12-2020

Information Theory Problem Description Parser

Gary Brent Hurst
University of Tennessee, Knoxville, tmn678@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Gary Brent Hurst entitled "Information Theory Problem Description Parser." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

James S. Plank, Major Professor

We have read this thesis and recommend its acceptance:

James S. Plank, Chao Tian, Michael R. Jantz

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Information Theory Problem Description Parser

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Gary Brent Hurst

December 2020

Abstract

Data corruption and data loss create huge problems when they occur, so naturally safeguards are usually in place to recover lost data. This often involves allowing less space for data in order to allow space for an encoding that can be used to recover any data that might be lost. The question arises, then, about how to most efficiently implement these safeguards with respect to storage, network bandwidth, or some linear combination of those two things. This work has two main goals for the information theory community: to produce an intuitive-to-use problem description parser that facilitates research in the area, and to demonstrate the parser's utility.

To these ends, I completed three objectives: First, I hardened an existing problem description parser to sanitize input. Then, after that code was released open-source to the community, I rewrote the parser using C++, making it more efficient and more agreeable to a Python-based workflow. Once this new parser had also been open-sourced, I wrote a program to generate problem description files to show how to use the parser and how useful it can be. This paper will give an account of the results of those three objectives.

Table of Contents

1 Introduction **1**

2 Objective **1**

3 Background: Entropy **1**

4 CAI **2**

5 Mission 1: Hardening CAI v0.1 **2**

6 Mission 2: New Version of CAI **2**
6.1 Structure of v0.1 . . . 2
6.2 Structure of v1.0 . . . 4
6.3 Speed-Up from v0.1 to v1.0 . . . 4
6.4 Sending Data Structure to the Linear Solvers . . . 5
6.5 CAI v1.0 User Guide . . . 5
6.5.1 `RV`: Random Variables . . . 5
6.5.2 `AL`: Additional Linear Programming Variables . . . 6
6.5.3 `O`: Objective Function . . . 6
6.5.4 `D`: Dependencies . . . 6
6.5.5 `I`: Independences . . . 7
6.5.6 `S`: Symmetries . . . 7
6.5.7 `BC`: Constant Bounds . . . 7
6.5.8 `BP`: Bounds to Prove . . . 7
6.5.9 `CMD` and `OPT`: Commands and Options . . . 8
6.5.10 Commands Other Than `PD` . . . 8
6.5.11 Parser Executable . . . 8
6.5.12 Linking Executables . . . 9
6.5.13 Command-Line Modifiers . . . 9
6.5.14 Compatability for Windows Command Prompt Users . . . 10

7 Mission 3: Demonstrate CAI's Utility **10**
7.1 Problem 1: RAID-6 . . . 10
7.1.1 Problem Overview . . . 10
7.1.2 How to Distill This Down for CAI . . . 10
7.1.3 Problem Description File . . . 12
7.1.4 Solution . . . 12
7.2 Problem 2: Regenerating Codes . . . 12
7.2.1 Problem Overview . . . 12
7.2.2 How to Distill This Down for CAI . . . 13
7.2.3 Problem Description File . . . 15
7.2.4 Performance . . . 16
7.3 Run-Time Complexity . . . 16

8 Conclusion **16**

References **18**

Vita **19**

1 Introduction

Modern computing often requires data to be stored across multiple disks, many of which might not be physically together and thus only mutually accessible through a network. The question, then, becomes how to handle disk failure. If no guard is in place to protect data from disk failures, then a disk failing means that data has been permanently lost. In response to this, the field of information theory provides clever ways to efficiently include some redundancy to allow for the recovery of a failed disk's data [4]. Each design decision, though, comes with trade-offs. For example, as will be discussed in more detail in Section 7.2 on a class of codes called regenerating codes, one might want to minimize the amount of data that each disk needs to store, but that might come at the expense of increasing how much data each disk needs to send over a network in order to recover lost data. This being the case, one can construct a system of linear equations representing bounds, which can then be solved to show the bounds for the minimum for disk storage, for the amount of data necessary to send, and for linear combinations of those things. Doing this by hand is difficult and very tedious, so Dr. Chao Tian developed a toolkit called CAI ("Computer-Aided Investigation") [7] which automates it, facilitating research in this area. The overarching goals of this work are to make CAI ready for release to the community in such a way that it fits well into a Python-based workflow, and then to show how CAI can be used as an aid in this field of study.

2 Objective

The objective of this work was three-fold. My first goal (discussed in more detail in Section 5) was to work on CAI v0.1 in order to ready it for open-source release. CAI v0.1 is written entirely in C and it originally did work for most correct input files; the main issue that I was tasked with was ensuring that errors in input files were caught by the software so that, if there were errors, the user would know what the issue was and, more importantly, so that the software wouldn't send bad information to a linear solver, causing the user to unknowingly receive back bad data, which clearly could have unfortunate consequences.

After hardening the toolkit for its initial release as v0.1, my next and most strenuous goal (detailed in Section 6) was to restructure the code to be more efficient, more user-friendly, less error-prone, and, ultimately, to work towards enabling a Python-based workflow and ensuring that the code maintained high levels of portability and interoperability. In v0.1, the format of the input files was intuitive to read and not difficult to produce, but it wasn't quite as efficient to produce, modify, and parse as other methods. For the newly-restructured release, v1.0, we modified the input format to use JSON, a widely-used data format that's simple to work with and much quicker to parse than the input format of v0.1. To confirm that v1.0 correctly matched v0.1, I wrote code to convert to the old data structures.

Finally, with the CAI v1.0 toolkit up and running, we wanted to show its utility by using it to solve some information theory problems. In Section 7, we assess the complexity of the code and its performance, and we show why we claim CAI to be a useful tool in the arsenal of information theorists.

