Generating Automated Problem Sets for Rapid Content Delivery and Adaptive Learning Modules

Dr. Philip Jackson, University of Florida

Dr. Philip B. Jackson earned B.S. degrees in Aerospace Engineering and Mechanical Engineering as well as an M.S. and Ph.D. in Mechanical Engineering, all from the University of Florida. He is currently a faculty member at the Institute for Excellence in Engineering Education at the University of Florida. There he specializes in implementing innovative methods of instruction in undergraduate courses on dynamics, heat transfer, and thermodynamics. His research interests include numerical heat transfer, fluids, and magnetohydrodynamic simulations and facilitating undergraduate students to engage in similar projects. He is also focused on the implementation of engineering freshman design experiences.

Generating Automated Problem Sets for Rapid Content Delivery and Adaptive Learning Modules

Abstract

Problem solving plays a critical role in the education of young engineers. Word problem sets are a vehicle that educators use to teach and assess that skill. While textbooks, problem repositories, and online learning systems provide a host of interesting problems there will always be a need to generate new problems to increase variety, to prevent students from cheating, and to facilitate robust student learning environments. While current online learning systems provide educators with problems that allow for individual numerical values to be randomized, they do not allow for randomized problem structures that challenge comprehension.

This paper develops a method to create new problem sets through the development of software tools that apply a series of automated generation algorithms. Several common undergraduate engineering word problems are distilled into archetypes (generalized problem descriptions that will facilitate automated authoring). One archetypical problem in each of the following disciplines is chosen: in particle dynamics (two-dimensional projectile motion), in thermodynamics (ideal-gases in piston-cylinder devices), and in circuits (equivalent resistance). For each archetype, algorithms codify problem parameters and generate a compatible list of inputs and outputs, problem diagrams, word problem text, and solution sets. Problem text is created using natural language programming (with varying levels of human intervention) and problem solution sets are constructed using a computer algebra system.

The problem sets are then presented to undergraduate mechanical engineering students as traditional course content such as homework, quizzes, and tests, and as part of adaptive learning modules or games in learning management systems. The relative difficulty of each automatically created problem is estimated with a heuristic and compared against student performance.

Background and Introduction

The following study seeks to organize, codify, and implement a method to adapt several undergraduate problem types into algorithms capable of automatically generating problem sets. Three fundamental undergraduate courses heavy in need for varied problem sets have been chosen as venues for the study: dynamics, thermodynamics, and circuits. As anyone who has created courses in one of these core topics knows, writing problems to test students on class learning objectives is not a trivial task. Original problems, of which each faculty member has their own style, can be time-consuming to write. Problems not only have to be written, but care must be taken so that each problem properly addresses the topics desired, problems are often desired to be sufficiently unique or exciting, they must be error free and solutions to problems must also be calculated.

There are several sources that faculty members can draw upon to find new problems. Textbooks are the first source that come to mind and each publisher painstakingly compiles hosts of

problem sets in each and new editions with augmented problem sets are published yearly. Many publishers also supply online learning systems for their textbooks that offer computer-based modules that contain problems. Often, the problems contained within the online learning systems can even have their input values generated randomly [1]. Some educational groups have also compiled repositories of problem sets that can be used as sources for instructors. These sources are not without their flaws. Textbook problem sets will always be incomplete with either too few or too many problems of a type or difficulty. Repositories may lack quality or organization even when they provide variety [2]. Perhaps one the largest drawbacks of these two methods of obtaining problems is the fact that students find and publish solutions to these sets online [2]. For the savvy internet searcher, the challenge of these problems then becomes trivial. Online learning systems also have a disadvantage when it comes to one of their greatest strengths in their ability to generate problems with random inputs. In this case savvy students will find a friend (or often examples provided by online system itself) whose solutions are available with different numbers. Creating a solution for that random input set now only requires an additional step of blindly recalculating a recipe of equations from a solution.

While many useful problem sets are available to educators, the need for instructors to write their own problems or adapt new problems from the existing resources will always be present. If an instructor seeks to do anything other than directly pull from the sources, the old problems of writing, editing, correcting, and solving problems repeats again. The only true way to eliminate the possibility that students can find the solution to a problem online is if each new problem they encounter is truly unique and cannot be found online through a simple cut-and-paste search. Where this work seeks to push the boundary of what is available is in developing a means to take the plethora of problem sets that exist, distill them down into their basic components, and organize that data into a useful format so that it may be used in a series of algorithms that can automatically generate new problem configurations along with problem diagrams, text, and solutions. To do so would be to turn a large repository of problems into a set of all possible permutations of the problem.

In addition, for such a system to be able to rapidly generate new course content, adaptive learning and game based learning are two of the major learning strategies that could provide the most motivation for this work. Adaptive learning usually refers to computer based systems that, in an automated fashion, facilitate learning by customizing the experience for each student. This can be accomplished through adaptive testing, tracking student behaviors, errors, and successes, and choosing new content for students based on their performance [7]. Adaptive learning strategies can also be used for the gamification of course content.

The bottleneck, however, in many adaptive learning systems is the variety of new content paths available to the students based on their decisions. The classic model can be exemplified through adaptive testing which is implemented in many computer-based standardized testing. At its most basic form, a student is assessed on their failure or success at answering a question and based on their performance either problem A or problem B may be chosen for the next assessment. But still each path a student takes and each problem along that path must still come from a static or minimally dynamic (in the case of problems with random inputs) repository of problems [9].

To achieve the algorithms required for automated problem generation, two major artificial-intelligence systems are used: a computer algebra system (CAS) and a natural language generation system (NLG). Computer algebra systems allow programmers to solve problems using symbolic mathematics, as a human would, rather than more traditional solving techniques at which digital computers excel, such as numerical methods or iterative methods [10]. This is necessary for several reasons. First, problems generated randomly, not just with random input values, but with an input parameter list that is random will all have different solution procedures. An algorithm that can generate such a random problem configuration must also be able to solve that configuration. It is impractical for a human to solve all possible permutations of the problem and therefore the automated system must accomplish that task as well. A computer algebra system would allow for automated algorithms to be built that (1) specify each applicable equation as part of the problem data set and (2) choose the proper order and variable by which to solve that set of equations. It is advantageous to create an algorithm that mimics the logic and method that a human would use to solve such problems. This not only allows for the computer to solve the problem itself but also allows for the creation of a solution report that could be written as a series of steps that a student could follow.

