Computerized Tutoring

Leah Campbell
Centre College
leah.campbell@centre.edu

Ashley Kieler
Wartburg College
ashley.kieler@wartburg.edu

Adam Shull
Valparaiso University
adam.shull@valpo.edu

July 30, 2007

Abstract

Our research was part of a project to advance a computerized assistant for mathematics tutors. Our work was in the areas of computational linguistics and mathematics education. In the former, we isolated phrases in student-tutor dialogues that referred to equations being passed between the student and tutor. We then wrote a parser to isolate the terms being referred to in the equations. The goal of these two steps was to link the dialogue phrases with the terms they referred to. In the mathematics education area, we identified and categorized student errors during tutoring sessions. In this article, we describe our work and results. The purpose of this research is to allow the computerized tutoring assistant to monitor the student-tutor conversation and automatically offer appropriate suggestions to the tutor when the student is stuck.

1 Introduction

As technology continues to advance in society today, more and more activities are becoming computerized. Tutoring is no exception. However, completely computer-to-student tutoring has not been very successful. A key reason for this lack of success is that tutoring is a dialogue. In order for a student to benefit the most from a tutoring session, it is usually best to talk through any problems he or she may have. Yet most computer systems are not flexible enough to handle this kind of interaction alone. Creating such a flexible program would be a large time and financial investment.

The Wooz program was created as another option to the computer-to-student tutoring system. Instead of computer-taught, Wooz is computer-mediated. This means that a tutor and student, the clients, are talking back and forth, similar to an instant messaging program. All of the dialogue is passed through a server

before it is sent to the other client. Currently, the Wooz program just collects the dialogue and compiles it into a log file. The next step for Wooz, however, is to actually have the computer mediate the conversation. Instead of being a passive program it will actively listen to the conversation and assist the tutor when appropriate.

Another way to think of Wooz is as a computerized tutor assistant. The ultimate goal of the Wooz project is to recognize when a student is having difficulty understanding a concept or procedure and have the program offer a helpful suggestion to the tutor. A major benefit of this architecture is that if the computer misinterprets a piece of the dialogue and provides an irrelevant suggestion to the tutor, then the tutor has the option to reject the suggestion. That way the tutor acts as a buffer between the computer and student so that the computer does not confuse the student further.

In order for the computer to monitor the dialogue between a student and tutor and give suggestions to the tutor, it must be able to understand human dialogue as well as recognize when a student is confused. Thus, our research this summer was focused in two main areas: computational linguistics and mathematics education. Training the computer to understand dialogue is a very large project. Therefore, our research in computational linguistics was a small piece of the overall project. In mathematics tutoring the student and tutor often refer to equations, expressions, or parts of both. So the first step for the computer to understand mathematics dialogue is to train it to link the pieces of dialogue, or referring expressions, that refer to equations to the corresponding parts of the equations. But even if the computer could understand the conversations perfectly, it would still need to be able to recognize student misconceptions, or impasses. Thus, the second part of our research was the study of student impasses. By understanding the types of errors students make and which types are the most common, we can train the computer to handle errors in the most effective manner. The remainder of this paper looks at these two pieces in more depth and discusses how they will both be used in the overall Wooz tutoring program.

2 Computational Linguistics

2.1 Background

Computational linguistics is a field dealing with the computational aspects of human language. It is closely related to natural language processing. Much work has been done in modeling natural language, although it is such a complex field that the work is far from complete. The computational linguistics aspect of the Wooz project is especially difficult because the language that occurs during tutoring is full of mathematical terminology. This makes the type of language the Wooz program must deal with significantly different from the ordinary language on which computational linguistics work has focused. However, many mathematical words that occur during a tutoring conversation can

be understood in the context of the math problem that the student is trying to solve. Therefore, a necessary part of the computational linguistics aspect of the Wooz project is associating parts of the dialogue with the equations* the student and tutor are discussing. This requires identifying the referring expressions in the dialogue as well as the parts of the equation to which they refer. We wrote code to accomplish this in the Python programming language.

