A Scalable Intelligent Tutoring System Framework for Computer Science Education

Nick Green[1], Omar AlZoubi[2], Mehrdad Alizadeh[1], Barbara Di Eugenio[1], Davide Fossati[2], and Rachel Harsley[1]

[1]*Computer Science, University of Illinois at Chicago, Chicago, IL, USA*

[2]*Computer Science, Carnegie Mellon University in Qatar, Doha, Qatar*

{*ngreen21, maliza2, bdieugen, rharsl2*}*@uic.edu*, {*oalzoubi, dfossati*}*@cmu.edu*

Keywords: Intelligent Tutoring Systems, Computer Science Education, Linked Lists, Systems Architecture, Teaching Strategies, Worked-out Examples

Abstract: Computer Science is a difficult subject with many fundamentals to be taught, usually involving a steep learning curve for many students. It is some of these initial challenges that can turn students away from computer science. We have been developing a new Intelligent Tutoring System, ChiQat-Tutor, that focuses on tutoring of Computer Science fundamentals. Here, we outline the system under development, while bringing particular attention to its architecture and how it attains the primary goals of being easily extensible and providing a low barrier of entry to the end user. The system is broadly broken down into lessons, teaching strategies, and utilities, which work together to promote seamless integration of components. We also cover currently developed components in the form of a case study, as well as detailing our experience of deploying it to an undergraduate Computer Science classroom, leading to learning gains on par with prior work.

1 INTRODUCTION

As all seasoned Computer Science (CS) professionals know, starting off in the field is not easy. There are fundamental concepts and principles that must be taught to all students, many of which feel unintuitive and require a new way of thinking. Unfortunately, it is at these early steps that students may decide to choose a different path in their career. This is not just impacting the number of professionals in the computing industry, but society as a whole, since there is now a significant worldwide skills shortage (Beaubouef and Mason, 2005).

These issues may be alleviated by providing higher quality and more accessible CS education in the early stages. A very effective method for learning is to employ human tutors, especially when one-to-one tutoring is possible. Such tutoring can be expensive and impractical, which leads us onto a viable alternative, the Intelligent Tutoring System (ITS) (VanLehn, 2011).

We are developing an ITS, called ChiQat-Tutor, that attempts to provide access to foundational CS topics, with the aim of helping students overcome some of these initial obstacles, increasing the chances of succeeding in a career within the discipline. This will be done by providing lessons in the basics of CS, such as the teaching of fundamental data structures, without the need for personal one-to-one tutoring. The inspiration for this was spawned from a prior computer science ITS, iList (Fossati et al., 2008), which teaches the basics of the linked list data structure. Accessibility is paramount in order to lower the barrier of entry for all students, regardless of location and background. Our system bears this in mind in its fundamental systems architecture.

Here, we outline the ChiQat-Tutor system as a whole which utilizes three major building blocks; *lessons*, *teaching strategies*, and *utilities*, where each can be considered as system modules, or plug-ins. An overview of the system is also given, showing a rich modular architecture which is designed for ease of adding new lessons and strategies, as well as encouraging reuse of system components across all modules. We also give an overview of modules currently in development in the form of a case study, showing how the architecture can be utilized to produce a working ITS. Finally, we deploy the system to a classroom of one of our target demographics, undergraduate CS students, to see the realized product in action.

2 RELATED WORK

ChiQat-Tutor builds on our previous work, specifically, the iList ITS. iList has been shown to be effective in teaching the basics of the linked list data structure, yielding learning gains similarly to that of a human tutor. In iList, students were given a series of problems to be solved. Each problem presents the student with a linked list, as well as a task to complete. For example, the student may need to insert a node between the first and second nodes. This is accomplished by the student typing in Java-like commands, either one-by-one or as a block of code, that will modify the data structure until the desired structure has been constructed. The true power behind this system is its teaching strategies, in this case its various feedback models (Fossati et al., 2009; Fossati et al., 2010; Fossati et al., 2015). If the ITS sees the student making an error, or feels that the student needs some help, it provides natural language feedback to the student to help them get back on track as soon as possible.

iList contains a single lesson type and primarily uses feedback (positive or negative) to aid the user. Our new system takes the lessons learned from this work and expands on its successes. ChiQat-Tutor will be able to handle multiple types of lessons, extend the use of teaching strategies, and include system utilities that may help in experimentation and evaluation for researchers. Another contribution of this system is to increase user accessibility to the system and be susceptible to the addition of new components, both factors that have been catered for at its core.