Secondly, to automatically create problem sets, it is the problem text that poses the most difficult challenge. Problem parameter lists and values can all be generated with relatively simple algorithms, but creating problem text as if it were written by a human, instructor, or textbook author is a daunting task. Natural language generation refers to computer algorithms capable of converting digital information or data into prose or a text-based representation that appears as if it were written by a human [11]. NLG works best in situations where the underlying data can be organized succinctly and the required text can follow suit. As such, NLG finds its most common use in business or economics where simple text-based or chart-based reports can be generated from sales data or descriptive data [11,12]. For example, advertising descriptions for online home-buying adds can be generated from realtor data. The structural, tonal, and grammatical requirements for such advertisements are low and are thus lend themselves well for human-free generation. NLG is difficult but has been improving in sophistication over the last decade [reference]. Structure and tone vary wilds depending on the implementation of the NLG software.

Human guidance is often required in many systems that purport to generate text from data automatically. Structure of the raw data and the organization of the human lexicon are the two most important factors that affect how much human intervention is necessary to create the desired proficiency of the text [12]. This was found to be true in the present study as well. While the algorithms used in this study have grown in complexity, the need for human intervention in producing final products is ever present and elimination of that weakness continues to be a goal.

In the following study the researcher attempts to produce automatically generated problem sets for adaptive learning systems or game-based systems that can be used in courses in dynamics, thermodynamics and circuits. Six problem types, or archetypes, are chosen, two from each discipline to adapt for this method. For each archetype, the following procedure is followed:

1. All problem parameters and value ranges are listed and organized
2. Problem parameter sets and value ranges are randomized
3. All unknown values are solved with a computer algebra system
4. Problem solutions reports are generated
5. Error checking is performed
6. Problem difficulty is assessed
7. Problem schematics and diagrams are generated
8. Problem text is generated using natural language algorithms

The final output consists of a problem description in text form, a problem diagram, and a text-based solution procedure.

Methodology and Implementation

To generate automated problem sets that are solvable, robust, unique, and correct is a complicated task with numerous steps. The basic procedure that outlines the method for both the representation of each problem type and the automated generation algorithm is shown in Figure X. The first step in automatically generating problem sets is represented by the left-most third of Figure X where the archetypical problem is properly defined.

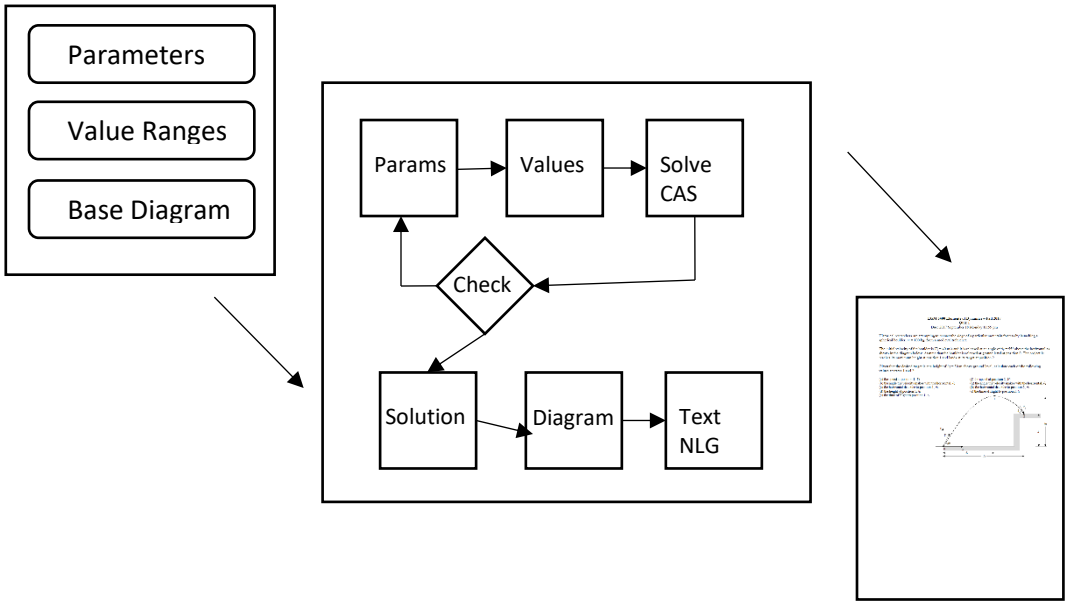

Figure 1: Flow diagram of each step in the automated generation process. Left-most block: codification of all necessary data for a problem archetype. Middle-block flow sub-diagram for randomly generating problem features. Right-most block: output of problem statement with solution

To define the problem archetype is to organize and codify all the necessary problem features into basic building blocks that lend themselves well to randomization and random customization. First, all possible problem parameters, every quantity that may be either given or be asked to find

in the problem needs to be given a variable and listed. Next, all possible practical ranges for each parameter needs to be specified. Next, it may be desired that distances not exceed some maximum limit, or angles not exceed 360°, or some physical constants not be negative. Next, all necessary fundamental equations that relate the specified parameters must be listed and organized into a format that can be manipulated by the chosen CAS. Next, the necessary graphical features must be specified. A library of geometric features must be stored in data structures that can be superimposed on default problem diagrams depending on the outcome of NLG. As a prerequisite for any NLG system is a foundational lexicon from which to pull simple textual elements such as nouns and verbs and more complicated textual elements such as technical engineering jargon. The organization of this data is important for the efficient implementation of the generation algorithms. Thankfully, the problem spaces used here are sufficiently small that even some of the brute-force or "stupid" algorithms created here run so fast that data organization was not so critical. All parameters, equations, and data ranges are stored in string arrays, equation object arrays, and vectors, respectively.

