AC 2011-818: USING GRAPH THEORY VISUALIZATION TO MOTIVATE SOFTWARE ENGINEERING CONCEPTS

Shane Markstrum, Bucknell University

Shane Markstrum is an Assistant Professor of Computer Science at Bucknell University. His primary research focus is on the intersection of programming languages and software engineering–language tools. His recent work in this area includes the JavaCOP pluggable type framework for Java, and refactoring support in Eclipse for the X10 language. At Bucknell, he has taught the Introduction to Computer Science courses, as well as courses on the theory of computation and theory of programming languages. Prior to arriving at Bucknell, Prof. Markstrum received his Ph.D. in Computer Science from UCLA.

Prof. Gary M. Haggard, Bucknell University

Using Graph Theory Visualization to Motivate Software Engineering Concepts

Abstract

Implementing a software system that is the solution of an open ended problem gives students the opportunity to use a variety of programming tools and practice a software engineering methodology. In this paper, we describe a course *Graph Theory, Algorithms, and Software Engineering* intended as an elective for second and third year students that has these opportunities as its learning objectives. The course is structured around the implementation of a visualization tool for use in explaining and demonstrating fundamental concepts and classical results of graph theory. Possible graph theory topics to be included in the tool are search techniques, Euler circuits, and minimal cost spanning trees. Many of the decisions about the project's design and the topics covered are dependent on the students. As a result, the resulting tool can appear quite different from one offering to the next.

The project is intended to be developed by one or more student groups with all groups using the same interface for the graphs and visualizations. Strict enforcement of interface usage allows the opportunity to teach many design patterns that will aid in seamless integration of the code developed by different groups. Additionally, user testing of prototype implementations adds a sense of real development to the project. Since simple graph theory topics are an important aspect of a computer science education, the project also gives an opportunity for the students to present their tool at local and regional computer science meetings that encourage student participation.

1 Introduction

Many computer science programs include some type of capstone course in the senior year as a way to challenge their students to apply all of the knowledge they have gained on a substantial project. However, there are typically not many opportunities in the second and third year of these programs to introduce students to a team-based, open-ended project in a setting that helps them prepare to handle their culminating experience. From a development perspective, it is particularly difficult to find the right context for a fairly large software project that uses the techniques of object-oriented design or software development methodologies introduced earlier in the curriculum.

One approach to providing a larger software development experience is to assign a programming project lasting three to five weeks during a required course sometime in the second or third year. The scope of such projects requires a team-based approach for design and implementation. However, given the short timeframe associated with such projects, there is little opportunity to emphasize the software engineering process. These projects provide some open ended problem solving but the time constraint makes it very difficult to develop all aspects of the project with the attention one would like.

In this paper, we describe a different approach that eschews development of a familiar project in favor of a project that emphasizes design decisions and the software engineering process. It also reinforces essential mathematical foundations that a computer science student would be exposed

to early in their program. We present a new elective course–*Graph Theory, Algorithms, and Software Engineering*–that uses a whole term to develop a visualization project with clearly definable problems from the domain of graph theory. The problems chosen can each be solved in reasonably short time intervals. The topics are relatively self-contained such that all aspects of development can be monitored without worrying about the number of topics the groups complete. The most important aspect of the project is that enough parts get completed by a group that students gain an understanding of how effective design can increase the flexibility and reuse of code. The students will see why development accelerates once a careful design foundation is laid.

The objective of the course project is to put various graph theory topics and algorithms together into an object-oriented visualization tool. This tool is intended to bring the abstraction of graphs to life for beginning students in a discrete structures or graph theory course. The focus on the user of the project, a group quite distinct from the developers, makes students see program development is more than just a problem solving exercise. The number and variety of topics covered in the finished product can be modified to suit the level of understanding of graph theory and programming experience of the students. If more time is needed with certain design topics, it is possible for the instructor to cut more detailed topics so that the project still maintains a reasonable scope. Specific testing of the intended users' group at one or more points of the term should guide students to evaluate the design choices they have made throughout development. Each term the course is offered, the class of students will bring a different set of experiences to the design process, resulting in a new final product.

Visualizations for algorithms and graph theory topics have been shown to be an effective pedagogical device. For example, the Algorithms in Action project[12] uses animations to explain data structures, sorting algorithms, and some graph algorithms. The Math Cove[8] system is a graph theory course taught primarily through visual examples and animations. Other visualization systems are designed to support advanced graph theory topics like Tutte polynomials,[13] or feature advanced assessment tools to measure student learning such as the JAVENGA[1] and JHAVE[10] tools. Determining the effectiveness of visualization on learning is difficult, but some studies have shown that students grasp advanced topics better with visualizations such as sorting algorithms[11] and network flow algorithms.[1] To this end, as the students develop their own visualization tool, they will show a deeper understanding of the material they are presenting with their tool.

Another advantage that developing visualizations for graph theory provides, in addition to those already discussed, is that the topics often work naturally with many of the standard design patterns. In particular, the object creation design patterns such as abstract factory, prototype, and singleton as well as certain behavioral design patterns are useful for building graphs and animating graph manipulation algorithms.

