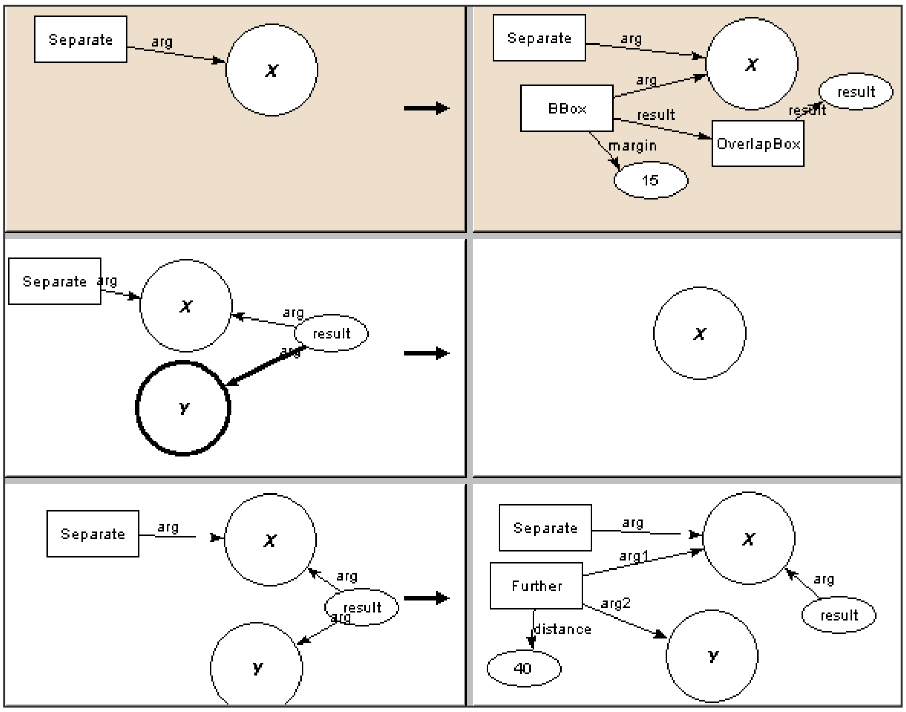

Fig. 2. The transformation 'Separate'. This finds the nodes that are closer than a set distance from a node and then moves them apart a constant distance. 'BBox' is a built in transformation that returns the nodes within the specified rectangle, 'OverlapBox' is also built in and returns the rectangle containing the specified nodes (or single node as in this case). The nodes within the rectangle are then separated with the built in 'Further' transformation that moves both argument nodes an identical distance from each other

The method treats arcs as springs between nodes, attracting them together, whist unconnected nodes are repelled. The algorithm presented here first iterates through the connected node pairs, bringing them closer, and then it iterates through all node pairs separating the nodes which are within a set distance of each other. This process is repeated a number of times, with the distance that the nodes are attracted reducing on each iteration. Our version of the force directed approach is a simple variant on those described in [5,14].

The two built in transformations that move nodes are 'Closer' and 'Further', which attract and repel node pairs respectively. They are used in the two transformations from the program, shown in Fig. 1 and Fig. 2. The start host graph is shown in Fig. 3 and the final host graph is shown in Fig. 4.

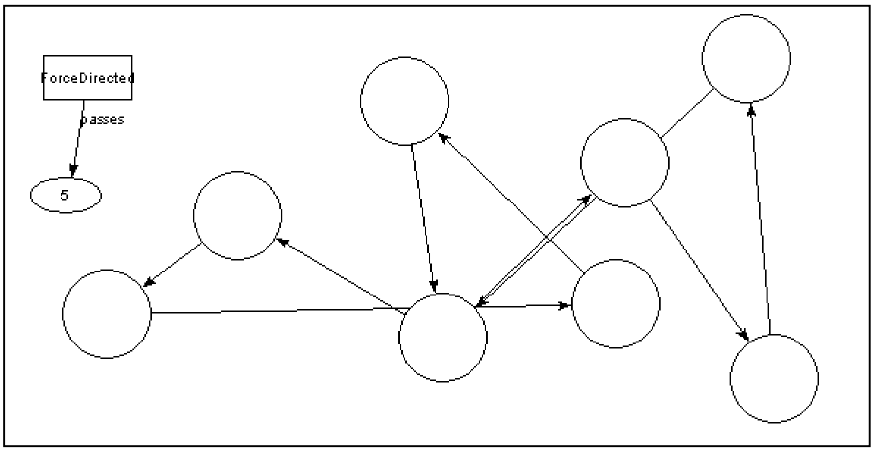

Fig. 3. At the start of execution. There will be 5 iterations of first closing the nodes connected by arcs and then separating nodes that are too close

