A Dialog System to Supplement Student Academic Advising

by

Brian C. McMahan

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Bachelor of Science
in
Cognitive Science
Computer Science Concentration

Minnesota State University, Mankato
Mankato, Minnesota
May 2011
Abstract

Dialog systems are an implementation of natural language processing theory which allow use of natural sentences for communication with a computer system. The purpose of this project was to design and implement a dialog system to supplement university student advising. Student advising is a relatively narrow domain of possible questions and responses. The dialog system focused on prescriptive advising rather than developmental advising to further narrow the domain to scheduling and registration issues. To better understand the domain, a professional advisor was recorded during a mock advising session in order to model student-advisor interaction. The natural phrases from the mock advising session were transcribed, and then encoded using Artificial Intelligence Markup Language (AIML). The dialog system was implemented in the programming language Python and was designed as a multi-layered state manager to facilitate the conversation and AIML interaction. The recorded interview was also analyzed to develop the conversational sub-goals with each sub-goal mapping to a computable value. The purpose of the state manager was to control the flow of the conversation and ensure that the various sub-goals are satisfied. An expert system based on rules and graph theory was implemented to process the result of a satisfied conversation (an array of numerical data) and make advising decisions and a visual representation of a proposed schedule to return to the student user. Finally, the system was tested on several students for feedback on ease of use. Future work would include expanding the conversational ability and the domain of knowledge so that the system can be useful outside of the College of Science, Engineering, and Technology and outside of Minnesota State University, Mankato.

Keywords: academic advising, dialog system, decision system, graph theory
Table of Contents

Chapter 1 Introduction .................................................................................................................. 5

Chapter 2 Background .................................................................................................................. 7
  2.1 Dialog Systems .......................................................................................................................... 7
  2.2 Decision Systems ....................................................................................................................... 9
  2.3 Implementation Tools ............................................................................................................... 11
    2.3.1 State Managers ...................................................................................................................... 11
    2.3.2 Pyjamas ................................................................................................................................. 12
    2.3.3 Javascript Object Notation Remote Procedure Call (JSON-RPC) ........................................ 12
    2.3.4 CherryPy HTTP Framework ................................................................................................. 12
    2.3.5 MySQL Database ................................................................................................................ 13
  2.4 Theories and Algorithms ........................................................................................................... 13
    2.4.1 Graph Theory ....................................................................................................................... 14
    2.4.2 Parsing ................................................................................................................................. 16
    2.4.3 Usability Testing .................................................................................................................... 17

Chapter 3 Methodology and Implementation ................................................................................... 18
  3.1 Dialog System Implementation ................................................................................................. 18
    3.1.1 Modeling the conversation ................................................................................................. 18
    3.1.2 Conversation Management ................................................................................................. 19
    3.1.3 User Interface ....................................................................................................................... 21
    3.1.4 Overall Design .................................................................................................................... 21
  3.2 Constructing the decision problem ............................................................................................ 23
    3.2.1 Modeling the degree ............................................................................................................ 24
    3.2.2 Modeling the courses ........................................................................................................... 25
    3.2.3 Simplification Decisions ....................................................................................................... 26
  3.3 Decision System Implementation ............................................................................................... 26
    3.3.1 Modeling the decision .......................................................................................................... 26
    3.3.2 The Advising Decision Algorithm ....................................................................................... 27
  3.4 Evaluation Methodology .......................................................................................................... 31
3.4.1 Student Evaluation ........................................................................................................... 31
Chapter 4 Results ......................................................................................................................... 33
Chapter 5 Conclusions and Discussion ......................................................................................... 34
  5.1 Future Features and Potential ............................................................................................. 35
  5.2 Conclusions ......................................................................................................................... 37
Appendix A ................................................................................................................................. 40
Appendix B ................................................................................................................................. 42
Appendix C ................................................................................................................................. 43
Appendix D ................................................................................................................................. 44
Appendix E ................................................................................................................................. 45
Chapter 1 Introduction

Advising is one of the most important facets of a university’s service to its students. Through this service, a student is aided in both a prescriptive and developmental manner. The developmental portion of advising deals with the academic, professional, and personal development of the students while the prescriptive deals with such things as course selection, requirements for graduation, and academic planning. The prescriptive portion is inherently the simpler of the two components of advising because it is based upon rules, but typically takes away from the developmental advising in a one-on-one advising session due to students not understanding the rules.

To allow advising sessions to be more centered on the development of the student, a supplemental dialog system (which will be hereafter referred to as 'the system') was developed. This project is concerned with the aspects of student advising that are rule-based, similar to the work in Siegfried (2003). Further, prescriptive advising was approached through interactive dialog because it offers the student a user-friendly interface and allows for more natural interactions than the supplemental automated advising system found at California State University-Dominguez Hills (Distance Education Report, 2004). The system enables a student to have most of their classes decided upon prior to an advising session which means more time for developmental advising (Murray & Le Blanc, 1995).

The goal of the system is to make advising easier and more efficient for both the student and the advisor. The system is designed to do this by engaging the student in conversation to gather preferential data then compute optimal schedules given both the preferential data and academic data (classes taken, classes needed, and classes offered). The student would then be able to print off an optimal schedule and bring it to the advising session. Future modifications
would allow the advisor to be able to log in and access the prospective schedules for approval. This would allow for quick, paperless access, streamlining the advising process. Also, with an expedited prescriptive advising process, there would be more time for developmental advising as well as more time for additional students.

Chapter 2 introduces the motivations and background, addressing the many components that went into making the system. There are many components that went into making the dialog system to supplement student advising. The methodology will be described in chapter 3, which builds on the background provided in chapter 2. Finally, the results and conclusions are discussed in chapters 4 and 5.
Chapter 2 Background

A dialog system to supplement student advising can be seen as composed of two problem sets: dialog and decision systems. Both problems have been researched since the beginning of the field of computer science and as such there are many approaches to both problems. Within these two realms of research, many applications for supplementing student advising have also been developed. This chapter gives an overview of the two types of systems with an outline of the various motivating and methodological tools and concepts that form the theoretical and practical basis of this project.

2.1 Dialog Systems

There are many different approaches to the problem of dialog systems, and more generally, of human language. To understand the approach to dialog systems, the difficulties of a computer manipulating language must first be described. The use of language can be deceptively simple. The complexity becomes clear when one tries to define language for a computer to manipulate. For instance, the work "cut" can take on many different meanings in different contexts: "cut the grass", "cut my hair", "his part of the cut", "a paper cut", etc. Beyond just the complexity of a single word, there is the complexity of phrases and sentences: "I would like ice cream", "Please give me ice cream", "Ice cream is what I want", and "I desire ice cream" can be understood as meaning the same thing, but each is made up of different parts in differing sequences. The gravity of the last part, *different parts in differing sequences* is very important to understand because it makes searching for the meaning of a sentence, or generating its understanding, very difficult. A computer typically interacts with language in one of two ways: by defining the language formally and defining rules explicitly or by treating it as a statistical
model where words are chosen and understood based upon the number of times they occur in a large set of training examples. There have been many attempted solutions to language and dialog systems using these two approaches. The focus of this work is on rule-based systems because they are most appropriate for this work, which is limited in domain and training examples.

Dialog systems were first set upon their path in 1950 when Alan Turing published his paper "Computing Machinery and Intelligence" and defined the ultimate test of dialog systems: the Turing Test. This is a test in which a human, the interrogator, converses with two players over a remote terminal with the assumption that one player is a human while the other is a computer. The interrogator is then to determine which player is the human and which is the computer. (Turing, 1950).