The rest of this paper is organized as follows. The second section explains some of the graph theory topics that an instructor could reasonably expect undergraduate students to implement. The third section deals with the the software engineering process and design issues with regard to implementing the project. The fourth section discusses some specific details about results from an earlier version of this course offered at Bucknell University. Finally, we conclude with a brief summation of the paper and future thoughts.

2 Graph Theory Topics

The graph theory and combinatorial topics in the course are designed to engage students both in learning about graphs and the software development process from the first week of the term. The first problem is designed to see what tools students bring with them and which tools they associate with successful software development. The first problem can be introduced during the first week of class and students can be asked to write specifications as preparation for implementation of the first topic. With the specifications, students can also write a black-box testing protocol that gives them some idea of the users' needs. Students will already begin to see that coding is only one part of the software development process.

One of the strengths of designing the software around graph theory in the course is that the project topics can be built around relatively little theory. The first topic can be built on something as simple as the degree of a vertex. The remaining topics primarily involve two major themes. The first theme is connectedness. Finding connected components involves some kind of a search of the graph and so depth first and breadth first search are topics that naturally build up to finding components. The second theme is tree structures. The classical minimal cost spanning tree problem has three classical solutions and each can be implemented and explored for strengths and comparisons.

The problem of drawing a graph and representing various features permeates all the topics. One of the natural problems that arises in graph theory is determining whether two drawings represent the same graph. This gives students a chance to discuss the isomorphism problem and show how it can be solved for small graphs. The design decision for representing graphs and solutions to problems should be kept as uniform as possible knowing this is an issue with each topic. With good design decisions incorporated into the specifications, it should be possible to revisit this and other issues while maintaining a uniform output design for each topic.

Since many graph problems are NP-complete, the students should not be faced with trying to develop efficient implementations for such problems. While it is worthwhile to discuss problems such as Hamiltonian cycle in relation to the other graph theory topics, we consider it best to forgo a requirement that the students implement solutions for even restricted classes of graphs.

2.1 Problem 1: Degree Sequences

Degree sequences provide a graph theory topic that can be discussed, and possibly implemented, before the discussion of design criteria for the implementation of graphs is complete. A degree sequence of length n for any integer n is a multiset of n numbers such that each value is between 0 and $n-1$. The total number of degree sequence for $n = 8$ is 12344. It is possible to devise a purely combinatorial algorithm to generate each degree sequence for a given n but the size of the solution set for relatively small n limits the exercise to $n \leq 8$. Some degree sequences of size n represent the degrees of the vertices of a graph on n vertices. The degree of a vertex v in a simple graph is the number of edges incident to v. The multiset of n numbers between 0 and $n-1$, consisting of the degrees of the vertices of a simple graph on n vertices, is a special kind of degree sequence called a graphical sequence. For example, $\{0,1\}$ is not the sequence of degrees of the vertices of a graph with two vertices but the multiset $\{1,1\}$ is. Interestingly enough, the combinatorial problem

of generating all degree sequences of a given size is a solvable problem–but the solution is not so obvious to prevent students from discovering different algorithms to use.[6] The fact that this is a purely combinatorial problem makes it ideal for a first project while design decisions are made about a common interface for graphs.

2.2 Problem 2: Drawing a Graph

Dealing with how to display a graph is a major area of interest in its own right. You must find a balance between the possible ways to display a graph and how to convey the important information in a way that is easily seen in the representation. One relatively simple approach for graphs of reasonable size–for graphs of at most 20 vertices–is to display the vertices equally spaced around a circle and represent edges as lines drawn between adjacent vertices inside the circle. This gives the students the task of finding the locations for the vertices and deciding how large the circle should be. The size of the display area should be part of the overall design discussion and can be specified by a parameter that can be adjusted according to design decisions. The students must also think about screen layout and text positions for explaining the graphs. The students find it challenging to start thinking of this project as not a simple input-compute-output program, but as a communication tool that explains complex ideas to an audience.

In most cases, the display of the vertices equally spaced around a circle of some radius gives rise to an effective way to represent a graph. Unfortunately, the display of vertices around a circle does not convey the information one expects to see when a depth first search or breadth first search is being carried out. A second output pattern is needed that displays vertices at different vertical locations depending on how many edges are between a new vertex and the root of the tree rather than having all the vertices around a circle. It is still possible to have a meaningful output if the tree edges of a search tree are displayed as other graphs. Not every feature need be implemented immediately, but the students should provide extension points for future enhancements in their design.

2.3 Problem 3: Depth First and Breadth First Searches

There are several ways to approach the topic of graph searches. For basic understanding, the user needs to be shown how the search proceeds from one vertex via edge traversal. Both the depth-first and breadth-first search processes can generate a number of different trees based on the traversal order. The output in the visualization tool should either show the search tree as it exists after the search or try to show the search tree in a step-by-step fashion as it is being formed one edge at a time–a sort of animation. The search implementation itself should not control the display, as the user may want to manually step through the process until a new edge is seen in relation to other tree edges or its position in the original graph. An additional feature of the display would be the opportunity for the user to choose different starting vertices for the search process. Finally, a third option for this topic would be to show both searches as they develop starting from the same vertex on the same output display panel. Here the step-by-step adding of edges would clarify the differences in the search orders. Since both searches have the same number of edges in their search trees, the step-by-step process would show the same stage of each tree's development simultaneously.