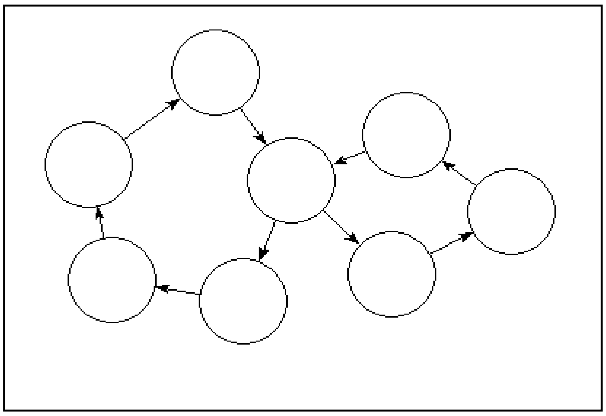

Fig. 4. At the end of execution

3.2 Shortest Path

This algorithm makes use of the highlighting to indicate the success of a graph search. In this case we are finding a shortest path between two nodes in an unweighted graph, so a simple depth first search will suffice. This is a version of an algorithm given in [17], hence we only show the major alteration, Fig. 5 which changes the algorithm by maintaining the structure of the graph and highlighting the path found, rather than deleting the nodes not participating in the path. This algorithm is a good example of using the match highlighting feature for animation purposes. Fig. 6 shows the host graph at the start of execution. Fig. 7 shows the host graph at the end of execution.

The search method can be clearly seen when the graph is stepped through in a slow manner, as only the matched nodes are visible, the path is added after the search has found the 'arg2' node.

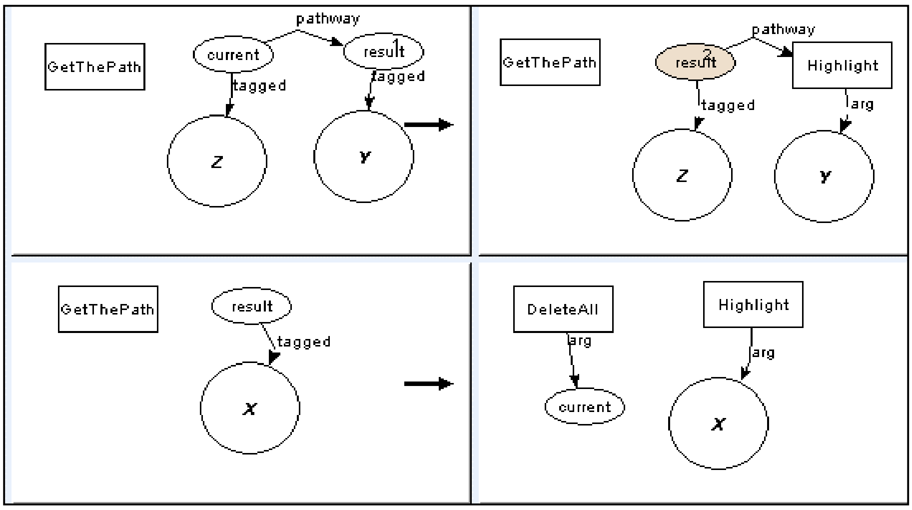

Fig. 5. The transformation 'GetThePath'. This is called after the path has been found, and traverses back along the search tree until the root (the 'arg1' node) is found

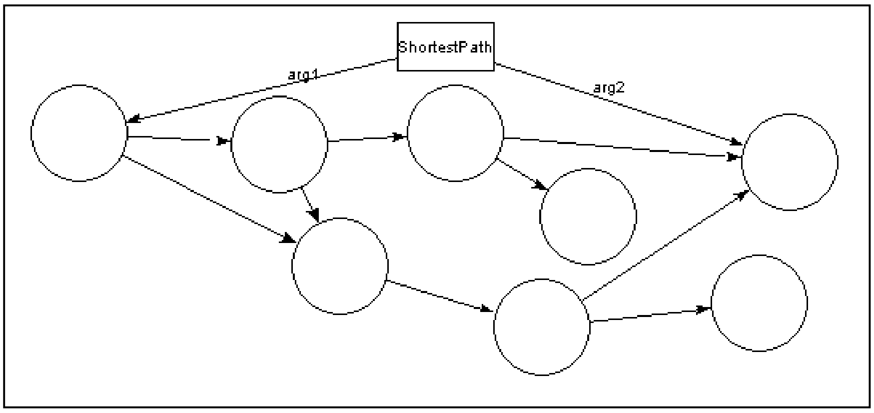

Fig. 6. The host graph at the start of execution. The program is searching for a path between the two round nodes connected to the trigger by 'arg1' and 'arg2' arcs. The search is from the 'arg1' node to the 'arg2' node

