**Topic Overview**

A conversational chatbot is a program that can understand language and generate intelligent, human-like responses. The input and output of the program are specific to an application. For instance, many chatbots are designed as personal assistants such as Google Assistant [1], Cortana [2], Siri [3], and more. These take in either text or spoken language and generate rapid responses, usually by utilizing an internet connection to the company’s servers. Weather, news, daily events reminders, scheduling, and just having a conversation are examples of what these types of chatbots are often used for. Businesses use chatbots to improve customer service like Geico’s assistant [4] or RightClick [5], which is a chatbot that asks you questions to build a website for a business. Even simpler cases where preprogrammed answers are the only inputs are used for applications in customer service for directing callers to specific departments, or answering questions. These generally have inputs like ‘yes’ and ‘no’ and will not know how to response to anything but those. We decided to investigate making a conversational chatbot that is intelligent enough to respond to any user typed input. The first thing that directed the project was the application. The main objective was to implement a parallel program, so we chose fun over functional in terms of application. A chatbot that you could simple talk to and have it respond with seemingly human-like responses. To build a chatbot which exhibits indistinguishable human-like responses with correct grammar and a wide variety of responses, one would need to include a large amount of training in combination with the right model tuning. Such chatbots are made and improved every year. The Loebner prize [6] is an award given to whomever can produce the best quality chatbot that is indistinguishable from a human. It is inspired by the Turing test, created by Alan Turing, a British mathematician who was largely contributive to field of natural language processing. Our goal, for the scope of this investigation, was to develop a chatbot that would utilize a parallel program to improve over its counterpart, the sequential program. The next section will discuss the computation problem we aimed to solve. Following that is a literature review of three papers we studied. Then the program is described for both the sequential and parallel versions. Following that is an analysis of strong and weak scaling with consideration of reasons for non-ideal scaling. Then we finish with a discussion of future work, and conclude with what was learned during this investigation.

**Computational Problem**

The program takes in user inputted text and uses machine learning to generate a response. Using a corpus of text files as training data, taken from Cleverbot [7] conversations, this program traverses a sentence tree to generate a response. The program generates many responses (controlled by the user) and decides on the best response by performing a reduction on fitness. This is done with a heuristic that we chose to represent aspects such as word mixture, word relevance, and sentence length. There is one final output for each input to simulate a typical conversation. The computation problem we solved specifically is traversing the sentence tree based on the input text and number of test responses to generate. This is done both in sequence and in parallel using the Parallel Java 2 (PJ2) library [8], [9], [10] to learn more about the effects of parallel programming.
Literature Analysis

Paper 1

The first piece of literature that was considered, Towards a Method for Evaluating Naturalness in Conversational Dialog Systems [11], addressed the problem of determining naturalness in a conversational chatbot. This refers to how well the chatbot can maintain a natural conversation flow, which cannot always be answered in a strictly objective manner. A quantitative metric for performance does not give a complete picture of the overall performance. The purpose of a conversational chatbot is to respond to the user in an intelligent way, that also adheres strongly to the application at hand. Consider some of the applications mentioned in the Topic Overview section. For an automated call system where the user is expected to respond with only selective responses, like ‘yes’ or ‘no’, then purely quantitative metrics would do just fine. On the other hand, if the user was talking to a chatbot that was developed to respond as human-like as possible then the evaluation would have to include qualitative metrics as well. This can be done through something like a questionnaire that the user fills out after having a conversation with the bot. These are subjective evaluations that will certainly differ by the user. The stronger the evaluation the better improved the chatbot can become. Large amounts of user feedback and usage data can make this happen for the state of the art chatbots. The short answer is there is no general way to judge the performance of one chatbot against another unless they are for the same application. This paper surveyed current practices in evaluating conversation bot responses. It discussed the duality of quantitative and qualitative aspects of evaluation. The authors provided a methodology to determine the effectiveness and naturalness of a dialog system by proposing a performance function to evaluate the total effectiveness of a dialog system in relation to a task. We used this information to validate the performance of our chatbot. Rather than make surveys and test on humans we did subjective testing on ourselves. Potential future work, that will be discussed towards the end of this report, is in extending the survey of this chatbot to further improve the effect of the heuristic we chose in determining fitness. To determine the effectiveness of the sentence tree search we used quantitative metrics, like timing the program for different input and output constraints such as maximum sentence length. The following papers describe the theory behind the implementation of making an intelligent agent, and what is referred to as fuzzy logic in natural language processing.