The literature on ITS is vast and we cannot do justice to it here. Specific concerns relevant to architectural considerations and arising in previous work include relying on online connectivity, as in (Graesser et al., 2005) (Brusilovsky et al., 1996) (Nakabayashi et al., 1997); this is a requirement that we aim to avoid due to potential accessibility issues for some populations (Nye, 2014). Expandability has already been considered in some work, such as with the Cognitive Tutor Authoring Tools (CTAT) (Aleven et al., 2006) that allows the platform to incorporate other lessons. Including new lessons is extremely important for longevity and wider adoption of such a system. Our system's architecture looks at such cases to derive a system which may yield greater value.

Other ITS', such as (Badaracco and Martnez, 2011), also take a view of the system architecture as key to its success. However, they have taken a different approach to us by developing a flexible ITS that works in multiple domains with a fixed pedagogical model. We have an opposing focus; a single domain with flexibility in employable pedagogical strategies.

3 ARCHITECTURE

In developing the new system, we listed fundamental requirements that the system must possess. These requirements indicated that the system architecture would be key. Even though we focus on a limited number of topics and teaching strategies at first, we identified the need for the system to be scalable and flexible, allowing developers to easily add new functionality. Of utmost importance is for the addition of new lessons and strategies while maintaining minimal coupling with the rest of the system. Software reuse is also important in order to help develop new components at speed and with a degree of consistency. Furthermore, it is important for the system to feel as a single application, rather than several applications brought together via bootstrapping. Also, as lowing the barrier of entry is an important quality, we take on board lessons learned from (Nye, 2014), by not relying on high quality Internet connectivity.

Given these basic requirements, we decided to adopt a highly modular desktop application, with a sparse communication with a central server. Minimal communication may promote wider adoption, e.g. students without an Internet connection, and having more processing occur on the client side will also increase responsiveness of the system.

Figure 1 shows an abstract overview of the systems major components for both client and server applications. Given our desire to keep as much processing as possible to be local to the user, the majority of the architecture is concentrated on the client side application. The server side is minimized to act as a system coordinator and data store.

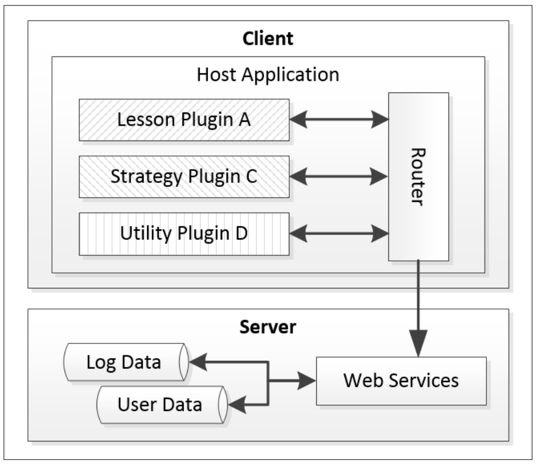


Figure 1: System Block Diagram.

3.1 Client

The novel core of the system is in the client side application, whose strength is in its modular architecture, where components can be created independently, however, can communicate together for a high degree of software reuse. This is incredibly valuable for experimentation among multiple researchers working on components for the project.