3 Background: Entropy

Before moving on, it is necessary to present a simple discussion of entropy in information theory. At a very basic level, entropy can be described as a measure of the "size" of a random variable, or how much information a random variable can store. For example, the entropy of a byte is 8, because there are 8 bits that can take different values. If, however, 7 of the bits in the byte can be anything while the last bit is used for some sort of parity or redundancy, then the entropy of the byte is 7 because the eighth bit adds no new information. The entropy of a disk holding x bits is x. For notation, if the entropy of a disk D holding x bits is x, we write $H(D) = x$. To denote the entropy of a disk D given a disk E, we write $H(D|E)$.

4 CAI

CAI is a software toolkit originally developed by Dr. Chao Tian [7] that takes a description of an information theory problem, converts it into its corresponding system of linear equations, sends those equations to a linear solver, and then prints out to the user whatever data point the user wanted to minimize (generally bandwidth, storage per node, or something similar). Its inputs are data points describing the problem, such as random variables (usually standing in for nodes or disks), bounds on the entropies of those variables, dependencies between those variables, and the function to be minimized. (A full account of the inputs will be discussed in later Sections.) CAI's outputs are the bounds on whatever random variable entropy property the user noted to be minimized.

5 Mission 1: Hardening CAI v0.1

Before I was added to this project, the original version of CAI had been partially written in C, with error-checking the input being the main part of the code that was lacking at the time. I was tasked with ensuring that if there was an error in the input file, the program would inform the user what the problem was and would exit gracefully; the biggest concern was to ensure that no error in the input file was overlooked, which could have caused a false result to be returned from the linear solvers. I first made a few modifications to make the code Unix and OSx compatible (since it was originally written for Windows in Visual Studio), and then over the span of a few months I spent time going through the code and hardening it, and I created over a hundred test files to ensure that all types of errors and edge cases were being caught and handled appropriately.

6 Mission 2: New Version of CAI

As mentioned above, my most involved goal was to rewrite the parser in C++ to utilize classes and JSON to make the code easier to work with and faster in parsing. The effects of the rewrite are demonstrated most strikingly in that the new version parses a real-world input file in under a second, while the old version takes 55 seconds on the same file. This will be detailed below.

6.1 Structure of v0.1

The original version of CAI was written in C and so made use of many pointers and double pointers to represent lists of random variables, bounds, etc. Input files were set up very intuitively, but reading the file necessitated two passes through the file itself and many linear searches through `int*` and `char**`. This was an unfortunate side effect of not having access to either custom classes or to the C++ Standard Template Library. The problem description struct was laid out very intelligently, as shown below with comments, but one can see that searching through each list will have to be done in linear time.

```
// problem_description struct
typedef struct
{

// Random Variables
int num_rvs;
char** list_rvs;

// Additional Linear Programming Variables
int num_add_LPvars;
char** list_add_LPvars;

// Objective Function
// list_pos, list_value are size num_items.
// list_pos[i] and list_value[i] correspond.
// list_value[i]: coefficient of list_pos[i].
// for all i < (1 << numrvs), list_pos[i]
// represents H(set shown by the bits of i).
//   Coeff*H(setA|setB) is represented as
//   Coeff*H(setA U setB) - Coeff*H(setB).
// for all i >= (1 << numrvs), i represents AL
// with index (i - (1 << numrvs)).
// num_items is the number of distinct sets and
// ALs ended up with.
int num_items_in_objective;
unsigned int* list_pos_in_objective;
double* list_value_in_objective;

// Dependencies
// list_dependency: array, size 2*num_dependency.
// For each dependency, [2*i] is a bit list of
// all vars in Dependent_Vars and Given_Vars,
// and [2*i+1] is a bit list of all vars in
// Given_Vars only.
int num_dependency;
int* list_dependency;

// Independences
// list_independence, list_type_independence, and
// list_num_items_in_independence are all
// size num_independence.
// list_independence[i] has size
// list_num_items_in_independence[i].
//   if type is (int)0, size is n+1.
//     1st n are (1<<index) of corresponding var.
//     list[n] is all those ORd together.
//   if type is (int)1, size is n+2.
//     1st n are (1<<index) of corresponding var
//     ORd together with list[n].
//     list[n] is (1<<index) for all "given"s.
//     list[n+1] is all those ORd together.
int num_independence;
unsigned int** list_independence;
char* list_type_independence;
int* list_num_items_in_independence;

// Constant Bounds
// All 5 lists are size num_constantbounds
// numitems: as in the objective function.
// list_pos: as in the objective function.
// list_value: as in the objective function.
// list_type: (int)0 for >=, (int)2 for =.
//     all <= changed to >= with negation.
// list_rhs is a constant.
int num_constantbounds;
int* list_num_items_in_constantbounds;
unsigned int** list_pos_constantbounds;
double** list_value_constantbounds;
char* list_type_constantbounds;
double* list_rhs;

// Bounds to Prove
// Variables used as in constant bounds.
int num_bounds;
int* list_num_items_in_bounds;
unsigned int** list_pos_bounds;
double** list_value_bounds;
char* list_type_bounds;
double* list_rhs_2prove;

// Symmetries
// Each row is a permutation of indices.
int num_symmetrymap;
char** list_symmetry;

} problem_description;
```

6.2 Structure of v1.0

In contrast, v1.0 makes use of classes and the STL. Random variables are stored in a class `Variable` and one of its members is `std::string Name`. The set of all random variables is stored in two ways in the `Problem_Description` class:

```
std::vector <Variable*>            All_Variables_V;
std::map <std::string, Variable*>  All_Variables_M;
```

The vector is useful for fast iteration, and with the map containing pointers to instances of the `Variable` class with the actual name of the random variable as a key, lookup code is more efficient.

The following classes exist in v1.0 of CAI:

`Variable`	`Dependency`	`Symmetry`
`Term`	`Independence`	`Cmd_Line_Modifier`
`Expression`	`Bound`	`Problem_Description`

Within `Problem_Description` we store instances of all other classes as vectors or maps (or both) of pointers. As already mentioned, this significantly aids with efficiency.