2.4 Problem 4: Connected and Non-separable Components

As students study more complex graph algorithms, they often find it is sufficient to solve the problem only for connected or non-separable graphs. The reduction occurs because the solution for the original graph can be found by putting together solutions for either each connected or each non-separable component of the original graph. To make use of this technique, a better understanding of what these components are and how they are found for a particular graph is needed.

Finding the connected components of a graph is a straightforward modification of a depth first search algorithm so that it keeps track of the vertices reached from a fixed starting vertex. If the algorithm is modified in this way, this set of vertices is a connected component of the graph. If the set of vertices does not include all vertices of the graph, the process can be repeated on a vertex that has not yet been visited. Eventually all the connected components will be identified.

Finding non-separable components is an even more interesting exercise. The reason is that you are not only interested in which vertices you can visit but also how far back towards the root of the search tree vertices can get by being part of a cycle from a vertex in the search tree back to a vertex earlier encountered in the search. The equivalence relation that defines non-separable components is an equivalence relation on edges. Because the relation is defined on edges, a vertex can be in more than one non-separable component. The vertices in more than one non-separable component are called cut vertices–i.e., vertices whose removal along with their incident edges will disconnect the graph.

2.5 Problem 5: Identifying Graphs with Six or Fewer Vertices

Graph isomorphism is one of the really interesting problems in graph theory because it really is an open problem for which a complexity result is not known. Identifying a graph as an isomorph of a particular canonical graph is a useful termination condition in algorithms. The project associated with graph isomorphism makes use of a catalog of graphs with six or fewer vertices.[7] Usually graphs with up to six vertices can be effectively used as termination conditions because the identification process relies on few invariants of graphs. Fortunately for students most of the graphs with five or fewer vertices can be readily distinguished by their degree sequence alone. The purpose of this topic is to take any graph with six or fewer vertices and minimum degree 1–thus, eliminating isolated vertices–and not only draw them, but also identify certain properties of each graph. In addition, the nullity of the graph ($|E| - |V| + k$ where k is the number of connected components) and other invariants like girth, circumference, and eccentricity can be listed after short computations.

2.6 Problem 6: Minimal Cost Spanning Trees

The construction of a minimal cost spanning tree is one of the problems that every computer science student encounters at some point. One of the nice features about this problem is that the greedy algorithm is optimal. There are three major algorithms for finding a minimal cost spanning tree. The algorithm of Prim uses a priority queue to test edges in increasing order until enough edges are found to form a spanning tree. The algorithm of Kruskal maintains connectedness as an invariant in the process of finding enough edges for a spanning tree. The idea for the Kruskal algorithm is to add the edge with smallest weight to one of the vertices already connected by previously

chosen edges. Finally Borůvka's algorithm uses the Kruskal approach but goes through each step by adding all the edges that can join two connected components of the process so far generated subject to a minimality constraint. When all these edges are added, the process is repeated with at most half as many disconnected components to consider. These three algorithms use priority queues, the union/find algorithm, and stacks during execution. The very different order of selection for the same set of edges is instructive for the students. The output for the algorithms shows how edges are added one at a time until a minimal cost spanning tree is found.

As a special feature for the output, it is interesting to show the Prim and Kruskal algorithms operating on the same screen. Much of the code used to display depth first search and breadth first search on a single screen can be reused for this option. The steps of each algorithm would show the status of the partial solution.

2.7 Problem 7: Euler Circuits

Finding an Euler circuit in a graph is a well known as the "first" problem of graph theory. The necessary and sufficient conditions that a graph be connected and that each vertex have even degree is easy to verify. In fact, this is one of the instances where code written for another topic can be enlisted to provide part of the answer. The verification of the existence of an Euler circuit makes use of the code to determine if a graph is connected, but more interesting is to actually find an Euler circuit and figure out how to display that information in a useful way.

There are two algorithms for finding an Euler circuit. The first simply starts out from a vertex and adds edges until a circuit is formed and the traversal is located at the starting vertex with no more edges left to be added to the circuit. The complement of this circuit is a union of the connected components of its complement for which each connected component has an Euler circuit that can be recursively processed. All of the separate Euler circuits can be spliced together to find an Euler circuit of the original graph. Pseudo code for this algorithm can be found in Haggard et al.[5] The second algorithm builds an Euler circuit so that the partial result is always connected. To make this algorithm work requires that an edge considered as the next possible edge in the Euler circuit be deleted and the remaining graph tested to see if it is connected. If the resulting graph is connected, the edge is included in the solution. If the resulting graph is not connected and if there are more edges incident to the vertex that is currently the end of the solution so far, another edge is tried so that if at all possible after adding the new edge, the partial solution is a connected graph. This solution is attributed to Fleury.[3]

The algorithms are both straightforward to code. More interesting is the question of how to display the solution. After all, the solution is just all the edges of the original graph. This is another opportunity for students to use the one-at-a-time, simple animation method used to show how to build search trees and minimal cost spanning trees.