Paper 2

Making a Clever Intelligent Agent: The Theory Behind the Implementation [12] addresses several problems. The first is collaborative communication, which is the concept that each participant in a conversation wants mutual understanding of the topic, and the purpose of conversation is often to allow both participants to understand the topic by the end. The next is the concept of availability, anchoring and adjustment. Availability is the idea that people usually believe things to be more like their own understanding than they might actually be. Anchoring and adjustment is a common approach to understanding, which involves making a model of the current topic - an anchor - and slightly modifying it based on the conversation. The third item that the paper addresses is the Maxim of Relation, which indicates that a statement doesn’t need to fully reflect all possible information because both participants can infer from context what the statement means. The solution presented to create clever agents is not to imitate these patterns, but take advantage of them. By acknowledging that people communicating with the chat bot will fulfill these observations, the clever agent can gain more information from the conversation than is strictly said, as well as appear to know more than it may actually know. In our project, we use this in our response heuristic for evaluating responses. The concept of anchoring and adjustment can be utilized to keep conversation on-topic and guide the
human’s adjustment of the topic. The Maxim of Relation is also used to primarily provide positive or acknowledgment of responses instead of attempting to express all possible information.

**Paper 3**

*Fuzzy Logic in Natural Language Processing* [13] describes using fuzzy sets for the purpose of natural language processing. Fuzzy sets are appropriate for language, because language often does not have a single correct solution, and words and phrases often have multiple usages. Fuzzy sets can be used to break down and classify words, and apply natural logic to the fuzzy sets. In our program, we use the concept of fuzzy logic and sets in our heuristic. We classify words based on length, popularity, and weighting based on our language. This allows us to assign a fitness based on how well the words fit into the various sets, and then compare the fitness values for the responses to choose the response with the highest fitness. Figure 1: Fuzzy logic visual illustrates the type of logic used in natural language processing. ‘The’ has high usage, short length, is generally evenly distributed within sentences and usually precedes longer words. The word ‘understand’ has low usage, often used with negative words (like I don’t understand), is often used in questions, and has a relatively long length.

![Figure 1: Fuzzy logic visual](image)

**Program**

The program was designed to take three arguments; The number of responses to test per prompt, N, a file containing the input prompts, and a set of files containing training data. The second argument, the file containing the input prompts, is a single file with several input lines. This allowed us to test many input prompts in one run to focus on the parallel program, rather than the human-computer interaction. In a real conversation, however, the program would generally respond to one input at a time. The third argument, the training data, is read by the program first. Then a ‘Words’ object is populated. This is the program’s known language. The responses that it generates can only be comprised of words from this training corpus. This is a sequential part of the program for both versions (parallelizing this is discussed in the Future Works section). Once that is finished, for each input prompt the program performs the following; Initialize a reduction variable, and for each value of n from 1 to N, the program generates a response that corresponds to that n value, calculates the fitness of the response based on our heuristic, then reduces with the reduction variable (or thread-local reduction variables). The best result is printed at the end. The only difference between the sequential version and the parallel version is in the step where N responses are generated and reduced to one. For the sequential version, each response is generated one after the other and only the most fit is kept by the end due to the reduction variable. In
the parallel version, a parallelFor loop is used to divide up the N responses among team threads that each do what the sequential version does and finish by reducing all best results from each team thread to the response with the overall maximum fitness. Figure 2: Program flow chart shows this process in a flow chart where nested loops are represented in the horizontal direction. The last column of actions is either handled in sequence or in parallel between the two versions. In the middle column, either a regular for loop with a reduction variable is used or a parallelFor loop with thread-local reduction variables is used.