2.2 Dialogue

One of the first steps toward linking referring expressions to equation elements is to isolate the referring expressions, or noun phrases, in tutor-student dialogues. This goal encompasses several steps. As a preparatory step, we wrote a small Python program to convert the dialogues from an annotated format to a simple text file of sentences that would be more flexible for us to work with. Once the dialogues were in the right form, we needed to take each sentence and assign every word a part of speech so that the noun phrases could be grouped together and isolated. For this task we chose the NLTK (Natural Language Toolkit) package, which included a tagger and a chunker [1]. A tagger assigns, or tags, each word with a part of speech, and a chunker groups together words according to part of speech. In order for the tagger to assign the right part of speech to a word, it needs to be trained on an already-tagged body of work. This training provides the tagger with the knowledge of which parts of speech correspond to words in an established corpus. The training corpus is large enough that for general speech most words are identifiable; however, math language is not typical language found in standard corpora, which poses a special problem. The tagger will not recognize words which it has not seen in training. Key mathematical words like "denominator" or "factor" may not be recognized and thus may be tagged with the wrong part of speech and not chunked correctly. For the present, we compensated for this weakness by simply tagging any unrecognized word as a noun, which in the majority of cases turned out to be correct. In addition, some words can be several parts of speech, and may also be tagged incorrectly. Below, an example from a tutoring transcript demonstrates the mistagging of the word "factor."

```
Sentence:  Wayne, you cannot factor out a 2, because for the middle
term, 7x, 2 does not factor out.

Tagged Sentence:  ('Wayne', 'np'), ('you', 'ppss'), ('cannot', 'md*'),
('factor', 'nn'), ('out', 'rp'), ('a', 'at'), ('2', 'cd'), ('because',
'cs'), ('for', 'in'), ('the', 'at'), ('middle', 'jj'), ('term', 'nn'),
('7x', 'expr'), ('2', 'cd'), ('does', 'doz'), ('not', '*'), ('factor',
'nn'), ('out', 'rp')
```

The tags are indicated by the letters following each word, where 'nn' stands

*For the purposes of this paper, the term "equations" includes any meaningful group of mathematical symbols, including expressions and inequalities, that represents the problem the student is trying to solve.

for "noun." As can be seen in the italicized terms above, "factor" was tagged as a noun, although it is clearly used as a verb in the sentence. This indicates that the tagger did not recognize "factor" as a verb. A way to rectify this problem is to further train the tagger using this word as a verb.

We experimented with several different taggers and training corpora to find the most useful tagger for our project. First we created a Brill tagger trained on the Brown corpus. The Brill tagger uses transformational learning; in other words, it guesses the part of speech of each word and then goes back through with a set of rules to find and correct mistakes. Our next step was to write some code to access our text file containing the tutoring dialogues, convert each sentence into a list of words, and tag each word. Once our words were tagged, we needed to isolate, or chunk, the noun phrases.

We used the chunker provided in the NLTK package to identify our noun phrases. This particular chunker was based on grammar rules, meaning we were required to specify all combinations of parts of speech that constituted a noun phrase. To help in this process, we analyzed algebra tutoring transcripts from North Carolina A&T State University to find all instances of noun phrases. Once we had defined our grammar rules, we extended our tagging program to chunk the tagged words and output a list of noun phrases found in each sentence.

```
Chunked Sentence:
(S:
  ('Wayne', 'np')
  ('you', 'ppss')
  ('cannot', 'md*')
  (NP: ('factor', 'nn'))
  ('out', 'rp')
  (NP: ('a', 'at') ('2', 'cd'))
  ('because', 'cs')
  (PP: ('for', 'in'))
  (NP: ('the', 'at') ('middle', 'jj') ('term', 'nn'))
  (NP: ('7x', 'expr'))
  (NP: ('2', 'cd'))
  ('does', 'doz')
  ('not', '*')
  (NP: ('factor', 'nn'))
  ('out', 'rp'))
```

```
Noun Phrase List:  [('factor', 'nn')], [('a', 'at'), ('2', 'cd')],
[('the', 'at'), ('middle', 'jj'), ('term', 'nn')], [('7x', 'expr')],
[('2', 'cd')], [('factor', 'nn')]
```

This chunker managed to isolate most noun phrases in the tutoring dialogues, but because in the previous example the word "factor" was incorrectly tagged as a noun, it was also chunked and incorrectly identified as a noun phrase, as can be seen in the italicized terms above. This misidentification can cause difficulties

when noun phrases are being linked to equation terms, since in this example factors in the equation are not actually being referred to. We were curious if a different training corpus would increase our tagging accuracy. Thus, we created a second Brill tagger and trained it on the CoNLL-2000 corpus. We found that there was not much difference between the two taggers so we continued to use the tagger trained on the second corpus.