3 Design

The graph theory problems described earlier are a reasonable set of topics for which implementation would take up most or all of an academic term. None of the topics involves very complex

algorithmic material, but just enough for students to begin to understand how to leverage the development of one algorithm in sufficient generality that other topics could easily make use of the code.

Since there are many designs possible, rather than make design a group decision, we choose to make design a class decision so code can be shared and reused. There are two ways to organize this part of the project. The first is to have the students code two topics with the understanding that the two topics should be accessible from a single driver class that has a visual interface. This will lead to a number of possible but quite different solutions. The issue then becomes about convergence on a single design. This design process is a good opportunity for students to start to see the importance of clear specifications so that everyone is building the same structure. Another possible approach is to give the basic structure of the final design and allow a class discussion of specific options to decide how the design should be finalized. If this second approach is used after the first two topics, the backtracking to get everyone on the same design scheme will not be too difficult. It is important that whatever decisions are made at this time become the specifications against which everything is measured. Black-box tests based on the specifications written for each topic can be developed and used to verify that the specifications are met. At the end of the term it is always healthy to have each student present a one slide list of both design decisions that helped and design decisions that the student would like to make in a different way. The most successful offerings of the course will result in at least one student finding a better or different alternative for each major design decision. The lesson is usually not lost that freezing the design was the only feasible way to get the whole project done and that changes can be more easily incorporated because of the design and specification work done in the beginning.

3.1 General Decisions

Beyond the design decisions that the students must make, some choices must be made by the instructor of the course that will facilitate development of the project. These decisions include the language to use; how student collaboration will be facilitated; what set of software tools should be used; and whether students will follow a particular kind of software development methodology. For example, in previous preparations for this course, the students were required to do planning with an off-the-shelf UML tool and develop in Java, where use of the Java standard library was emphasized. They were also required to use the Subversion tool for concurrent group development. However, the approach is adaptable to any language which offers good support of object-oriented design and user interface libraries. Even if the tools used in the course have not been previously seen by all of the students, start up costs are minimal since students with experience using the tools help their classmates to learn them. As a result, the less experienced class members increase their proficiency over the whole term.

The instructor should also help to identify the target audience at the school for the visualization tool. This should be done in collaboration with another instructor so that the students have a ready set of students who can act as subjects for a practical user study. The decision about the intended audience is important because it helps the students determine an appropriate test plan and keeps the level of documentation at an appropriate level. The design with a set of users in mind, as opposed to their instructor or themselves, is a new idea for most students as they primarily program to

satisfy a course requirement in most of their early courses.

When undertaking an open ended and rather extensive system development project, students need to grapple from the start with developing specifications carefully. In addition, students need to realize that they must build a system that matches the specification without modifications along the way to the specification because of problems encountered during design activities or implementation itself. The usual *ad hoc* manner of dealing with design problems becomes an issue that cannot be resolved at the group level but must be filtered through project leaders, mirroring how such decisions are made in a professional development group. Exercises during the implementation of the components of the project that include co-mingling code developed by different groups will demonstrate the strength or weakness of the design being implemented. This emphasis on specifications can be instilled in groups by requiring black-box testing plans along with the specification. Once a part of the project is implemented, the black-box testing should be undertaken and the results should be recorded. When there is a design change or a modification of the specifications, this level of testing can be used to see if the project is back on track.

3.2 Project Design

Developing specifications and design documents such as UML diagrams for a project is often the first step in the software process. However, we believe that the students in this course should implement simple, text-based solutions to a couple of graph theory problems before trying to design the more flexible visualization tool. When the students have implemented two of the topics as separate classes with independent implementations, they will recognize the commonalities in their solutions. This provides a clue as to how interfaces for components in graph theory topics should be extracted or created. It also removes any possible confusion the students may encounter in the sometimes tedious thinking required to implemented a graphical user interface.

Once the students have tackled the initial challenge of implementing the graph theory algorithms, they must begin thinking about what the users of a visualization for graph theory topics would want. Presenting choices to the user requires a feature associated with each topic that introduces the topic so the users can determine which topics they would like to explore. For example, in a previous iteration of the course, the students decided to use a hyperlinked set of screens containing enough documentation about each topic that the user could get an elementary understanding of the topic. Implementation details associated with this choice required the students to design a menu with a large number of listeners waiting to be activated by a mouse click. Because any design decision the students make for a modern graphical user interface will involve event-driven programming, the students will naturally encounter the observer design pattern and become familiar with callbacks.[4]

As with any good user interface, the students should apply consistent interface design across the varied topics. For example, the same pattern of screens and the same placement of features should be sought for each topic. When the students have implemented more features, the design of the interface can be evaluated for clarity and usability. An ideal solution would maintain only superficial interaction between the graph theory components and the visualization components. In such a situation, the students will be able to change their interface design to better suit their user

base. One feature of robust design is that any one of the topics can be broken away and used separately without making any changes to the overall system. This is necessary for independent development of topics by different groups in the class.