![Program flow chart](image)

Sentences are generated using a sentence generation tree, or hash table. Figure 3: Sentence generation tree shows a small example of what this tree might look like. Keys are indicated in the tree, which are either START, END or a word. Each key has values, and each value has weights. Each word is gathered into a response sentence one at a time. Whichever word is the current word, the program uses its values and their associated weights to select the next word. The weights provide likelihoods of word selection based on a weighted random draw. Sentences start with START words and end on END words. We limited sentence length to 25 words to keep the responses reasonable.
The next step in the program is to reduce on fitness. Response fitness was determined using a heuristic weight function. Figure 4: Heuristic weight function shows the exact parameters we used in our final adaptation of the chatbot. Sentence weight corresponds to the last column in Figure 3: Sentence generation tree. Relevance is the number of like words between the output response as the input prompt. Word mixture refers to the variability of word length in the response. Sentence length refers to how similar in length the prompt and response are and usage mixture refers to the mix of common and uncommon words in the response.

\[
\text{Weight of a response} = (0.2 \times \text{Sentence weight from Words}) + (0.35 \times \text{Relevance weight}) + (0.1 \times \text{Word mixture weight}) + (0.15 \times \text{Sentence length weight}) + (0.2 \times \text{Usage mixture weight})
\]

The program includes a heuristic reduction variable class. This class implements Vbl in the PJ2 library. It contains two fields; a response string and a fitness value. The class can get response for printing at the finish of the program, and perform a reduction. This takes a new fitness associated with a new response and compares to the current fitness. The maximum fitness and response is kept. Please refer to the appendix for a developer and user manual for the program.
Scaling

By either increasing the number of cores used (strong scaling) or increasing the number of responses to evaluate along with the number of cores (weak scaling) we can quantitatively evaluate the performance of this chatbot.

Strong

Figure 5: Strong scaling running times and efficiencies vs. number of cores shows the running times against number of cores on a logarithmic scale as well as the efficiency of the program against number of cores. We see a non-ideal efficiency drop as we increase the number of cores. This is likely due to sequential overhead from the ‘Words’ object. The best way to cure issue would be training separately from generating a response. This is typically done using a server and database and training before the program is used. Another cause for non-ideal scaling is the prompt-response flow. We chose to input several prompts to the program rather than one at a time, to assess it easier. This requires the program to do sequential portions more in one run that if it were to just generate a single response. The last reason we suspect for non-ideal scaling is the varying-length responses. Some responses might only be 2 or 3 words long, while others could be closer to 25 words. The amount of time it takes to evaluate the heuristic on a given response varies by the length of the response, and while this isn't large in comparison to the total number of responses tested, the distribution of length of responses across the tested indexes of responses is effectively random (based on the test data). If the randomness happened to make longer responses fall on higher indexes, the program would scale worse for that testing. Table 1: Strong scaling results in the Scaling Results section of the Appendix gives exact values for these plots.

Figure 5: Strong scaling running times and efficiencies vs. number of cores
Weak

Figure 6: Weak scaling running time and efficiencies vs. number of cores shows the same results as Figure 5: Strong scaling running times and efficiencies vs. number of cores under weak scaling. The runtimes are not too bad, but we do see some non-ideal scaling as the number of cores and problem size (number of responses to test) increase. Likewise, in the efficiency plot we see very non-ideal scaling. The cause of this under weak scaling is similar to that of strong scaling, however, here the sequential overhead does not play as big of a role in reducing efficiency. Because the sequential portion that deals with populating the 'Words' object stays the same the part of the problem that is increasing is what the parallelFor loop executes. If we were to increase the training size we would expect to see an even more non-ideal scaling effect. The second and third reasons discussed before were about the prompt-response flow and varying-length responses. These are still affecting the efficiency here because the program must still compute responses for each input in sequence since this simulates a real conversation flow. The latter, varying-length responses is still in effect here as the program must still evaluate each response heuristically.