2.3 Equation Terms

In order for us to parse equations, it was first necessary to examine how the server represents them. Because we were looking at tutoring sessions that had already occurred, we read data from the log files produced by the server. The data is represented using the Extensible Markup Language (XML). Three types of data are stored: a declaration that one of the parties has begun or ended their turn, the text of the chat between the parties, and the equation in the equation box when the message is sent. Each of these types of data is stored as a message, either within an XML tag `<msg>`, or between `<msg>` and `</msg>`. Each `<msg>` also contains attributes identifying the type of message and which party sent it. Here is an example of the way each type of data is stored:

Begin Turn

```
<msg type="begin-turn" party="tutor"/>
```

Chat

```
<msg type="chat" party="tutor"> Tutor: Greeting
```

Equation

```
<msg from="tutor" type="eq" source="1"> &lt;math&gt;&lt;mn&gt;
2&lt;/mn&gt;&lt;mi&gt;x&lt;/mi&gt;&lt;/math&gt;</msg>
```

Our program reads from the log file one message at a time and creates a Document Object Model (DOM) representing the message. It then determines what to do with the message based on the type: it ignores begin-turn and end-turn messages, it passes the dialogue from chat messages to the chunker, and it extracts the actual equation from equation messages.

While the dialogue is stored as a string, making it easy to obtain, equations are stored as MathML, which is an XML application used to represent mathematical symbols. In the example above, where the equation is $2x$, the MathML is the part of the message that contains `<math>` and similar expressions. This is the escaped version of the MathML tag `<math>`. The MathML is escaped in the log files so that when the XML DOM is formed, the MathML is considered part of the message element and not separate elements. The creation of the XML DOM unescapes the MathML, changing the MathML from the above example to `<math><mn>2</mn><mi>x</mi><math>`. The program then uses a new DOM to create a MathML parse tree for the equation. It then goes through the equation and extracts the actual mathematical symbols and puts them together to form the equation. Each symbol is the contents of a token element whose tag name represents the category of symbol: `mn` for a number, `mi` for an identifier such as

a variable, and `mo` for an operator. These tags are helpful because they allow the program to easily distinguish these categories of math symbols. There are also separate element names for representing exponents, fractions, and radicals, which the program identifies and takes care of appropriately; for instance, the program represents x^2 as `(x)^(2)`.

After the program forms the equation from the MathML, we then wrote classes in the program code to represent different parts of the equation. The classes we made are Equation, Term, MinusTerm (a subclass of Term for terms beginning with minus signs), Factor, Superscript, Fraction and Root. If the equation were $2x^2 - 7x + 6$, then there would be the following objects: an Equation object representing the whole equation; Term objects representing $2x^2$ and 6; a MinusTerm object representing $-7x$; Factor objects representing 2, 7, x, and 6; and a Superscript object representing x^2.* All the part objects form a tree, with the whole equation as the root node and parts representing single symbols as leaf nodes. Here is a representation produced by the program of the part tree for $2x^2 - 7x + 6$, with the name of the class of each part next to it.

```
2((x)^(2))-7x+6                 Equation
        2((x)^(2))              Term
                2               Factor
                (x)^(2)         Superscript
                        x       Factor
                        2       Factor
        -7x                     MinusTerm
                7               Factor
                x               Factor
        6                       Term
```

2.4 Linking Dialogue and Equation Terms

Once the referring expressions and equation parts are found, they will be matched up with each other. This will help the computer to understand the referring expressions, as they will then have meaning specific to the problem the student is trying to solve. This way, the computer can determine whether or not the student's ideas for the specific problem are correct, and if they are not, the computer will better know what suggestions to give to the tutor. Table 1 shows some examples of referring expressions and their corresponding equation parts:

*Note that even though x^2 is a factor of $2x^2$, it is a Superscript object because Superscript is used instead of Factor to represent a factor with an exponent. Fraction and Root are similar.