The first attempted solution to this test was ELIZA, developed by Joseph Weizenbaum in the 1960s. ELIZA has several programs, the most famous of which was a DOCTOR script that mimicked a psychiatrist in asking questions about a patient (Paillard, 2009). The approach was shallow and only made use of matching phrases given by the user to system responses. There was little processing and no information gained from the user input or the answer given. This is because ELIZA is stateless, meaning each user input sentence is treated the same as the last and the next. There is no goal being progressed toward.

The first attempt to incorporate more information into a dialog system was as early as 1978 when Michael Maudlin created PET and introduced reasoning into the conversations (Paillard, 2009). The following style of conversation is what made PET famous:
Subject: I like my horse

Subject: I like my friend

PET: I have heard that your horse is your friend

However, PET was still limited in that it could not draw broad conclusions across the dialog, merely posit simple logic across similar sentences.

In the late 1990s Dr. Richard S. Wallace began creating the Artificial Intelligence Markup Language (AIML) (Wallace, n.d.; Wallace, 2004). AIML is Dr. Wallace's formalization of the pattern matching that decades of dialog systems had utilized. AIML allows for the storage of potential input sentences and their correct responses. It also allows for states to be defined and the past sentence to be considered. This is all done through the mark-up structure which uses identification tags within angle brackets (< >).

The dialog systems that have been described so far are strict pattern matching systems. AIML is one of the later evolutions of these, but it is pattern matching all the same. There are many other types of systems: some that require deep knowledge computation while others require a large corpus of data to generate accurate statistics necessary to accurately predict conversations. Because of the limited task domain, this work will use AIML.

2.2 Decision Systems

The second major difficulty in a dialog system to supplement student advising is making advising decisions. Decision systems have been studied since 1957 with the creation of the General-purpose Problem Solver (GPS). Herbert Simon, J.C. Shaw, and Allen Newell created the GPS computer program as a universal problem solver (Newell, Simon, & Shaw, 1959). GPS could theoretically solve any problem that could be represented with symbolic logic. It was able
to solve simple problems such as the Tower of Hanoi, but it failed to solve more complex problems in which every possibility would have to be explored. This is an inherent problem with very hard problems and solution strategies that enumerate all possibilities.

Since the creation of the GPS, modern day decision systems have been labeled "expert systems". Researchers realized that a system intended to solve problems must do so in a limited, or narrowed, domain. This means that an expert system needs to only deal with a limited amount of information and possibilities, thus escaping the downfalls the GPS had experienced.

Academic scheduling is a narrowed domain in which several attempts have been made to construct expert systems, see Siegfried (2003), Distance Education Report (2004), and Murray & Le Black (1995). Although it is a narrowed domain, it is still very complex because of the many permutations allowed for the typical undergraduate student. This complexity is also why an expert system for student academic advising is valuable: it will allow advisors to spend more time on developmental advising and addressing more students.

Students must be able to balance their necessary core requirements for their major while satisfying the general education requirements and choosing electives of interest to them. There is also a certain order in which the required classes should be taken due to pre-requisite requirements in upper-level classes. Poor academic planning can result in additional costly semesters in school. Further, as students enter the university, they must follow the academic bulletin of that year, which lists the requirements for general education and for their major. Students can choose to switch their bulletin in subsequent years if the requirements change and they wish to graduate under the new requirements. With students from at least four different entering years, an advisor can expect to work out of multiple bulletins.
These factors play into a complex system that gives rise to the difficulties a student faces in selecting courses. Even more so now, with budget cuts affecting the number of times a class can be offered, students must plan out their schedules far in advance. If they do not plan to take classes ahead of time, their graduation date could potentially be pushed back due to core major requirements that come in sequences. This makes the motivation for any automated advising system dual-focused: to supplement student advising, allowing for more developmental advising, and to make it easier for students to efficiently schedule their required classes over their academic career.

2.3 Implementation Tools

There are many categories of tools and methods that go into designing and implementing dialog and decision systems. Some of these tools and methods are reviewed in the following subsections and serve as the basis for the methodology undertaken for this project.

2.3.1 State Managers

The practice of dividing a problem into separate states has been common since Alan Turing's "Turing Machine" was first described in 1936 (Hodges, 2000). The concepts behind using states can be understood as formalizing a problem so that only one variable will change at a time. Each state is a description of the "state of affairs," or the current value of the variables. Each new input could either deterministically change the state or keep it in the same state until a final state, or goal state, is reached.

State managers are an implementation of Turing's original idea: break up the problem into smaller pieces that can be handled one at a time. Where Turing Machines were much simpler, state managers can be layered and handle more complex input. This dividing of the
problem into smaller pieces that can be handled individually or perhaps handled by another state manager can be a useful solution model.

2.3.2 Pyjamas

Pyjamas is an open-source project that began in 2007. It is a Python port of the Google Web Toolkit (GWTK) which is written in Java. This means it is translated from the Java programming language to the Python programming language. The Google Web Toolkit, and subsequently Pyjamas, allows for users to program in non-web programming language (Java in GWTK's case and Python in Pyjamas's case) and have it be converted to a Javascript program that will run in any browser. While non-visual components and purely functional components can be made with it, it is typically used as a user interface for web applications (Pyjamas, 2010).

2.3.3 Javascript Object Notation Remote Procedure Call (JSON-RPC)

For online web applications that are dominated by Javascript, there are many different ways for communicating between services. One of the most popular and fastest ways is the Javascript Object Notation Remote Procedure Call (JSON-RPC). This protocol communicates over HTTP to a service to obtain information that cannot be calculated or found on the web application. Many applications allow for this type of communication, such as Twitter, Facebook, and YouTube (JSON-RPC, 2009).

2.3.4 CherryPy HTTP Framework

There are many different HTTP frameworks freely available, such as Apache (The Apache Software Foundation, 2011), IIS (Microsoft, 2011), nginx (Sysoev, 2011), and lighttpd (Kneschke, 2011). However, most of these frameworks are stateless. This means that every
request sent to the server for web information (whether it be a service or a page request) is treated exactly as the last; there is no memory for past requests.

Stateless HTTP servers are common because HTTP was originally designed as a stateless model to serve visual and audio content to whomever is requesting it. However, as the internet evolved, circumstances arose that required HTTP servers to have session memory. This can be done manually by establishing cookies, rewriting the URL, or hidden form fields, but there are also several object-oriented frameworks specifically built for keeping session information. One of the most lightweight and easy to implement object-oriented HTTP frameworks is CherryPy built in the Python programming language. CherryPy automates session memory by keeping session IDs and storing cookies on a user's computer. CherryPy can also respond to service calls from JavaScript applications by means of JSON-RPC. This is extremely powerful because it allows for backend computations that will not bog down a user's JavaScript interface.

2.3.5 MySQL Database

MySQL is a relational database management system that allows for the storage of data in tables. Data is extracted from the tables using a structured query language. The benefit of using a MySQL database is that it allows large amounts of information to be stored and retrieved in negligible time (Oracle Corporation, 2010).

2.4 Theories and Algorithms

Within the field of computer science, a common focus is on solving problems. From this common focus, many sub-fields were created and others adopted from varying disciplines such as mathematics. Notably, the topics of graph theory and parsing were used in this work. Usability testing theories were also used, which will be covered in section 2.4.3.
2.4.1 Graph Theory

Graph theory is the study of mathematical structures called graphs that consist of vertices and edges. Each vertex typically represents a concept while each edge can represent a semantic relation between the vertices. This is a useful way of representing different problems because of the properties that are associated with graphs that can be exploited. Graph theory also serves as a basis for many computer science algorithms by first representing a problem as a graph then using different features to search for an answer.