3.3 Using Interfaces vs. Abstract Classes

From the definition of a graph, students will quickly determine what methods and data are needed for a graph class. In terms of data, some representation of vertices and edges will be necessary. In variants of a simple graph, other information might need to be stored, such as directionality or weight. General methods associated with a graph will also be apparent. For example, the students will determine that a graph should be able to determine whether vertices are adjacent. Thus, any graph object will need to supply a method for answering this question. Beyond the graph itself, the students will have to provide a class implementation for either vertices or edges. In the past, students have chosen to provide an implementation for vertices, since edges can be defined as pairs of vertices.

When writing specifications for an edge or vertex class, the students need to pay special attention to use of interfaces and abstract classes for the implementation. The students will quickly learn that some form of interface abstraction is necessary to separate the data from the application. Choosing to use an interface or an abstract class should be left to the students so that they can learn the advantages and disadvantages each offers. However, for pedagogical purposes, the instructor should require that at least one topic be implemented using each approach. The minimum cost spanning tree problem is especially ideal for use of an abstract class since the algorithms all share the same template. Ultimately, they will find that either option will work, but that their choice may affect ease of implementation of other components. In prior offerings of this course, students have chosen to work primarily with interfaces as they felt they created a more visible separation between the data and the application.

The students find that interface abstraction is useful here because graph theory topics involve the use of different kinds of graphs. Simple and multigraphs can have the same representation and are processed the same way. Weighted graphs should behave the same as simple graphs for problems that do not take into account weights. A digraph can also be converted into a simple graph by splitting each vertex into a connected input vertex and and output vertex. Thus, a digraph can be used where a simple graph is expected if an adapter is provided for the digraph objects.

Graphs with more features require class and interface extension, but should maintain a subtype relationship with a simple graph and its components. For example, a weighted graph should make use of weighted edges. If both graphs and edges are independent classes, then weighted graphs need to be used only with weighted edges. Depending on the language used in developing the project, the students will discover that it is often difficult to define and use such families of types properly.

3.4 Using Libraries

One powerful tool for coding purposes in Java, and most other widely taught/used languages, is the extensive libraries that accompany them. Since use of libraries is pervasive in professional software development, the students should be required to use as much functionality from libraries as possible. This requirement has three additional benefits beyond matching industry best practices. The first benefit is that students will learn a great deal about writing good interfaces from library usage. The second benefit is that code from one group will be much more understandable to other groups if libraries are used. The third benefit is that debugging of code will be easier.

The students should further view the creation of their tool as implementation of a new library (or set of libraries) specific to graph theory visualization. This analogy extends to viewing their interfaces as a common language with which to present and discuss their work. It cannot be stressed enough that all this common language makes it much easier for collaboration on aspects of a problem. The idea is that the project is not a competition but a learning experience that is enhanced by collaboration. Common testing protocols for the groups require that a certain amount of similarity exists among the groups' software. Use of libraries, and more generally good teamwork, can be encouraged through in-class exercises that stress components from the libraries.

3.5 Design Patterns

As all software engineers know, design patterns are one of the most effective tools for good object-oriented software development. However, as an abstraction, this idea does not gain too much traction with students because they do not see many applications that benefit much from use of design patterns. With graphs it is very easy to introduce some very useful design patterns. In particular, the abstract factory and builder patterns are of particular benefit for coding graph theory applications.

The reason for the value of the builder design pattern is that graph theorists over the years have developed a myriad of input formats for large libraries of graphs. It is not realistic to expect all these libraries to be converted to a common format nor is it really necessary for such a conversion to take place. The builder design pattern handles this problem in an extensible way so that developers need not worry about using a library with a different format for storing graphs. With the builder design pattern, the program can look at an input file; identify what format is being used for storing a graph; and select the the appropriate concrete builder based on that format type. The students can then write and use a director object that calls the builder methods and returns a graph.

Constructing graphs can be tricky, though, because edge and vertex objects for a graph need to maintain certain relationships during construction. While a builder is useful for handling the multiple input formats, it does not really address this issue. In this case, the students will find use of the abstract factory design pattern to be helpful. An abstract factory will not only allow the code to remain independent of the actual type of graph being constructed, but will remove any direct creation and manipulation of vertices and edges which might result in an incorrect graph. As far as the end programmer is concerned, a graph built by a factory is a list of edges that can be updated and queried as needed. The students will need to define concrete factories for simple

graphs, weighted graphs, and directed graphs.

Since the implementations of the builder and abstract factory design patterns are relatively straightforward, students immediately see the benefits of interface abstraction. Using these patterns decouples the difficulty of parsing files and constructing graphs in the implementation. Instead, the files are parsed by a concrete builder which in turn delegates the hard and tedious work of building graphs to a factory.

An additional design pattern which is beneficial for a graph visualization tool is the prototype design pattern. With the prototype design pattern, a graph can be cloned as necessary for use in many of the (somewhat) destructive graph theory algorithms. Students can also see how prototyping can be used in combination with the abstract factory to quickly construct graphs via caching or memoization.