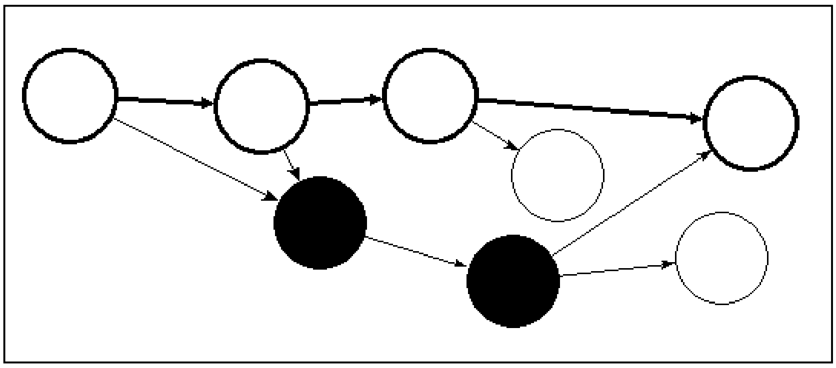

Fig. 7. The host graph at the end of execution. The black nodes indicate the nodes visited which are not in the path, the thick lined nodes indicate the nodes in the path, and the unchanged nodes are those that have not been visited. The algorithm finds only one shortest path of possible candidates, hence the path given was chosen over the alternative (using the black nodes) by the Grrr node ordering system which ensures the matching process is deterministic. The animation uses different colours when appearing on the screen, but we are limited to a black and white display for this paper

3.3 Bubble Sort

Here we give an example of a purely custom visualisation task. This is the sort of algorithm animation that is a useful teaching aid. Sorting is not an ideal task to perform with Grrr, as the relative lack of complexity of the data structure that is manipulated makes it less suited to our form of graph rewriting. The housekeeping concerned with list iteration, for example, dealing with all cases of nodes with or without predecessors or successors, means that transformations often have many rewrites, one for each case. We show all the transformations in this program to indicate some of the difficulties of producing this sort of custom visualisation task.

The program sorts a list represented by a set of nodes connected by arcs. Bubble sorting performs several iterations through a list, swapping the position of neighbouring list members that are in the wrong position until an iteration swaps no more members. The algorithm animation here is that of indicating the pair of nodes that are being tested and demonstrating swaps via the physical moving of the positions of swapped nodes. Both node highlighting and swapping is defined explicitly in the program.

The full bubble sort program is shown in Fig. 8, Fig. 9 and Fig. 10. Some illustrative stages in execution are shown in Fig. 11, Fig. 12 and Fig. 13.

