StanPlan

Adam Keppler
Department of Computer Science
Stanford University
akeppler@stanford.edu

Julia Truitt
Department of Computer Science
Stanford University
jtruitt@stanford.edu

Ricky Grannis-Vu
Department of Computer Science
Stanford University
rickygv@stanford.edu

Abstract

StanPlan consolidates the information students need to organize and create their four-year course plan into one convenient UI. Our web app automatically generates a schedule for students based on the courses they select and helps students make sure they satisfy necessary prerequisites in the right order. With StanPlan, the quarterly course-picking scramble currently experienced by so many undergraduates will be transformed into one smooth, long-term planning experience.

1 Introduction and Motivation

Imagine you're a Stanford freshman beginning to plan your path to graduation over the next four years. You begin by looking at a high-level outline of your degree requirements on ExploreDegrees [1] and are overwhelmed by the amount of classes you must choose between to satisfy various requirements. To narrow down your options, you want to view more details about each class, including its description and quarters it is offered. To do so, you navigate to ExploreCourses [2]. However, descriptions on ExploreCourses [2] only offer a summary of the class. To find out more about a class' reputation from the student's perspective, you search for the course on CARTA [3]. After collecting all of this information, you finally know enough to decide whether you want to take this individual course, but you still lack the necessary information to compile a four-year plan. Moreover, after the agonizing process of piecing together your four-year plan on a makeshift Excel spreadsheet, you must enter the information all over again on yet another website to submit your official degree plan.

Unfortunately, this scenario is far from hypothetical; it is the experience of nearly every Stanford student trying to navigate the vast array of university course offerings and somehow generate a sensible four-year plan. Though there are many resources for collecting information about a single course, relevant information is scattered across multiple web pages (including ExploreCourses [2], ExploreDegrees [1], and CARTA [3]), and there is little guidance on how to create a long-term plan.

To tackle this issue, we set out to create an app that would consolidate relevant course information in a single interface and provide students with a streamlined tool for developing their four-year plan.

2 Design and Implementation

2.1 User Interface

We took an iterative design approach to developing our user interface. First, we went through several iterations of sketching, before beginning our implementation of the user interface. We then walked through several potential user flows via our paper model. These user flows helped us to consider the different degree requirement types required by different tracks. This approach allowed us to quickly and cheaply experiment with how users might interact with our app and identify what scenarios our user interface needed to support.

Ultimately, we found that we could represent the course selection process using three different types of inputs: single-choice course selection, multiple-choice course-selection, and open-ended course selection. We implemented single-choice course selection and multiple-choice course selection using radio buttons and multiple choice boxes respectively. These radio buttons and multiple choice boxes are attached to rows displaying basic course information: course code, course title, and unit count, and these rows can be expand to display the course description when clicked. For open-ended course selection, we created two text fields, side by side, one of which prompts the user to enter a course code and the other prompts the user to enter the number of units the course for which to take the course. We send the course code to the server, which checks to make sure that the course code exists in our list of available courses, and use this as a form of validation to provide immediate feedback to the user about whether or not their entered course is still offered.

Since our user interface consists of many common components with minor differences (e.g. course information rows with varying titles, units, and descriptions), we decided to use the React framework, which is a popular component-based user interface framework. By using React, we were able to maximize component reuse and minimize code repetition, while still remaining free to customize components where necessary. This component reuse proved useful both for cleanly implementing our user interface and for making the process of adding majors and tracks very scalable (discussed in further detail in section 2.5).

Two of the challenges we faced were saving a user's course selections if the user leaves the page and comes back to it and selecting a course across multiple pages if the same course is found on multiple pages and the user selects it on one of those pages. To solve these problems, we decided to use the Redux framework, which is a powerful state-management system which integrates well with React. Redux allowed us to maintain a global state across the entire app so that different pages could easily communicate with each other by pushing updates to the global state. We were able to resolve these challenges using Redux by storing the list of selected courses in the global state, pushing updates to it whenever a user selects a course on any page, and using this global list of selected courses when determining which courses should be selected on each page.

Once the user has picked out a list of courses to take, the user interface sends this list to a server, which we implemented using the Express framework for Node.js. To facilitate communication between the server and client, we created several API routes that the client can invoke. For example, after the user selects a major and track on the Welcome Page, the client side calls one such API routine, which sends the selected major and track to the server. The server then responds with a JSON file containing the information that the client needs to populate the user interface with the requirements for that major and track combination. To take another example, after the user has finished selecting courses and has indicated their scheduling preferences, the client sends the server the list of selected courses and the user's scheduling preferences. The server inputs this data into the scheduling algorithm and sends the resulting schedule to the client as a JSON structure representing the user's four-year plan.