When alternative algorithms are implemented to solve the same problem, the template design pattern gives structure to the code that makes the implementation more readable. For example, with the three minimal cost spanning tree algorithms, the students can use the template design pattern because each algorithm is an instance of a greedy algorithm that follows the same basic outline. The first step should be to select the next edge to examine, followed by a step that examines the edge. The final step incorporates selected edges into the solution. For two of the algorithms the selection step not only chooses a next candidate but chooses a candidate that will be part of the final solution. In all three cases, the algorithms have three parts that can be defined in a parent class as abstract methods and, thus, given a different implementation in the concrete subclasses that correspond to the different algorithms.

There are a number of design pattern textbooks at the appropriate level for students in this second or third year elective course. Previous offerings of this course have relied on a free online textbook describing design patterns in a Java context can easily be required reading for each student.[2]

3.6 A Template for Additional Topics

Since the number of topics that can be included is in no way exhausted by the list given, a clear description of how to add a feature would make the addition of new topics easier for students who were not involved in the original development. The students should be tasked with providing documentation and hooks for additional add-ons to their system that do not require an overhaul of their original specification. This discussion and its results may include the study of additional design patterns such as the chain of responsibility and state patterns.

3.7 Test Results

One feature of this project that is not usually part of a course project is realistic testing of at least part of the software with a typical group of users. The process of using human subjects will often require the instructor to be certified on campus as qualified to carry out experiments on human subjects. The students' part in the testing starts with the decision of what to focus on in the test.

The easiest decision at first is to focus on how effectively the user can navigate through the system. Adding additional questions can focus on the use of color or animation for the display of certain results. Testing can be viewed more as an iterative process with the convergence of the designer's perspective and the users' perspective being brought together through a series of testing sessions as the developers gets a better idea of the users' perceptions.

One feature that is particularly difficult to get right is the level of documentation provided with help screens or some help feature. Elementary graph theory is a relatively straightforward subject but most students, and even other faculty who are not regularly involved with the subject, sometimes have a hard time remembering what all the terminology means and what some of the outputs represent. The development of help screens is a bigger issue than students typically expect. Since the tool being developed is designed as a visualization aid, the help screens should contain adequate illustrations so that the abstractions defined are coupled with a picture. This strategy for documentation helps with, but does not always overcome, the problem of familiarity with the material.

3.8 User Manual

As with all large software projects, some form of documentation for the end user is necessary to accompany its release. The students should prepare a thorough user manual that demonstrates they know how to explain their tool to the end user. The documentation should show an understanding that a user manual is different than a design specification in terms of the details provided about the tool and the level of readership associated with the document. Many tutorials for software are built around screen captures showing how the software operates. This model works especially well for visualization tools such as the one described here. In addition, a manual provides an interesting model for future classes tasked with implementing a similar project. Such a manual can be developed one topic at a time so that prototype user manuals would be available at the time of formal user testing.

4 Sample Implementation: Graph Works

A notable implementation of this project from a previous offering of this course is the Graph Works project. In earlier parts of the paper, certain aspects of the design and implementation of Graph Works were described as examples of choices the students can make. Some screenshots of this prototype visualization tool are shown in Appendix A.

The entire tool was accomplished in one semester by two student groups. First year computer science students in their second computer science course were chosen as the users of this system and participated in a user study coordinated with the instructor of that course.

Because of the visual nature of the project, it is easy for the students to prepare a poster of their work for presentation at regional, national, or international levels. In the case of Graph Works, the students were able to present a poster at the 2010 ACM Symposium on Software Visualization, which gave them exposure to the wider world of research in software visualization.[9]

5 Conclusion

In this paper, we have described a course in which students develop a large, open-ended visualization tool for graph theory topics. The development of the project enables the students to gain a deeper understanding of graph theory and its algorithms. Additionally, it provides a natural way to explore fundamental decisions of the object-oriented software process and common design patterns. Students enhance their understanding of the software engineering techniques introduced in earlier courses. A real benefit to the project is that students can bring their experience to capstone projects and to future software development opportunities. Since the material covered in the course is fundamental to nearly all computer science and software engineering curricula, we believe that the *Graph Theory, Algorithms, and Software Engineering* course is a viable elective for any program after students have taken the core introductory classes.

References

[1] T. Baloukas. Javenga: Java-based visualization environment for network and graph algorithms. *Computer Applications in Engineering Education* n/a. Wiley Subscription Services, Inc. Hoboken, NJ. doi: 10.1002/cae.20392

[2] J. W. Cooper. The Design Patterns Java Companion. *http://www.patterndepot.com/put/8/JavaPatterns.htm*,IBM Thomas J. Watson Research Center.

[3] Fleury. Deux problemes de geometrie de situation. *J. de Mathematiques Elementaires.* (1883), 257-261.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Publ., 1994.

[5] G. Haggard, J. Schlipf, S. Whitesides. Discrete Mathematics for Computer Science. Brooks Cole Publ.,2007.