Most Recent Equation	Noun Phrase	Reference
$(2x-1)(x-6)$	your first terms	2x, x
$2x^2-5x+1=0$	the leading coefficient	2
$2x^2-7x+6$	the last term of the trinomial	6
$x-\frac{5}{4}=\sqrt{\frac{17}{16}}$	the denominator under the radical	16
$(2x-3)(x-2)$	the product of your inner terms	$-3x$

Table 1

Since there is still more work to be done on the chunker and equation parser, we have not yet been able to make the program link the referring expressions and equation parts. Even when that is done, there will still be certain referring expressions that cannot be easily interpreted by the computer and certain equation parts that the program is not yet able to identify. Therefore, there is still much more work to be done on this part of the Wooz project, both in expanding the number of noun phrases the program can understand and creating more methods to extract more parts from the equation.

3 Mathematics Education

3.1 Background

Categorizing student misconceptions and errors in algebra is not a new idea. McArthur, Stasz, and Zmuidzinas identified four categories of student errors in their work on algebra tutor expertise [3]. VanLehn et al. stated that learning is associated with students reaching an impasse, or lack of understanding. They then go on to precisely define the term impasse as the point when a student gets stuck, detects an error, or expresses uncertainty about an action that has been performed correctly [4]. Chae et al. used this definition to categorize student algebra impasses in their work on novice versus expert tutor behavior [2]; similarly, we built upon the research of VanLehn et al. and expanded the term "impasse" for our own project.

3.2 Student Impasses

For the purposes of the Wooz tutor assistant, we have defined impasse as a point in the tutoring conversation when progress toward solving a given problem is halted, including when a student answers a question incorrectly, makes an error in implementing a procedure, or asks a question. We looked at a number of algebra tutoring transcripts from North Carolina A&T State University to identify all of the impasses that occurred. These transcripts were from tutoring sessions conducted by tutors ranging from expert to novice. We then developed several categories into which student impasses in our transcripts could be fit: concept, stuck, implementation, arithmetic, trial and error, and near-miss. A *concept* impasse is when a student questions the reasoning behind an idea or

procedure. A *stuck* impasse occurs when a student does not know what to do to begin or continue a problem. There are two subtypes of stuck impasses: deadlock and off-track. In a *deadlock* impasse, the student indicates that he or she does not know what to do to begin or continue solving the problem and waits for the tutor to provide guidance. In an *off-track* impasse, the student provides a clearly incorrect answer, implying that he or she does not understand the problem, yet is still trying to solve it. The third category of impasse is an *implementation* impasse. This type of impasse occurs when a student knows what to do to solve a problem but not how to do it. For example, the student asks how to implement a procedure, answers a question incorrectly but still knows what to do to complete the problem, or completes a problem incorrectly but in such a way that implies he or she understands the overall procedure. An *arithmetic* impasse occurs when a student knows what to do and how to do it but makes an error, such as a sign error or multiplication error. In addition to these four main impasse categories, we defined two other special cases, the *trial and error* impasse and the *near-miss*. While using the trial and error method, students often reach a unique kind of impasse. This impasse occurs when a student guesses incorrect factors and is corrected by the tutor. Such factoring errors are inherent in the trial and error method as the student endeavors to find the correct factors; thus, this type of error merits its own special category. A near-miss occurs when a student gives an answer that is not entirely incorrect, but not what the tutor was looking for, and is thus corrected by the tutor. Examples of each of these impasse categories can be found in Table 2 below.