2.2 Data Acquisition

In order for us to perform course validation for user-given courses, prerequisite extraction, and scheduling, we needed to extract a corpus of information about Stanford's course listings/catalog. In order for us to acquire this data, we explored both a administrative approach and an automatic one. Both approaches proved fruitful. We contacted Ms. Claire Stager, who manages the CS depatarments course information, and learned that only Stanford's registrar has a collection of all course information across departments. We were also able to confirm that ExploreCourses [2] contained an accurate and complete image of the course information our system needed.

Our automated approach was facilitated by Jeremy Ephron's open source python API for the Explore Courses website [4]. We used the ExploreCourses API [4] to scrape all information relevant to our work from ExploreCourses [2]. In addition to acquiring the data from ExploreCourses [2], we hand labeled 483 course requisite relationships to generate a gold dataset for the evaluation of our extraction system.

We also heavily consulted ExploreDegrees [1] when designing our degree requirement pages; ExploreDegrees details all of Stanford's degree plans and is currently the primary source for degree planning information, in tandem with ExploreCourses [2].

2.3 Prerequisite Extraction

Currently Stanford has no database or corpus mapping classes to their prerequisite or co-requisite requirements. Prerequisite requirements indicate the necessary courses that must be taken before another course should be taken, and co-requisite classes indicate classes that must be taken in conjunction with a course in the same quarter. A single class can often have multiple prerequisites and/or co-requisites; it is also possible for a requisite requirement to have multiple parts, such as a requirement that course C must be taken with either course A or course B, but not necessarily both. Codifying these requirements was a vital step in being able to implement automatic scheduling and validation of course plans. There is also an additional class of requirement, which we have named soft co-requisites, as they can be scheduled as either prerequisites or co-requisites. To simplify our scheduling algorithm, we chose to treat soft co-requisites as prerequisites.

Information about prerequisites and co-requisites is only available in course descriptions, and must be extracted to produce a dataset/database. Stanford currently offers approximately 15,000 courses; manually annotating this entire corpus would be time-consuming and labor-intensive. Rather than hand labeling this entire corpus, we pursued the development of an automated extraction system and hand labeled a limited subset for use in evaluating the quality of our extraction model.

Automated extraction would entail not only extracting a list of the relevant mentioned classes, but also understanding the relationships between them. Prerequisites and co-requisites can be complex structures, with their own internal relationships. "AND" relationships are the classic case, where all courses indicated are required. "OR" relationships are also quite common and indicate instances, where only one course in a specific sequence needs to be taken to satisfy the requirement.

We first considered the use of a deep learning approach, however the absence of a dataset for this task or a sufficiently similar task meant that a deep learning model would be unable to successfully train on the task. As such, we began to exploring a variety of other methods.

We first explored Snorkel [5], a system for relation extraction, automated labelling, and dataset generation. At first glance, the Snorkel package sounded like a perfect fit for our task, but unfortunately Snorkel was developed for classification tasks and was unable to handle the complex relationships between requisite classes. However, our experience with Snorkel provided us with insight and inspiration for the development of a system to test a rules-based approach to extraction.

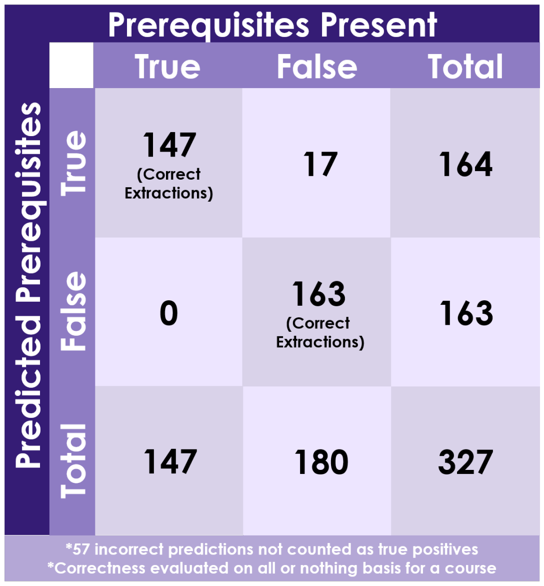
Next, we developed our own testing framework to evaluate our custom designed labelling functions. We began with a rules-based approach intended to produce a single highly effective rule and found that the results were effective, but that handling "OR" and "AND" relationships was challenging, and error analysis frequently revealed unusual edge cases that would require extensive revision to support. To help handle "OR" and "AND" relations, we explored the use of a neural parser developed by the Stanford NLP Lab [6] to attempt to detect and parse phrases that defined the requirement. This approach unfortunately proved to be ineffective, as the grammar used in course descriptions is neither ubiquitously correct nor consistent across descriptions. Given that neural approaches proved ineffective, we reverted back to our rules-based model and continued to further improve our prediction system.