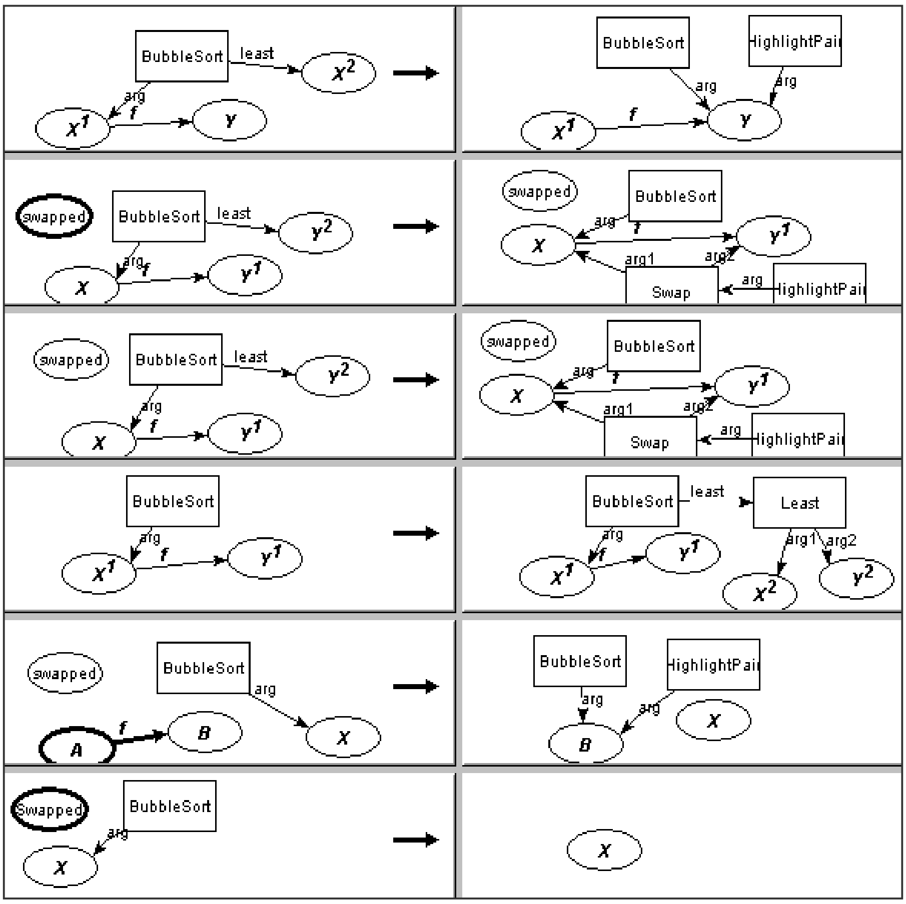

Fig. 8. The transformation 'BubbleSort'. This is the top level transformation in the program and performs the tasks of iterating through the list, calling the transformation 'Swap', which swaps the pairs and 'HighlightPair' which indicates which pair of nodes are being swapped

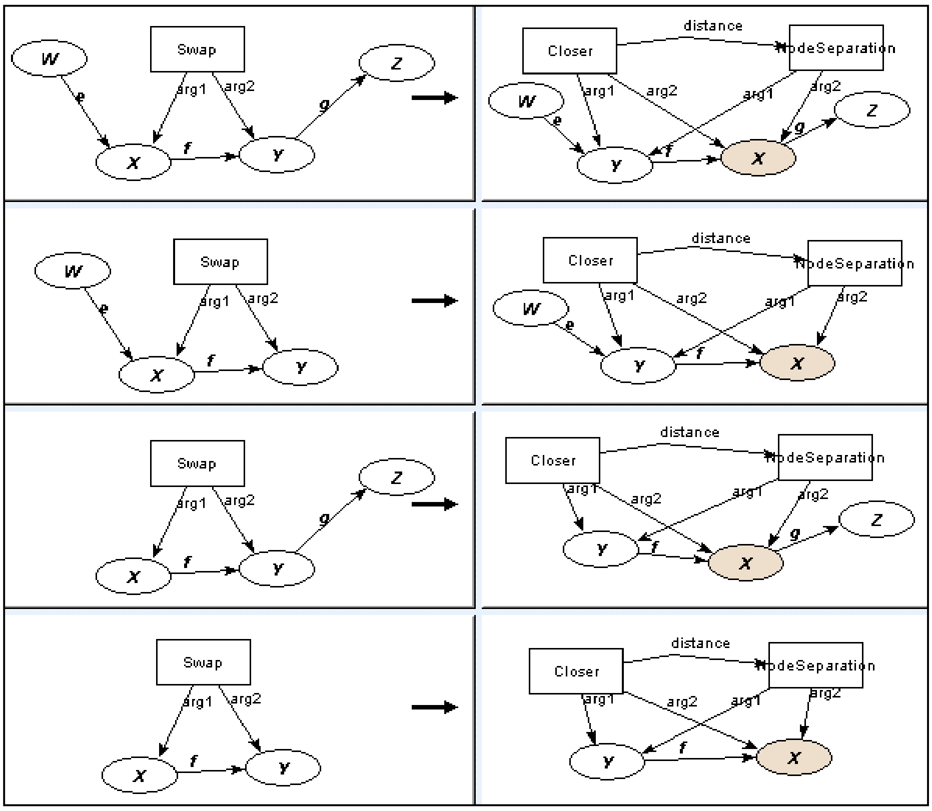

Fig. 9. The transformation 'Swap'. It has to deal with three cases: where is a node to either side of the swapped pair, where there are nodes at either end, and where the pair is alone. It calls two built in transformations: 'NodeSeparation' which finds the distance between the nodes, and 'Closer' which moves each node closer to the other by that distance, so animating the swap by node movement.

