# – Implementation

## Introduction

The game flow analysis deserves particular attention for serious games (ABT, 1987), which are games used for purposes other than entertainment while still providing pleasure. Serious games have been used for aiding students to learn and understand concepts taught in classrooms (BAKER; NAVARRO; VAN DER HOEK, 2003; NAVARRO, 2006; KOHWALTER; CLUA; MURTA, 2011) due to their characteristic of stimulating curiosity and providing motivation for learning (PRENSKY, 2001). Understanding the educational results obtained in a serious game is important to assimilate the knowledge and concepts passed in the game. In addition, examining the game flow allows for the identification of good and bad attitudes made by the player or by game developers. This knowledge can be used in future game sessions to avoid making the same mistakes or even to adjust gameplay features.

In this chapter, the *provenance in games* conceptual framework is instantiated in the *Software Development Manager* (SDM) game (KOHWALTER; CLUA; MURTA, 2011) as a proof of concept. The SDM game focuses on introducing Software Engineering concepts and skills to undergraduate students. The version of SDM presented in this chapter includes the provenance gathering, allowing students to visualize their actions and identify steps that lead to successful or unsuccessful outcomes. A visualization tool for the provenance graph, named *Proof Viewer*, is also described in this chapter. *Proof Viewer* was customized to be compatible with SDM. However, it can support other games with few modifications on the interface, filters, which use SDM nomenclature, and vertices, which contains information also customized for SDM.

This chapter is organized as follows: Section 5.2 briefly describes the SDM and details of how the provenance information is gathered, generating the *game flow log*. Section 5.3 provides a simple example of game session in SDM. Section 5.4 describes details about *Proof Viewer*. Lastly, Section 5.5 presents the final considerations of this chapter.

## SDM

In SDM, which was developed using the game engine Unity3D (HIGGINS, 2010), the player has a team of employees that are used to develop software according to contracts made with customers. The gameplay and game mechanics are modeled presenting possibilities to the player to decide strategies for development and to define the roles for each staff member. As in any contract, the software has requirements that must be followed during development. From a gameplay point of view, these requirements help to balance the mechanics and rules. When the software is completed and delivered to the customer, there is a quality assessment of the software and a project completion payment accordingly to the product quality.

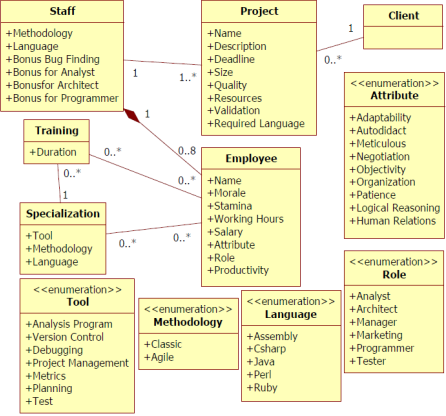


Figure 1: SDM simplified class diagram

Since SDM focuses in people management, the main elements of the game are the employees, which represent the player’s labor force. Employees can perform different roles (analyst, architect, manager, marketing, programmer, and tester.), which uses the employee’s attributes to calculate his performance depending on the performed role. Another element present in the game is specialization, used to define the employee working competence. With the specialization system, it is possible for employees to undergo training to learn new sets of skills. Also the concepts of working hours, morale, and stamina are used to modify the employee’s productivity. Figure 1 shows a simplified version of SDM’s class diagram focusing on the employee, showing his human attributes, types of specializations, the possibility of training to acquire specializations, and that the employee is affect by other employees that belong to the staff team. It also illustrates the project and its characteristics and requirement.

### Provenance Gathering

The data structure used in SDM is similar to the one explained at chapter 4. Each project contains a list of the employees involved in its development. In turn, each employee has a list of his actions executed throughout the development. If any action had an external influence during its execution, then the action also has a pointer to the action that influenced it. Throughout the game, when actions are executed or new employees are hired, information about the event is collected and stored for later usage. Actions go to their respective lists while new employees are added to the project list, creating their own list of actions. Each day passed in the game also records the state of the software development at the end of the day.

Since the information collected is used for the generation of the provenance graph, its content is mapped to the three possible types of provenance vertex (*activities*, *agents*, and *entities*). This mapping is done according to the data model explained at chapter 4. The following paragraphs describe information details that are extracted from the game and their respective roles in the provenance graph.