After considering the frequency of edge cases revealed via error analysis on our limited hand labelled dataset, we began to grow concerned about the generalization of our rules. As such, we began to take a new approach that focused on developing rules that are guaranteed to both be theoretically and empirically validated as having 100% precision. This approach favors the development of many highly specialized functions that detect and extract the relations in a very specific pattern. While these functions individually do not often provide a high number of predictions, their predictions are made with an extreme level of confidence and would not require human validation. We currently have a few such functions based on some common cases found through our earlier error analysis.

Our best such function is a predictor of when classes have no requisite courses. This function simply checks for the absence of class codes, class code suffixes/numbers, and the keywords "prerequisite(s)" and "co-requisite(s)." The absence of these attributes indicates from a logical standpoint that no course requirement could be stated in the description; our empirical testing also validated this assertion. This function is able to correctly predict 163 of our 483 hand labeled courses, or roughly 1/3 of our gold data. We believe the frequency of courses with no requisites in our gold data is representative of their frequency/distribution overall and that this function may alone be able to resolve 1/3 of our dataset.

Another advantage of our current approach is that any prediction from these high confidence functions would not require manual verification. As such, we can use these functions to filter our dataset to a more limited subset, such that human validation is significantly more feasible. We can also further assist human annotators by outputting a confidence prediction and providing a supplementary BRAT [7] Annotation. The BRAT Annotator [7] is a tool for highlighting important elements of a text and would further help human annotators to spot course codes and keywords.

Our new approach combined with our preliminary predictor - which did not guarantee 100% precision - is able to correctly extract/generate the course relations information for 310 classes out of the 384 classes that it produced a prediction for. Thus, the combined accuracy of both systems is 80.7% when a prediction is made. It is important to note that this also means neither of our rules made predictions for 99 of our gold dataset's courses. Figure 1 shows a confusion matrix for our predictions; 17 of our predictions are false positives, while 57 of our errors are due to imperfect extractions. It is also important to note that for these 57 predictions, they are not always entirely inaccurate. It is often the case for these predictions that the system was able to detect the right course codes, but failed to correctly resolve the "AND" or "OR" relationship structure. There are a variety of different issues that can lead to the classification, but we believe that by further building perfect high confidence predictors, many of these inaccurate predictions could be resolved, helping to decrease the number of instances that require human verification.



		Prerequisites Present		
		True	False	Total
Predicted Prerequisites	True	147 (Correct Extractions)	17	164
	False	0	163 (Correct Extractions)	163
	Total	147	180	327

*57 incorrect predictions not counted as true positives
*Correctness evaluated on all or nothing basis for a course

Figure 1: *Confusion Matrix for Requisite Extraction*

2.4 Scheduling Algorithm

One of the core benefits of StanPlan is its ability to schedule a full four-year plan for students. In addition to selecting courses that satisfy degree requirements, designing a four-year plan means considering the expected term offerings of courses, quarterly unit load, and prerequisite and co-requisite requirements. These considerations can make planning challenging for students, as Stanford currently offers little support for long-term planning. Students and course advisers currently need to produce and validate these schedules by hand; this process is tedious and error-prone and can lead to students discovering unsatisfied requirements late in their academic career. StanPlan offers a reliable, automated, and simplified approach to help both students and advisers produce workable schedules.

One critical component of our scheduler is its ability to ensure that prerequisite and co-requisite requirements are met. In order for students to correctly generate their schedules, they need to be aware of the prerequisite and co-requisite requirements for each course. It is easy for students to miss these important relations, which can cause unnecessary stress. To ensure that students have correct schedules that reflect necessary prerequisite courses and satisfy co-requisite requirements, our scheduler by default offers the option to automatically add all prerequisite and co-requisite classes to ensure that all such requirements are met, even if the student does not explicitly select the requirement. Our system also offers a way to disable this feature, so students who have already completed or waived these requirements can see their schedule without the additional classes. If requirement assurance is disabled, we still ensure that any prerequisite or co-requisite relations between courses explicitly selected by the student will be maintained. We believe that this approach allows students peace of mind, ease of use, and flexibility.

2.4.1 Requisite Satisfaction and Graph Structure

Scheduling incorporates elements from both Constraint Satisfaction and Scheduling Algorithms. We use a series of Graph Structures to represent the relationships between courses, which we will refer to predominantly as nodes. Our most fundamental structure is a Directed Acyclic Graph (DAG) representing the prerequisite relationships between courses. In this graph, every edge represents the flow of scheduling information from one node to another. For example a perquisite constraint that CS 106 must be taken before CS 107 would result in a directed edge from CS 106 to CS 107. This edge allows CS 106 to notify CS 107 of changes to its scheduling status. If CS 107 receives notification that CS 106 is scheduled it can then mark the constraint as satisfied and re-evaluate its status.