We also have a struct named `Problem_Description_C_Style` which, as the name implies, is identical to the v0.1 struct (with a few additions, but no modifications to what was already there). This is used for two purposes. First, we originally used this struct to check the correctness of v1.0 by giving both versions input files and printing out the values of every variable and running a `diff`. Second, this struct is also used to actually send data to the linear solvers, as discussed further in Subsection 6.4 below.

6.3 Speed-Up from v0.1 to v1.0

As previously mentioned, we worked with a problem description (`PD_PIR4x2`) that v0.1 required around 55 seconds to read while v1.0 could read it in well under a second. The following is a description of `PD_PIR4x2`:

23 random variables	1 independence
2 additional linear programming variables	568 constant bounds
2 terms in the objective function	48 permutations in the symmetry relation
12 dependencies	1 bound to prove

Code Block 1:

```
LP_vars = (double*)calloc((1 << num_rvs) + num_add_LPvars, sizeof(double));
```

Code Block 2:

```
for (int i = 0; i < (1 << num_rvs) + num_add_LPvars; i++) {
   if (fabs(LP_vars[i]) > TOLERANCE_LPGENERATION) {
      objective_pos[num_items_obj] = i;
      objective_values[num_items_obj] = LP_vars[i];
      num_items_obj++;
   }
}
```

Code Block 3:

```
free(LP_vars);
```

There are two main slow-downs here. First, because `num_rvs` in this case is `23`, we're allocating approximately 2^{23} `doubles` (generally 2^{26} bytes, or 64 megabytes) for the variable `LP_vars` (as seen in Code Block 1). That makes `free`ing `LP_vars` a very time consuming task: Code Block 3 (the `free` call alone) usually took around 6.3 milliseconds to run.

Second, the `for`-loop in Code Block 2 iterates over $2^{23} + 2$ integers. This is necessary because the terms of the function (at this point) are stored based on `OR`ing together the random variables included in entropy terms. This creates a huge search every time a function is entered, no matter how simple it might be. This loop usually takes over 90 milliseconds to run.

While around 96 milliseconds might not seem too bad in itself, this function can end up being called very often because it's used to parse the left side of bounds in addition to the objective function. In the file discussed in this Section, `parse_objective` is called 1 time for the objective function, 1 time for the bound to prove, and 568 times for the constant bounds, yielding $570 * 0.096s = 54.72s$, which is almost the time that the entire file takes to parse from start to finish.

This isn't a problem in v1.0 for one main reason: Easy access to pointers to class instances. Instead of storing (for example) `H(A|B)` in a 64MB array, the new version would store it as a `Term` class instance with `Operation` equal to `"H"`, `Coefficient` equal to `1`, the `Entropy_Of vector` having size 1 containing a pointer to the `Variable` with `Name "A"`, and the `Given_Vars vector` having size 1 containing a pointer to the `Variable` with `Name "B"`. This is huge for speed.

6.4 Sending Data Structure to the Linear Solvers

Once the parser has correctly read and stored the problem description from the input file, if the parser is called in the linear solver linker code, it then calls a method to create a `Problem_Description_C_Style` instance and fills its variables appropriately. Since there is no reading of the input file involved at this stage, and all that it needs to do is change the representation of variables, this part is done quickly. With the `Problem_Description_C_Style` created, the parser sends the struct to the linear solver. Instructions on how to call a linear solver linker executable can be found in Subsection 6.5.12.

6.5 CAI v1.0 User Guide

This new version of CAI (described above in Section 6) consists of two types of executables: one parser executable and various "linking" executables to send our data structure to the linear solvers. This Section will first focus on the input file format, which is the same for all executables.

Input problem description files must include the characters `PD` (which stand for "problem description"), followed by a JSON detailing the problem description. The JSON can either be on the same line as `PD` or start on the next line, but it can't begin on the same line as `PD` and then continue on the next line.

The problem description JSON can contain the following keys: `RV`, `AL`, `O`, `D`, `I`, `S`, `BC`, and `BP`.

6.5.1 `RV`: Random Variables

The `RV` key is used to add random variables, which must be alphanumeric strings not beginning with a number. `RV`'s value must be a JSON array of strings, even if there's only one random variable. For example, the following are allowed:

```
"RV" : ["A","B","var3"]
"RV" : ["C"]
```

The following are not allowed:

```
"RV" : "A"      // Not an array
"RV" : A        // Not an array, no quotes
"RV" : [A]      // No quotes
"RV" : [A,B]    // No quotes
```

6.5.2 `AL`: Additional Linear Programming Variables

The `AL` key is used to add additional linear programming variables, which must be alphanumeric strings not beginning with a number. `AL`'s value must be a JSON array of strings, even if there's only one variable. For example, the following are allowed:

```
"AL" : ["A","B","var3"]
"AL" : ["C"]
```

The following are not allowed:

```
"AL" : "A"      // Not an array
"AL" : A        // Not an array, no quotes
"AL" : [A]      // No quotes
"AL" : [A,B]    // No quotes
```

6.5.3 `O`: Objective Function

The `O` key is used to add the objective function. Its value should be a string written as expected. Spacing is ignored, though recommended for human readability. The following is allowed:

```
"O" :  "A + 5B + H(C) - 2H(C,D|E,F)"
```

6.5.4 `D`: Dependencies

The `D` key is used to add dependencies. Its value must be an array of JSON objects, each of which specifies a dependency. Each dependency JSON object should have a `dependent` key and a `given` key, each of which has a value which is an array of random variables. For example, the dependencies

$$H(A,B|C) = 0$$
$$H(E,F) = 0$$

would be represented by

```
"D" : [
          {"dependent" : ["A","B"] , "given" : ["C"]} ,
          {"dependent" : ["E","F"] , "given" : []}
      ]
```