Impasse Type	Impasse
Concept	**Tutor**: the 5th root of 32 is 2. **Student**: is the 5th root of 32 2 because 32 is a perfect square?
Stuck: Deadlock	**Tutor**: Please solve the linear equation **Student**: Student: 1st Im not really sure how to start
Stuck: Off-track	**Tutor**: What method do we use here? **Student**: We use the distributive property. **Tutor**: No, the distributive property is used when we are multiplying a term across addition or subtraction.
Implementation	**Tutor**: What is the first step when completing the square? **Student**: everything must be on one side and set equal to zero **Tutor**: Well, when completing the square , we want x-terms on one side and the constant on the other.
Arithmetic	**Student**: $(2x-3)(x-2)$ $2x^2+4x-3x+6$ **Tutor**: Check your sign when you multiplied the two outer terms.
Trial And Error	**Student**: $2x^2-7x+6$ $(2x-2)(x-3)$ $2x^2-6x-3x+6$ **Tutor**: Try rearranging your factors for 6.
Near-Miss	**Tutor**: What operation are we performing here on the two polynomials? **Student**: foil **Tutor**: Well the operation is multiplication; however, we do use FOIL to multiply two binomials.

Table 2

Within four of the six impasse categories, we also used subcategories to specify which area the error was in. The subcategories were introduced as necessary; in other words, they were not predefined. Because trial and error and near-miss impasses are special categories, we excluded them from this extra characterization. Table 3 below lists the subcategories used with each impasse category.

Impasse Type	**Subcategory**
Concept	Radical
Stuck: Deadlock	Factoring LCD (Least Common Denominator) Operation Completing the Square Exponent Substitution Radical Solving Equation
Stuck: Off-track	Expression Definition Factoring LCD (Least Common Denominator) Fraction Solving Equation
Implementation	LCD (Least Common Denominator) Solving Equation Completing the Square Fraction Factoring Distribution Radical FOIL Definition Exponent Substitution Expression
Arithmetic	Addition Subtraction Multiplication Division Sign Error Like Terms Substitution
Trial And Error	None
Near-Miss	None

Table 3

In addition to these six *understanding-driven* impasse categories, we used VanLehn's impasse examples as a springboard for defining a separate set of categories to define how impasses were signaled. VanLehn's three categories are when a student gets stuck, detects an error, or expresses uncertainty about an action that has been performed correctly [4]. Our *signal* categories expand upon this idea to include when a student detects his or her own error (SE), when

the tutor detects an error (TE), when the student expresses uncertainty and provides some sort of statement or guess that the tutor considers correct (UC), when the student expresses uncertainty and provides some sort of statement or guess that the tutor considers incorrect (UI), and when the student expresses uncertainty but provides nothing for the tutor to evaluate as correct or incorrect (UW). Table 4 below provides examples of each of these categories from tutoring sessions. Once again, we did not use this extra categorization with the trial and error and near-miss impasses.

Impasse Category	Impasse Example
Student Detects Error (SE)	**Student**: $(2x+3)(x-2)$ $2x^2-4x+3x-1$ THIS IS NOT CORRECT
Tutor Detects Error (TE)	**Student**: $3x^2+15x-2x+10$ **Tutor**: Check your sign for the last term
Uncertainty, Correct (UC)	**Student**: $3x^2+15x-2x-10$ $3x^2+13x-10$ I am not sure if my terms are correct for the inner terms **Tutor**: Your terms are correct.
Uncertainty, Incorrect (UI)	**Tutor**: what exponent rule does the problem use? **Student**: Is it the one when you multiply the exponent by the other number? **Tutor**: No, the rule you are using is for solving equations.
Uncertainty, No Work (UW)	**Tutor**: do you know how to start? **Student**: I don't know

Table 4

Using these two sets of categories, we marked up the tutoring transcripts. Each impasse was given one of the six understanding-driven categories; then, if it was not a trial and error or near-miss impasse, it was given a signal category as well. Our results were recorded in spreadsheets, which provided a simple way to review our data and look at impasse trends.

3.3 Impasse Analysis

As would be expected, the most common error subcategories in each problem corresponded to the problem type. For example, in the problem that required factoring, factoring errors were the most prevalent. However, in the overall problem set, we looked at the six basic understanding-driven categories rather than specific subcategories.