Once the underlying data space has been organized and all problem archetypes have been properly defined, the next step is the sequential generation of each main element of the problem content. First, based on the list of all possible parameters an archetype, the minimum number of problem parameters are specified and a set of that number of parameters are chosen at random from the list. For each random parameter in the list, data values are then chosen. Parameters in the list are chosen based on a uniform probability distribution while the individual parameter values are chosen either from uniform probability distributions or gaussian distributions depending on how much variety is desired for a particular parameter. Each of these decisions are based on elementary algorithms that warrant no elaboration due to their simplicity.

What is not simple, however, is how solutions are achieved through the implementation of a computer algebra system (CAS). All algorithms here are implemented in Python, which was chosen partially for the ease with which the SymPy CAS has been developed for that language. The algorithm for solving the system of each set of equations involved iterative symbolic equation manipulation through SymPy until all required unknowns are found. While SymPy handles the actual equation manipulation, the decision still needed to be made as to what equation would be used to solve for what variable or to identify when two or more equations had to be solved simultaneously. The algorithm logic follows the following priority queue:

1. Make a list of all unknown variables, iterate the following steps for each variable
2. Use SymPy to solve each equation in the list the unknown variable
3. Solve for as many single variables as possible
4. If not done, iterate through all combinations of two remaining variables from the list
5. Find two equations that contain only those two variables, solve using SymPy
6. If not done, iterate through all combinations of three remaining variables from the list
7. Find two equations that contain only those two variables, solve using SymPy

This procedure either found the solution of the random problem configuration or completed with a set of parameters yet uncalculated. Error checking was next performed to assess the outcome of the core given and found data and values. The most common reason why a problem core failed

the series of error tests that were written was because of a partial or full ill-defined system. But this was not always considered a failure. Failure conditions were usually situations where the problem parameter and value pairs chosen randomly produced a scenario that either (A) produced a series of equations that could not be solved or (B) produced values for the parameters to be found that were outside the bounds specified in the problem codification step. Conditions that did not pass the error checking step, but were still considered successes and sent on to continue generation involved primarily scenarios where one or two independent variables could not be calculated. For example, in the projectile motion archetype to be described in the next section, only two locations in the trajectory need be defined out of the three, A, B, and C. Occasionally a variable for location C, say, would be requested when no other information about C was generated in the given parameter list. That single C variable could not be found, but that did not inhibit the generation of the remainder of the problem, human intervention could catch these cases and success was allowed.

With the problem checked for correctness all core parameters and values could then immediately be used to create a problem. If the algorithm stopped at this step, educators could still find it useful as much of the hard work in making sure a problems numbers and configuration are authored properly is a large portion of the work.

The next step in completing the automated problem generation was a determination of difficulty. A simple heuristic was implemented to estimate the probable difficulty each problem configuration would be for the average undergraduate student that simply counted the number of steps generated to produce a solution. If a solution step involved solving systems of two or three sets of equations, rather than one, those steps were weighted higher by a factor of two and three, respectively. Difficulty ratings were then normalized between one and five, five being the most difficult. The simplest way to achieve a gauge of problem difficulty would be to use an ensemble of automatically generated problems. Consider the exercise of allowing the algorithm to generate a large (say 1000) number of problems of one archetype and then compare them or sort them using criteria such as the length of solution, whether certain solution steps are present (such as solving a quadratic equation), or whether certain equations are implied based on problem text. Sorting the ensemble in this way can then at least order the numerous problem sets based on difficulty and clear lines or marks of delineation between different difficulty ratings can by chosen the researcher.

The last tasks were to generate the actual problem statement and diagrams. To tackle the diagrams, a 2D Python graphics library, PyGame was used. PyGame is simple to use and fairly extensive which allowed for the creation of simple 2D graphical artifacts to build problem diagrams. As a start, simplicity was favored over elaboration. The most visually appealing graphics output requires large amounts of time and art content that simple geometric primitives which were limited to boxes, lines, circles, text, arrows, and others were used. Rudimentary diagrams based on simple graphical primitives would suffice initially and lead the way to more sophisticated outputs as development continued. A sequence-based procedure was used to create each diagram using black primitives on plain, white backgrounds. Lines, rectangles, and parabolas of predefined coordinates were drawn in pixel space using PyGame’s drawing tools

and saved as a single frame as a PNG file. That PNG file can then be added automatically to the final output image file.

Problem diagrams are often, at a glance, the strongest visual indicator of the similarity or difference of two comparable problems. Varying the visual styles, such as line width, colors, primitive styles, and subtle features can make two problems with similar input values and similar solution procedures seem quite very different.

The final step was the most complicated step and also the one that required the most development time for the smallest gains and that was the automated generation of problem text. An open source Java API was used for NLG named simplenlg. Simplenlg was sufficiently robust to be able to handle all desired functions, with one small disadvantage. Both the native lexicon of vocabulary and the most commonly used extended lexicon found difficulty producing some of the jargon used in typical thermodynamics problems. For example, the term “polytropic” was not found in either lexicon and was deemed to be a fairly important term in the development of the ideal gas, piston-cylinder archetype detailed in the next section. Two approaches were proposed to remedy this problem: (1) augment the API’s lexicon to handle usage of additional vocabulary or (2) find suitable, less technical replacements for common terms. The latter option was by far the simplest and was used here.

Problem Archetypes

Two-dimensional projectile motion is a classic problem used in most introductory physics classes and sees additional coverage in most undergraduate mechanical dynamics courses, usually within the first week. The problem is simple enough in its nature, but rather complex solution procedures can be required of students that many permutations of the problem are not only non-trivial, but prove to be an invaluable tool in teaching problem solving techniques, the solution of simultaneous equations, the proper way to set-up and organize engineering problems, and visualizing motion and vectors in space.