Our DAG very effectively resolves "AND" relationship prerequisites, where all root nodes are eligible for scheduling. However, "OR" relationships require additional management overhead. When an "OR" relationship is satisfied, it is often that case that we no longer need to schedule the alternative classes. "OR" relationships necessitate that we not only notify the courses that required the scheduled class, but also all of the alternative classes. This relationship between alternative offerings is by definition cyclic and as such cannot be encoded by the prerequisite DAG. As such we use a second Directed Graph structure to represent these relationships. This Directed Graph can be imagined as a collection of self-referential clusters, where if any one node in a cluster is scheduled, the remainder of the cluster no longer needs to be scheduled. The edges in these clusters represent delete if notified relationships, but it is important to note that notifications are not recursive, such that each node must be in contact with all other nodes in the relationships. Moreover, if a node is a part of multiple relationships or has been explicitly specified by the user, the notification of the completion of one cluster will not stop it from being scheduled, but merely update its internal state, so it is able to continuously re-evaluate its status. To ensure that nodes are not eliminated prematurely and recognize the completion of a cluster correctly, we further implement to additional mechanisms.

In some cases, a class that is a part of one "OR" relationship may be the part of another "OR" or "AND" relationship, or have been explicitly required by the user. In these cases, the course should not be removed if an alternate is scheduled. To ensure that these cases are handled correctly, each node stores a collection of the relationships that is a part of, whether or not it was explicitly scheduled by the user, and a list of the nodes that necessitate it's scheduling. If all relationships and nodes in need of it have been scheduled and the node was not explicitly selected by the user, then the node will no longer be considered schedulable as it is no longer necessary to schedule it.

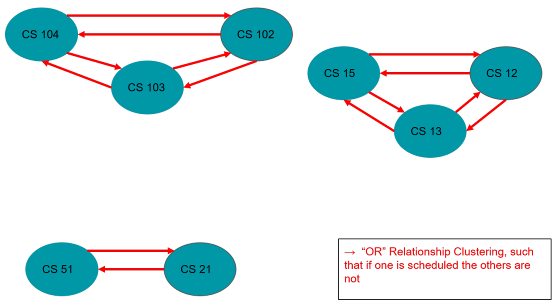

CS 104
CS 102
CS 103
CS 15
CS 12
CS 13
CS 51
CS 21
→ "OR" Relationship Clustering, such that if one is scheduled the others are not

Figure 2: *A visualization of 3 OR relationship cluster's as self-referential clusters that ensure that only one node is scheduled.*

Furthermore, due to our implementation of scheduling eligibility evaluation and the graph structures, we need to have an additional mechanism for notifying "OR" relationship classes of the status of the relationship. This method is a supplementary component that ensures that multiple alternates are not scheduled in the same quarter, it does not however accomplish the same delete if scheduled relationship described above. This mechanism is a lookup table that allows each node to check the status of its "OR" relationships. This method allows the node to check whether or not each of the "OR" relationships that it is a part of have been satisfied. This step is necessary, as it allows us to update the status of "OR" relationships without simultaneous processing updates of dependencies; this distinct update process helps to ensure that multiple classes satisfying the same "OR" relationship do not get scheduled in the same quarter and that classes with a prerequisite dependency on the class scheduled in the current quarter are not notified of the dependency being resolved until the end of the quarter.

To increase the memory efficiency of our program and ensure that the graph structures are synchronized we implemented the two graphs using a single set of nodes, that maintain separate lists of their incoming edges based on the graph that the edge is associated with, this allows for an annotated understanding of the type of relationship held with regards to the calling node. This also offered the advantage of localizing all information for a specific course to a specific node. This approach also allowed us to streamline the management of outgoing edges. Each node simply maintains a list of all nodes that need to be notified that it has been scheduled and invokes there inform() method, while passing along its course name. Given the name of the caller the receiving node is able to determine the graph associated with the edge and the appropriate behaviour that it should undertake. This behaviour can involve satisfying a constraint or changing its status to no longer in need of scheduling depending on the relationship with the calling node.