Each action executed during the game is represented by an *activity* vertex. The information collected during its execution includes: who executed it, which task and role the employee was occupying, and the current morale and stamina status of the employee that executed the action. The worked hours in the day the action was generated and credits spent to execute the action are also stored. Lastly, the progress the employee made during his task. These details are illustrated in Figure 2. Besides those, if the action had any external influences, such as the use of an artifact (prototypes or test cases, for example), then SDM stores a link to the action or artifact that affected its execution.

Each employee that participated in the development of the software is mapped to an *agent* vertex in the provenance graph. The collected information includes the employee’s name, his current staff grade (junior, mid-level, or senior), his current level (and experience points), traits, and specializations. Lastly, the *entity* vertex in the provenance graph represents one of the three possible artifacts in SDM: Prototypes produced by architects and used by analysts; Test Cases produced by analysts, architects, and programmers and used by testers; and Project, which represents the instances of the software development progress recorded each day.

The daily project information collected includes the day of its instance, the project’s deadline, how much coding was produced and the code overall quality. It also stores the clients requirements identified and modeled by analysts, how many credits the player had by the end of that day, and the state of each type of bugs in the software. For prototypes and test cases, only the day they were created and their names are stored, since actions contain the information of when they were used. Figure 2 illustrates the information collected in SDM for a Project vertex (*Entity*), Agent vertex (*Agent*), and a Process vertex (*Activity*), and shown in Proof Viewer according to the vertex’s type.

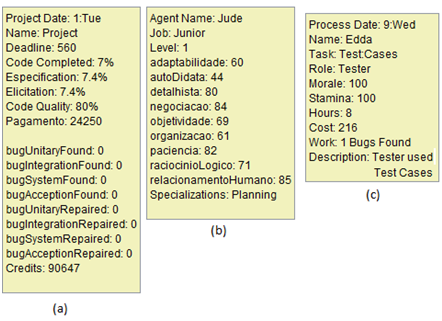


Figure 2: Information data extracted and visible at the provenance graph. The *entity* vertex representation for the project’s data (a), the employee’s *agent* vertex data (b), and the action’s *activity* vertex data (c).

At the end of a gaming session, the data collected during the game session, also known as *game flow log*, is exported to an external visualization tool, the *Proof Viewer*. *Proof Viewer*, in turn, processes the data and generates the corresponding provenance graph that represents the game session. The next section describes a game scenario in SDM that is used as an example for describing the features present in *Proof Viewer*.

## Guiding Example

In this section, we exemplify a SDM game session, which explores…

Starting the game, the player has at his disposal four employees: Yesha, Tornik, Mirax, and Emmy. The first thing he does is to assign roles for each employee. Yesha is assigned as the staff’s manager and has the task of aiding analysts. Tornik is assigned as an analyst, Mirax as marketing (which aids analysts and generate cash), and, lastly, Emmy is assigned as programmer to develop the software. Then the player asks Yesha to hire tree new employees: Arden, which is placed in training, Marke, an architect, and lastly Daniel, an analyst that will work for 14 hours a day. Almost two weeks passed before Arden finished his training and was allocated to work as programmer.

Starting the third week in the game, the player begins to have financial problems. He is running out of cash. Daniel, due to the extra hours, is tired and later quits. The game continues with a few rearrangements in task. Later on, Arden and Marke also leave the staff because of financial problems. At the start of the next month, and after receiving cash from the contract with the client, the player hires another employee (Miera) as a programmer to replace Arden. At the same week, the player sets Mirax to negotiate with the client, asking to extend the project’s deadline by one extra week, since the deadline was ending. Because of the deadline extension, the staff manages to complete the software in time, delivering the software to the client.

At the end of the session, the *game flow log* was generated by using the collected information from the game (employees, actions, and the project daily progression). The examples and illustrations present at the next section are based on the *game flow log* exported from the game session described in this section, which are in the first weeks of the development process.

## Proof Viewer

The *game flow log* is used by *Proof Viewer* to generate a provenance graph corresponding to the game session. In order to do this, *Proof Viewer* processes the information and interprets it to generate the vertices and edges of the graph, as illustrated by Figure 3. As can be seen, the process is divided by three phases: Processing the *game flow log*, generating the graph, and drawing the graph.