[6] S. Hakimi. On the realizability of a set of integers as degrees of the vertices of a graph. *J. SIAM Appl. Math.* **10** (1962), 496-506.

[7] F. Harary. *Graph Theory*. Addison Wesley Publ., 1969.

[8] C. Mawata. *Math Cove*. *http://www.mathcove.net/petersen/lessons/index*, 1998.

[9] D. Medani, G. Haggard, C. Bassett, P. Koch, N. Lampert, T. Medlock, S. Pierce, R. Smith, A. Yehl. Graph works - pilot graph theory visualization tool. In *SOFTVIS'10: Proceedings of the 5th International Symposium on Software Visualization*. Salt Lake City, UT, October 2010.

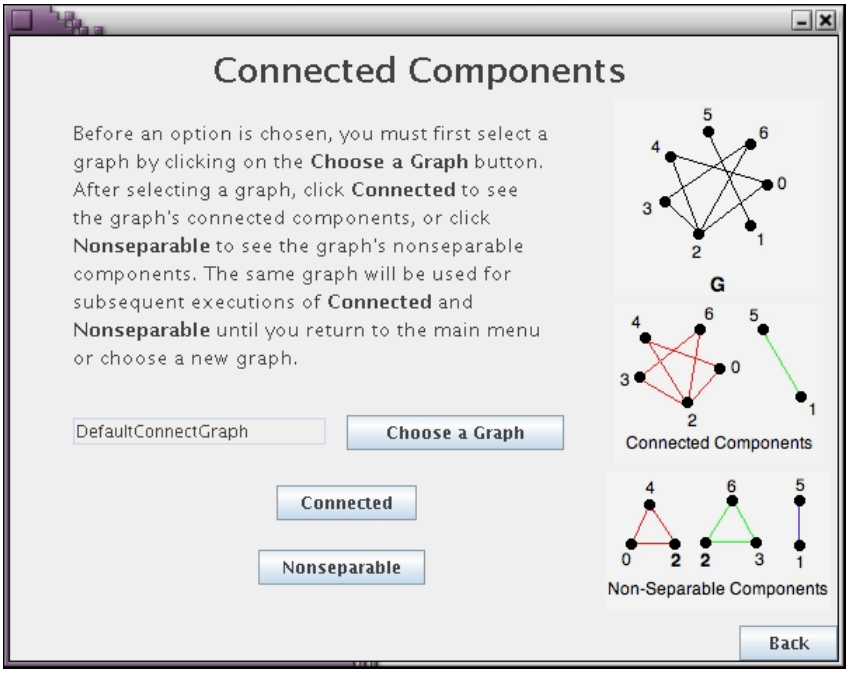
[10] T. Naps, J. Eagan, L. Norton. JHAVE—An environment to actively engage students in web-based algorithm visualizations. *ACM SIGCSE Bulletin* **32** (2000), 109-113.

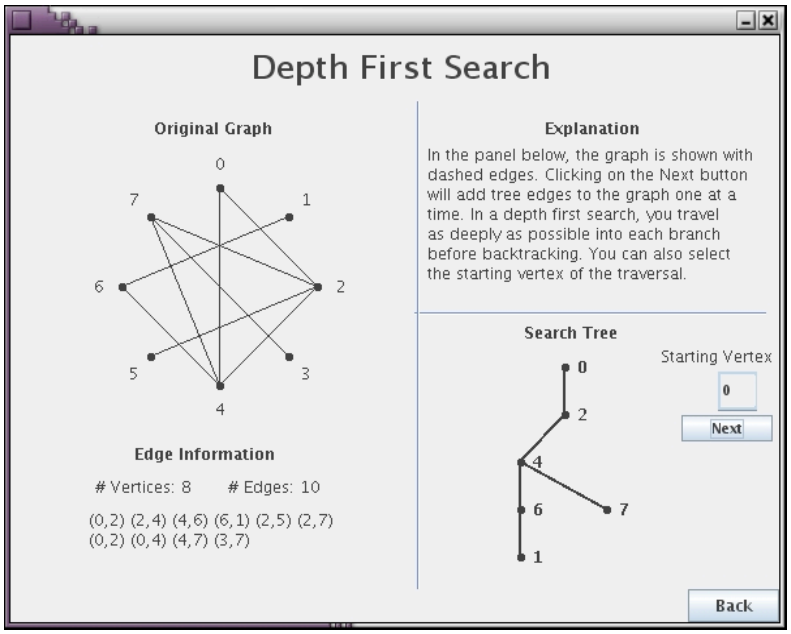
[11] R. Baecker. *Sorting Out SORTING: A Case Study for Teaching Software Visualization in Computer Science*, in: J. T. Stasko, M. H. Brown, and B. A. Price, editors. *Software Visualization*, MIT Press, Cambridge, MA, 1997