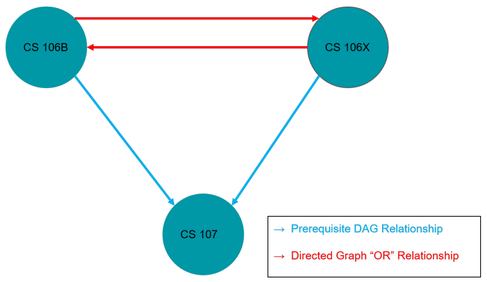

CS 106B
CS 106X
CS 107
→ Prerequisite DAG Relationship
→ Directed Graph "OR" Relationship

Figure 3: *A visualization of both graphs' edges overlaid on a single set of nodes to represent the prerequisite constraint that CS 106B or CS 106X must be taken before CS 107.*

In addition to our graph structure, we maintain a list of all nodes that need to be scheduled. It is important to note that not all nodes in this list will immediately be ready for scheduling, but as our scheduling algorithm runs, the status of the nodes will change as requirements become satisfied. This list is also important, as it allows us a way to remove nodes that have been scheduled or no longer need to be scheduled without trimming our DAG. The status of nodes in the DAG does change, but all nodes and edges remain in the DAG after its construction. By using this approach, we always have access to all of the root nodes that are ready to be scheduled.

In addition to encoding prerequisite relationships, our graph structure encodes the co-requisite relationships for classes. This is achieved by merging nodes that are co-requisite to one another; the resulting node from such a merge can be referred to as a clumped node. These clumped nodes represent multiple courses that are to be scheduled as a single unit. Clumped nodes are treated as a single class/node for the purpose of scheduling evaluation, ensuring that when the classes are scheduled the requirement is satisfied. The clumped node itself represents the information for all of the sub-nodes and has edges corresponding to the union of the sub-nodes' edges.

When we encounter "OR" relations for co-requisites, we create all possible clumps or co-requisite pairings that could satisfy the relationship and add "OR" relation edges between them to ensure that only one clump is ever scheduled. Similarly, because the clump maintains all of its' sub-node edges, it will notify all possible dependants, receive information from all the sub-nodes' parents, and process received information according to all appropriate relations in the sub-nodes.

These clumped nodes act very much like normal nodes and the "OR"-relationship is effectively the same as a standard "OR"-relationship. We have included a diagram of this relationship in Figure 4 to demonstrate the similarity. The diagram shows how course CS 41 with a co-requisite of CS 51 or CS 21, CS 104 with a co-requisite of CS 105, CS 103, or CS 102, and CS 19 with co-requisite of CS15, CS 13, or CS 12 would be represented. If any one of the nodes in the cluster is selected the co-requisite is satisfied and the remaining nodes in the cluster will not be scheduled.

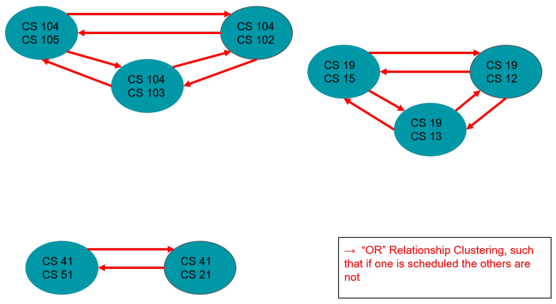


Figure 4: *A visualization of an "OR" cluster using Clumped Nodes.*

2.4.2 List Scheduling

Having developed a mechanism to represent the relationships between classes, we implemented a list scheduling algorithm to determine which classes to place in which quarter. Our algorithm took a quarter by quarter approach to scheduling. The algorithm begins by querying all courses in the list of nodes to schedule for their ability to be scheduled in the current quarter. Each node determines its eligibility to be scheduled in the quarter by first ensuring that the current term is listed in its list of offered terms. We use a heuristic approach to determine the expected terms that each course will be offered in; our approach examines the offerings listed in our database to determine the expected terms. This heuristic only provides moderate insight, but is still useful in helping students develop reasonable plans. A node also checks to verify that it has no outstanding dependencies (i.e. that all of its prerequisites have been satisfied) and has not had any of its deletion conditions triggered. If all conditions are met, the node is considered schedulable, and its priority is calculated. Once the node is verified as schedulable, we also check to see if adding it would cause us to exceed the maximum number of units for the quarter. Should the number of units be too large, the node is not scheduled in this particular quarter. This helps to ensure that students do not have quarters that are too heavy.

A node's priority is determined by the amount of future class it will allow to be scheduled. Classes that will be affected by the scheduling of the current node can be considered its dependants. All nodes have a priority value of the number of courses in the node, which is one except for clumped nodes, plus the sum of the priorities of its dependants. We calculate the priority of the node, by traversing the edges of the prerequisites DAG and querying those nodes for their priority. Only nodes that have a dependency relationships to the calling node will respond with their value.

After identifying the highest priority node that is eligible for scheduling, the list scheduler schedules the class, updates the "OR" table, deletes the scheduled class from the list of eligible classes, and repeats the above process until there are no longer any eligible classes for the current quarter. This ineligibility may be due to reaching the max units per quarter limit, prerequisite ineligibility, or term restrictions. We then update the DAG, by telling each of the scheduled nodes to inform its dependants that it has been scheduled, and repeat the process for the next quarter. This method ensures that all dependencies are met before a class is scheduled, prerequisites and co-requisites are taken as early as possible, and user unit limits are respected.