Firstly, the *game flow log* is processed, classifying the information to their corresponding vertex types (*activity*, *entity*, or *agent*), and generating the edges that link each vertex in the graph. To simplify this step, the information extracted from the game is arranged in pairs, where each pair represents two vertices followed by the edge that links them. As such, *Proof Viewer* generates the vertices and edge every time it processes a pair of vertices. Each time *Proof Viewer* process a vertex, it searches the database to check if the vertex was already processed. If the vertex was previously processed, then *Proof Viewer* uses the processed vertex instead of creating a new entry. Otherwise, it creates the vertex. This avoids duplicates in this step, since a single vertex can appear multiple times in the *game flow log* due to the nature of how the *log* is structured: a vertex, another vertex, and the edge that links them. So, with the vertex had multiple edges connecting it, then it would appear multiple times in the log.

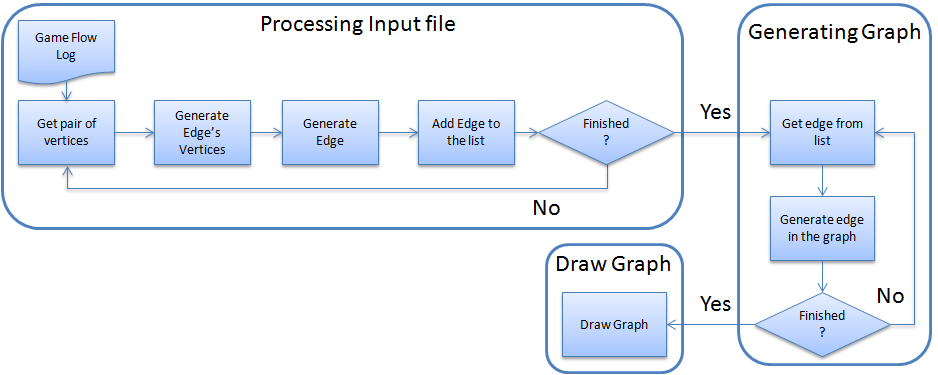


Figure 3: *Proof Viewer* processing the *game flow log* and generating the graph.

After processing both vertices, *Proof Viewer* creates the edge and stores it in a list of edges that is later used to generate the graph. In *Proof Viewer*, an edge contains pointers to the source vertex, target vertex, and edge’s information (value and type). The source and target are the previously processed vertices from the pair. This is done until the entire *game flow log* is processed and all edges are placed in a list of edges that is used to generate the graph. All information from the *game flow log* is processed in this stage, even if they don’t initially appear in the graph, due to filters.

As mentioned earlier, vertices can belong to three types: *activities*, *entities*, and *agents*. When generating the *game flow log* that contains the information extracted from the game, an additional tag is added to distinguish the vertices types. This distinction is used by *Proof Viewer* when generating the vertex in order to choose the correct vertex type, without the need to process the information and decide which vertex it is according to its characteristics, saving processing time. Note however that the input format can be customized, as long as it generates a list of edges, where each element in the list has the vertex source, the vertex target, and the edge information. Table 1 show the number of fields each element has in the generated *game flow log* file from SDM, which is a simple tab separated value file. The structure of each line in the file is composed of: Tag + Vertex + Vertex + Edge.

Table 1: Number of fields for each element type in the input file

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Tag | Vertex: Activity | Vertex: Agent | Vertex: Entity | Edge |
| # fields | 1 | 12 | 15 | 1 | 1 |

The next step is the generation of the graph. *Proof Viewer* uses the generated list of edges, creating each edge in the graph, and, consequentially, the vertices from the edges. It is done this way because *Proof Viewer* uses the JUNG framework (JOSHUA O’MADADHAIN; DANYEL FISHER; TOM NELSON, 2010), where an edge is created by the method *addEdge(source, target, edge)*, which adds the edge in the graph from *source* to *target* with the information *edge*. If *source* and/or *target* are not in the graph, then JUNG automatically generates the vertex. This avoids the need of creating each vertex before creating the edge in the graph, while at the same time, checking for duplicates. After creating the graph, it is drawn on the screen and displayed to the user.