Future Work

As mentioned earlier we aim to improve this program by evaluating it more extensively. One of the ways to do this is by testing it on humans and having them fill out questionnaires about their experience. Human feedback is very beneficial to this type of software since its purpose for most applications is to appear to be human. Quantitative evaluation would improve by changing the programs structure. First, creating a database for training data will greatly improve the efficiency of the program as discussed in the Scaling section. This allows the program to spend most of its time on evaluating a response to the user’s input prompt, rather than training at the start of each run. Furthermore, training could be sped up by designing a program to train in parallel as well. We considered this at the start of the investigation and eventually focused in more on the response generation side of things. Continuous learning is a possibility too. The chatbot was trained on Cleverbot conversations between two chatbots. Taking in
user conversations that happen with this chatbot can be used to further improve its capabilities. This would allow it to learn as it goes. The prompt-response flow, which we suspected contributed to the non-ideal scaling, could be changed so the program operates in real time with inputs one at a time. Lastly we note that we did not take into account punctuation and grammar. Adjustments like capitalizing starting words, adding periods to the end of sentences could all be added to the capabilities of this program. These processes would likely contribute to the sequential operation of the program but in theory should not take up much time at all. Spell check is very common and works very quickly even for large documents. Anyone who has written in text editors and office tool software knows this from experience. The potential for these chatbots is incredible and ever-growing in business and social communities. We hope to improve our chatbot program and its use of parallel programming to better our understanding of the subject, and to have fun. It can be quite enjoyable to read the responses and see how they change and improve over time.

Conclusion

With the growth of artificial intelligence in the recent past, the results of the techniques and applications discussed in this investigation will certainly be seen in other applications to come. It is how most chatbots work today and with the growth of machine learning and natural language processing it can only get better. We learned that even with a small amount of training data satisfactory responses can be achieved in a basic conversational chatbot. Getting relevant response can be more difficult that getting understandable (or grammatically-correct) responses. The type of training and how it is utilized by the main program, whether it be online training or offline training, can significantly impact the effectiveness and efficiency of the chatbot. We saw how the prompt-response flow of our program was bit hit to performance. This is one of several pieces that can be refined to make an effective chatbot. We also learned how the heuristic directly effects the output of the program. In our final version we increased the weight on word relevance to be greater than all others. This was observed directly in the responses. We saw a lot of similar words in the response. For example, “Hello, how are you doing?” would likely return something along the lines of “Hello, I am doing fine.” This, along with the other pieces to the heuristic can be tuned to make the chatbot act in the appropriate manner, which generally reliant on the application in specific.
References


Appendix

Manual

Developer
This is a developer’s manual for compiling the program.

Source Code Files:

CleverSeq.java - main sequential program
CleverSmp.java - main parallel program
Formatting.java - utilities for formatting sentences
Heuristic.java - heuristics to evaluate responses
HeuristicVbl.java - Vbl to reduce the best response
Words.java - representation of the known language

Compiling:

Before compiling, the environment must be set up with pj2 in the classpath. See https://www.cs.rit.edu/~ark/runningpj2.shtml for full instructions about setting up the environment on the CS machines. On the multicore parallel machines, an example setup command is:
$ export CLASSPATH=./var/tmp/parajava/pj2/pj2.jar
Once the classpath is set, and all the above java files are in the current working directory, compile with
$ javac -cp pj2.jar *.java

Additional Notes:

This program does not utilize clusters nor GPU programming, so there is no need to package in a .jar nor compile any GPU code.

User

This is a user’s manual for running the program:

Running:

Once the software has been successfully compiled, and pj2 is on the classpath (see the Developer Manual for how to do this), the programs are run as follows:
$ java pj2 CleverSeq <N> <promptfile> <data files...>
$ java pj2 threads=<K> CleverSmp <N> <promptfile> <data files...>

N = the number of responses to test for each prompt
K = number of threads (only applies to parallel program)
promptfile = file containing one prompt user input per line
data files... = one or more files containing one training response per line
Output:

While running, the program will first read in all the data from the data files and generate its language. Then, it will print each prompt and determine the best response for that prompt. Each prompt is printed on a new line beginning with "> " . Each response will be printed below the prompt with no prefix. 
Example output:

> Hello.
Hello my shield my human.
> How are you?
Watches you teaches you.
> Do your responses make sense?
Make sense whatsoever your responses and glasses.
> Please help me.
Help me please.