2.4.3 Simulation and Testing Infrastructure

Our scheduling algorithm is a critical component of our application that involves many important and distinct facets. To ensure the accuracy, effectiveness, and reliability of our algorithm, we developed a testing harness and wrote a variety of test cases to stress test our system. The algorithm takes as input a JSON file that contains a list of the classes selected by the user and their associated unit counts, as well as a database file. The database file maps the course IDs to their corresponding prerequisite/co-requisite requirements and expected terms. The user input JSON can be considered our node list, and the database the edge list. By defining our inputs in such a way, we can write custom input files and databases to test very specific scenarios and specify unique node and edge combinations.

This method of testing allowed us a great deal of flexibility and allowed us to validate the robustness of our scheduler with a high degree of confidence.

2.5 Scalability

Our design approach is highly scalable to additional majors and tracks. Each major is represented by a subfolder labeled by its name. Inside each major's folder is a list of JSON files, all but one of them representing a track or concentration inside of that major. The last one of the JSON files represents the general requirements a user must satisfy for that major. Inside each JSON file, requirements are formatted as a list of *Sections*, representing different pages, each containing a list of *Subsections*, representing different course selection groups on that page. For example, a page requesting the user to select a Physics course could be represented as follows:

```
{
  "title": "General Computer Science Requirements: Science",
  "instructions": "Select one of the following Physics courses.",
  "section": [
    {
      "subsection": [
        {
          "id": "PHYSICS 41",
          "required": false
        },
        {
          "id": "PHYSICS 41E",
          "required": false
        },
        {
          "id": "PHYSICS 21",
          "required": false
        },
        {
          "id": "PHYSICS 61",
          "required": false
        }
      ],
      "type": "radio"
    }
  ]
}
```

This standardized and simple format allows developers to easily add new majors and tracks to StanPlan and further expand its applicability to more universities with little to no prior programming experience.

3 Results

We succeeded in developing a working prototype of StanPlan that allows students to schedule for year plans for the CS Major. The user is able to select the Computer Science major and a track within the major on the Welcome Page. The user is then guided through the process of selecting their courses, through a variety of single-choice, multiple-choice, and open-ended course options. Course descriptions and unit counts are provided to the user to assist them during this process. Required courses are pre-selected and marked as such, and user selections are tracked across pages to avoid duplicate selection and simplify the scheduling process. We also give the user the ability to skip any section and deselect any of the required courses to allow maximum flexibility.

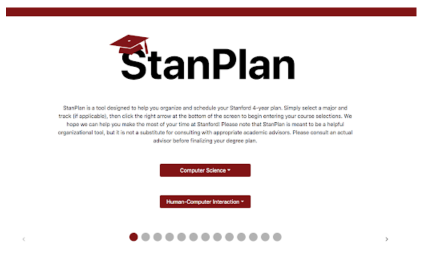
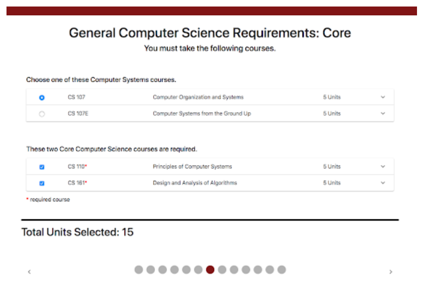

StanPlan is a tool designed to help you organize and schedule your Stanford 4-year plan. Simply select a major and track (if applicable), then click the right arrow at the bottom of the screen to begin entering your course selections. We hope we can help you make the most of your time at Stanford! Please note that StanPlan is meant to be a helpful organizational tool, but it is not a substitute for consulting with appropriate academic advisors. Please consult an actual advisor before finalizing your degree plan.

Figure 5: *Welcome Page*


General Computer Science Requirements: Core

You must take the following courses.

Choose one of these Computer Systems courses.

●	CS 107	Computer Organization and Systems	5 Units
○	CS 107E	Computer Systems from the Ground Up	5 Units

These two Core Computer Science courses are required.