In summary, the process for reading the *game flow log* and generating the graph is: Firstly, *Proof Viewer* reads the *game flow log* and gets the first pair of vertices and the edge information, if any. Then it classifies the vertices according to their types and generates them. In the next stage, it generates the edge with the source and target being the vertices previously created and the information that came along with the pair. Then it adds the generated edge to the list and checks if there are any more vertices pairs to be processed. It repeats this process until the entire *game flow log* is processed. After creating all edges and storing them in the list, it proceeds to the next stage, which is the graph generation. For each edge in the list, it creates an edge in the graph corresponding to the element in the list. After all edges, and consequentially vertices, were created in the graph, *Proof Viewer* draws the graph in the screen.

### Graph Visualization and Representations

After the game flow log is processed and the graph is generated, it is drawn on the screen so the user can analyze it. Figure 4 illustrates the graphical user interface (GUI) of *Proof Viewer*, using the provenance graph generated from the scenario discussed in section 5.3. The provenance graph is displayed at the center of the screen but only a part of it is visible due to the graph size. However it is possible to zoom in and out and navigate through the graph. The graph layout is set to be similar to a spread sheet, were each “line” represents the activities of each agent and each “column” represents a day in the game. The layout can be customized by creating new layouts or using existing ones available from JUNG. The filters, which are customized for SDM, and other features are located at the lower region of the interface. Starting with the buttons, the first one is “Granularity: 7 days”. This button is only an example of grouping vertices together for the same agent. In this case, it groups vertices from the same week. This is useful for huge graphs, which allows summarizing displayed information in a weekly basis.

The “CollapseAgent” button collapses all the agent’s vertices into the agent itself. It can be useful to detect if an agent had any influence throughout the game, instead of looking vertex by vertex. The “Collapse” button allows the user to collapse selected vertices, while the “Extend” button remove the last collapse made to generate the selected vertex. The “Reset” button removes all modifications made in the graph, returning it to the original state. The “MainGraph” and “TutorialGraph” buttons change the displayed graph, where the main graph is the graph to be analyzed and the tutorial graph is a smaller graph used as example. The “Edge Style” is used to change the edge’s arrow curvature from a quad-curve to straight lines. Lastly, the “Mouse Mode” is for changing the function that the mouse will perform. There are two functions: Transform, which is used to navigate the graph, and Picking, which is used to select vertices in the graph.