6.5.5 `I`: Independences

The `I` key is used to add independences. Its value must be an array of JSON objects, each of which specifies an independence. Each independence JSON object should have an `independent` key and a `given` key, each of which has a value which is an array of random variables. For example, the independences

```
I(A;B|C) = 0
I(E;F) = 0
```

would be represented by

```
"I" : [
          {"independent" : ["A","B"] , "given" : ["C"]} ,
          {"independent" : ["E","F"] , "given" : []}
      ]
```

6.5.6 `S`: Symmetries

The `S` key is used to add symmetries. Its value must be an array of arrays of random variables, with each inner array containing every random variable exactly once. For example, assuming the set of all random variables is `["A","B","C","D"]`, The following is a valid set of symmetries:

```
"S" : [
          ["A","B","C","D"],
          ["A","C","B","D"],
          ["A","B","D","C"],
          ["A","C","D","B"],
          ["A","D","B","C"],
          ["A","D","C","B"]
      ]
```

6.5.7 `BC`: Constant Bounds

The `BC` key is used to add constant bounds. Its value should be an array of strings of (in)equalities. The right-hand side must be a number. For example:

```
"BC" : [
          "H(A,B) + 2C <= 0" ,
          "H(D) - X <= 5.2"
      ]
```

6.5.8 `BP`: Bounds to Prove

The `BP` key is used to add bounds to prove. Its value should be an array of strings of (in)equalities. The right-hand side must be a number. For example:

```
"BP" : [
          "H(A,B) + 2C <= 0" ,
          "H(D) - X <= 5.2"
      ]
```

6.5.9 CMD and OPT: Commands and Options

The keys CMD and OPT are identical and are both included for legacy reasons. (I will be using CMD for examples and explanation without loss of generality.) Their value should be an array of commands, and the values of this array can consist only of SER, PDC, and CS, and will tell the parser to run that command (or those commands) after parsing the rest of the PD. The functionality of SER, PDC, and CS is described in the next Subsection.

6.5.10 Commands Other Than PD

Before and after the PD JSON, there are a few other commands that can be placed in the problem description file, and comments are allowed. To comment, simply start the line with a #. (Note that comments are not allowed within a JSON).

To read in commands from a second file, use the command DESER following by at least one file name. For example:

```
DESER file1 [file2 ...]
```

To print out the problem description state, use the SER command, followed by an optional file to output to (with an append or truncate flag). If no output file specified, this will output to standard out. For example:

```
SER [-a|-t file]
```

To print the current state in the old C (v0.1) format, use the command PDC. For example:

```
PDC
```

To force the executable to check that the symmetries entered are valid, use the CS command. For example:

```
CS
```

Example problem description files are given in Sections 7.1.3 and 7.2.3.

6.5.11 Parser Executable

The parser executable itself is in parser.out. This interactive command-line-interface-style parser can be called in the following way:

```
./parser.out [optional_prompt_string]
```

All allowable commands from the problem description file are allowable commands to the parser (PD, DESER, SER, PDC, CS). Three other commands are also allowed: DESTROY, to wipe the current state of the problem description held in memory, ? to print out commands, and Q to quit.

In addition, anything that exists as a key within the PD JSON (RV, AL, O, D, I, S, BC, BP) is also an allowable command to the parser. They work the same way as with the PD JSON, but without quotes around the commands, without a colon to separate them from the keys, and without allowing parts of the key to be on the same line and parts on the next lines. For example, this PD JSON

```
PD
{
    "RV" : ["A","B","C"] ,
    "D"  : [
                { "dependent" : ["A","B"] , "given" : "C" }
           ]
}
```

may be equivalently written either as

```
RV
["A","B","C"]
D
[
    { "dependent" : ["A","B"] , "given" : "C" }
]
```

or

```
RV ["A","B","C"]
D  [ { "dependent" : ["A","B"] , "given" : "C" } ]
```

6.5.12 Linking Executables

The objective of the linking executables is to use the problem description held in memory to create a data structure that can be sent to a linear solver. The two linear solvers that are currently used in CAI are Cplex [3] and Gurobi [2]. The executable for Cplex is stored in `cplexcompute.out`, and the executable for Gurobi is stored in `gurobicompute.out`. Each of these executables is called by the user in the same way, so when this Section needs to demonstrate a call to an executable, `cplexcompute.out` will be used without loss of generality.

The linking executables can be run in this way:

```
./cplexcompute.out [hull|random|prove|sensitivity] inputfile [cmd-line-modifier-1 ...]
```

The first element is the name of the executable, followed by an optional compute type (`hull`, `random`, `prove`, `sensitivity`). If no compute type is specified, `regular` will be assumed. The next argument is the input file (the problem description file), which should have the same format as described above. (Note that, whereas for the parser executable, the input file was read on standard in, the input file to the linking executables is a command-line argument.) After the input file, command-line modifiers can be added.

6.5.13 Command-Line Modifiers

When running a linear solver, the input file can be modified in memory before sending data to the linear solver by using command-line modifiers. Each command-line modifier will need to be a JSON whose keys are a subset of the following: `RV`, `AL`, `O`, `D`, `I`, `S`, `BC`, `BP`, `+RV`, `+AL`, `+D`, `+I`, `+S`, `+BC`, `+BP`, `+CMD`, `+OPT`. (Note the exclusion of `+O`, `CMD`, and `OPT`.) These are the same keys allowed in the PD JSON (with the exclusion of `CMD` and `OPT`), and those keys preceded by a `+` (with the exclusion of `+O`). The value for each of these keys must be the same as for the PD JSON equivalent: a JSON array containing elements in the same format as in the PD JSON (with the exception that the `O` key can have a value which is just a string; the array isn't necessary).

Keys that don't include a `+` sign (i.e. keys that are identical to the PD JSON keys) will replace that line from the input file, while keys that do inclue a `+` sign will append the item(s) from its value to the appropriate part of the problem description.

The only allowable elements of the JSON array which is the value of `+CMD` or `+OPT` are `SER`, `PDC`, and `CS`.