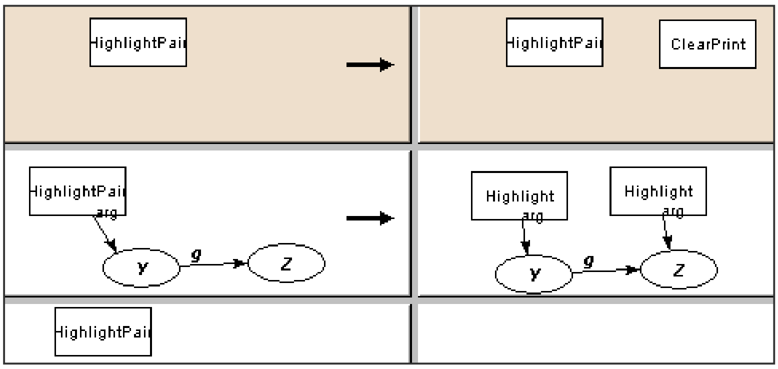

Fig. 10. The transformation 'HighlightPair'. The first rewrite clears the current highlight and is not called again because it is once only. The second rewrite highlights the indicated two nodes and removes the 'HighlightPair' trigger. The final rewrite is present to deal with the case that the chosen node does not have a following node in the list. Here the trigger is deleted with nothing highlighted

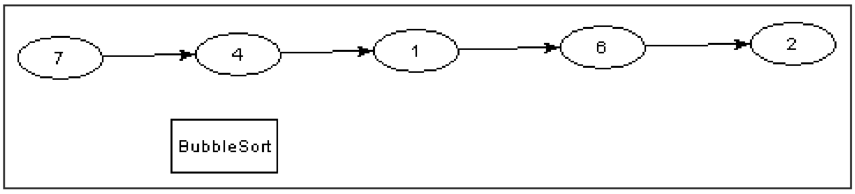

Fig. 11. At the start of the 'BubbleSort' program

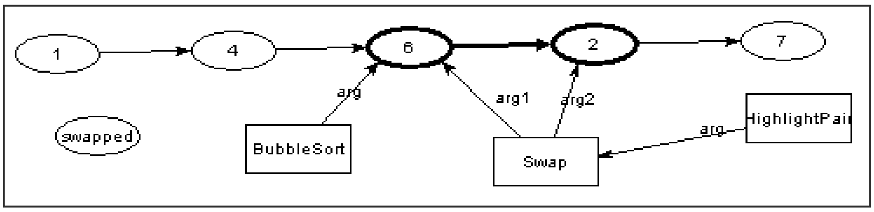

Fig. 12. The host graph at step 76 in the rewriting process. This is the middle of the second iteration through the list. The next few steps will exchange both the graph theoretic and geometric positions of two highlighted nodes. The 'swapped' node indicates that a swap has already been performed on this iteration. The 'HighlightPairs' trigger will be executed after the swap and highlight the next pair to be tested

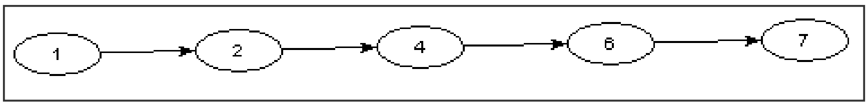

Fig. 13. The host graph after the program has finished on step 166. The list is sorted

4 Conclusions and Further Work

The animation techniques presented here are not new, however the graph rewriting method used to create them is novel and the animation is easy to achieve, as in many cases it requires no extra effort from the programmer. We see this as an application area that plays on the strengths of graph rewriting programming languages, as they are the only current systems which combine a visual view of graph data structures and computational completeness, so potentially allowing all possible graph based algorithms to be animated. Indeed, such animation is a great debugging aid when programming with Grrr.

The algorithm animation capabilities presented here fall into two main visualisation techniques: animating node movement, and highlighting visited or chosen subgraphs. The methods used for producing animations can be partitioned into those that can be used on existing algorithms, such as showing the node movement in graph drawing, or displaying visited nodes and those which are defined by the user, such as placing nodes for illustration and selecting specific nodes to be highlighted. The techniques described here can be combined as wished.

There are many possible areas of future work concerned with improving the usability of this programming language for the task of algorithm animation. The first important requirement for development of graph based algorithms is graph editing. The current Grrr editor is proving tricky to use for the high volume graph production required for animation. Further flexibility in cutting and pasting, and general user interface improvements are required.

The definition of node movement and highlighting in transformations is currently explicit, that is, the nodes are moved and highlighted by built in transformations. One can envisage an implicit method for defining node movement, where a difference in position of a node from the LHS to a RHS would mean the node was moved in the host graph. Implicit highlighting is also possible, where a node highlight in a RHS would be reflected by the corresponding node being highlighted in the host graph.