Each line beginning with > was a line from the prompts file, and each unmodified line was the best response that was found within the N responses tested.
Scaling Results

This section gives shows the values recorded that are distilled into plots in the Scaling section.

Table 1: Strong scaling results

<table>
<thead>
<tr>
<th>N (# Sentences)</th>
<th>K (cores)</th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>N (# Sentences)</th>
<th>K (cores)</th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000 seq 1040</td>
<td>1024</td>
<td>1.0169026</td>
<td>1.0169026</td>
<td>1.00025358</td>
<td>50000 seq 2048</td>
<td>1024</td>
<td>2.0444</td>
<td>1.00025358</td>
<td>1.00025358</td>
</tr>
<tr>
<td>1</td>
<td>815</td>
<td>1.2760736</td>
<td>0.63903509</td>
<td>1.00025358</td>
<td>2</td>
<td>1024</td>
<td>2.0444</td>
<td>1.00025358</td>
<td>1.00025358</td>
</tr>
<tr>
<td>2</td>
<td>716</td>
<td>1.45251366</td>
<td>0.454713222</td>
<td>1.00025358</td>
<td>3</td>
<td>756</td>
<td>2.70803782</td>
<td>0.902865795</td>
<td>1.00025358</td>
</tr>
<tr>
<td>4</td>
<td>700</td>
<td>1.46714266</td>
<td>0.3714265714</td>
<td>1.00025358</td>
<td>4</td>
<td>6035</td>
<td>3.39720795</td>
<td>0.6476501866</td>
<td>1.00025358</td>
</tr>
<tr>
<td>5</td>
<td>662</td>
<td>1.570696979</td>
<td>0.31419591956</td>
<td>1.00025358</td>
<td>5</td>
<td>5021</td>
<td>4.07482872</td>
<td>0.6150965943</td>
<td>1.00025358</td>
</tr>
<tr>
<td>6</td>
<td>659</td>
<td>1.57814871</td>
<td>0.263024785</td>
<td>1.00025358</td>
<td>6</td>
<td>4286</td>
<td>4.775485616</td>
<td>0.795915966</td>
<td>1.00025358</td>
</tr>
<tr>
<td>7</td>
<td>637</td>
<td>1.632653061</td>
<td>0.2332361516</td>
<td>1.00025358</td>
<td>7</td>
<td>4110</td>
<td>4.975532117</td>
<td>0.711261751</td>
<td>1.00025358</td>
</tr>
<tr>
<td>8</td>
<td>662</td>
<td>1.570696979</td>
<td>0.1903746224</td>
<td>1.00025358</td>
<td>8</td>
<td>3684</td>
<td>5.05406261</td>
<td>0.6943200326</td>
<td>1.00025358</td>
</tr>
</tbody>
</table>