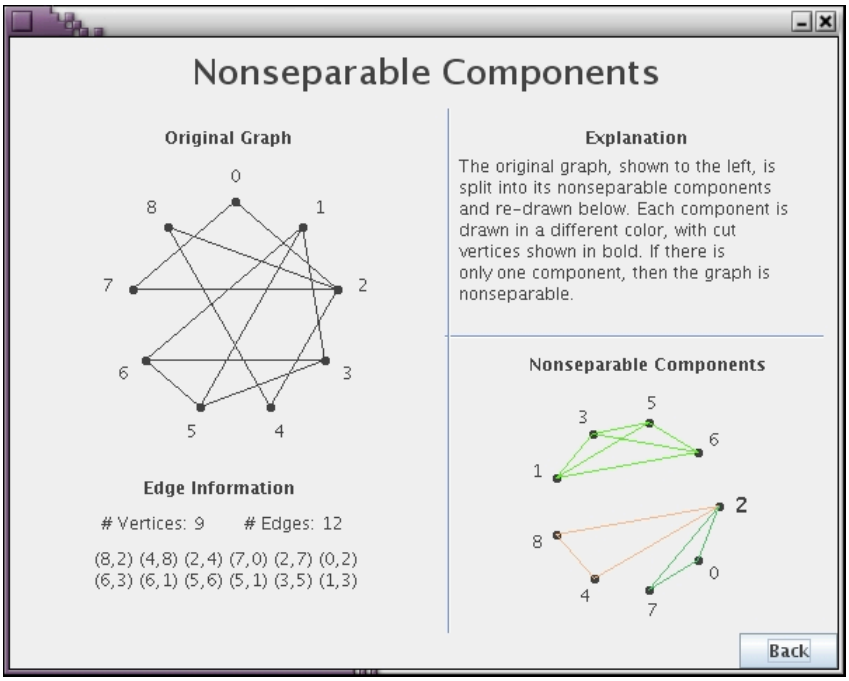
[12] L. Stern, L. Naish, H. Sondergaard. *Algorithms in Action. http://www.csse.monash.edu.au/˜dwa/Animations/index.html*, 2000.

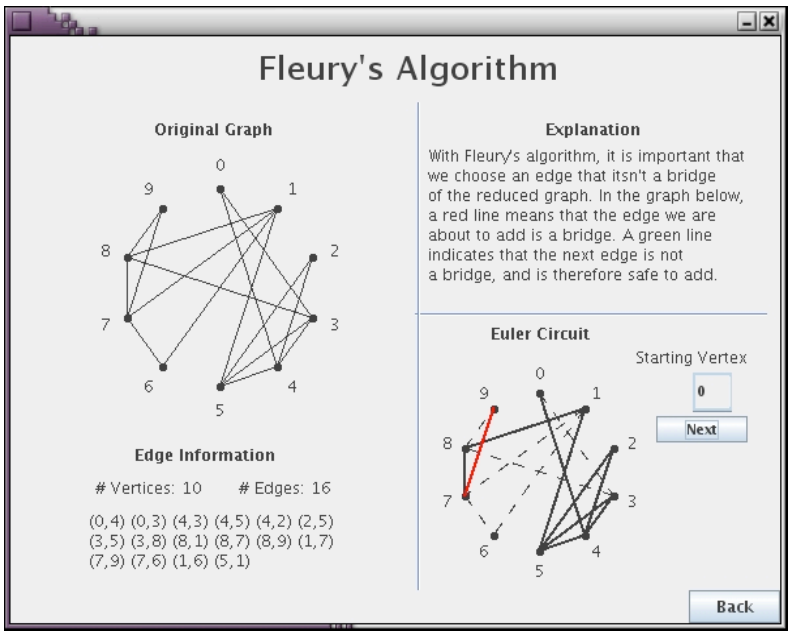
[13] B. Thompson, D. J. Pearce, C. Anslow, G. Haggard. Visualizing the computation tree of the Tutte polynomial. In *SOFTVIS'08: Proceedings of the ACM Symposium on Software Visualization*. Herrsching am Ammersee, Germany, September 2008.

A Screenshots






Nonseparable Components
Original Graph
0
1
2
3
4
5
6
7
8
Edge Information
Vertices: 9 # Edges: 12
(8,2) (4,8) (2,4) (7,0) (2,7) (0,2)
(6,3) (6,1) (5,6) (5,1) (3,5) (1,3)
Explanation
The original graph, shown to the left, is split into its nonseparable components and re-drawn below. Each component is drawn in a different color, with cut vertices shown in bold. If there is only one component, then the graph is nonseparable.
Nonseparable Components
Back


Fleury's Algorithm
Original Graph
0
1
2
3
4
5
6
7
8
9
Edge Information
Vertices: 10 # Edges: 16
(0,4) (0,3) (4,3) (4,5) (4,2) (2,5)
(3,5) (3,8) (8,1) (8,7) (8,9) (1,7)
(7,9) (7,6) (1,6) (5,1)
Explanation
With Fleury's algorithm, it is important that we choose an edge that itsn't a bridge of the reduced graph. In the graph below, a red line means that the edge we are about to add is a bridge. A green line indicates that the next edge is not a bridge, and is therefore safe to add.
Euler Circuit
Starting Vertex
0
Next
Back