Some of the common features and terms in graph theory are parents, children, trees, roots, branches, and leaves. In a directed graph, which is a graph in which all the edges can only be followed from one vertex to another, the originating vertex is considered the parent while the destination vertex is the child. In some graphs, each vertex may have multiple incoming edges and multiple outgoing edges. However, there is a form of directed graphs called trees that can be represented in a hierarchical manner because each vertex only has one parent. Each child is hierarchically a level lower than its parent. Then, the vertex on the top level, which by definition can only be one, is considered the root because all other vertices are descendents of it. Each child of a vertex is then a branch in the tree. This also leads to the idea that the further a vertex is away from the root, the deeper it is in the tree. If a vertex has no children then it is considered a leaf vertex. Roots, parents, children, branches, and leaves are important concepts for search approaches (Tucker, 2007).
2.4.1.1 Search Approaches

One method for searching a directed graph is called depth-first search (DFS). The procedure is to start with a vertex in a graph, whether it be arbitrary or chosen for a reason. This vertex is then considered the root vertex. The purpose of the depth first search is to traverse vertices until either all vertices are traversed or the desired vertex is traversed. The resulting traversal is a tree. To begin, an edge is followed away from the root vertex to a child vertex that has not been traversed yet. If there is more than one child, then a tie-breaking scheme is established such as comparing the values of the children. The traversal then keeps following edges to children until a vertex is reached that either has no outgoing edges or all edges lead to vertices that have already been traversed. This end-of-the-line condition can be designated as a depth-terminating condition. While other depth-terminating conditions can be specified, the end-of-the-line condition is typical. This vertex is a leaf in the resulting tree. The path to this vertex is then backtracked until it reaches a vertex that has an edge to a vertex that has not been traversed already. The algorithm will then descend in the same depth-first style until the same terminating condition (in the general case, the end-of-the-line condition) is fulfilled. The search is considered done when the backtracking leads back to the root vertex and there are no more available edges to descend (Tucker, 2007).

The alternative method to DFS is breadth-first search. It proceeds in much the same way as DFS but it will instead traverse all children, the breadth, of a vertex before descending further. There are advantages to either method, but a depth-first method will typically find leaf vertices faster than the breadth-first search. This is useful in certain circumstances, such as trying to find a possible travel route with each node representing cities, while disadvantageous in others, such as trying to find the shortest possible travel route in the same condition. (Tucker, 2007).
2.4.1.2 Trees, Forests, and Pruning

In a directed graph in which there are many outgoing links between vertices and no clear distinction as to which vertex should be the root node, the possibilities are enumerated such that all viable vertices are chosen as the root vertex. This creates many different trees that can either be seen as branches of a much larger tree with a null vertex as the root or as a forest of trees. For simplicity, the result is described as a forest rather than a larger tree.

There are circumstances in which the resulting forest is too large and needs to be reduced to a forest of a smaller size. This reduction is called pruning. The process is simple: determine which branches to cut such that vertices will lose children and the number of leaf vertices will be reduced. This is an important optimization principle if the forest represents choices and results can be optimized by taking out unnecessary options.

2.4.2 Parsing

In many cases there is a need to parse large amounts of data. This problem has been thoroughly studied for compilers since the early 1960s when Donald Knuth published his paper "On the Translation of Languages from Left to Right" and described the initial processes of an LR Parser (reads left to right and provides rightmost derivation) (Knuth, 1965). An LR parser tokenizes the input and then looks up an associated action for each token in the input. The actions typically include shift, reduce, and accept. Shift means to move the LR Parser to a different state, which changes the look-up values for each token. This can be visualized as each row being a different state and each column being the separate tokens. The action lookup process is then finding the corresponding cell of the table. The reduce action will apply a grammar rule to the input. This can be abstracted to just applying a procedure to the input such
that the input is reduced and the parsing is closer to being done. The accept is then the final action which terminates the parsing for that input.

Typically, an LR parser is used in making a compiler for a programming language, but it can also be designed to parse documents. Each token is simply an expected word in the document and the reductions are the actions provided to remove the information from the document.

2.4.3 Usability Testing

Evaluating software and user interfaces has well-established methods and principles. Some of the usability principles, such as an iterative design process, can be seen in the United States Department of Health & Human Services' on usability testing (2011). An iterative design process has a distinct development cycle. First, a prototype is developed and tested. The information from the testing is then used to develop another prototype. The process is repeated until an acceptable application is developed. The usability testing documents provided by the U.S. Department of Health and Human Services are designed for websites, but the principles apply to software usability testing as well.
Chapter 3 Methodology and Implementation

A dialog system to supplement student academic advising would need to balance both what the student wants and what the student needs. This was the motivating principle behind the methodology and implementation. First, the dialog system needed to gather student preference data through a dialog with the student, which is described in section 3.1. Next, the system needed to decide upon and offer possible optimal schedules using the student's academic record and preferences. Sections 3.2 and 3.3 describe these optimal schedule decisions.

3.1 Dialog System Implementation

"How can a computer converse with a human?" was the first question that was asked in the creation of this dialog system. To begin answering this question the Artificial Intelligence Markup Language (AIML) was chosen. AIML allows for possible sentences and their responses to be coded into structured data files. However, AIML only results in structured data files and needs a corresponding interpreter to be fully functional. For this, an implementation in the Python programming language (Python Software Foundation, 1990) was used because it was the only open source implementation for an AIML interpreter freely available (Stratton, 2003).

3.1.1 Modeling the conversation

The first step in designing the dialog system was to understand what happens during a typical academic advising session. To understand the advising interaction, there were conversations with three college-level academic advisors in which a conceptual model of the advising session was derived. This model was used to design a mock interview.
The mock interview session consisted of several different scenarios with each scenario representing a possible student seeking advising. The various scenarios and the mock interview script are in Appendix A. Level 1 human subject approval was obtained through the Institutional Review Board (IRB) to conduct this mock interview with an academic advisor. The interview was conducted, recorded, and transcribed not only so that the sentences could be incorporated into the dialog system but also to model the flow of the conversation.

3.1.2 Conversation Management

Using a model of the advisor-student conversation, the conversation management system was designed and implemented. A multi-layered model was chosen because it allowed for better conversation control. A visual representation of the model is shown in Figure 1. The bottom, or most basic, layer was the Artificial Intelligence Markup Language (AIML) that allowed for the representation of sentences and their responses. Figure 1.b is a visual representation of the transition between AIML states. Each box represents a state within the AIML in which it is waiting for a user generated sentence. AIML would provide the transition sentence into the next state. AIML also allowed for topics to be defined that broke the sentence-response pairs into larger chunks. This key concept was utilized by a state manager that could switch between AIML topics that it locally represented as states.
The top, or the most complex, layer was the state manager which acted as a middle man between the user and the AIML is represented in Figure 1.a. The state manager was an encapsulated set of states that directed the flow of the conversation, linearly or non-linearly, through the various conversation states. Additionally, each topic within the AIML can be viewed as a collection of states as well. This meant that within each top-layer state the AIML progressed through bottom-layer states towards a goal response. The goal response would have
a token at the end of it that the state manager would recognize. Each box titled "Scheduling Preference Sub-goal" represents a grouping of AIML sentences and responses. The state manager is designed to move onto the next state when the AIML unit has reached termination. The goal token also had the capability of imbedding data to send to the state manager. This allowed for a discrete range of values to be gained from a conversational state. This was the basis for the student preference data gathering for which the dialog system was designed.

Upon recognition of a goal token and completion of a conversation state, the state manager would reference itself for the next state. This process was designed to allow for any returned data from AIML to influence both the state that would follow the current state and future ordering of states.

3.1.3 User Interface

A dialog system designed to interface naturally with a user needs to have a convenient interface. For this purpose, a user interface was designed as a web application and the Pyjamas tool was chosen for implementation. JavaScript created by Pyjamas runs natively in every browser (given that the user has JavaScript enabled). Further, Pyjamas has a library of various JavaScript tools and visual widgets that gave great flexibility to the interface.