Note that command-line modifiers don't modify the input file itself; they modify only the internal data structures held in memory by the program. Also note that command-line modifiers must be wrapped in single-quotes (for Linux and Mac users).

Here is an example with a description of what happens:

```
./cplexcompute.out inputfile.pd '{"+BC" : ["H(X) = 0", "H(Y) = 0"] , "D" :
[{"dependent" : ["A","B"] , "given" : ["C"]}]}' '{"+CMD" : ["PDC"]}'
```

1. `./cplexcompute.out` reads `inputfile.pd`.
2. `"H(X) = 0"` and `"H(Y) = 0"` are added to the `BC` currently stored.
3. All dependencies in memory from `inputfile.pd` are deleted.
4. The dependency in the command-line modifier becomes the only dependency.
5. The program records that it needs to run `PDC` before sending data to Cplex.

6.5.14 Compatability for Windows Command Prompt Users

The Windows Command Prompt handles quotes differently. For Windows Command Prompt users, each command-line modifier needs to be surrounded by double-quotes, and the inner quotes should be escaped with a backslash. The above example would need to be written as follows for the Windows Command Prompt:

```
CplexCompute.exe inputfile.pd "\{\"+BC\" : [\"H(X) = 0\", \"H(Y) = 0\"] , \"D\" :
[\{\"dependent\" : [\"A\",\"B\"] , \"given\" : [\"C\"]\}]\}" "\{\"+CMD\" : [\"PDC\"]\}"
```

7 Mission 3: Demonstrate CAI's Utility

Now that the CAI toolkit has been thoroughly discussed, it seems appropriate to provide examples of it being used. This Section contains two types of problems descriptions, explains how to write them for use with CAI, and relates their respective performances.

7.1 Problem 1: RAID-6

7.1.1 Problem Overview

In this presentation, we will include a very simple example of a five-disk RAID-6 storage system. In such a storage system, there are three disks that hold data, and there are two that hold parity; the parity disks are computed from the data. We will label the data disks A, B, and C, and the parity disks P and Q. To be a valid RAID-6 system, the loss of any two disks must be tolerated. In other words, if any two disks fail, the contents of A, B, and C must be able to be determined from the surviving disks. We will use CAI to answer the following question: If A, B, and C are all x bytes in size, then what is the minimum storage capacity required of P and Q? The answer to this question is well-known: Any MDS erasure code will implement RAID-6 with all five disks being the same size. However, we will use CAI to derive this result, and we will frame it in terms of entropy: In a five-disk RAID-6 system, if the respective entropies of A, B, and C are identical, what is the minimum entropy of P and Q?

7.1.2 How to Distill This Down for CAI

RV: The random variables for this problem are the five disks: A, B, C, P, and Q.

AL: We introduce one additional linear programming variable, which we will call `S`. This will represent the larger of the entropies of P and Q.

D: As a reminder, the dependencies allow the user to specify when the entropy of one set of variables, given a second set of variables, is zero. In such a case, we say that the first set "depends on" the second set. Because our system must tolerate the failure of any two disks (that is, the system must be able to recreate any two disks given any three disks), we have that any set of two random variables must depend on the remaining three random variables. This yields the following ten dependencies:

```
{"dependent" : ["A","B"] , "given" : ["C","P","Q"] }
{"dependent" : ["A","C"] , "given" : ["B","P","Q"] }
{"dependent" : ["A","P"] , "given" : ["B","C","Q"] }
{"dependent" : ["A","Q"] , "given" : ["B","C","P"] }
{"dependent" : ["B","C"] , "given" : ["A","P","Q"] }
{"dependent" : ["B","P"] , "given" : ["A","C","Q"] }
{"dependent" : ["B","Q"] , "given" : ["A","C","P"] }
{"dependent" : ["C","P"] , "given" : ["A","B","Q"] }
{"dependent" : ["C","Q"] , "given" : ["A","B","P"] }
{"dependent" : ["P","Q"] , "given" : ["A","B","C"] }
```

BC: These are equations that state global conditions of the system and help frame the problem being solved. In the RAID-6 example, we use the constant bounds to state that drives A, B, and C are the same size, and we normalize their entropy to one. This yields the following three equations:

```
"H(A) = 1"
"H(B) = 1"
"H(C) = 1"
```

Additionally, we state that drives P and Q are less than or equal to `S`:

```
"H(P) <= S"
"H(Q) <= S"
```

O: The objective function to minimize is simply `S`, which, again, is the maximum entropy of the P and Q drives.

S: There are times when the relationships between random variables allow them to be permuted without changing their functionalities. As the symmetries are used to reduce the number of necessary bounds and dependencies (and, for probelms requiring them, independences), they're unnecessary for CAI to run, but symmetries are a big asset of CAI in that they can significantly cut run-time by simplifying the linear system CAI is solving. In this problem, the identities of A, B, and C are interchangeable, as are the identities of P and Q. We express this to CAI by stating that the following 12 permutations are identical:

```
["A","B","C","P","Q"] | ["A","B","C","Q","P"]
["A","C","B","P","Q"] | ["A","C","B","Q","P"]
["B","A","C","P","Q"] | ["B","A","C","Q","P"]
["B","C","A","P","Q"] | ["B","C","A","Q","P"]
["C","A","B","P","Q"] | ["C","A","B","Q","P"]
["C","B","A","P","Q"] | ["C","B","A","Q","P"]
```

These symmetries allow us to reduce the number of dependencies to 3: one dependency with two data disks being lost, a second with two coding disks being lost, and a third with one of each.

```
{"dependent" : ["A","B"] , "given" : ["C","P","Q"] }
{"dependent" : ["P","Q"] , "given" : ["A","B","C"] }
{"dependent" : ["A","P"] , "given" : ["B","C","Q"] }
```

The symmetries also allow us to reduce the number of constant bounds to the following three:

```
"H(A) = 1"
"H(A,B,C) = 3"
"H(P) <= S"
```