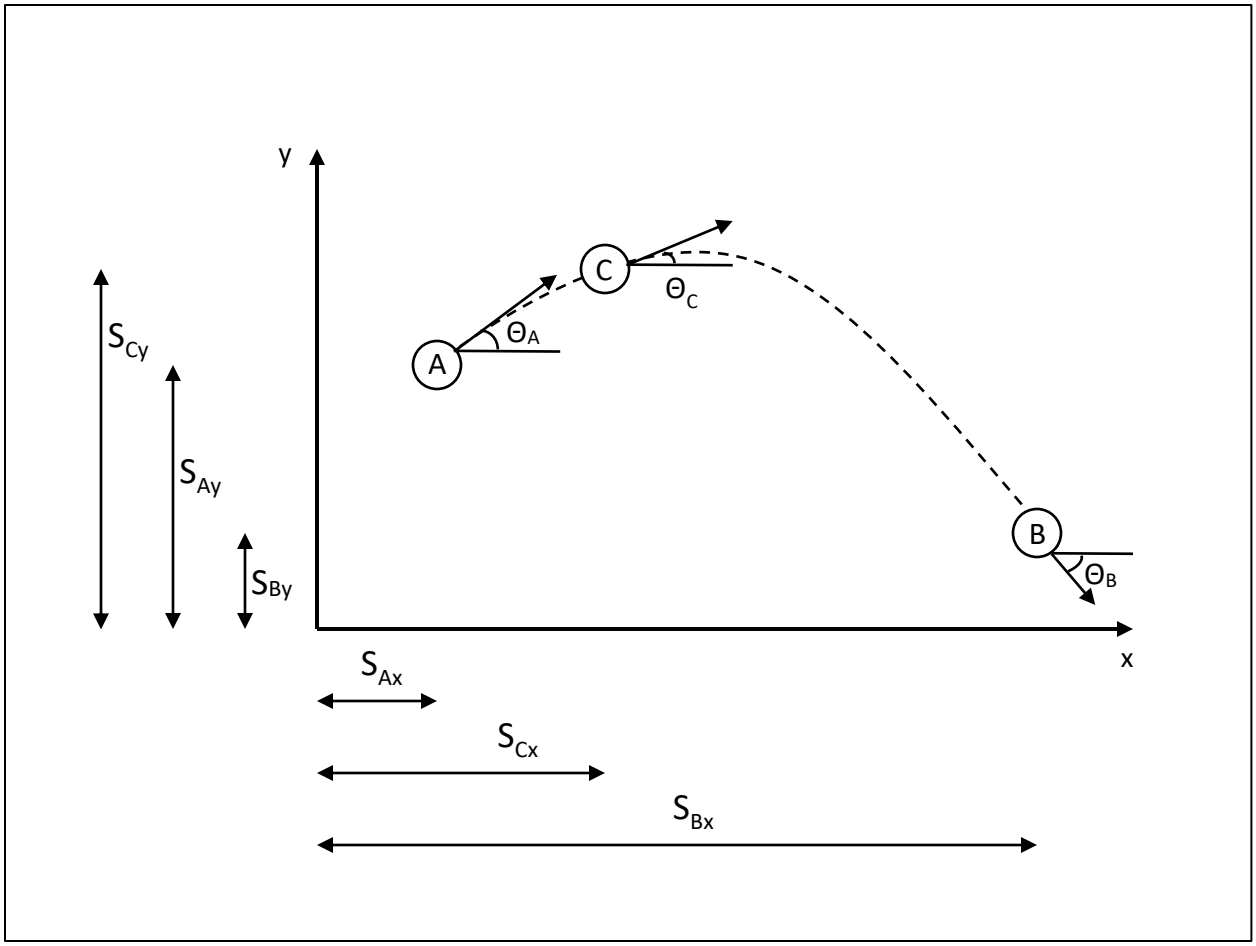

Figure 2: Projectile motion archetype

Projectile motion is confined here to two-dimensions for simplicity as not much more fundamental complexity is gained from including a third dimension to the particle's motion. The motion of the particle is constrained to have constant velocity and solutions are usually found using three classic kinematics equations. The problem description below shows three points of interest in the trajectory, A, B, and C. Each point can have specified its position, *S*, or velocity, *V*. Since acceleration in projectile motion is constant, only two constant values, the *x*- and *y*-components, are necessary to define the acceleration at all points along the trajectory. If additional variables for the time of flight between each point, the maximum height, *H*, and range, *R* of the trajectory are included the problem can be fully defined in terms of 25 total parameters and 18 total equations requiring a total of 7 problem values to be defined in the problem statement in the most difficult scenario. Automatically generated problems that do not ask for the maximum number of unknowns may obviously require a smaller set of equations to achieve solution. Simpler problems may also define redundant values or values in the form of contextual clues that then require fewer numerical values to be explicitly stated. Stating that a particular projectile is launched on the surface of the earth (or even omitting this fact, as it is often implied) then implies the acceleration of the particle in the y-direction is that of gravity and is therefore assumed.

Parameter Lists	Description
S_{Ax}, S_{Bx}, S_{Cx}, S_{Ay}, S_{By}, S_{Cy},	x- and y-coordinate positions of three points of interest: the initial location, A, the final location, B, and a third intermediate location, C, that may be included in the problem
V_A, V_B, V_C, θ_A, θ_B, θ_C, V_{Ax}, V_{Bx}, V_{Cx}, V_{Ay}, V_{By}, V_{Cy},	Velocity of each point A, B, and C along with the angle each velocity makes with the horizontal and the x- and y-components of those velocities
a_x, a_y	The x- and y-components of acceleration (these are included as variables for problems that might have an orientation angled with gravity or if the problem takes place on another planet)
t_{AB}, t_{AC}, t_{BC}	The time of flight from A to B, A to C, and B to C, respectively.
H, R	The maximum height and range of the projectiles motion.

Equations:

$$S_0 = S_f + V_0 t + \frac{1}{2} a t^2 \text{ (for each interval in each coordinate, 4 eq's)}$$

$$V_f^2 = V_0^2 + 2a\left(S_f - S_0\right) \text{ (for each interval in each coordinate, 4 eq's)}$$

$$V_f = V_0 + at \text{ (for each interval in each coordinate, 4 eq's)}$$

$$V_x = V\cos\theta \text{ (for each point)}$$

$$V_y = V\sin\theta \text{ (for each point)}$$

Table 1: Parameters and equations that fully define the projectile motion archetype problem. There are 25 total parameters and 18 total equations.