3.1.4 Overall Design

With the implementation choice of Pyjamas, the conversation management system was forced to run outside of the user interface because the AIML interpreter was not supported by Pyjamas and has no JavaScript equivalents. A back end server was then needed to run the conversation management system. However, the server could not be stateless (it had to be able to keep track of the progress through the conversation and not re-instantiate all the variables with
every server call). If there server were stateless, it would treat every input sentence sent from the user interface exactly the same because the state manager and AIML would not be kept in memory between messages.

For this desired state-ful characteristic, CherryPy was chosen. CherryPy allowed for sessions by means of cookies and session IDs. To communicate between the user interface and the CherryPy server, JavaScript Object Notation Remote Procedure Call (JSON-RPC) was used. JSON-RPC is a communication protocol that passes messages over HTTP between two entities. Pyjamas had native support for JSON-RPC and to implement on CherryPy required little set up time. A visual representation of the user interface and the CherryPy server is in Figure 2. The arrows in this figure represent the JSON-RPC communicating between the two.
3.2 Constructing the decision problem

Once the dialog system has gained preference data from the user and has gained courses taken and chosen major information from the academic record provided by the user, possible schedules are formulated and offered to the user. The approach to formulating possible schedules was to formally model the information.

Section 3.2.1 describes how the degree was modeled. A robust notation was developed to model the over 100 degrees at Minnesota State University, Mankato. Section 3.2.2 describes how the courses were modeled, which involved an inherent measure of benefit. Finally, section
3.2.3 outlines the simplifying decisions that were made in the course modeling to eliminate complexities that compounded the difficulty of the problem.

Once the degrees and courses were modeled, an LR-parser was constructed to store the information in useful forms in a MySQL database. The database was chosen so information could be retrieved at any point in computation. The LR-parser was used because it gave a fast and easy approach to gathering a variety of information. It was in the parsing of all possible courses that many of the simplification decisions that will be described in 3.2.3 took place.

3.2.1 Modeling the degree

Degrees at Minnesota State University, Mankato follow the same pattern: core classes, electives, and possible degree emphases. Using these general categories as the basis for the major, a notation was derived and named Course Language Assisting SQL-like Searches (CLASS). An example can be seen in Figure 3.

```
NAME PSYCHOLOGY
ADMISSION
C-RULES ALL PSYC 201
CREDITS 32

CATEGORY REQUIRED GENERAL EDUCATION
C-RULES ALL PSYC 101

CATEGORY REQUIREMENTS
C-RULES ALL PSYC 201 PSYC 211 PSYC 409

CATEGORY LEARNING AND COGNITION
C-RULES SELECT 1-COURSE PSYC 413 PSYC 414 PSYC 415 PSYC 416

CATEGORY PERSONALITY/SOCIAL
C-RULES SELECT 1-COURSE PSYC 340 PSYC 435 PSYC 456 PSYC 458

CATEGORY BIOLOGICAL
C-RULES SELECT 1-COURSE PSYC 420 PSYC 421 PSYC 423

CATEGORY DEVELOPMENTAL
C-RULES SELECT 1-COURSE PSYC 433 PSYC 436 PSYC 466

CATEGORY ELECTIVES
C-RULES SELECT 40-CREDITS RANGE PSYC 100-499 EXCEPT PSYC 101
```

Figure 3 - CLASS representation of the B.S. degree in Psychology
Each category had a variable number of constraints. Each constraint can be one of three options: ALL, SELECT, and SELECTSEQUENCE. ALL represents the core classes and is followed by a discrete list of courses that are required. SELECT and SELECTSEQUENCE represent two different types of electives. SELECT was followed by a token describing what kind of elective it was: a minimum number of credits or a minimum number of courses. It would then be followed by a discrete list of courses that are possible to take. SELECTSEQUENCE will be described further in section 3.2.3.

There are circumstances in which a set of classes are limited in or even restricted from fulfilling an elective constraint. There are also circumstances in which a range of courses are specified. Each of these circumstances also have their own command words: LIMIT, EXCEPT, and RANGE respectively.

### 3.2.2 Modeling the courses

Courses have an inherent benefit in relation to pursuing a degree. A course can either satisfy a core or an elective constraint or it could not satisfy any constraint. Furthermore, if a course is a prerequisite for a course that does satisfy a constraint, then the prerequisite has an inherent benefit towards completing the degree.

I modeled this inherent benefit by placing a higher numeric value on the courses satisfying a core constraint than those satisfying an elective constraint. Courses that were prerequisites for courses designated as dependents had additional benefit calculated by compounding a portion of the dependent's benefit added to it. This is sometimes a recursive calculation because a dependent could possibly be a prerequisite for further courses.
3.2.3 Simplification Decisions

There were several simplification decisions made during this project. First, general education requirements were not included in the modeling and decision making process. However, general education requirements are conceptually formulated in the same manner as degrees are and are thus easily describable by the CLASS notation. The decision algorithm would need to be changed slightly to allow for two collections of categories and to correctly model the benefit between them.

Also, while parsing all of the courses that could be offered at Minnesota State University, Mankato there were many difficulties. The information for the courses was stored in portable document formats (PDFs). For each degree there was a PDF and inside the PDF was the course information. Many different degrees described prerequisites in a variety of ways. While there were many consistencies, the variability proved a challenge that is a project unto itself. Any prerequisite not described in the standard form was ignored.

3.3 Decision System Implementation

From the modeled problem a solution was constructed. Section 3.3.1 outlines the form of the solution and section 3.3.2 describes the optimal schedules algorithm.

3.3.1 Modeling the decision

To begin the decision making, users are prompted for their academic record and it is parsed to gain the user's academic data about classes taken and declared degree. This step occurs before the dialog begins so that the dialog system can incorporate conversation states that are based upon the information within the academic record.
From the CLASS description of the user's declared degree, an encapsulated degree object is created that represents the categories and constraints of which the degree is composed. From this object, a list of all courses that could fulfill the degree is formed. This object, located within the user's browser, sends the list over JSON-RPC to the server. The server then compiles all of the information for the courses, such as title and prerequisites. It also compiles information of all about prerequisite courses. The server returns this compiled course information to the user's browser, which updates the degree object with the information.

The taken courses from the user's academic record are removed from the degree object and the constraints are updated. This updates the degree object to match the current progress of the student. This updated list of courses is sent over JSON-RPC to the server and is compared to the list of courses offered for the following semester to discover the courses both needed by the student and offered for the following semester. This final list of courses is the model from which possible optimal scheduling decisions can be made.

3.3.2 The Advising Decision Algorithm

Each course in the final set has a numerical benefit (calculated by its relation to the degree and the benefit of the courses for which it is a prerequisite) as well as one or more scheduled times. This set of courses is translated to a graph. Each node in the graph represents a course section. Each node also has a numerical value attached to it, the benefit of the course in relation to the student's degree, as well as other descriptive properties.

To complete the graph, edges must connect the various nodes. There are three types of edges, but all of them denote a conflict between nodes. The first type connects sections of the same course. The second type connects course sections that have overlapping scheduling time.
This is done by connecting each node to time nodes representing one hour blocks of one day. Once every course section node has been projected onto all of the time nodes that represents its scheduling time, each time node is iterated over and all nodes connecting to it are connected to each other. This ensures that all time conflicts are included in the model.

Figure 4 - An example of two course section nodes

The last edge indicates that two courses can independently fulfill a requirement of the degree, but it is a little trickier and is not represented as a structural link. Rather, each course section node has a connection to its parent constraint. If a node does not have a parent constraint, then it must be a prerequisite and it is still affected by the dependent courses' parent constraint. The parent constraint handles the fulfillment status and can return a list of courses that are no longer needed when it is fulfilled.
A portion of the formed graph can be seen in Figure 4. The two course section nodes are linked in the snapshot because they are two different sections of the same course. The properties that they have are scheduling times and benefit. These properties are used in the modification and search of the graph.