7.1.3 Problem Description File

```
PD
{
    "RV" : ["A","B","C","P","Q"] ,
    "AL" : ["S"] ,
    "O"  : "S" ,
    "D"  :  [
        {"dependent" : ["A","B"] , "given" : ["C","P","Q"] } ,
        {"dependent" : ["P","Q"] , "given" : ["A","B","C"] } ,
        {"dependent" : ["A","P"] , "given" : ["B","C","Q"] }
    ] ,
    "BC" : [
        "H(A) = 1" ,
        "H(A,B,C) = 3" ,
        "H(P) - S <= 0"
        ] ,
    "S"  : [
        ["A","B","C","P","Q"] ,
        ["A","B","C","Q","P"] ,
        ["A","C","B","P","Q"] ,
        ["A","C","B","Q","P"] ,
        ["B","A","C","P","Q"] ,
        ["B","A","C","Q","P"] ,
        ["B","C","A","P","Q"] ,
        ["B","C","A","Q","P"] ,
        ["C","A","B","P","Q"] ,
        ["C","A","B","Q","P"] ,
        ["C","B","A","P","Q"] ,
        ["C","B","A","Q","P"]
    ]
}
```

7.1.4 Solution

When this file is run in CAI, the objective function (s) is minimized to 1, meaning that the minimum size of the P and Q drives must equal the size of the data drives. As stated above, this is a well-known result, so we haven't learned anything new from CAI, but this example is a simple demonstration of CAI.

7.2 Problem 2: Regenerating Codes

7.2.1 Problem Overview

Regenerating codes are employed in distributed storage systems [1]. Data is encoded and distributed among the storage nodes, and two fundamental operations are supported:

1. The data is requested from a client, and the storage system must decode the data from a subset of the storage nodes.

2. A storage node fails, and a replacement must be populated with its contents by using the non-failed nodes.

The RAID techniques discussed above can be used to implement regenerating codes; however, additional coding techniques can enrich the storage system with improved performance along several dimensions. Specifically, techniques have been developed so that an (n, k, d) regenerating code has the following properties:

- Let the size of the data be denoted by S_D.
- The storage system is composed of n storage nodes, labeled $N_0, \ldots, N_{n-1}$.
- Each storage node N_i stores S_N bytes of storage.
- k nodes are required to decode the data and send it to a client. Each of them sends all S_N bytes to the client for decoding.
- If a node fails, then $d \geq k$ nodes are required so that a replacement can recover its contents. Each node sends S_R bytes to the replacement.

We draw the actions of encoding, decoding, and node replacement for a $(4, 2, 3)$ regenerating code in Figure 1. In the figure, a 20 MB file is encoded onto four 10 MB regions in four storage nodes. The data may be decoded from any two of the four nodes, and if a node fails, then it may be replaced by having each of the remaining three nodes send 5MB for the replacement to decode. Thus, using the variables above, $S_D = 20MB$, $S_N = 10MB$, and $S_R = 5MB$.

With regenerating codes, one may trade off the amount of storage required at the storage nodes with the amount of encoded data sent to restore a failed node. In the example above, it is also possible to have each node store 12 MB, and then replace a failed node by having each of the three survivors send just 4 MB to the replacement.

7.2.2 How to Distill This Down for CAI

Here we will describe the details of the input file for a $(4, 2, 3)$ regenerating code.

RV: Our random variables to this function correspond to the 12 possibilities of nodes being used to repair each other. We denominate each random variable $S_{i,j}$, where $i \in [0, 3] \cap \mathbb{Z}$ and $j \in [0, 3] \cap \mathbb{Z} : j \neq i$.

AL: As shown in [6], for the storage requirement and the bandwidth constraint, we need two additional linear programming variables. Without repeating details here, Tian shows three relationships establishing the need for these additional linear programming variables: First, the entropy of each set of random variables with the same first index is less than or equal to $\log k$ (for example, $H(S_{0,1}, S_{0,2}, S_{0,3}) \leq \log k$). Second, the entropy of any of our random variables is less than or equal to $\log d$. Lastly, Tian shows that $\log k$ is "essentially" S_N and $\log d$ is "essentially" S_d. (Note that Tian has $\alpha = S_N$ and $\beta = S_d$.) We will define A to be $\log k$ (corresponding to Tian's α), and we define B to be $\log d$ (corresponding to Tian's β). Our two `ALs` are A and B. These will be used both in the objective function and in the section on constant bounds.

O: Since we want to find the lowest combination of the storage and bandwidth overhead, our objective function to be minimized is `"A + B"`.

S: Each symmetry line corresponds to a permutation of $0, 1, 2, 3$. For permutation a, b, c, d, the first index of the first three random variables is a, the first index of the second three random variables is b, the first index of random variables 6-8 is c, and the first index of the random variables 9-11 id d. So we currently have

```
["Sa_","Sa_","Sa_","Sb_","Sb_","Sb_","Sc_","Sc_","Sc_","Sd_","Sd_","Sd_"]
```

Next, look at each grouping of variables with the same first index separately. Place a, b, c, and d in the order they're in in the permutation, skipping the number corresponding to the first index. So for the b set, the first variable should be `Sba`, the second skips b and becomes `Sbc`, and the third becomes `Sbd`. The full list of permutations can be found in the full problem description file in the next Subsection.

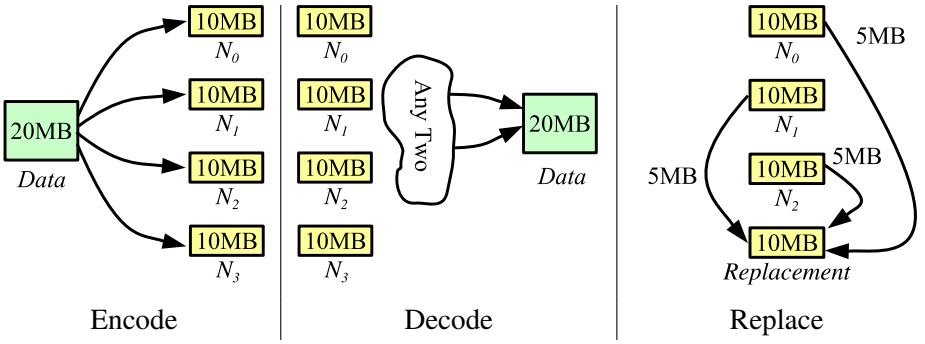

Figure 1: The actions in a storage system based on a $(4, 2, 3)$ regenerating code. It is also possible for each node to store 12 MB of encoded data and then send 4MB to a replacement to regenerate a failed node.