An ideal gas in a piston-cylinder device undergoing a two-step process is a problem encountered usually within the first third of an undergraduate course on thermodynamics. Each state can be specified by some combination of the thermodynamic properties of pressure, *P*, volume, *V*, and temperature, *T*. The device can be drawn and described as either driven by a work-based process or a heat-based process. Either way, the governing equations amount to a single ideal-gas equation of state, a single expression of the first law of thermodynamics, and definitions for density and internal energy.

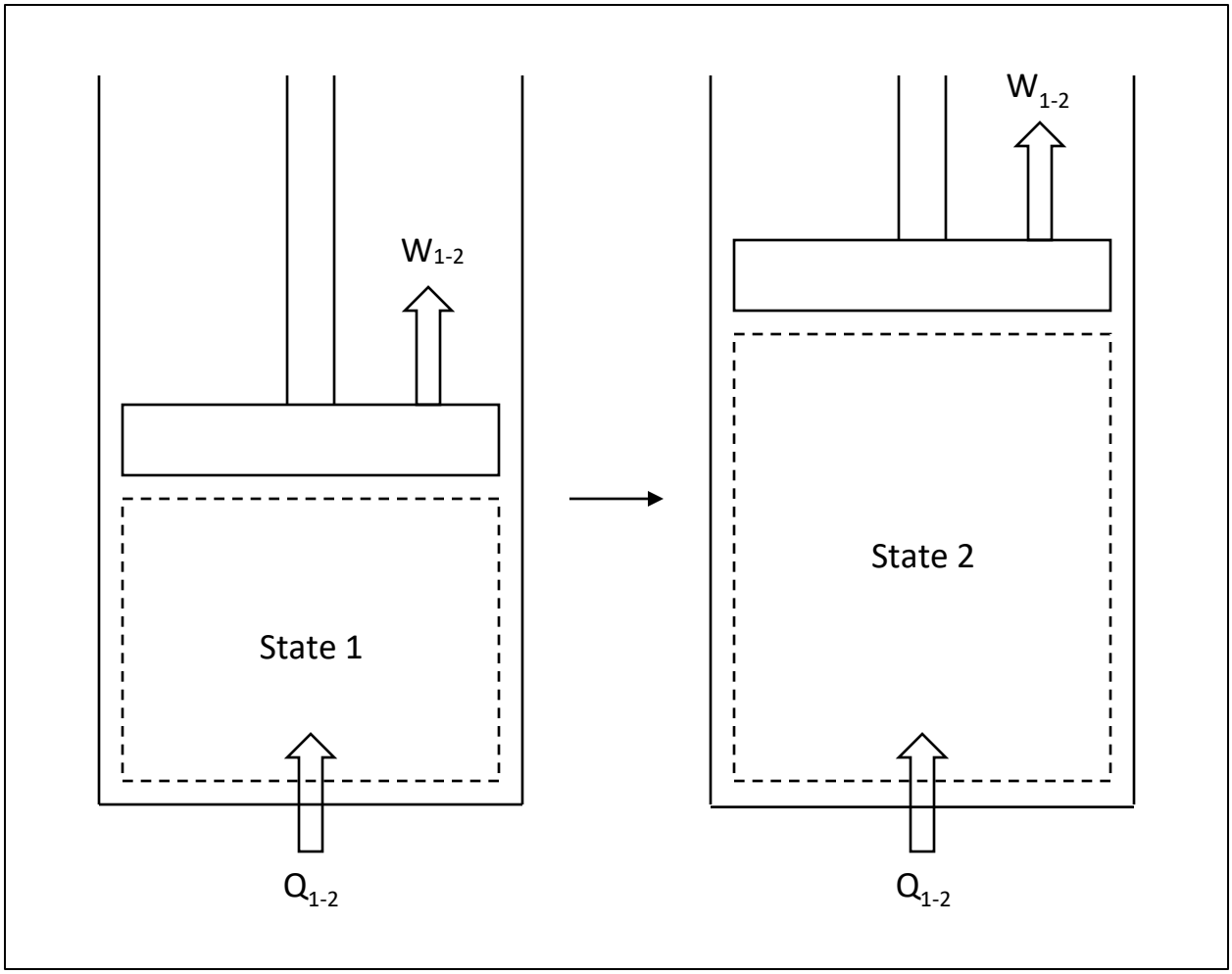

Figure 3: Ideal gas in a piston-cylinder device archetype

This problem makes a great candidate for inclusion in the study as its governing equations are quite simple and the possible number of permutations available for possible problem configurations are vast. Thermodynamic properties of pressure, volume, and temperature as well as all of the other parameters listed make great candidates for either input parameters or values to be found. The possibility for unique problem configurations is vast as the gas identity as specified by the gas constant, R, and the specific heat capacity, c_p, can also either be given or be a value to be found. The problem can be fully defined with 15 parameters and 10 equations.

Parameter Lists	Description
P_1, V_1, T_1, P_2, V_2, T_2	Pressure, volume, and temperature at state 1 and state 2
m, ρ_1, ρ_2, v_1, v_2,	Constant mass, the density, and specific volume at state 1 and state 2
W_{12}, Q_{12}, $\Delta U_{1\text{-}2}$, $\Delta u_{1\text{-}2}$	Work done, heat transferred, and the change in internal energy and specific

	internal energy in a process from state 1 to state 2
c_v, c_p, R	Constant specific heat capacities and ideal gas constant of the substance
Equation List: $$P_1V_1 = mRT_1$$ $$P_2V_2 = mRT_2$$ $$V_1 = mv_1$$ $$V_2 = mv_2$$ $$\rho_1 = \frac{1}{v_1}$$ $$\rho_2 = \frac{1}{v_2}$$ $$\Delta U_{12} = m\Delta u_{12}$$ $$\Delta u_{12} = c_v\Delta T_{12}$$ $$\Delta T_{12} = T_2 - T_1$$ $$Q_{12} - W_{12} = \Delta U_{12}$$	

Table 2: Parameters and equations that fully define the ideal gas problem. There are 15 total parameters and 10 total equations.