✓	CS 110*	Principles of Computer Systems	5 Units
✓	CS 161*	Design and Analysis of Algorithms	5 Units

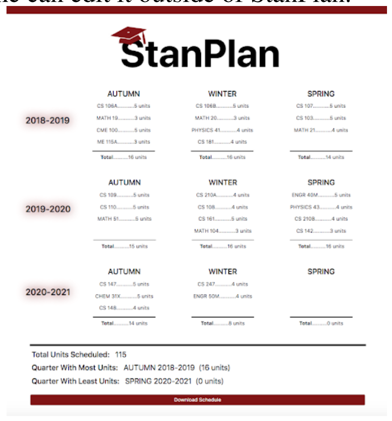
* required course

Total Units Selected: 15

Figure 6: *Course Selection Page*

After selecting their courses, the user is able to set several scheduling preferences that our automated scheduler will honor. The user is able to set a max unit cap per quarter, such that if the user chooses a max unit cap of 16 units, then the scheduler will assign no more than 16 units per quarter. The user is also able to choose whether or not they want to schedule courses during the summer quarter. The user is also able to assert whether or not they want the scheduler to add missing prerequisite or co-requisite courses for the selected courses. For example, if the user selects MATH 20 without selecting MATH 19 (a prerequisite for MATH 20), the scheduler can and will by default automatically add MATH 19 to the user's schedule.

After taking in the user's preferences, StanPlan then automatically generates a schedule for the user. In the schedule, which can be see in Figure 4, each course's prerequisites are satisfied before the course is scheduled. This allows the user to be well prepared for each course he or she decides to take. Additionally, we allow the user to download his or her schedule as a csv file, so that he or she can edit it outside of StanPlan.



	AUTUMN	WINTER	SPRING
2018-2019	CS 106A..........5 units	CS 106B..........5 units	CS 107..........5 units
	MATH 19..........3 units	MATH 20..........3 units	CS 103..........5 units
	CME 100..........5 units	PHYSICS 41..........4 units	MATH 21..........4 units
	ME 115A..........3 units	CS 181..........4 units	
	Total..........16 units	Total..........16 units	Total..........14 units
2019-2020	CS 109..........5 units	CS 210A..........4 units	ENGR 40M..........5 units
	CS 110..........5 units	CS 108..........4 units	PHYSICS 43..........4 units
	MATH 51..........5 units	CS 161..........5 units	CS 210B..........4 units
		MATH 104..........3 units	CS 142..........3 units
	Total..........15 units	Total..........16 units	Total..........16 units
2020-2021	CS 147..........5 units	CS 247..........4 units	
	CHEM 31X..........5 units	ENGR 50M..........4 units	
	CS 148..........4 units		
	Total..........14 units	Total..........8 units	Total..........0 units

Total Units Scheduled: 115

Quarter With Most Units: AUTUMN 2018-2019 (16 units)

Quarter With Least Units: SPRING 2020-2021 (0 units)

Figure 7: *Schedule Page*

Currently, our prototype supports the full Computer Science major at Stanford University. This program includes all of the general education requirements (THINK, WAYS, PWR, and Foreign Language), the general Computer Science requirements, and all of the Computer Science track options and requirements.

4 Lessons Learned

In creating StanPlan, we learned a variety of valuable skills. On the web development side, we learned how and when to use React, a popular component-based user interface framework, and Redux, a powerful state management system which works in tandem with React. We also learned how to create a client-server framework with NodeJS and Express. Finally, we learned how to conduct user testing and how to use this feedback to develop and refine an attractive and efficient user interface.

We also discovered the difficulty of working with unlabelled textual corpora and the challenges associated with developing datasets for Deep Learning. In developing our text extraction system, we learned how to optimally generate a large scale corpus using automated solutions and human annotation. We also learned about the advantages and limitations of Snorkel and Neural Parsers for textual extraction in a real-world setting.

Finally, we learned how to implement a complex course scheduling algorithm. Based on feedback from Professor Lam, we ultimately settled on the list scheduling algorithm, which proved very effective in satisfying all prerequisites before their courses. We learned how to represent all the course data and prerequisites as a directed acyclic graph and how to implement additional features such as co-requisites into our list scheduling algorithm. We also learned how to test for edge cases, such as the case where the user selects a course which is no longer offered or the case where the user fails to select any courses at all. We learned how to write compact and efficient tests for these edge cases, which we combined into a testing suite so that all the tests can be automatically run at once.

5 Related Work

Course Catalogs. Course catalogs are used to record and share information about courses to students at nearly all universities. At Stanford, the course catalog is called ExploreCourses [2]. The advantage of course catalogs such as ExploreCourses [2]is that they provide information about all the current course offerings, including course code, course title, course description, unit count, and quarters offered. The disadvantage of these course catalogs is that they are purely descriptive and do not allow the user to actually select courses to take in the future.

Degree Catalogs. Degree catalogs are used to record and share sequences of courses needed to be taken to satisfy degree requirements. At Stanford, the degree catalog is called ExploreDegrees [1]. The advantage of degree catalogs such as ExploreDegrees [1] is that they provide information about which courses to take to graduate. Like course catalogs, the disadvantage of these degree catalogs is that they are purely descriptive and do not allow the user to actually select a degree program or courses to take within that degree program.