To prime the graph for the algorithm, it is reduced by deleting the nodes that conflict with the user preferences. The simplest of user preferences are those of morning and night courses. This reduction avoids unnecessary computation of course sections that the user does not prefer. However, this opens up the possibility of a graph being reduced too far and no possible schedules being available. This issue is addressed in Chapter 5.

A recursive search of the graph is performed to find optimal schedules. This search is a modified depth first search and the algorithm can be seen in Figure 5. The algorithm is recursive, so the first check it performs is whether the tree that is passed into the function is a possible schedule. Schedules are represented as trees. If the total number of credits in the tree is not larger than the minimum number of credits needed for a schedule, then it proceeds with the rest of the algorithm.
For each course section node in the graph, starting with the one with the highest benefit, a possible schedule is attempted. This is done by the "use" function of the graph and tree objects. For the graph object, the "use" function will return a new graph without the course section node and all of the nodes to which it is incident (with which it shares edges). The tree object's "use" function returns a new tree that includes the course section node in it. This new graph and new tree are passed to another call of the recursive search function.

To better explain the recursive search function, the first iteration will be described. First, the node with the highest benefit is chosen. If there is more than one, one is chosen arbitrarily as the rest will be chosen eventually. This node is the root of a tree. All nodes that have an edge incident to the one chosen are then removed from the graph. The process is repeated with choosing a node with the highest possible benefit. This next node is the child of the first one chosen.

The algorithm repeats until a depth-terminating condition is met. This is the comparison at the beginning of the algorithm in Figure 5. Once the depth-terminating condition is met, the

```java
function rsg(graph, tree)
  if tree.credits ≥ minCreditsNeeded
    addToForest(tree)
  else
    for each course in graph.orderedList
      newGraph = graph.use(course)
      newTree = tree.use(course)
      rsg(newGraph, newTree)
```

**Figure 5 - Recursive Search Graph Algorithm**
algorithm will try a different course at the last point it chose a course to use. This process can be thought of as walking down a path with many forks. At every fork, the best option is chosen. However, the path can only be walked a certain distance (the depth-terminating condition). Once that condition is met, backtracking occurs to the last fork and another option is tried. This is continued until all the options at that fork have been tried. Backtracking occurs again to another fork earlier in the path taken. The algorithm behaves like this and continues backtracking to choose a different course at every possible point with multiple courses from which to choose. The result is many different trees, each with a total benefit. The trees with highest total benefit are presented to the user as possible scheduling options, each optimized for benefit to the academic degree.

3.4 Evaluation Methodology

A methodology was developed to evaluate the dialog system for its usability and accuracy of advising assistance. This methodology was approved by the Institutional Review Board and conducted with 5 students. It was modeled after the iterative design process described in (U.S. Department of Health & Human Services, 2011) with the dialog system taken as a prototype. Section 3.4.1 describes the methodology.

3.4.1 Student Evaluation

To start the evaluation a student was given a consent form to read and sign. Through this, a student is made aware that they can stop the evaluation at any time. After signing the consent form, the usability testing session began. The script used can be seen in Appendix B. During this testing session the student used fabricated academic records which can be found in
Appendix C. The student was observed as they used the system and observations were recorded. After the dialog system had offered possible schedules, the student was prompted to fill out a survey to rate the experience and provide qualitative feedback.
Chapter 4 Results

Five students participated in the evaluation of this project. Most users were able to use the system without much trouble. An iterative development process was used between user testing sessions. The first student tester did not experience the exact same system as the last student tester. Instead, results from each testing session were used to improve the system.

After each testing session a survey was administered containing both Likert-scale questions and open ended questions. The results for the Likert-scale questions are in Table 1. These results show that users found the system easy to use, got possible optimal schedules, and understood how to use the system. The results for the open ended questions are summarized in Table 2. The survey can be seen in Appendix D.

<table>
<thead>
<tr>
<th>Likert-Scale Questions (1=Agree, 5=Disagree)</th>
<th>Averages</th>
<th>Standard Deviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>It was easy to talk with the system</td>
<td>1.4</td>
<td>0.55</td>
</tr>
<tr>
<td>It gave me a possible optimal schedule</td>
<td>1.6</td>
<td>0.89</td>
</tr>
<tr>
<td>I had a hard time answering questions</td>
<td>4.8</td>
<td>0.45</td>
</tr>
<tr>
<td>I didn't understand how to use it</td>
<td>4.6</td>
<td>0.89</td>
</tr>
</tbody>
</table>

Table 2 - Usability Testing Open Ended Survey Results

<table>
<thead>
<tr>
<th>Open Ended Questions</th>
<th>Most common answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Were there any problems with the system?</td>
<td>&quot;yep&quot; broke the system, unresponsive wait time</td>
</tr>
<tr>
<td>What could improve your experience</td>
<td>&quot;shaded&quot; input, auto focus on text entry, &quot;wait&quot; messages, scheduling speed</td>
</tr>
<tr>
<td>What kinds of things do you take into consideration when scheduling classes that the system did not ask you about?</td>
<td>0 classes on one day, balance between major/minor and general education, work</td>
</tr>
</tbody>
</table>
Chapter 5 Conclusions and Discussion

With a project that consists of many parts, it is easier to assess each part individually and then as part of the whole. In this dialog system to supplement academic advising, the dialog, modeling, and decision systems are assessed.

The implementation choice of AIML came with many advantages and drawbacks for the dialog system. The major advantage was that it allowed for quick development. However, this quick development did not capture the breadth that natural language tends to have. During the usability testing, for example, a user responded with a response that was not anticipated and it caused the system to move into an unknown state from which it could not recover. Future work on this portion of the project will entail allowing for more variability in the conversation. It will also include allowing for more preference data to be gathered.

There were two main simplification decisions made during the modeling of the decision problem: not including general education requirements and ignoring hard-to-parse course information. These simplification decisions were made to allow for a simpler prototype and proof of concept to be developed. However, for this project to be a realistic supplement to academic advising it must be able to model the entire problem accurately. As was described by a user in the usability survey, a balance between general education and degree-fulfilling courses is considered optimal by some students. Additional development of the model will take the form of allowing for general education requirements and computational representation of harder-to-represent course information (e.g. "Prerequisites: 8 credits of PSYC").

The decision algorithm worked exactly as it was designed to work: incorporating the user preferences, it pruned a graph modeled from the courses a student could take and then found an
optimal schedule based on the benefit of the classes. The downfall of the algorithm is that it is computationally complex. It considers all possible combinations of courses in the graph that satisfy the credit constraints. The algorithm needs to be redesigned and optimized to run in a more efficient manner. The next step of this project will include the optimization of this algorithm to decrease running time and address user concerns about the wait time.

Future work on this project will cover all three major areas: the dialog system, the model of the problem, and the decision system. Improvement to each of these three areas is needed to propagate the current prototype into a usable program for students. Once these three areas have been improved, another round of usability testing can be conducted. The eventual goal is to create a fully-functional system that can become a tool to supplement advising.

5.1 Future Features and Potential

There are several scenarios that will come up in future development that must be addressed. These include double majors, no-benefit course choices, too-restrictive preferences, interactive optimal course rearrangement, and optimal schedules based on additional factors.

Double majors often take courses that fulfill a requirement for one major but only have a conceptual benefit towards the other. This implication is hard to define mathematically because the benefit to the second major is purely in the content and not in the rule-based relationship of courses and degrees. However, using machine learning algorithms and analyzing course selection by others of the same declared majors could reveal courses that are preferred but not required.

Courses that have no benefit toward the second major can be further generalized by viewing them as a course that has no-benefit towards that second major. This could then apply
to students with one major. There are many times in which students are encouraged to take courses outside of their major to gain broader perspective. This can be solved in the same manner mentioned earlier by using machine learning algorithms. Also, surveys could be done to gather student preferences on a much larger scale. This data could then be used to improve the system further.