Equivalent resistance problems are another example of a simple problem occurring in all undergraduate circuits classes and often near the beginning of the course. The challenge with equivalent resistance problems is not in the generation of textual problem statements as some variation of “Find the equivalent resistance of the following circuit segment” is usually a sufficient instruction. Rather, the difficulty in automatically creating equivalent circuit problems is randomly generating the diagram and the solution procedure.

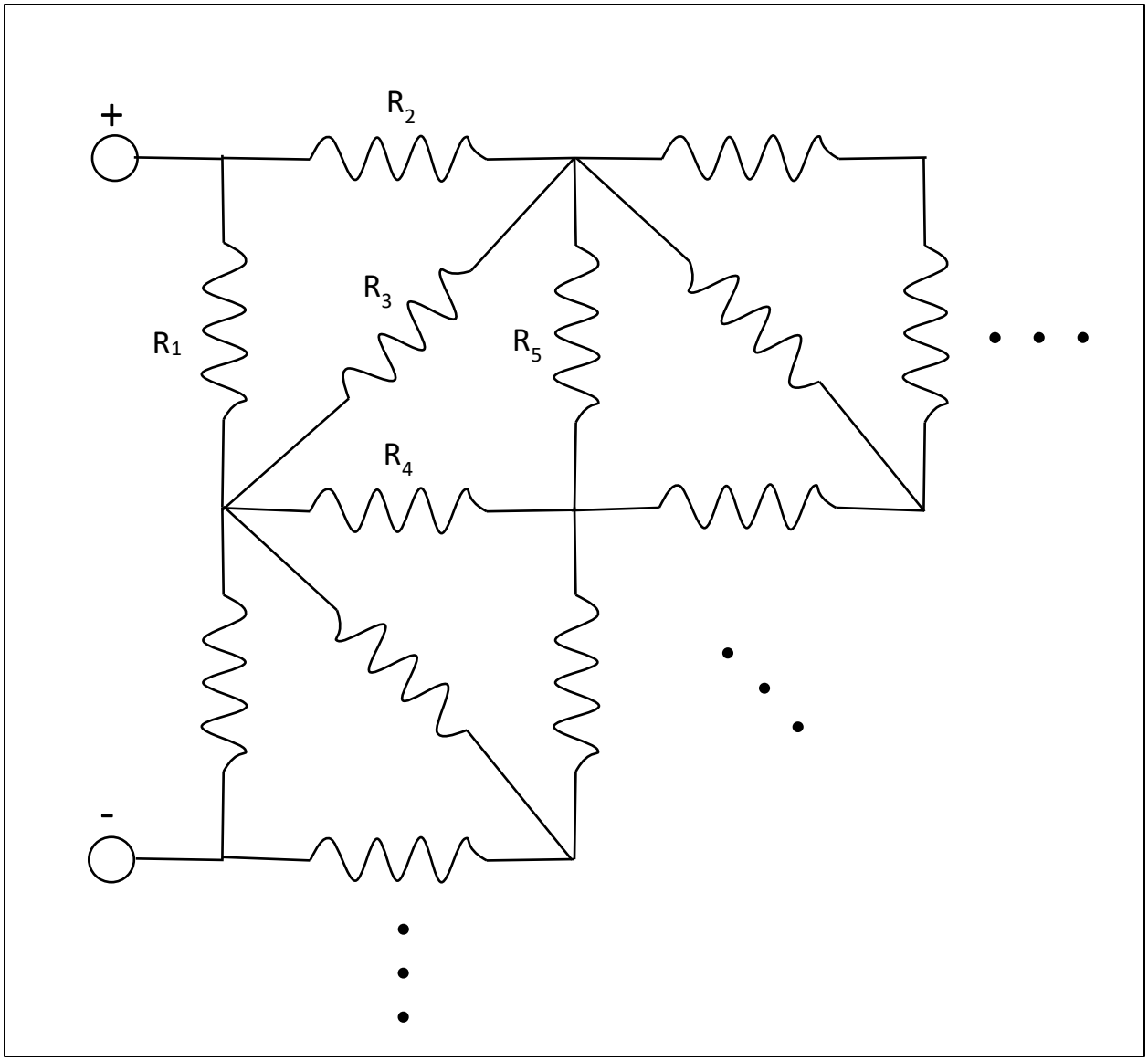

Figure 4: Equivalent resistance archetype

The solution procedure, implemented in such a way that a student can most easily follow, involves collapsing two or more resisters into either series or parallel equivalent resistors until there is only one left. There are multiple ways to do this and each substitution must be clear to the student. Beyond this difficulty, generating problem diagrams based on the characteristic set of cells in the diagram below is straightforward. The size of the circuit diagram, the total number of resisters, N, is the major factor in determining complexity and is itself a parameter in the problem. The problem is fully defined by N+2 parameters and N-1 equations.

Parameter Lists	Description
R_1, R_2, R_3, …	Resistance value of each resistor present

R_{eq}, N	The equivalent resistance and the total number of resistors, respectively
Equation List: (N-1 equations of the following types, depending on layout) $$R_{eq} = R_1 + R_2 \text{ (series)}$$ $$\frac{1}{R_{eq}} = \frac{1}{R_1} + \frac{1}{R_2} \text{ (parallel)}$$	

Table 1: Parameters and equations that fully define the equivalent resistance problem. There are N+2 total parameters and N-1 total equations.

Results and Analysis

The three archetypical problems detailed in the previous section were implemented in Python using the SymPy computer algebra system, the simplenlg Java natural language generation system, and the PyGame 2D graphics engine. Only data for the second problem archetype, the ideal gas, piston-cylinder problem, is shown for brevity.