Node movement could be made easier by allowing a movement path to be defined by an arc. The node might follow the bends in the arc so as to produce more sophisticated user defined animation.

Because many of the built in transformations do not change the structure of the graph, it can be difficult at times to ensure that they are called in the right order. For example, the current position of a node should be found before that node is moved. Hence we are considering adding some method for specifying the order of trigger node execution in RHS graphs.

Acknowledgements

This work was supported by funding from the UK Engineering and Physical Sciences Research Council (EPSRC), grant reference GR/M23564.

References

1. Banach R.: MONSTR I -- Fundemental Issues and the Design of MONSTR. Journal of Universal Computer Science 2,4 (1996) 164-216.
2. Brown M.H.: The 1992 SRC Algorithm Animation Festival. Proceedings of the 9th IEEE Symposium on Visual Languages. IEEE Computer Society Press (1993) 116-123.
3. Brown M.H and Sedgewick R.: Techniques for Algorithm Animation. IEEE Software 2,1 (1985) 28-39.
4. Brandenburg F.J.: Layout Graph Grammars: The Placement Approach. Proceedings 4th International Workshop on Graph Grammars and their application to Computer Science. LNCS 532. 144-156.
5. Eades P.: A Heuristic for Graph Drawing. Congressus Numerantium 42 (1984) 149-160.
6. Feiner S., Salesin D. and Banchoff T.: Dial: A Diagrammatic Animation Language. IEEE Computer Graphics and Applications 2,9 (1982) 43-54.
7. Glauert J.R., Kennaway J.R. and Sleep M.R.: Dactl: An Experimental Graph Rewriting Language. Proceedings 4th International Workshop on Graph Grammars and their application to Computer Science. LNCS 532. 378-395.
8. Gurka J.S and Citrin W.: Testing Effectiveness of Algorithm Animation. Proceedings of the 12th IEEE Symposium on Visual Languages (1996) 182-189.
9. Helttula E., Hyrskykari A. and Räihä K.-J.: Graphical Specification of Algorithm Animations with ALADDIN. Proceedings 2nd International Conference on System Sciences, Vol. 2 (1989) 892-901.
10. Höfting F., Wanke E., Balmo•an A. and Bergmann C.: 1st Grade - A System for Implementation, Testing and Animation of Graph Algorithms. LNCS 665 (1993) 706-707.
11. Kaplan S.M., Goering S.K. & Cambell R.H. Specifying Concurrent Systems with Δ Grammars. Proceedings of the Fifth International Workshop on Software Specification and Design. Society Press (1989). 20-27.
12. Lahtinen S.P., Sutinen E. and Tarhio J.: Automated Animation of Algorithms with Eliot. Journal of Visual Languages and Computing Vol. 9 Iss. 3 (1998) 337-349.
13. Lee M.-C.: An Algorithm Animation Programming Environment. LNCS 602 (1992) 368-379.
14. Purchase H.C.: Performance of Layout Algorithms: Comprehension, not Computation. Journal of Visual Languages and Computing (1998) 9, 647-657.
15. Paredaens J., Van den Bussche J., Andries M., Gyssens M. and Thyssens I.. An Overview of GOOD. ACM SIGMOD Record, 21,1. (March 1992) 25-31
16. Rodgers, P.J.: A Graph Rewriting Programming Language for Graph Drawing. Proceedings of the 14th IEEE Symposium on Visual Languages, Halifax, Nova Scotia, Canada. IEEE Computer Society Press (1998) 32-39.
17. Rodgers P.J. and King P.J.H.: A Graph Rewriting Programming Language for Database Programming. The Journal of Visual Languages and Computing 8(6), 1997. 641-674.
18. Schürr A.: Rapid Programming with Graph Rewrite Rules. Proceedings USENIX Symposium on Very High Level Languages (VHLL), Santa Fe. October 1994. 83-100.
19. Stasko J. and Badre A.: Do Algorithm Animations Assist Learning? An Empirical Study and Analysis. Proceedings of ACM INTERCHI'93 (1993) 61-66.
20. McWhirter J.D.: AlgorithmExplorer: A Student Centered Algorithm Animation System. 12th IEEE Symposium on Visual Languages (1996) 174-181.
21. Zinßmeister G. and McCreary C.L.: Drawing Graphs with Attribute Graph Grammars. Graph Drawing '94. LNCS 894. (1995) 266-269