Looking at our entire problem set, implementation impasses were by far

the most common category of impasses, occurring 59.2% of the time. This percentage suggests that the Wooz tutoring assistant must be able to provide an extensive body of information on how to complete mathematical processes. Specifically which processes need to be provided depends upon the problem being completed. Stuck impasses made up 15.7% of the total impasses, indicating that Wooz must be able to help guide students who are at a standstill. From this 15.7%, 8.5% were deadlock impasses and 7.2% were off-track impasses; again the subcategories depended upon the problem being completed. In the concept, arithmetic, trial and error, and near-miss categories, impasses occurred 0.4%, 9.9%, 10.8%, and 4.0% of the time, respectively. Although less common, Wooz's treatment of these impasses must be considered. Wooz needs to be able to respond to conceptual questions as well as identify arithmetic errors. If the student is to effectively use the trial and error method, Wooz must allow a certain number of factoring errors before intervening. In addition, Wooz must be able to gracefully handle situations in which the student does not give an expected answer or perform the expected action but is not incorrect.

Beyond the impasse categories above, some interesting data emerged when we looked at the signal categories. Students rarely noticed their own mistakes, in fact only 1% of the time. Interestingly, however, tutors stopped a student because of an error 53.4% of the time and students expressed uncertainty to the tutor 45.6% of the time. This means that the number of instances where tutors signaled an impasse and those where students signaled an impasse were roughly equal in our data. Thus, Wooz must be adequately equipped to both identify errors and respond to student uncertainty. Within the 45.6% of uncertainty cases, students were correct only 10% of the time, were incorrect 14% of the time, and simply showed uncertainty without providing anything for the tutor to evaluate the remaining 22% of the time.

In addition to examining the most common types of impasses, we also tried to find a correlation between how different levels of students solve problems. To accomplish this, the teacher of the students who participated in the tutoring sessions assigned each student a level number according to that student's math capabilities. A level of one was given to the students showcasing the lowest math capabilities, while a level of five was given to the students showcasing the highest math capabilities. Since we were dealing with a relatively small data set, we compressed these five levels into two groups: low level, consisting of levels one through three, and high level, consisting of levels four and five. We used a contingency table to determine if the percentages of impasse types of the two groups came from the same underlying expected distribution. In order to obtain the best results using a contingency table, a caveat of the contingency tables is that only one-fifth of the counts can be less than five. However, our data sets were small and there was only one concept impasse in our transcripts. Therefore, excluding the concept category, we had 225 impasses among five categories. A χ^2 test on the 5×2 contingincy table showed that there was a 95% chance ($p < 0.5$) that the low level and high level students exhibit different patterns of impasses. All this means is that these high and low level students did in fact behave differently when they worked on these problems. Though we

do not expect this information to be used for the Wooz project, it is interesting from a math education standpoint.

4 Conclusion

Our work in linking referring expressions to equations and categorizing student impasses is a small part of the overall Wooz project. However, it can contribute in important ways. Firstly, by keeping track of which phrases in the tutoring dialogue refer to which parts of the equations being passed back and forth, Wooz can more actively monitor the tutor-student conversation. This switch from passive program to active monitor will allow the computer to automatically provide more appropriate suggestions to the tutor based on what is currently being discussed. In addition, by identifying which types of student impasses are the most prevalent, Wooz can be trained to handle these errors most effectively. Each area in which Wooz can be fine-tuned will make it more effective for the next group of students who use it.

References

[1] Steven Bird, Ewan Klein, and Edward Loper, *Natural language processing in Python*, 2007, Available from http://nltk.sourceforge.net.

[2] Hyeun Me Chae, Jung Hee Kim, and Michael Glass, *Effective behaviors in a comparison between novice and expert algebra tutors*, Proceedings of the Sixteenth Midwest AI and Cognitive Science Conference, 2005, pp. 25–30.

[3] David McArthur, Cathleen Stasz, and Mary Zmuidzinas, *Tutoring techniques in algebra*, Cognition and Instruction **7** (1990), 197–244.

[4] Kurt VanLehn, Stephanie Siler, Charles Murray, Takashi Yamauchi, and William Baggett, *Why do only some events cause learning during human tutoring?*, Cognition and Instruction **21** (2003), no. 3, 209–249.