In figure 5 the input file is shown that includes most of the necessary data that defines the archetype. The parameter “P-2digpc” in the third line is a parameter that tells the system that the type of problem to be generated is of the ideal gas, piston cylinder archetype and to include several default values for parameter ranges.

```
## Input File:
## 2D ideal gas piston-cylinder archetype

"P-2digpc"

## Parameter List

P1,V1,T1,P2,V2,T2,m,rho1,rho2,W,Q,U1,U2,v1,v2,
w,q,u1,u2,deltaU,deltau,deltaT,deltaP,deltaV,d
eltav,cv,cp,R

## Equation List

[P1*V1=m*R*T1]
[P2*V2=m*R*T2]
[V1=m*v1]
[V2=m*v2]
[W=m*w]
[Q=m*q]
[rho1=1/v1]
[rho2=1/v2]
[U1=m*u1]
[U2=m*u2]
[deltaU=U2-U1]
[deltaU=m*deltau]
[deltaT=T2-T1]
[deltaP=P2-P1]
[deltaV=V2-V1]
[deltaV=m*deltav]
[deltaU=cp*deltaT]
[Q-W=deltaU]

## Constraints

"isometric process"
```

Figure 5: The input file used for the automatic generation of problems based on the ideal gas, piston-cylinder archetype.

Each parameter to be included in the possible generation of the problem statement is included and each equation relating the parameters is listed. The final item specifies keywords that simplify the problem generation process, which can also be randomized. The “isometric process” keyword specifies two additional equations to be added to the list, namely that the two volume values are the same and that the work done in the process is zero.

```
Given: P1,T1,m,P2,cv,R
Given: 100, 467, 2.3, 150, 1.0, 0.2870

Find: V1,V2,T2,rho1,W,Q,deltaU

Difficulty: 4

Solution:

1. [P1*V1=m*R*T1]
2. [V1=m*R*T1/P1]
3. [V1=V2]
4. [V2=V1]
5. [P2*V2=m*R*T2]
6. [T2=P2*V2/m/R]
7. [V1=m*v1]
8. [v1=V1/m]
9. [rho1=1/v1]
10. [W=0]
11. [deltaT=T2-T1]
12. [deltaU=cv*deltaT]
13. [Q-W=deltaU]
14. [Q=deltaU+W]

Consider an ideal gas in the piston-cylinder
device in the picture. If the gas is air and
the process is isometric process and P1=100
and T1=467 and m=2.3 and P2=150 find V1 V2 T2
rho1 W Q deltau. The process is "isometric
process".
```

Figure 6: The text-based output file after all automated generation is complete.

Figure 6 shows one output file the from automated problem generation step. The first two lines specify the randomized parameter list that was chosen by the algorithm and the values for each parameter chosen based on default parameter ranges. A subset of the available parameter that could possibly be found was chosen randomly. The problem shown here as output is a typical problem that one might find in an undergraduate thermodynamics textbook. The algorithm was run numerous times to produce an output product that was a practical mimic to a typical textbook problem.

The solution procedure, shown also in Figure 6, is detailed in a step-by-step linear fashion. Each variable in the list of parameters to find is find in order with the notable exception of Q and

deltaU as the algorithm is smart enough to know that the final value must be found before the penultimate. The last paragraph is that text that is output from the current implementation of simplenlg. There are several noticeable errors in the problem description, but with a small amount of editing, the text is almost immediately viable for inclusion in a course assignment.

While Figure 6 shows the raw output from the software in text form, Figure 7 shows the final formatted version the software is capable of generating, still without human involvement. The problem description fits one of several predefined generic sentence structures augmented to include all the necessary information the algorithm dictates. The picture included in the problem statement is generated using simple drawing tools such as line, rectangle, and textbox primitives from PyGame. Note that the process in the example is isometric, or constant volume. The diagram reflects this fact in that the position of the piston has not risen or fallen. As the problem parameters change, that affects not only the data values shown in the figure but also the position of the piston accordingly.

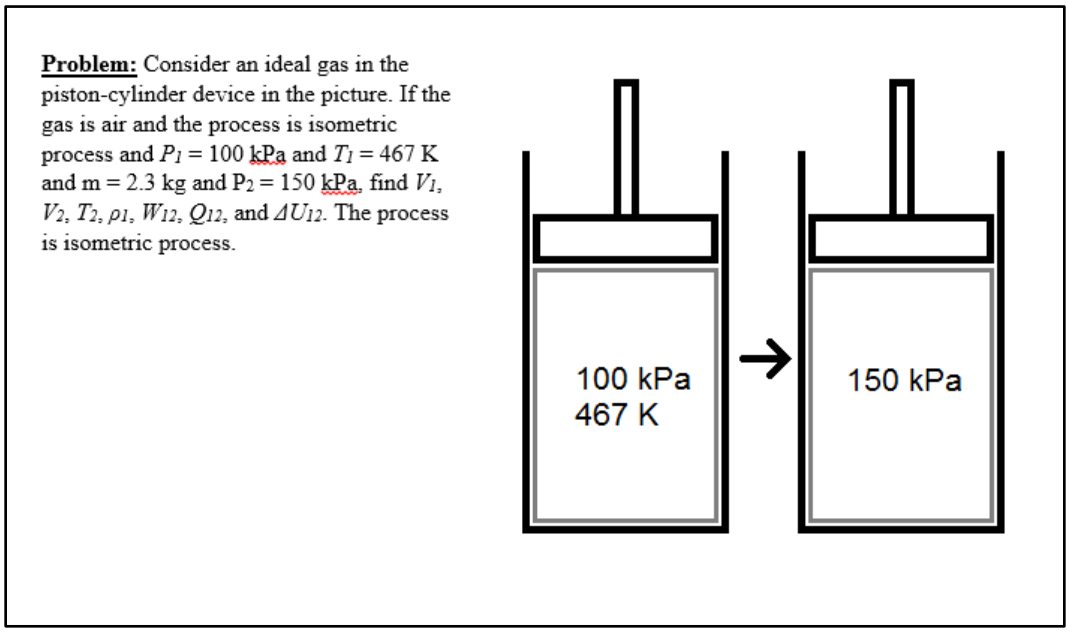

Figure 7: Formatted output of problem description

As was mentioned previously, rarely does automatically generated text not require some amount of human intervention in the form of additional post-processing to correct grammar mistakes, to possibly correct formatting, or to add a human flair or poetry to otherwise mechanical prose.