There are two main components in the client application; plug-ins and a host application. The host application acts as the entry point and glue of all client components. On execution, it will dynamically find all applicable plug-ins, automatically load, and initialize them. This process of plug-in selection will be invisible to the end-user, yet is configurable within the system. Application specific functionality is encapsulated in *plug-ins*. All plug-ins are derived from a common plug-in type, however, broadly speaking there are three types of plug-ins, those being lesson, teaching strategy, and utility plug-ins. Each plug-in can work with other plug-ins via an internal messaging API. Later, we will look at a case study showing an instantiated application and how these plug-ins can communicate.

3.1.1 Client Host Application

The host application provides a framework for the client side functionality to reside within. Its main function is to: (1) host plug-ins, (2) act as a container for a plug-in's user interface, (3) route messages between various components of the system.

Upon creation of the host application, it will load applicable plug-ins and initialize them as defined in the configuration file. An application can load any plug-in which exposes a set interface and has been defined within the system's blueprint configuration file. When plug-ins wish to communicate with each other, they can send a message through this interface to the application's router, which is hosted in the host application. The application router can either route the message to the destination plug-in, broadcast the message to all plug-ins, or handle the message itself. An example of the host application handling a message could be an application change of layout event.

A plug-in can expose various interfaces that the host application can access. The application can query for these interfaces, and update its own user interface. An appropriate case of this would be when a plug-in exposes a view that it wants shown to the end user when a button is clicked. The host application can choose how it displays the view, for example, as a modal dialog box or within the applications main window. This layer of abstraction exists to handle potential future interfaces, such as touch screen displays or multi-screen setups. The key feature of this style of application is that it can easily be rewritten, for example when porting to a new device.

3.1.2 Client Plug-in

Plug-ins contain all ITS specific business logic and user interfaces for a well defined piece of functionality. Each plug-in is decoupled from all others and will only be able to communicate via an internal messaging system. A plug-in contains a queryable user interface which can be exposed to the host application, as well as a messaging interface which other plug-ins may pass messages through.

The major requirement on the part of the plug-in developer is to fulfill the plug-in's interfaces, which can be done by deriving from the abstract class *BasePluginInstance* (Figure 2) (abstract methods are prefixed with a '-'). In terms of the message passing interface, this will require overriding of the OnMessage method to receive messages. Sending messages can be achieved by either calling the *PostMessage* or *BroadcastMessage* methods. Also, the plug-in can pass back a set of queryable user interfaces, which are JavaFX (Topley, 2010) containers, by deriving from the *PluginViewFx* class and registering it with the BasePluginInterface. The interface, which will be displayed within the application will be wholly operated within the plug-in and changes cannot be made to the application in order to use new interface features. To complement the interface, another abstract class, *PluginController*, can be implemented to provide back-end implementation details.

3.1.3 Lesson Plug-in

When plug-in developers wish to develop a new plug-in, they will have to create a new plug-in and implement the generic plug-in interface. However, lesson developers can bypass much of the standard work by deriving from a specialized lesson controller (Figure 3) and view (Figure 4). This plug-in provides an additional framework where individual lessons can be created within. Many of the lesson basics which have been developed have been abstracted to this level, including a standard user interface with abstracted components for further specialization, along with integration of existing teaching strategies, such as the worked-out example plug-in.

This is completely optional, yet should provide a consistent look and feel to the application, while speeding up the development process.

```java
// BasePluginInstance.java
public abstract class BasePluginInstance
    implements IMessageReceiver
{
   List<PluginViewDefinition> m_ViewDefs;
   List<PluginViewFx> m_Views;
   List<PluginController> m_Controllers;
   BaseMessageReceiver m_Receiver;
   List<PluginMenuItemDefinition>
       m_MenuItemDefs;
   BluePrintNode m_BluePrint;
   AppDataStore m_AppDataStore;

-  boolean Init( PluginConnector connector,
     List<KeyValuePair> args, BluePrintNode
     node, AppDataStore appDataStore );
-  boolean Destroy();
-  void OnMessage(Message msg);
   boolean PostMessage( String dest, Message
       msg );
   boolean BroadcastMessage( Message msg );
}
```