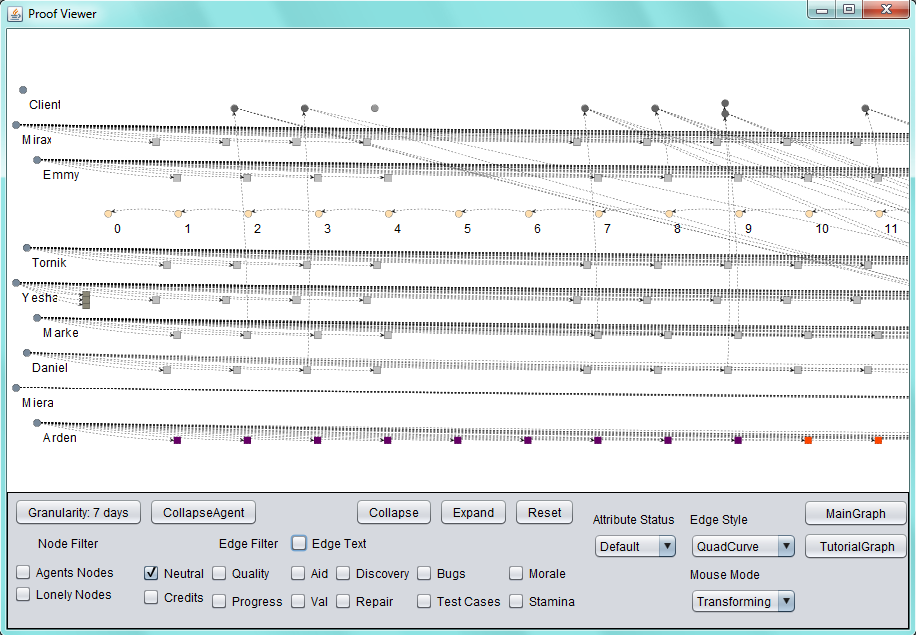


Figure 4: Proof Viewer’s GUI



Figure 5: Same graph from but with all edges

The next items in the interface are the checkboxes used for filters. Starting with “Node Filters”, the “Agents Nodes” hides all agents in the graph. This is just to illustrate the possibility of hiding vertices by type. In this case, it hides *agent* type vertices. The “Lonely Nodes” hides all vertices that have no edge linking them to other vertices in the current displayed graph. This is useful to clean the graph from vertices that have no edges/influences from the selected types being displayed, reducing the number of vertices on the screen. The “Edge Filter” sets the displayed graph to show only the selected types of edges. This is done changing the display status of each edge in the graph, displaying only the types selected while hiding the rest. No information is lost in this process, the information is only hidden from the user. An edge is composed of a value and a type. For example, and edge labeled as *342 validation* has a value of *342* and the *validation* type. In the case of SDM, the edge’s types can be: Credits, Quality, Progress, Aid, Val (short for Validation), Discovery, Repair, Bugs, Test Cases, Morale, and Stamina. Neutral edges are edges that represent association between vertices, while the others represent influences. Black edges are neutral edges, which are also dotted, or edges with a value equal to zero. Note that only the “Neutral” type is selected in the displayed graph shown in Figure 4. This means that the graph is showing only neutral edges. The graph is set to always start the visualization with only the neutral type selected, pre-filtering all other edges. This is just an example of possible pre-filtering. Any type of filter can be used during the initialization of the graph. This is useful to reduce the graph granularity, hiding information from the user to avoid overwhelming him. The full graph can be seen if all edge’s types are selected, resulting in the section of the graph illustrated by Figure 5. The “Edge Text”, when selected, displays the edge label, containing its value and type. This information is also shown as a tooltip when moving the cursor to the edge. Vertices details are also available by moving the cursor over it.

The edge filter is important because it allows for the identification of types of influences in the graph, filtering other influences that are not relevant for the desired analysis. For example, at days 10 and 11, the employee Daniel had drastic changes in his performance, dropping from *342 validation* to *34 validation*, as shown in Figure 6. This was detected by activating the “Val” edge filter. The reason for this sudden drop can be traced to Mirax and Yesha by changing the filter to “Aid”. Yesha provided an aid of 298% in day 10 and a negative aid of 248% at day 11 to Daniel. Moreover, Mirax provided 227% and 136%, respectively for days 10 and 11. By combining these factors, at day 10 Daniel receive a bonus of 525% in his task, while at day 11 he receiver -112%. Thoroughly, Daniel productivity without any bonus was 65 at day 10 and 53 at day 11, which is within his productivity margin.

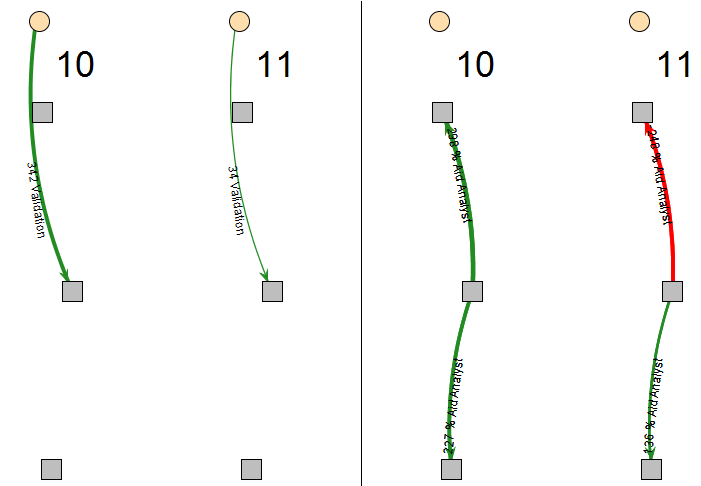


Figure 6: Analyzing Daniel's productivity. Left picture has the "Val" edge filter on. The right picture has the "Aid" edge filter on. Employees are: Yesha (upper tasks), Daniel (Middle), and Mirax (bottom).

The “Attribute Status” changes the vertex color according to their values from the selected attribute. In SDM they can be: Morale, Stamina, Hours (short for Working Hours), Weekend (highlights “Saturday” and “Sunday” vertices), Credits, and Role. The vertex color does not change if it does not have the selected attribute. The default mode colors common activities with a shade of gray and uncommon activities with different colors. Common activities in SDM are normal tasks executed by employees during their roles, while uncommon activities are activities that do not happen frequently. For example, in SDM the uncommon activities are: Idle (red color), Training (purple color), Fired (brown color), Promotion (green color), Hired (“cornsilk” color), and Negotiation (“honeydrew” color). This color difference between vertices is useful to quickly identity non-ordinary events. For example, by looking at the graph shown in Figure 7 it is possible to quickly identify that an employee trained during one week and was idle during the consecutive four days after the training was complete. In addition, Daniel was fired and Mirax was promoted in the same day. Finally, Yesha hired three new employees at the beginning of the game.