D: The dependencies for this problem are best described as being four groups of four, with each group demonstrating that any of the four nodes can be determined from the remaining nodes. Within each group, there's one dependency showing that a node can be repaired from the other three nodes, and then there are three more dependencies which show that it only takes two nodes to read the entire data. The list of dependencies can be found in the next Subsection showing the problem description file.

BC: As explained in [6], this problem has three bounds. First, because k nodes are required to read the data, we have that $H(S01, S02, S03) \leq \log k$, which we've defined to be A. (By our symmetries, we actually have a family of bounds encapsulated in that one inequality.) Second, because d nodes are required to fixed a lost node, we have $H(Sij) \leq \log d \forall\ i, j$. This family of bounds can be written with just one inequality: $H(S12) \leq \log d$. Third, we know the entropy of all RVs together is greater than or equal to 1.

7.2.3 Problem Description File

```
PD
{
    "RV":["S01","S02","S03","S10","S12","S13","S20","S21","S23","S30","S31","S32"],
    "AL":["A","B"],
    "O" :"A + B",
    "D" : [
              {"dependent":["S01","S02","S03"],"given":["S10","S20","S30"]},
              {"dependent":["S01","S02","S03"],"given":["S10","S12","S13","S20","S21","S23"]},
              {"dependent":["S01","S02","S03"],"given":["S10","S12","S13","S30","S31","S32"]},
              {"dependent":["S01","S02","S03"],"given":["S20","S21","S23","S30","S31","S32"]},
              {"dependent":["S10","S12","S13"],"given":["S01","S21","S31"]},
              {"dependent":["S10","S12","S13"],"given":["S01","S02","S03","S20","S21","S23"]},
              {"dependent":["S10","S12","S13"],"given":["S01","S02","S03","S30","S31","S32"]},
              {"dependent":["S10","S12","S13"],"given":["S20","S21","S23","S30","S31","S32"]},
              {"dependent":["S20","S21","S23"],"given":["S02","S12","S32"]},
              {"dependent":["S20","S21","S23"],"given":["S01","S02","S03","S10","S12","S13"]},
              {"dependent":["S20","S21","S23"],"given":["S01","S02","S03","S30","S31","S32"]},
              {"dependent":["S20","S21","S23"],"given":["S10","S12","S13","S30","S31","S32"]},
              {"dependent":["S30","S31","S32"],"given":["S03","S13","S23"]},
              {"dependent":["S30","S31","S32"],"given":["S01","S02","S03","S10","S12","S13"]},
              {"dependent":["S30","S31","S32"],"given":["S01","S02","S03","S20","S21","S23"]},
              {"dependent":["S30","S31","S32"],"given":["S10","S12","S13","S20","S21","S23"]} ],
    "BC": [
              "H(S01,S02,S03) - A <= 0",
              "H(S01) - B <= 0",
              "H(S01,S02,S03,S10,S12,S13,S20,S21,S23,S30,S31,S32) >= 1" ],
    "S" : [
             ["S01","S02","S03","S10","S12","S13","S20","S21","S23","S30","S31","S32"],
             ["S01","S03","S02","S10","S13","S12","S30","S31","S32","S20","S21","S23"],
             ["S02","S01","S03","S20","S21","S23","S10","S12","S13","S30","S32","S31"],
             ["S03","S02","S01","S30","S32","S31","S20","S23","S21","S10","S13","S12"],
             ["S02","S03","S01","S20","S23","S21","S30","S32","S31","S10","S12","S13"],
             ["S03","S01","S02","S30","S31","S32","S10","S13","S12","S20","S23","S21"],
             ["S10","S12","S13","S01","S02","S03","S21","S20","S23","S31","S30","S32"],
             ["S13","S12","S10","S31","S32","S30","S21","S23","S20","S01","S03","S02"],
             ["S10","S13","S12","S01","S03","S02","S31","S30","S32","S21","S20","S23"],
             ["S13","S10","S12","S31","S30","S32","S01","S03","S02","S21","S23","S20"],
             ["S12","S10","S13","S21","S20","S23","S01","S02","S03","S31","S32","S30"],
             ["S12","S13","S10","S21","S23","S20","S31","S32","S30","S01","S02","S03"],
             ["S21","S20","S23","S12","S10","S13","S02","S01","S03","S32","S31","S30"],
             ["S21","S23","S20","S12","S13","S10","S32","S31","S30","S02","S01","S03"],
             ["S20","S21","S23","S02","S01","S03","S12","S10","S13","S32","S30","S31"],
             ["S20","S23","S21","S02","S03","S01","S32","S30","S31","S12","S10","S13"],
             ["S23","S20","S21","S32","S30","S31","S02","S03","S01","S12","S13","S10"],
             ["S23","S21","S20","S32","S31","S30","S12","S13","S10","S02","S03","S01"],
             ["S31","S32","S30","S13","S12","S10","S23","S21","S20","S03","S01","S02"],
             ["S31","S30","S32","S13","S10","S12","S03","S01","S02","S23","S21","S20"],
             ["S30","S32","S31","S03","S02","S01","S23","S20","S21","S13","S10","S12"],
             ["S30","S31","S32","S03","S01","S02","S13","S10","S12","S23","S20","S21"],
             ["S32","S30","S31","S23","S20","S21","S03","S02","S01","S13","S12","S10"],
             ["S32","S31","S30","S23","S21","S20","S13","S12","S10","S03","S02","S01"] ]
}
```

7.2.4 Performance

There have been quite a few papers written to prove the theoretical limits of the tradeoffs with regenerating codes [1, 6]. We are able to perform this analysis using CAI for many small regenerating code systems. This has resulted in some discoveries new to the field. Unfortunately, the number of random variables for a system with N disks is $(N * (N - 1))$, limiting the storage systems to five nodes and fewer. Regardless, we present the optimal tradeoffs of S_N vs. S_R (normalizing S_D to 1) in Figure 2.