Figure 2: ChiQat Plug-in Interface.

```java
// LessonController.java
public abstract class LessonController
    extends PluginController implements
    IInputInterface
{
   LessonView m_View = null;
   User m_User = null;

-  void OnMessage( Message msg );
-  void CreateBlankProblem();
-  void ExecuteStatement( String s );
-  void GetLessonId();

   LessonController( BasePluginInstance inst,
       String controllerId );
   LessonView GetView();
   void CloseLesson();
   void OnRestart();
   void OnSubmit();
   void OnExample();
   void OnSetProblem( String problemId );
   void EnableInput( boolean bSet );
}
```

Figure 3: ChiQat Lesson Controller Interface.

3.2 Server

The next major component of the system is the server, which has the role of acting as a centralized data manager within the systems ecosystem. The primary role of the server is to provide an online user authentication and profile store, repository for log messages, and binary asset data storage (such as plug-ins and computational models).

```java
// LessonView.java
class LessonView extends PluginViewFx
{
-  boolean Init();
-  void HighlightNode( String nodeId, boolean
     bSet, boolean bPulse, boolean bSize,
     boolean bBrightness );

   void OnViewShown();
   void DisplayFeedback( TutorMessage msg );
   void OnMessage( Message msg );
   void EnableProblemInput( boolean bEnable );
   void SetScratchText( String text, int
       iPageNum, int iLineNum );
   void ClearScratch( int iPageNum );
   void EnableDone( boolean bSet );
   void EnableRestart( boolean bSet );
   void ShowProblemMenu();
   void EnableExamples( boolean bSet );
   void HighlightUiComponent( String
       componentName );
}
```

Figure 4: ChiQat Lesson View Interface.

It is important to note that the client does not require a persistent connection to the server, but rather will provide data to and receive data from a client when possible. The advantage of this is to (1) take computational load off of the server, (2) give the user a more responsive experience, (3) have a highly available service for users with limited online connectivity.

In order to improve the system, it is very important for us to know how users are using the system. Thus we include a very flexible logging system, where the client can send entries to the server for persistent storage. This data can then be mined for user behavior that may help during offline analysis. The system will also be able to log data while offline by storing entries locally until online connectivity can be obtained.

4 CASE STUDY

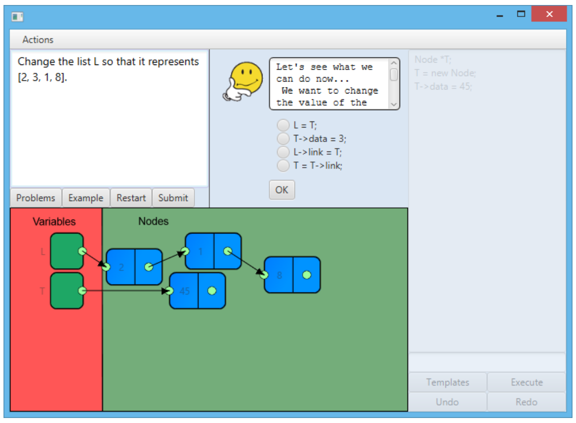
From the ChiQat-Tutor framework, we created a working ITS with numerous facet to demonstrate some of the strengths of the architecture in a practical setting. The ChiQat application covers three different problematic areas for Computer Science students, that being the linked list and binary search tree data structures, as well as recursion theory. Along with these lessons, we have also developed a reusable teaching strategy using *worked-out examples* (Atkinson et al., 2000). Several utilities have also been developed to collect usage data for further analysis, as well as providing an easy to use application for student and researcher. Figure 5 gives an overview of the full system and the relations between components. Here, we describe each of these plug-ins in further detail.

4.1 LESSONS

Initially, the system provides lessons in three topics that students tend to find difficult when encounter them for the first time, these being linked lists, recursion, and binary search trees. Each of which contained as separate modules.