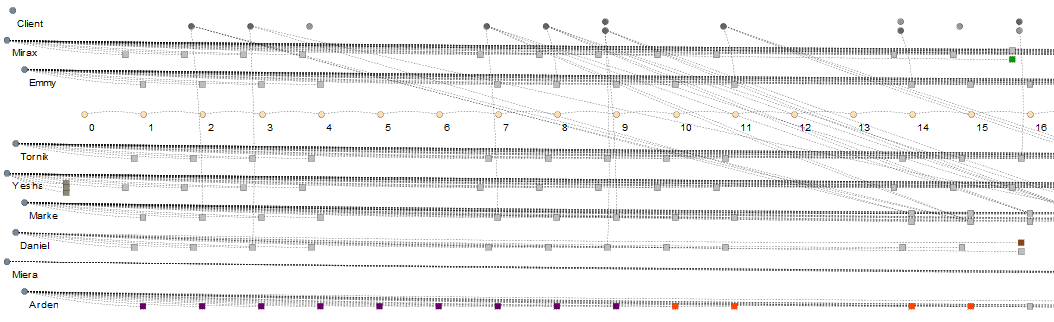


Figure 7: Same graph from with Attribute Status set to default mode.

This type of visualization, based on the evaluation of attributes, is useful to quickly identify particular sections in the graph. Another example in the same scenario is to check the player’s financial status. By changing the visualization to evaluate Credits instead of the default mode, the vertices that have the player’s credits value changes their color according to its status. In SDM, the vertex that contains such information is the Project vertex. By looking at Figure 8, it is possible to see that the player ran out of credits after day 10. It is also possible to identify the source of this problem by activating the “Credits” edge filter, which is illustrated in Figure 9.

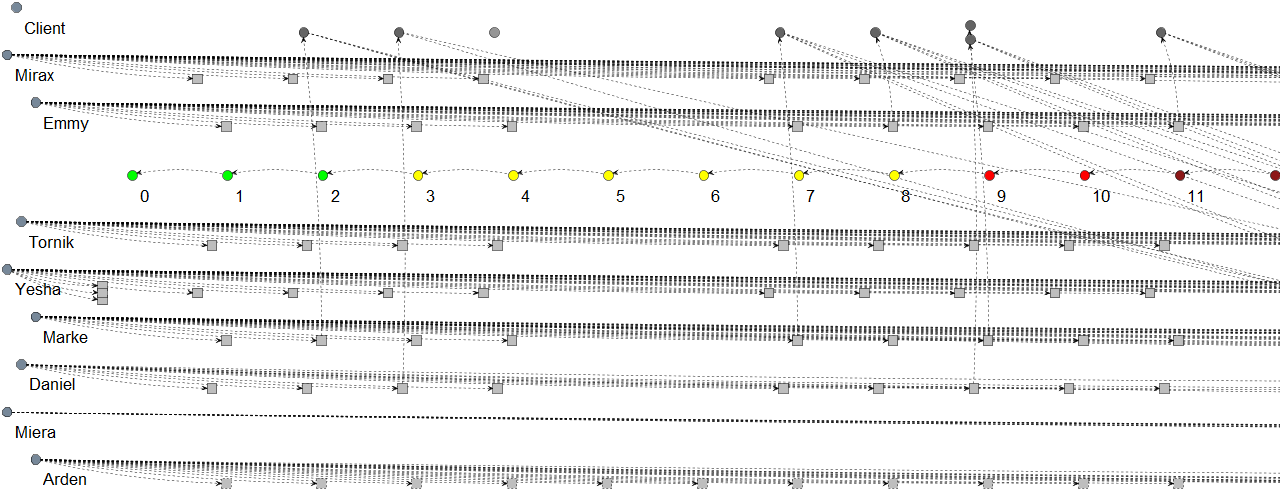


Figure 8: Same graph from Figure 4 with Attribute Status set to Credits mode



Figure 9: Same graph from Figure 4 with Attribute Status set to default mode and Credits edge filter on

As can be seen in Figure 9, the player had many expenses with his employees. Hiring three new employees and training another, as illustrated by the thicker edges at days 0 and 1, was a key factor to increase this problem. It is possible to group the vertices together to better visualize the expenses, as illustrated in Figure 10, which grouped these 11 days. In total, 24,170 credits were spent with hiring, 8,036 credits with Arden, 0 credits with Miera, because she was not hired until this moment, 3,240 credits with Daniel, 1,971 credits with Marke, 1,899 credits with Yesha, 1,809 credits with Tornick, and 2,007 credits with Emmy. Mirax actually generated 6,840 credits for the player by performing her role as marketing, which generates cash and aid analysts.