Program Sheets. Program sheets are typically used to certify that a student has taken all the courses he or she needs to graduate. At Stanford, there is no standard program sheet and different majors have different program sheets. The advantage of program sheets is that they allow students to check off which courses they have taken. The disadvantage of program sheets is that they neither provide information about the courses nor allow students insight into whether their filled-out program sheet can actually be completed within four years.

Course Selectors. Course selectors are used to allow students to sign up for courses they want to take in the next quarter. At Stanford, the course selector is called Axess [8]. The advantage of course selectors is that they allow students to see course descriptions and whether they can fit the chosen courses into the next quarter. The disadvantage of course selectors is that they neither certify whether the courses will actually help the user satisfy his or her degree requirements nor allow the user to plan for four years, or even further in advance than the next quarter.

6 Future Work

In the future, we plan to improve upon our existing implementation of StanPlan. Initially, we hope to conduct further user testing of our user interface to get a better idea of what features we should prioritize in the future. As we continue implementing new features in StanPlan, we will continually conduct user testing. We have already formed a group of several current freshmen who are interested in testing StanPlan.

Based on feedback we have already received, we plan to allow greater flexibility on the schedule page of our user interface. We will add the ability to drag, drop, add, and delete courses on the schedule page.

Among the features we plan to add, we would like to complete a full dataset of Stanford's prerequisite structure. Currently, we only have the prerequisite structure of four majors. We hope to use this dataset to train an AI model to automatically extract prerequisites and co-requisites from course descriptions.

Additionally, we would like to provide a more complete offering of Stanford's degrees. Currently, we only support the Computer Science degree and its associated tracks. In the future, we plan to include all majors and tracks offered at Stanford. Furthermore, we plan on providing support for graduate and coterminal student schedules as well.

We will also add a login functionality, to allow users to save their schedules online. Additionally, we will allow users to save more than one schedule. This would be helpful if the user has different versions of a schedule he or she would like to save or if the user wants to compare schedules for different majors or tracks.

Currently, the user is able to download their schedule as a csv file. In the future, we will also allow users to edit this csv file and to re-upload it to StanPlan in order to verify that it still satisfies all requirements/prerequisites. We will also allow users to upload their transcripts, such that we can extract the course data from their transcripts and save them as schedules to StanPlan.

Our end goal of this process is to eventually expand to more universities. We hope to create a start-up company that licenses the StanPlan software to other colleges. Right now, we are planning on approaching UCI with a prototype of our software and launching StanPlan for before the Stanford and UCI campuses this upcoming fall quarter to help the incoming freshmen with their course plans.

7 Conclusion

At many universities, especially public four-year universities, many students have difficulty graduating within four years. Multiple factors have been mentioned during discussions with students, from difficulty navigating the extensive degree process to a lack of enough course advisors to meet the growing student population. As the number of students attending universities continues to increase, it has become more and more imperative that we find a solution to address this issue. The aim of StanPlan is to guide students through the process of selecting courses that will satisfy their degree in one clean, simple, and consolidated user interface.

This paper presents a fully functional prototype of StanPlan. The key concepts presented include the web interface, prerequisite extraction, the scheduling algorithm, and a design for scalability.

StanPlan does not seek to be a replacement for course advisors; indeed, we advise students to verify their StanPlan-generated schedule with a course advisor. Instead, StanPlan seeks to be a supplement for course advisors. By helping students plan out their four-year plans on their own, StanPlan can help save both advisors and students time. Allowing course advisors to meet with more students and to better help students who may be struggling. We hope that the introduction of StanPlan to universities nationwide will help more students graduate within four years.

Acknowledgements

We would like to thank Professor Monica Lam for her generous mentorship, guidance, and feedback throughout the quarter and project. We would also like to thank Sam Premutico for his support, excitement, and willingness to experiment with StanPlan. We would also like to thank Ms. Claire Stager for her insights on how Stanford manages its course information.

References

[1] Stanford. Explore degrees. https://exploredegrees.stanford.edu/.

[2] Stanford. Explore courses. https://explorecourses.stanford.edu/.

[3] Stanford. Carta. https://carta.stanford.edu.

[4] Jermey Ephron. Explore courses api. https://github.com/jeremyephron/explore-courses-api, 2019.

[5] Alexander J Ratner, Stephen H Bach, Henry R Ehrenberg, and Chris Ré. Snorkel: Fast training set generation for information extraction. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1683–1686. ACM, 2017.

[6] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014.

[7] Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. Brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107. Association for Computational Linguistics, 2012.

[8] Stanford. Axess. https://axess.sahr.stanford.edu.