4.1.1 Linked Lists

Based on iList, this plug-in has been developed within the new system's framework (Figure 6). This lesson gives students the opportunity to visually manipulate linked lists in code, with the tutor providing various hints and corrections as needed. Students may experiment within a sandbox or work on one of the 7 provided problems. The advantage in this port is that it can leverage the systems common framework, services, teaching strategies, and common user interface.


Change the list L so that it represents [2, 3, 1, 8].

```
Let's see what we
can do now...
 We want to change
the value of the
```

L = T;
T->data = 3;
L->link = T;
T = T->link;

```
Node *T;
T = new Node;
T->data = 45;
```

Figure 6: Linked List Tutorial.

4.1.2 Recursion

Recursion is a notoriously difficult subject to teach a young computer scientist (Gal-Ezer and Harel, 1998). The recursion module is one of the first modules to have been developed, and has been trialled in an introductory programming class of 60 students (AlZoubi et al., 2014). It teaches students recursion in a visual manner using recursion graphs (Hsin, 2008). Users can use the recursion graph in 4 ways; tracing, validating, constructing, and animating. Questions are also posed to the student to include some reinforcement.

4.1.3 Binary Search Trees

Another module in development is the binary search tree lesson. Working similarly to the linked list tutorial, it gives students the ability to manipulate and perform searching on binary tree structures.

4.2 TEACHING STRATEGIES

Another important aspect of the system is how lessons are taught. There is not just one good way of teaching, there are many strategies available that can be used in different circumstances (Kameenui and Carnine, 1998). Once again, these strategies are implemented within the system's ecosystem as reusable modules. Plug-ins which derive from the abstract lesson toolkit can take advantage of these strategies with minimal effort.

4.2.1 Worked-out Examples

A worked-out example is a fundamental teaching strategy that teaches a concept via examples (Sweller, 2006), (Atkinson et al., 2000). Such an example is broken into three stages; problem formulation, solution steps, and final solution.

Using worked-out examples has been shown to induce greater learning gains for some students. It is also noted in (Renkl, 2005) that due to the step-by-step nature of the strategy, it is best used for algorithmic problem solving, which suits our target domain well. This type of strategy is often employed by teachers to teach some basic concepts. Prior work showed that there may be a correlation in the use of examples leading to learning gains in computer science tutoring (Di Eugenio et al., 2013).

The Worked-out Example plug-in gives the ability for any system plug-in to support this teaching strategy by supporting a well-defined interface via the system's messaging system. The plug-in is split into three major components; WOE engine, definition asset, and an editor for the definition asset.

At its core, a WOE is described via a definition asset (in XML) which is based on a cyclic directed graph, where each node is considered a WOE step. Each step is related to an action, such as the virtual tutor giving some instruction or an instruction being added to a list of solution steps. The steps are then connected to other steps, with the simple case being a series of steps leading from problem to solution in a serial fashion, or more complexity can be added by branching at various steps if instruction needs to be tailored to the user. The configuration file is given to the engine in order for processing. To ease the cre-

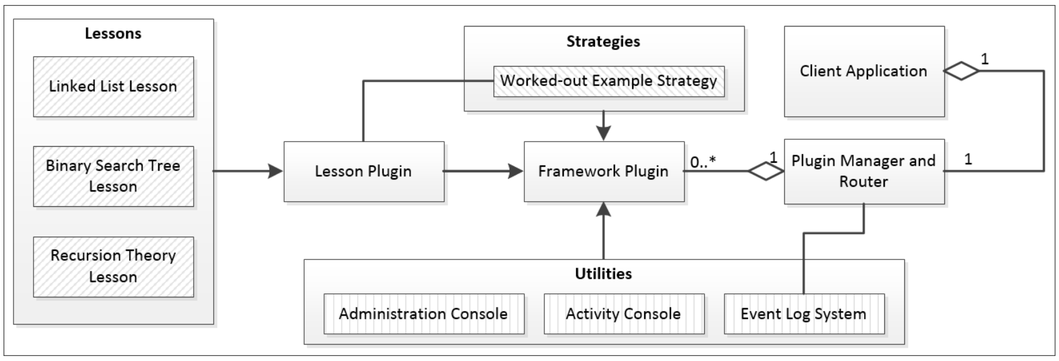


Figure 5: ChiQat-Tutor Plug-ins.

ation of such assets, a GUI editor is also provided to the lesson designer.