No individual result Figure 2 is new information to the community, but, to our knowledge, this is the first work that provides an overview of this data in one place. It should be mentioned here that these results are theoretical optimality tradeoffs; actually constructing regenerating codes that achieve these optimality points is another matter.

7.3 Run-Time Complexity

The high-level picture of CAI's operation is as follows:

1. A linear programming (LP) variable is created for each non-empty subset of random variables. There are $2^{\|RV\|}$ of these.
2. These variables are partitioned into sets of equivalent LP variables according to both the symmetry and the dependencies. For example, in the RAID-6 layout above, the LP variables $\{A\}$, $\{B\}$, and $\{C\}$ are equivalent, as are $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$, and as are $\{P\}$ and $\{Q\}$. Moreover, from the dependency that $\{A, B\}$ depends on $\{P, Q, C\}$, we can union any LP variable that contains $\{P, Q, C\}$ with the same variable that also contains $\{A, B\}$. When the process is done, there are six sets of LP variables as displayed in Table 1.
3. A linear system is created from these sets, the additional variables, and the constant bounds.
4. The linear system is solved by an external linear solver.
5. The results are mapped back from the sets above to relevant LP variables.

We omit a lot of details, explained more precisely in [5]. We include the detail about the LP variables in order to convey the importance of the symmetry specification and the dependencies. We always start with $O(2^{\|RV\|})$ LP variables. However, the symmetry and dependencies allow us to merge these variables into equivalent sets, thereby reducing the size of the linear system that must be solved.

8 Conclusion

As has been shown, the CAI toolkit is extremely useful in showing bounds in information theory problems that can be used in erasure coding. CAI is very user-friendly, and v1.0, which has been released open-source to the community, is optimized for use by researchers accustomed to using Python. We've demonstrated in Section 7 an example of how CAI might be used, and we encourage its use by others. In short, CAI is an effective addition to the information theory corpus.

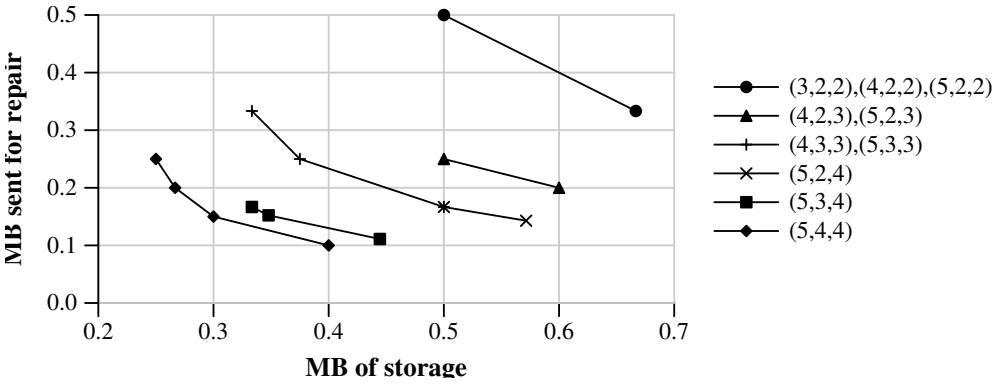

Figure 2: Optimality tradeoff in megabytes of storage per storage node (S_N) vs. megabytes sent for replacement, per storage node (S_R), assuming 1MB of data is stored on the system.

$\{A\}$,$\{B\}$ and $\{C\}$
$\{A, B\}$,$\{A, C\}$ and $\{B, C\}$
$\{P\}$ and $\{Q\}$
$\{P, Q\}$
$\{A, P\}$,$\{A, Q\}$,$\{B, P\}$,$\{B, Q\}$,$\{C, P\}$ and $\{C, Q\}$
All of the LP's that contain three or more random variables.

Table 1: The six sets of equivalent LP variables (non-empty subsets of random variables) for a horizontal, 5-disk RAID-6 system.

References

[1] DIMAKIS, A. G., GODFREY, P. B., WU, Y., WAINWRIGHT, M., AND K., R. Network coding for distributed storage systems. *IEEE Trans. Information Theory 56*, 9 (September 2010), 4539–4551.

[2] GUROBI OPTIMIZATION, LLC. Gurobi optimizer reference manual. http://www.gurobi.com, 2020.

[3] IBM. IBM ILOG CPLEX 12.7 user's manual. IBM Corp., 2017.

[4] PLANK, J. S. Erasure codes for storage systems: A brief primer. *Magazine of USENIX and SAGE 38*, 6 (December 2013), 44–50.

[5] PLANK, J. S., TIAN, C., AND HURST, B. Practical information theory results for computer systems researchers using the cai software toolkit, September 2020.

[6] TIAN, C. Characterizing the rate region of the (4,3,3) exact-repair regenerating codes. *IEEE Journal on Selected Areas in Communications 32*, 5 (May 2014), 967–975.

[7] TIAN, C., PLANK, J. S., HURST, B., AND ZHOU, R. An open-source toolbox for computer-aided investigation on the fundamental limits of information systems, version 1.0.

Vita

Brent Hurst spent his early life in Sequatchie County, Tennessee, before moving to Knoxville to attend the University of Tennessee. There, he received a Bachelor of Science degree and a Master of Science degree, both in Computer Science. His research area included the development of open-source software to facilitate further study in the field of information theory.