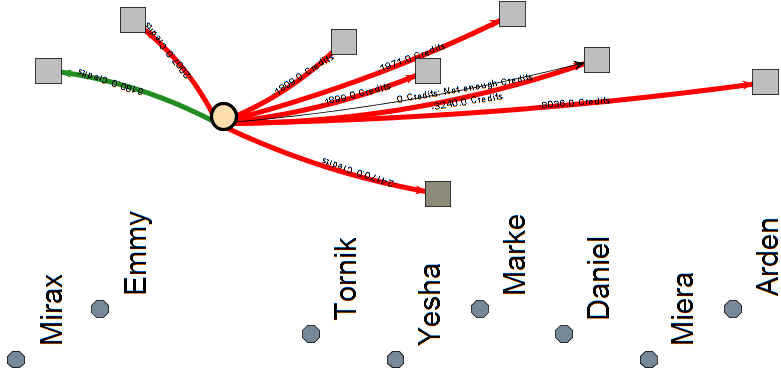


Figure 10: Same graph from Figure 4 with Attribute Status set to default mode, Credit edge filter on, and vertices collapsed (from day 0 to day 11). Figure was rotated by 90º to the left and the edge’s labels were enabled.

## Final Considerations

In this chapter presented details about Proof Viewer, a tool used to display the provenance graph generated by the game SDM. Using the conceptual framework present at chapter 4, SDM is able to generate a *game flow log* to be used by the graph visualization tool *Proof Viewer*. The contents from the *game flow log* are directly related to the information available in the graph. It was also presented features present in *Proof Viewer*, like visualization details and the usage of filters to change the displayed graph.

Even though it was only showed a section of the provenance graph (11 days), the original graph is 40 days long and contains 273 vertices. This number of vertices does not compromise the visualization of the graph and the user analysis. However, graphs with more vertices might generate problems for the user in terms of visualization, overwhelming him with information. To deal with this, it is possible to cluster vertices and the edges together in order to simplify the graph. However, currently, these clustering must be done manually or by using the granularity feature. Nevertheless, *Proof Viewer* provides the necessary features to create complex clustering algorithms, such as clustering vertices if they satisfy specific rules or behaviors.

Even though *Proof Viewer* was customized for SDM, it can be adapted to work with other games. Most resources present in it were designed to work independently of the game context. The few features that are context dependable, like filters, have templates and are based on abstract classes for easy implementation.

# References

ABT, Clark C. *Serious Games*. 1. ed. Abt Books: University Press of America, 1987.

BAKER, Alex; NAVARRO, Emily; VAN DER HOEK, André. Problems and Programmers: An Educational Software Engineering Card Game. *International Conference on Software Engineering(ICSE)*, p. 614–621, 2003.

HIGGINS, T. *Unity - 3D Game Engine*. Disponível em: <http://unity3d.com/>. Acesso em: 5 maio 2011.

JOSHUA O’MADADHAIN; DANYEL FISHER; TOM NELSON. *JUNG: Java Universal Network/Graph Framework*. sourceforge: Open-source, 2010. Disponível em: <http://jung.sourceforge.net/>.

KOHWALTER, Troy; CLUA, Esteban; MURTA, Leonardo. SDM – An Educational Game for Software Engineering. In: 2011 X BRAZILIAN SYMPOSIUM ON GAMES AND DIGITAL ENTERTAINMENT (SBGAMES), nov. 2011, Salvador. *Anais*... Salvador: In: X SBGames, nov. 2011.

NAVARRO, Emily. *Simse: a software engineering simulation environment for software process education*. 2006. California State University at Long Beach, Long Beach, CA, USA, 2006.

PRENSKY, Marc. Fun, Play and Games: What Makes Games Engaging. *Digital Game-Based Learning*, p. 1–31, 2001.