Finally, the WOE engine is used to play out a WOE session to the student. This works by executing actions at each step while making appropriate transitions to other steps in the WOE graph. Such actions include printing to the console, displaying a tutoring message, and executing a command for the problem. Lessons can integrate this strategy by acting upon messages sent from the engine.

4.3 UTILITIES

Although students will be learning via lessons, and being taught via teaching strategies, another important aspect are the general system utility plug-ins. Such utilities can be used to perform background tasks or add some additional usability to the system.

4.3.1 Activity Console

The entry point of the system is through a plug-in that exposes an automatic starting property. Here, we have made an Activity Console that is shown on startup. The console handles user log in, account creation, as well as shows all possible lessons and lesson options.

4.3.2 Event Log Utility

ChiQat-Tutor supports global event logging, where text can be logged to the server's log database for future analysis. Such messages may include when a lesson is started, stopped, completed, and when UI components are moved by the user. To ease development, generic ChiQat messages such as starting and stopping a lesson, are handled within the framework.

4.3.3 Administration Console

The ChiQat framework supports the notion of user privileges, whereby a user may be given standard user or administrative privileges. An administrator will have access to the administration console, which allows them to edit system properties, such as individual user settings and control parameters. This is useful while setting up experiments.

5 USER STUDY

The framework and plug-ins have been shown to work together to provide a useful tool for researchers to develop new, reusable ITS concepts that can be experimented with. Another important aspect is how test subjects and end users can use the system. iList has already shown promise in teaching the concept of linked lists to students while using different types of feedback (Fossati et al., 2009; Fossati et al., 2010; Fossati et al., 2015). We conducted a controlled experiment, similar to those iList had been evaluated on, by giving students access to the linked list lesson in ChiQat-Tutor.

5.1 Setup

A set of experiments were run over two lab sessions as part of a second year class in computer science software development. Part of this course included material on the linked list data structure. Over the two lab sessions, students participated in our research by completing a pre-test, activity, and post-test. As part of the activity, students were given access to the linked list lesson, and they could use the system for approximately 40 minutes. The activity was a solo activity where they were able to use the linked list lesson in any way they wished within the constraints of the system. There was only a single condition and all students had access to the same resources.

Each of the two sessions consisted of a 10 minute pre-test, 40 minute activity, and 10 minute post-test. Participating students had their usage of the system recorded, such as actions performed and feedback they received from the system, in the form of log data. Pre and post tests were randomized and anonymized, which were then graded between three independent graders. Two graders were assigned to each test, and in the event of a difference in grades, the third grader would also grade that test. What we report here are our preliminary results from grading.

The conducted experiment is very similar to the ones done with iList (Fossati et al., 2010). The pre/post-tests, duration, and setup, are the same in both sets of experiments. This therefore allows us to compare the results from both sets of experiments with relative ease. However, there are a few subtle differences that may affect the outcome. Firstly, six years have elapsed between the two sets of experiments; iList was evaluated around 2008 and experiments on our new ChiQat-Tutor were conducted in 2014. This six year difference is a variable to bear in mind as students behavior may have changed over time, e.g. the pervasiveness of touchscreen technology may have changed the way students use a user interface. Secondly, there are some demographic differences in the test subjects. The prior system was tested on undergraduate students as in our experiments, but the prior evaluation also included subjects from the United States Naval Academy.