Conclusions and Recommendations for Future Work

The method outlined in this study provides a framework for the creation of algorithms capable of automatically generating problem sets that can be used to rapidly produce new class content. Automatically generated class content lends itself well for use in adaptive learning systems, game-based learning systems, or even creating traditional course materials such as homework problems, quiz problems, and test problems. For an automated problem generation algorithm to be considered successful it must produce a problem that is solvable and unique that includes all

necessary input parameters, parameter values, a text-based problem description, the necessary problem diagrams, and a problem solution that may be easily followed.

This study has sought to show that such algorithms are possible and that a general procedure for applying those algorithms to an ever-expanding list of new problems can be practical. Several challenges were encountered in this study that require further investigation and refinement. First, using a computer algebra system for the solution of randomly generated problem sets that are well-defined by a succinct set of equations is difficult and time consuming to refine. While the researcher found that it is relatively trivial to create an algorithm that finds one solution to the problem, representing that solution as a logical procedure automatically is often elusive. Moreover, solution procedures that employ methods of direct substitution are most easily found while more elegant solutions that involve creative manipulations of equations are more difficult.

Second, generating problem text using NLG is an exceedingly difficult endeavor. Overall text structure, logical sentence flow, and the inclusion of all necessary contextual elements are all easy to accomplish and can be generated rapidly using available libraries. Generating problem text that does not require some level of human involvement, however, is not practical. This difficulty is to be understood to be most prescient and therefore should be expected by all who seek to accomplish automated problem generation. With that in mind any level of automated generation below that of full human-free text creation can still be viewed as a success as most of the work and time commitment in writing new problem sets can be completed by computer. Typical human-required refinements involve completing sentence structure such as proper noun-verb agreement, including proper technical jargon, and simply making sentences "sound human." Similarly, generating problem diagrams is a difficulty that extends from that of creating problem text. While all necessary graphical elements can be included easily, just as all necessary nouns can be included easily in text, different portions of problem diagrams require human intervention to be "stitched" together properly.

Lastly, determining a heuristic for most accurately evaluating problem difficulty is an elusive prospect that requires further testing. Rudimentary assessment algorithms were employed here, but as problem complexity increases so does the complexity of the nuances that determine how difficult the problem appears to students. Simple algorithms that count the number of steps to a problem solution or the relative amount of time require to achieve solution are used here but a robust determining of problem difficulty will require testing a large set of automatically generated problems against actual student performance. Assessments must also be made as to how difficult students perceived each problem type or each problem permutation separately from the failure or success of the student to solve each problem. The purpose of automated problem sets is to provide students with a multitude of problems through which they can practice and hone their skills. Student progress and how that effects the assessment of problem difficulty can be simultaneously studied.

Additional future work must be done on an ever-expanding set of new problem archetypes. The next candidates for investigation are (1) dynamics: oblique impact of particles, (2) thermodynamics: simple forward and reverse heat engine representations, and (3) circuits: Kirchoff's current law. While only one problem archetype from each course has been studied in

this work and only one more per course has been suggested here, there are obviously multitudes more from each course that must be investigated, codified and implemented for a successful set of course modules to be complete.

In addition, it must also be noted that each of the presented archetypes have a maximum achievable difficulty. For example, two-dimensional projectile motion problems are only so difficult that a relatively small number of problem permutations need be practiced by students for sufficient mastery to be achieved. A simple way to increase complexity, then, would be to augment the automated generation algorithms to include multiple stacked projectile motion problems. A problem may include two projectiles thrown at different times all from a similar location, or perhaps two projectiles thrown from the same location at the same time to land at different final positions. A further difficulty may be creating a problem where two projectiles are launched in such a way that mid-air collision is achieved. The compound nature of these new problem sets has the potential for the unlimited increase in problem difficulty with minimal augmentation of the presented algorithms.

References

1. Felder, R. & Brent, R. (2016) *Teaching and Learning STEM: A Practical Guide*. San Francisco, CA: John Wiley & Sons, Inc.
2. Michael, T. & Williams, M. (2013) Student Equity: Discouraging Cheating in Online Courses. *Administrative Issues Journal: Education, Practice, and Research*, 3(2), 1-12.
3. Muilenburg, L. & Berge, Z. (2007) Student Barriers to Online Learning: A Factor Analytic Study. *Distance Education*, 26:1, 29-48.
4. Lavieri, E. D., Jr. (2014) *A Study of Adaptive Learning for Educational Game Design*, ProQuest Dissertations Publishing (Order No. 3628679).
5. Murray, M. C., & Pérez, J. (2015). *Informing and performing: A study comparing adaptive learning to traditional learning*. Informing Science: the International Journal of an Emerging Transdiscipline, 18, 111-125.
6. Walberg, H. J., Paschal, R. A.,& Weinstein, T., (1985) *Homework's Powerful Effects on Learning*, Educational Leadership, 85 (42):76-79.
7. Swanbom M. K. Moller D. W., Evans K., (2016) *Open-Source, Online Homework for Statics and Mechanics of Materials Using WeBWorK: Assessment of Student Learning*. Proceedings of the 2016 American Society for Engineering Education Annual Conference & Exposition. Paper ID 16092
8. Chew, K., Chen, H., Rieken, B., Turpin, A., & Sheppard, S., (2016) *Improving Students' Learning in Statics Skills: Using Homework and Exam Wrappers to Strengthen Self-Regulated Learning*. Proceedings of the 2016 American Society for Engineering Education Annual Conference & Exposition. Paper ID 15770
9. Roberts, M., Curras, C., & Parker, P. (2006) A Homework Problem Database: Design and Implementation: Proceedings of the 2006 American Soceity for Engineering Education Annual Conference & Exposition. 11.53.1 – 11.53.9
10. Cohen, J. (2003) *Computer Algebra and Symbolic Computation*. Natick, Massachusetts: A K Peters, Ltd.

11. Bird, S., Klein, E. & Loper, E. (2009) Natural Language Processing with Python. Sebastopol, CA: O'Reilly.
12. Reiter, E. (2010). Natural Language Generation. In The Handbook of Computational Linguistics and Natural Language Processing (eds A. Clark, C. Fox and S. Lappin).