There is a possibility that a student will have a choice of preferences that will result in too many course section nodes being pruned and no possible schedules. In this scenario a student will have to forgo a preference in order to get a possible course schedule. The issue then becomes computing which preference to change first. In this case, the dialog system could be reactivated. The dialog system would explain the situation and ask which preference should be overturned. The dynamic state manager was designed to allow for this.

A possible feature of the system could be an interactive schedule builder. The first step to this would be an analysis of all the possible optimal schedules and an identification of the courses that are integral to the largest number of schedules. From these primary course identifications, secondary courses can be identified. These secondary courses could be interchangeable and an interactive schedule builder based on the inclusion of primary courses and possible inclusion of secondary courses could be built.

Finally, many students base their schedules on many different factors. From the usability testing survey knowledge of some factors was gained, such as work schedule and the possibility of no-course days. However, there are also other factors such as professor ratings that could be included in the computation of an optimal schedule. These various factors can all be seen as modifying the benefit of the courses. Professor ratings can be gained from either user surveys or
websites that offer this information publicly and then used to multiply the benefit of courses they teach by factors larger than one if the ratings are high or lower than one if the ratings are low. A similar modification can be used for other factors.

5.2 Conclusions

This dialog system intended to supplement student academic advising has strong potential to benefit students and their advisors. This working prototype proves that the intersection of what a student wants and what a student needs for an academic schedule can be found computationally. The possible future features and potential of the system have been described as well as the algorithms that define the system functions. One day this system could truly supplement student academic advising, thus lifting the load of academic advisors and allowing them to address more students through developmental advising.
Works Cited


Appendix A

Student Advising Interview

I plan to ask participants the following questions. Anything to be said directly to the interviewee is shown in italics. Possible sub-prompts for the core questions are in the bulleted list following the question. While we have multiple scenarios, no more than three will be asked of each advisor.

Interviewer Script:

“Thank you for taking the time to talk with me today. I will be using this information to model interaction between advisors and students which will aid in the creation of an automatic dialog system intended for student advising. In this interview, I will establish different student roles and I would like you to best address the questions and issues with responses as natural as possible. You may stop this interview at any time, or choose not to answer one or more of the questions. All of your responses will be kept confidential. Is it OK with you if we tape this interview? The recorded responses will be transcribed and used in the automatic dialog system. Do you have any questions before we get started?” [Wait a moment for any questions.]

“We will cover up to three scenarios. After describing the scenario, I will give you the opportunity to ask me questions or initiate the session as you normally would. I will respond according to the character and will also ask questions. If you prefer, I can start with an opening question after I describe the character. Do you have a preference?” [Let the interviewee decide on the approach.]

1. In this scenario, I will be a student who is a freshman trying to schedule classes for Spring semester. The major isn’t specific, but if you feel the need to speak to a specific major please do so.
   a. I’m not sure what classes to take. I took a couple general education classes last semester. Should I just continue to take general education classes?
      i. Well, I was thinking about taking two lab classes. Is that a good idea?
      ii. Should I get started on my prerequisites?
      iii. I was thinking about getting a part-time or full-time job so I would have more money. Is this a good idea?

2. In this scenario, I will be a student who is a sophomore that had a pretty bad freshman year. My GPA is low and I need a more balanced semester, however I want to take a class load that is obviously too hard with a writing intensive course, an upper level math course, and a physic lab course.
   a. Is this schedule going to be good for me?
      i. What kind of classes should I take instead?
      ii. Should I be worried about applying to my major soon?
1. What steps should I be taking to make sure I get in?
2. Why is it so important?
3. In this scenario, I will be a student who needs to be encouraged to try more things, including in math and science. I have abilities but low self-confidence. In my first semester, I took a balanced load and surprised myself by getting a 3.8. I’m afraid of taking harder courses.
   a. I’m not sure what classes to take. I did pretty well last year, but I think it might have just been because they were easy classes
   i. Can’t I just keep taking easy gen ed classes and do my major classes later?
4. In this scenario, I will be a pre-engineering major who does not have the pre-requisites to take calculus I yet but really wants to be in classes with my friends.
   a. I really want to take this calculus. Is there any way I can take it?
   i. Are you sure there is no way we can waive the prerequisites?
      1. What if I take the prerequisite at the same time as calculus?
   ii. What should I take instead?
5. In this scenario, I will be a student who did really well on the ACT but have never gotten good grades. I don’t know what I want to major in but I really like video games.
   a. I’m not sure really what I want to go into. What do you think I should take? I don’t really like to work hard, though. Everything is kind of boring.
   b. I really don’t want any morning or Friday classes, is there a way I can avoid them?
Appendix B

Usability Testing Script

FACILITATOR: Hello, my name is Brian McMahan and I will be walking you through this usability testing session. I have asked you to come in today to participate in testing the usability of a dialog system implemented to supplement student advising. I want you to understand that the purpose is to test the system, not you. If you are unable to proceed with the course of the system, please indicate so and do not feel as if you are not contributing. There is more gained from finding the faults in the system as it will allow a better direction of development. I will be observing how you interact with the system. I will be writing things down, but our information and responses will be kept confidential. All of the data gathered from this session will be labeled with a number. The only place in which your name will be associated with the corresponding number will be a key code locked in Dr. Bates' office (Wissink Hall 231). The consent forms will be located in the same place. You can end this session at any time and there will be no penalty associated with this. I would like to reiterate this point again: you have the choice to end this session at any time and it will not influence your relationship with Minnesota State University, Mankato or any entities associated with it at all.

I will now guide you through the beginning of the system and then observe the rest of the interaction.

1. Start using the system by putting in a name into the box.

2. Please copy the Academic Record given in the open text editor and paste it into the corresponding box designated "Academic Record".

3. Please chat with the system. You can choose to answer truthfully. The questions it will ask are not recorded or stored.

Now that you are done with the system I will ask you to do one more thing. Please fill out this short paper questionnaire that reflects your experience with the dialog system.

Thank you for coming in today and helping with my project. I appreciate your contributions.
## Appendix C

### Sample Fabricated Academic Record

#### Sample 1

**Fall 2009**

<table>
<thead>
<tr>
<th>Course</th>
<th>Subject</th>
<th>Grade</th>
<th>Credits</th>
<th>GPA Pts</th>
<th>Term Att</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLTH 101</td>
<td>Health &amp; Environment</td>
<td>B</td>
<td>3.00</td>
<td>3.00</td>
<td>9.00</td>
</tr>
<tr>
<td>SPEE 102</td>
<td>Public Speaking</td>
<td>A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>ENG 101</td>
<td>Composition</td>
<td>A</td>
<td>4.00</td>
<td>4.00</td>
<td>16.00</td>
</tr>
<tr>
<td>PHYS 101</td>
<td>Introductory Physics</td>
<td>A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>PHIL 110</td>
<td>Logic &amp; Critical Thinking</td>
<td>3.00 A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
</tbody>
</table>

**UNDG** Term Att: 16.00 Earn: 16.00 GPA Crs: 16.00 GPA Pts: 61 GPA: 3.81

### Spring 2010

**MAJOR: PSYCHOLOGY**

<table>
<thead>
<tr>
<th>Course</th>
<th>Subject</th>
<th>Grade</th>
<th>Credits</th>
<th>GPA Pts</th>
<th>Term Att</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHIL 101W</td>
<td>Mind-Body Problems</td>
<td>A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>MUS 125</td>
<td>Pop Music USA: Jazz Blues</td>
<td>3.00 A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>ENG 114</td>
<td>Introduction to Film</td>
<td>B</td>
<td>4.00</td>
<td>4.00</td>
<td>12.00</td>
</tr>
<tr>
<td>PSYC 101</td>
<td>Psychology</td>
<td>A</td>
<td>4.00</td>
<td>4.00</td>
<td>16.00</td>
</tr>
</tbody>
</table>