5.2 Analysis

Log system data was collected for each consenting user of the system, which recorded their usage as well as their pre/post-test scores. After removal of erroneous collections, we collected valid datasets for 24 individual students.

Table 1 shows combined statistics for the pre/post-tests performed in all evaluated versions of iList, ChiQat-Tutor, as well as when a human tutor was used in the activity step. In total, there were 5 evaluated versions of iList; 1-3 providing varying levels of reactive feedback, and 4-5 also included varying levels of proactive procedural feedback. In this table, *N* gives the number of subjects in each experiment. Our results show there is a similar mean gain in the new system as with the prior system. This indicates that value has been successfully transferred to the new system.

However, there are some striking differences between the two systems when looking at their usage. Figure 7 tracks two statistics across the seven problem in the linked list lesson; the percentage of students who attempted a problem, and the percentage of students who solved a problem. Data on students who solved problems is measured out of all students

Table 1: Learning Gain of Students.

Tutor	**N**	**Pre-test**		**Post-test**		**Gain**	
		μ	σ	μ	σ	μ	σ
None	53	.34	.22	.35	.23	.01	.15
iList-1	61	.41	.23	.49	.27	.08	.14
iList-2	56	.31	.17	.41	.23	.10	.17
iList-3	19	.53	.29	.65	.26	.12	.24
iList-4	53	.53	.24	.63	.22	.10	.16
iList-5	30	.37	.24	.51	.26	.14	.17
ChiQat	**24**	**.45**	**.14**	**.56**	**.22**	**.11**	**.25**
Human	54	.40	.26	.54	.26	.14	.25

and not just those who attempted that problem. It appears that more students attempted problems in the prior system, however, the proportion of students who actually solved the problem is fairly similar. This indicates that fewer students failed to complete a problem if attempted in the new system. This could be an attractive quality as it may indicate greater engagement. Perhaps given more time, students may be able to solve more problems and make even further learning gains.

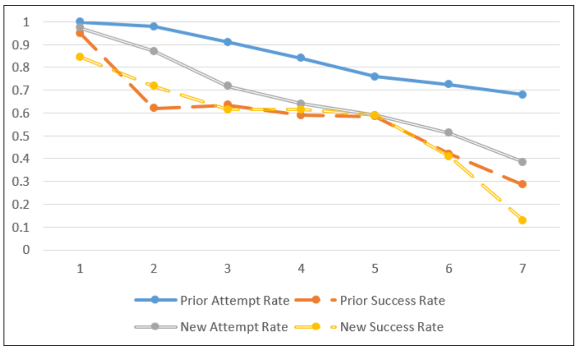


Figure 7: Problem Attempts and Successes in both Systems.

6 SUMMARY AND FUTURE WORK

We have described our new ITS, ChiQat-Tutor, an intelligent tutoring system with a focus on computer science education. The system architecture of ChiQat has been key in its development, as it is highly susceptible to change and encourages ease of reuse. It is through this reuse that new lessons, teaching strategies, and utilities can be developed within the system's ecosystem, which can also enrich existing functionality.

Several modules have already been developed for ChiQat-Tutor, which include 3 lessons, a reusable teaching strategy, and several utility modules. One of those, the linked list plug-in, has been stress-tested by one of our target user groups, undergraduate Computer Science students. By utilizing log analysis and pre/post-testing, we observed learning gains which are comparable to those from a prior system.

From here, there are several avenues of research that can be undertaken. Firstly, other ChiQat-Tutor lessons could be evaluated for potential learning gains, as well as the development of new lessons. Secondly, new teaching strategies could be developed to enhance the existing lessons, which can then be formally evaluated. However, these two paths are not disjoint, since some teaching strategies may be better than others on certain lesson types. Thirdly, given the architecture of the system, it will also be possible to include new utilities that may be of use for experimentation, for example, utilities that record user habits, such as typing speed in problems, or measuring student affect.

Furthermore, we could look into the students themselves as not all people learn in the same manner. Modules could be developed that develop student models which may then direct components of the system, such as suggesting what strategies to use at what times on what students.

