# CS4341 Project 5 Report

Odell Dotson, Ethan Prihar

**Program Instructions:**

==========
Please note that this program is using

Python 2.7.10
Numpy library (for some added maths)

There are no additional external dependancies that do not come with python for our feature generation code to work.
==========
This code will take a set of inputs, defining the relationships between the items and the bags.

Our code will use the information given to create several classifications for each item, and then use backtracking and forward checking and heuristics to reorder the list and then solve the given constraint satisfaction problem.
==========
In order to run the program, first extract the archive ZIP file. Open a terminal and navigate to the extracted folder.
Once in the folder, run the command with the following format:

python main.py <desired file>

In order to specify a file to run the program on, look at the following examples:

python main.py input1.txt

is referring to your desired text file, named input1.txt in the same directory.
For example, this might look like:

python main.py input1.txt

Or, if your data is in a subdirectory:

python main.py data/input1.txt

This will print the solution to the terminal. Please note that the file extension AND directory need to be specified.
==========

**Design Description:**

To solve the constraint satisfaction problem provided we used a backtracking search with a minimum remaining values heuristic, a least constraining value heuristic, and forward checking to create a solution. To implement the backtracking search we created two classes, one for items and one for bags. Each class contained all of the parameters of the objects as well as functions to determine whether or not an item could be placed in a bag, and wrappers to add or remove items from bags. Two lists were created, one of all the bags, and one of all the objects. The backtracking search worked its way through the items, placing each item in a bag as it went. When an item could not be placed into a bag, the program would remove

the last item that was placed in a bag from its bag and try to place it in the next bag in the list. When all of the items were placed in a bag, a final function was called to make sure that all of the parameters of the bags were met, such as every bag needing to be 90% full and needing a minimum number of items. If these criteria were met, the search would end. Otherwise the search would begin to backtrack again. If a state was reached where the very first item that was being placed in a bag was unable to be placed in any bag, this meant that there was no solution to the CSP. In this case, the program returns the statement that there is no solution to the problem.

To speed up the search two heuristics were added. The goal of the first heuristic was to organize the items by how constrained they are. By starting the backtracking search with the most constrained item, the need to backtrack later in order to accommodate that items constraints is eliminated. In order to determine which items were most constrained the following equation was used.

Where A is the number of bags that the item is allowed to be placed in, B is the number of items that this item must be placed with, C is the number of items that this item cannot be placed with, D is the number of items mutually inclusive with this item, and H is the value of the heuristic. The list of items was sorted from lowest to highest heuristic value so that the most constrained items would be at the beginning of the list. This helped to decrease the amount of backtracking required.

The other heuristic determined which bags were least constrained. The bags that were least constrained were more likely not to reject having an item placed in it. By attempting to fill the least constrained bags first, less item rejections occurred so the search was able to be completed faster. To determine which bags were the least constrained the following equation was used.

Where A is the capacity of the bag, meaning how many weight units of items it can hold, B is the maximum number of items the bag can hold, C is the minimum amount of items the bag must hold, and H is the value of the heuristic. The list of bags was sorted from lowest to highest heuristic value so that the least constrained items would be at the beginning of the list. This decreased the amount of item rejections and the search was completed faster.

In order to implement forward checking a function was created that would check if all items that were not in a bag could be placed into any bag. This function was called every time an item was placed in a bag. The goal of this function was to check if any of the items domains had been reduced to no bags. If this was the case than the last item to be placed in a bag was placed in the wrong bag and the program could begin to backtrack sooner instead of having to wait until it reached an item with an empty domain, which would take longer and involve many useless steps. This function helped to reduce the amount of time it took our program to solve the CSP.

**Tests and Performance:**

To test our program we ran our code using the inputs provided to us for testing. For every input we compared our programs output with the answer key. For each new constraint introduced in the input, such as unary inclusive or binary equals, we would need to debug the section of the code responsible for recording the constraints as well as the section of the code responsible for determining if an item was allowed in a bag based on those constraints. The process was fairly straight forward, when a new constraint was implemented a standard debugging procedure of printing statements after every line to see where there was an inconsistency or error was used. This process allowed for a streamline debugging process.

The program performed well for almost all of the test cases. With just backtracking, our program solved the test cases, but not as quickly as with heuristics and forward checking. Using the heuristics and forward checking lead to a decrease in the amount of time to run the program when the test case required more than 200 consistency checks. Our program is fairly strong when the CSP has a simple answer, but when the answers become more complicated, the program begins to become inadequate. In a professional situation our program would most likely need to be rewritten in a lower level programming language, but we are happy with the results of our program.

In order to see verbose sample outputs of our program, please look to the files:

myoutput5.txt, myoutput7.txt, myoutput12.txt, myoutput20.txt, myoutput24.txt

**CSP Algorithm Comparison:**

**File:**

| | BT alone | BT + heuristics | BT, heuristics and FC |
|---|---|---|---|
| input5.txt | 135 | 135 | 135 |
| input6.txt | 9 | 9 | 9 |
| input7.txt | 16 | 16 | 12 |
| input8.txt | 235 | 235 | 196 |
| input9.txt | 3 | 3 | 3 |
| input10.txt | 9 | 9 | 9 |