**UNDG** Term Att: 14.00 Earn: 14.00 GPA Crs: 14.00 GPA Pts: 61 GPA: 3.71

**Cum Att:** 30.00 Earn: 30.00 GPA Crs: 30.00 GPA Pts: 113 GPA: 3.77

#### Sample 2

**Fall 2008**

**Major: MARKETING**

<table>
<thead>
<tr>
<th>Course</th>
<th>Subject</th>
<th>Grade</th>
<th>Credits</th>
<th>GPA Pts</th>
<th>Term Att</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEE 100</td>
<td>Fund Speech Communication</td>
<td>B</td>
<td>3.00</td>
<td>3.00</td>
<td>9.00</td>
</tr>
<tr>
<td>MATH 112</td>
<td>College Algebra</td>
<td>B</td>
<td>4.00</td>
<td>4.00</td>
<td>12.00</td>
</tr>
<tr>
<td>ETHN 101</td>
<td>Intro Multicultural/Eth Studies</td>
<td>A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>ISYS 101</td>
<td>Personal Productivity IS</td>
<td>C</td>
<td>3.00</td>
<td>3.00</td>
<td>6.00</td>
</tr>
</tbody>
</table>

**UNDG** Term Att: 13.00 Earn: 13.00 GPA Crs: 13.00 GPA Pts: 39.00 GPA: 3.00

**Cum Att:** 13.00 Earn: 13.00 GPA Crs: 13.00 GPA Pts: 39.00 GPA: 3.00

### Spring 2009

<table>
<thead>
<tr>
<th>Course</th>
<th>Subject</th>
<th>Grade</th>
<th>Credits</th>
<th>GPA Pts</th>
<th>Term Att</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIOL 100</td>
<td>Our Natural World</td>
<td>B</td>
<td>4.00</td>
<td>4.00</td>
<td>12.00</td>
</tr>
<tr>
<td>BLAW 200</td>
<td>Legal Pol &amp; Reg Env Bus</td>
<td>A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>HLTH 101</td>
<td>Health &amp; Environment</td>
<td>B</td>
<td>3.00</td>
<td>3.00</td>
<td>9.00</td>
</tr>
<tr>
<td>ECON 201</td>
<td>Prin of Macroeconomics</td>
<td>C</td>
<td>3.00</td>
<td>3.00</td>
<td>6.00</td>
</tr>
</tbody>
</table>

**UNDG** Term Att: 13.00 Earn: 13.00 GPA Crs: 13.00 GPA Pts: 39.00 GPA: 3.00

**Cum Att:** 13.00 Earn: 13.00 GPA Crs: 13.00 GPA Pts: 39.00 GPA: 3.00

### Fall 2009

**Major: PSYCHOLOGY**

<table>
<thead>
<tr>
<th>Course</th>
<th>Subject</th>
<th>Grade</th>
<th>Credits</th>
<th>GPA Pts</th>
<th>Term Att</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSYC 101</td>
<td>Psychology</td>
<td>B</td>
<td>4.00</td>
<td>4.00</td>
<td>12.00</td>
</tr>
<tr>
<td>SOC 101</td>
<td>Intro to Sociology</td>
<td>A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>GEOG 101</td>
<td>Intro Physical Geography</td>
<td>B</td>
<td>3.00</td>
<td>3.00</td>
<td>9.00</td>
</tr>
<tr>
<td>CDIS 205</td>
<td>Beginning Signing</td>
<td>A</td>
<td>3.00</td>
<td>3.00</td>
<td>12.00</td>
</tr>
<tr>
<td>ENG 212W</td>
<td>Perspec: World Lit/Film</td>
<td>B</td>
<td>4.00</td>
<td>4.00</td>
<td>12.00</td>
</tr>
</tbody>
</table>

**UNDG** Term Att: 17.00 Earn: 17.00 GPA Crs: 17.00 GPA Pts: 57.00 GPA: 3.35

**Cum Att:** 43.00 Earn: 43.00 GPA Crs: 43.00 GPA Pts: 135.00 GPA: 3.14

### Spring 2010

<table>
<thead>
<tr>
<th>Course</th>
<th>Subject</th>
<th>Grade</th>
<th>Credits</th>
<th>GPA Pts</th>
<th>Term Att</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSYC 201</td>
<td>Statistics for Psychology</td>
<td>A</td>
<td>4.00</td>
<td>4.00</td>
<td>16.00</td>
</tr>
<tr>
<td>SPEE 101W</td>
<td>Interpersonal Communicatn</td>
<td>B</td>
<td>3.00</td>
<td>3.00</td>
<td>9.00</td>
</tr>
<tr>
<td>SPEE 190</td>
<td>Intro Communicatn Studies</td>
<td>B</td>
<td>3.00</td>
<td>3.00</td>
<td>9.00</td>
</tr>
<tr>
<td>PSYC 340</td>
<td>Social Psychology</td>
<td>A</td>
<td>4.00</td>
<td>4.00</td>
<td>16.00</td>
</tr>
</tbody>
</table>

**UNDG** Term Att: 14.00 Earn: 14.00 GPA Crs: 14.00 GPA Pts: 50.00 GPA: 3.57

**Cum Att:** 57.00 Earn: 57.00 GPA Crs: 57.00 GPA Pts: 185.00 GPA: 3.25
# Appendix D

A Dialog System to Supplement Student Advising Usability Survey

<table>
<thead>
<tr>
<th>Please Choose how much you agree or disagree with the following statements.</th>
<th>Agree</th>
<th>Neutral</th>
<th>Disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>It was easy to talk with the system.</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>It gave me a possible optimal schedule.</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>I had a hard time answering questions.</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>I didn't understand how to use it</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

**Long answers.**

*If you need more space, please ask for more paper*

Was there any problems with the system?

What could improve your experience?

What are kinds of things do you take into consideration when scheduling classes that the system did not ask you about?
Appendix E
Code Map

AIMLEditor.py
- Dependencies
  - Pyjamas
- Global Functions
  - boxify
- Classes
  - editor
    - Functions
      - __init__
      - makeMenu
      - toAIML
      - newState
      - setAndCreateState
      - log
      - loadContents
      - handleAIMLArray
      - onRemoteResponse
      - onRemoteError
      - loadAIMLS
  - AEService
    - Functions
      - __init__
  - State
    - Functions
      - __init__
      - addUnit
      - getUnits
      - returnAIML
      - toArray
      - makeNewUnit
  - Unit
    - Functions
      - __init__
      - setPattern
      - setTemplate
      - setThat
      - update
      - printThis
      - returnAIML
      - toArray
  - labeler
    - Functions
      - __init__
DialogUI.py

- Dependencies
  - Pyjamas
  - RequirementsHandler

- Global Variables
  - courseTypes
  - logOn
  - commonTransforms

- Classes
  - DialogUI
    - Functions
      - __init__
      - createInitialSkeleton
      - introDone
      - newInput
      - onRemoteResponse
      - onRemoteError
      - log
      - printDecision
  - DialogService
    - Functions
      - __init__
  - ARParser
    - Functions
      - __init__
      - parse
      - r1
      - r2
      - r3
      - r4
      - rej
      - findCred
      - specialSplit
  - chatManager
    - Functions
      - __init__
      - loadProfile
      - log
      - begin
      - startConversation
      - getResponse
      - takenRequest
      - onRemoteResponse
• onRemoteError
• printDecision

o introModule
  ▪ Functions
    • __init__
    • previousPage
    • nextPage
    • introCard
    • getAcademicRecord
    • notify

o chatModule
  ▪ Functions
    • __init__
    • updateConversation
    • sendInput
    • setResponder
    • log
    • disableFeedback
    • onKeyUp
    • say
    • onKeyDown
    • onKeyPress

o studentProfile
  ▪ Functions
    • __init__
    • setMajor
    • setMinor
    • setTaken
    • getMajor
    • getMinor
    • getTaken
    • getBundle