ACKNOWLEDGEMENTS

This work is supported by award NPRP 5-939-1-115 from the Qatar National Research Fund.

REFERENCES

Aleven, V., McLaren, B. M., Sewall, J., and Koedinger, K. R. (2006). The cognitive tutor authoring tools (CTAT): Preliminary evaluation of efficiency gains. In *Intelligent Tutoring Systems*, pages 61–70. Springer.

AlZoubi, O., Fossati, D., Di Eugenio, B., and Green, N. (2014). ChiQat-Tutor: An Integrated Environment for Learning Recursion. In *Proc. of the Second Workshop on AI-supported Education for Computer Science (AIEDCS) (at ITS 2014). Honolulu, HI, June 2014.*

Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of educational research*, 70(2):181–214.

Badaracco, M. and Martnez, L. (2011). An intelligent tutoring system architecture for competency-based learning. In *Knowlege-based and intelligent information and engineering systems*, pages 124–133. Springer.

Beaubouef, T. and Mason, J. (2005). Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin*, 37(2):103–106.

Brusilovsky, P., Schwarz, E., and Weber, G. (1996). ELM-ART: An intelligent tutoring system on World Wide Web. In *Intelligent tutoring systems*, pages 261–269. Springer.

Di Eugenio, B., Chen, L., Green, N., Fossati, D., and Al-Zoubi, O. (2013). Worked Out Examples in Computer Science Tutoring. In *Artificial Intelligence in Education*, pages 852–855. Springer.

Fossati, D., Di Eugenio, B., Brown, C., and Ohlsson, S. (2008). Learning linked lists: Experiments with the iList system. In *Intelligent tutoring systems*, pages 80–89. Springer.

Fossati, D., Di Eugenio, B., Brown, C. W., Ohlsson, S., Cosejo, D. G., and Chen, L. (2009). Supporting computer science curriculum: Exploring and learning linked lists with iList. *Learning Technologies, IEEE Transactions on*, 2(2):107–120.

Fossati, D., Di Eugenio, B., Ohlsson, S., Brown, C., and Chen, L. (2010). Generating proactive feedback to help students stay on track. In *Intelligent Tutoring Systems*, pages 315–317. Springer.

Fossati, D., Di Eugenio, B., Ohlsson, S., Brown, C., and Chen, L. (2015). Data driven automatic feedback generation in the iList intelligent tutoring system. *Technology, Instruction, Cognition and Learning*.

Gal-Ezer, J. and Harel, D. (1998). What (else) should CS educators know? *Communications of the ACM*, 41(9):77–84.

Graesser, A. C., Chipman, P., Haynes, B. C., and Olney, A. (2005). AutoTutor: An intelligent tutoring system with mixed-initiative dialogue. *Education, IEEE Transactions on*, 48(4):612–618.

Hsin, W.-J. (2008). Teaching recursion using recursion graphs. *Journal of Computing Sciences in Colleges*, 23(4):217–222.

Kameenui, E. J. and Carnine, D. W. (1998). *Effective teaching strategies that accommodate diverse learners.* ERIC.

Nakabayashi, K., Maruyama, M., Koike, Y., Kato, Y., Touhei, H., and Fukuhara, Y. (1997). Architecture of an Intelligent Tutoring System on the WWW. In *Proc. of*, pages 39–46.

Nye, B. D. (2014). Intelligent tutoring systems by and for the developing world: a review of trends and approaches for educational technology in a global context. *International Journal of Artificial Intelligence in Education*, pages 1–27.

Renkl, A. (2005). The worked-out-example principle in multimedia learning. *The Cambridge handbook of multimedia learning*, pages 229–245.

Sweller, J. (2006). The worked example effect and human cognition. *Learning and Instruction*, 16(2):165–169.

Topley, K. (2010). *JavaFX Developer's Guide*. Pearson Education.

VanLehn, K. (2011). The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221.