Table 1: Strong scaling results(continued)
Table 2: Weak scaling results

<table>
<thead>
<tr>
<th>N (ll Sentences)</th>
<th>K (cores)</th>
<th>Time (ms)</th>
<th>Sizeup</th>
<th>Efficiency</th>
<th>N (ll Sentences)</th>
<th>K (cores)</th>
<th>Time (ms)</th>
<th>Sizeup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>seq 1</td>
<td>531</td>
<td>0.9736346516</td>
<td>0.9736346516</td>
<td>50000</td>
<td>seq 2</td>
<td>2738</td>
<td>0.9978134111</td>
<td>0.9978134111</td>
</tr>
<tr>
<td>2000</td>
<td>2</td>
<td>563</td>
<td>1.8508959689</td>
<td>0.9182948494</td>
<td>100000</td>
<td>2</td>
<td>2737</td>
<td>2.0006757272</td>
<td>0.000656564</td>
</tr>
<tr>
<td>3000</td>
<td>3</td>
<td>566</td>
<td>2.7402826666</td>
<td>0.9134275618</td>
<td>150000</td>
<td>3</td>
<td>2736</td>
<td>2.9289337183</td>
<td>0.983112392</td>
</tr>
<tr>
<td>4000</td>
<td>4</td>
<td>594</td>
<td>3.4814814814</td>
<td>0.8703703704</td>
<td>200000</td>
<td>4</td>
<td>2831</td>
<td>3.868576699</td>
<td>0.967149172</td>
</tr>
<tr>
<td>5000</td>
<td>5</td>
<td>644</td>
<td>4.0139751056</td>
<td>0.8027950311</td>
<td>250000</td>
<td>5</td>
<td>2820</td>
<td>4.8846099999</td>
<td>0.9709219858</td>
</tr>
<tr>
<td>6000</td>
<td>6</td>
<td>661</td>
<td>5.0782307070</td>
<td>0.8461584615</td>
<td>300000</td>
<td>6</td>
<td>3065</td>
<td>5.398694943</td>
<td>0.933315824</td>
</tr>
<tr>
<td>7000</td>
<td>7</td>
<td>629</td>
<td>5.7857717070</td>
<td>0.8219396666</td>
<td>350000</td>
<td>7</td>
<td>3343</td>
<td>5.7317373796</td>
<td>0.819024828</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>620</td>
<td>6.6709677477</td>
<td>0.8338709767</td>
<td>400000</td>
<td>8</td>
<td>4048</td>
<td>5.410671744</td>
<td>0.6768833992</td>
</tr>
</tbody>
</table>

| 1000000          | seq 12    | 3915      | 0.9835098961 | 0.9853698961 | 5000000         | seq 22    | 2738     | 0.9978134111 | 0.9978134111 |
| 2000000          | 2         | 4154      | 1.8625475844 | 0.941273516  | 10000000        | 2         | 2737     | 2.0006757272 | 0.000656564  |
| 3000000          | 3         | 4100      | 2.3227228828 | 0.9442109839 | 15000000        | 3         | 2736     | 2.9289337183 | 0.983112392  |
| 4000000          | 4         | 4665      | 3.359639991  | 0.9390997776 | 20000000        | 4         | 2831     | 3.868576699 | 0.967149172  |
| 5000000          | 5         | 4517      | 4.242837663  | 0.848567507  | 25000000        | 5         | 3065     | 5.398694943 | 0.933315824  |
| 6000000          | 6         | 4777      | 4.921009677  | 0.9201582783 | 30000000        | 6         | 3343     | 5.7317373796 | 0.819024828  |
| 7000000          | 7         | 5213      | 5.261076374  | 0.7916230392 | 35000000        | 7         | 4048     | 5.410671744 | 0.6768833992 |
| 8000000          | 8         | 5738      | 5.4629861596 | 0.6828732748 | 40000000        | 8         | 4048     | 5.410671744 | 0.6768833992 |

Group Members Work Statement

We chose a chatbot because our interests in machine learning and natural language processing aligned. We attempted to have each member work on parts of the project that best suited our skills. For the literature search, we split that up evenly. We both compiled papers we liked and ultimately decided on three to focus in on. The program itself was originally mostly written by John. Some components were contributed by Chris, who also worked on the parameter tuning (heuristic weights) and response evaluation. Both group member edited the code until the definitive version was complete and all aspects of it were agreed on. Both members found reasonable files for training and worked together to pick the final documents that were used by the program. It was John’s idea to use Romeo and Juliet training data which turned out quite enjoyable for the in-class demo. The program was adjusted by both members to better fit the project’s focus. Decisions such as training on the spot, and using multiple input prompts was agreed on by both members in order to keep the focus on the parallel program. Chris originally wrote the final report, where parts were contributed by John and both members edited it to what it is now. John worked on weak scaling and Chris worked on strong scaling. All presentation were created by both members. One would set up the slides and pass it to the other, who would add content and pass it back and forth until it was completed. We worked together to speak half and half on the material in the presentations, where Chris started the presentation, and John finished.