BulletinParser.py
  • Global Variables
    o logOn
    o courseTypes
  • Classes
    o BulletinParser
      ▪ Functions
        • __init__
        • preReqCheck
        • log
- inParse
- sqlReadyArray
- processType
- parseCourseHeading
- parseCoReqs
- parsePreReqs
- preTokenized
- parseGESatisfaction
- parseAvailTypes
- parseDescLine

  o Tokenized
    • Functions
      • __init__
      • addWord
      • addOnlyWord
      • finish
      • getToken
      • getRest
      • getWord
      • getWordList

  o simpleStack
    • Functions
      • __init__
      • popleft
      • pop
      • returnIt
      • __len__

  o PreReqParser
    • Functions
      • parsePreReq
      • s
      • r
      • rule1
      • rule2
      • rule3
      • rule4
      • rule5
      • rule6
      • rule99
      • acc
      • preParse

  o parseTable
    • Functions
      • __init__
• getAction
  o Course
    ▪ Functions
      • __init__
      • setDescription
      • getPreReqs
      • setPreReqs
      • setAvail
      • setGESats
      • setCred
      • toArray

MajorParser.py
• Global Variables
  o courseTypes
• Classes
  o MajorParser
    ▪ Functions
      • __init__
      • printAll
      • returnForDB
      • inParse
      • parseName
      • parseCategory
      • parseAdmissions
      • parseRules
      • allCombine
      • recursiveRules
      • parseSelect
      • parseRange
      • parseSet
      • parseExcept
      • parseLimit
      • selectSequence
      • parseAll
  o Category
    ▪ Functions
      • __init__
      • getName
      • getConstraints
      • addConstraints
OfferCoursesParser.py
- Classes
  o OfferedCoursesParser
    • Functions
      - __init__
      - inParse
      - printOut
      - mysqlSafe
      - processType
      - doNothing
      - parseCourseHeader
      - parseCourseSection
      - parseAdditionalSectionInfo
  o Course
    • Functions
      - __init__
      - addSection
      - addDayToSection
      - toArray
      - toPrinterArray
  o Section
    • Functions
      - __init__
      - addDay
      - returnPrinterArray
      - returnArray

RequirementsHandler.py
- Dependencies
  o pyjamas.JSONService
  o SGTK
- Global Variables
  o _logger
  o _needed
  o _allCourses
- Global Functions
  o setAllCourseInfo
  o setCallback
  o log
  o setNeeded
- Classes
  o RequirementsHandler
    • Functions
• __init__
• pruneNeded
• printerFriendly
• loadTaken
• getMajor
• loadMajor
• loadCourses
• courseLoading
• onRemoteResponse
• startConversation
• getCourse
• onRemoteError
• loadPossible
• makeDecision
• prunePreferences
• log

  o Category
    • Functions
      • __init__
      • loadConstraints
      • returnForCourseLoading
      • loadCourses

  o Constraint
    • Functions
      • __init__
      • getUniformBenefit
      • default
      • loadCourses
      • addPreReqs
      • loadRules
      • ALL
      • SELECT
      • SELECTSEQUENCE
      • EXCEPT
      • LIMIT
      • update
      • updateAll
      • updateSelect
      • updateSelectSequence
      • returnForCourseLoading
      • limitFulfilled

  o Set
    • Functions
      • __init__
- loadDiscrete
- loadRange
- loadExcept
- loadLimit
- isInSet
- isDiscreteEmpty
- removeFromDiscrete
- removeFromRange
- returnForCourseLoading

  o Course
    ▪ Functions
      • __init__
      • updateInfo
      • printerFriendly
      • makeTrueBenefit
      • getParent
      • recursiveGetBenefit
      • updatePrereqs
      • updateSpecial
      • __cmp__
      • __str__
      • setParent
      • isJustPreReq
      • setCreditSort
      • setBenefitSort
      • getCredits
      • getBenefit
      • addDependent
      • notify
      • isReady

  o ruleLoader
    ▪ Functions
      • __init__
      • loadRules
      • s1
      • s0
      • acc
      • r1
      • r2
      • wait
      • ALL
      • getRule
      • SELECT
      • EXCEPT
• LIMIT
  o simpleStack
    • Functions
      • __init__
      • popleft
      • pop
      • returnIt
      • __getitem__
      • __setitem__
      • __len__
  o courseService
    • Functions
      • __init__

SGTK.py
• Global Functions
  o log
• Classes
  o SGTK
    • Functions
      • __init__
      • loadNodes
      • makeEdges
      • createForest
      • rsg
      • prunePreferences
  o Forest
    • Functions
      • __init__
      • push
      • pop
      • getTotal
  o Tree
    • Functions
      • __init__
      • numCourses
      • getCredits
      • loadValues
      • getStack
      • use
      • getBenefit
      • printThis
      • __eq__
      • __ne__
• __cmp__
  
  o Graph
    Functions
    • __init__
    • setReferenceVariables
    • formPruneList
    • pruneOne
    • pruneMany
    • ordered
    • use
    • calcUndesired
    • subGraph
    • copy
  
  o CNode
    Functions
    • __init__
    • printThis
    • dayTimePrinter
    • __repr__
    • __hash__
    • __str__
    • __cmp__
    • getName
    • getBenefit
    • getCredits
  
  o NodeLinker
    Functions
    • makeTimeNodes
    • convertTimes

EditServer.py

• Dependencies
  o cherrypy
  o json
  o os
  o MySQLdb
  o xml.sax
  o AIMLHandler
  o AdvisingAssistant

• Global Variables
  o Maverick
  o PYJSDIR
  o courseTypes

• Global Functions
  o connect
A DIALOG SYSTEM TO SUPPLEMENT STUDENT ACADEMIC ADVISING

- jsonrpchdl
  - Classes
    - Root
      - Functions
        - default
        - services
        - startConvo
        - getResponse
        - informationPoll
        - test
        - storeBulletinCourses
        - parse
        - getMajor
        - loadMajor
        - getCourseInfo
        - executeAndRetrievePRVersion
        - executeAndRetrieve
        - processPrereqs
        - preprocess
        - makeToQueryPRVersion
        - makeToQuery
        - makeToQueryForPossible
        - getPossible
        - storeOfferedCourses
        - makeOfferedQuery
  - sqlWorker
    - Functions
      - __init__
      - execute
      - hasNext
      - next
      - insert

AIMLHandler.py
- Dependencies
  - xml.sax.handler
- Classes
  - AIMLHandler
    - Functions
      - __init__
      - startElement
      - characters
      - endElement
      - printAll
      - getStates
A DIALOG SYSTEM TO SUPPLEMENT STUDENT ACADEMIC ADVISING

- toArray

  - State
    - Functions
      - __init__
      - addUnit
      - getUnits
      - returnAIML
      - toArray

  - Unit
    - Functions
      - __init__
      - setPattern
      - setTemplate
      - setThat
      - printThis
      - returnAIML
      - toArray

AdvisingAssistant.py
- Dependencies
  - aiml.Kernel

- Classes
  - AdvisingAssistant
    - Functions
      - __init__
      - Manager
      - informationPoll
      - advance
      - getResponse
      - beginConvos

  - LinearConversation
    - Functions
      - __init__
      - setOrder
      - storeStateData
      - advance
      - getCurrentState
      - dataVector

  - BasicDialog
    - Extends LinearConversation
    - Functions
      - __init__