Programming Language Compilation by Linear Parsing and Translation

Randall Porter

April 24, 2019

**Abstract**

In order for code written in any high level language to be run on a machine, it must first be processed into a form the machine can understand. Generally, this takes the form of compilation, interpretation, or a combination of the two. The job of a compiler is to read a set of instructions in the source language and convert them into a functionally equivalent set of instructions in the target language. The challenge is for the compiler to read and translate the source code quickly, and generate target code that is both fast and efficient. One common approach is the use of grammars and parsing trees. However, this paper instead examines an example of a compiler that operates by reading source code one character at a time, linearly, in a two passes.

**1 Components**

The compiler discussed in this paper is written in Windows PowerShell. It takes command line arguments for input and output files. The intended input language is Bur, a custom esoteric programming language created for this project. Bur has a very small set of valid instructions compared to more popular high level languages such as Java, Python, or the C family. The intended output of the compiler is a binary executable that can be run from a command line interface (GitBash, in this instance).

**1.1 Compiler**

The compiler being discussed in this project is written in Windows PowerShell because of PowerShell’s convenient syntax for working with string manipulation, file input/output, and external program calls. The compiler accepts command line arguments for an input file and the desired name for an output file. The input file must be written in Bur (discussed below) and the output file is a binary executable that can be run from a command line interface. The Bur file is read into the compiler, and converted to a one-dimensional array of characters.

This array is processed character by character, two times. The first pass simply scrubs the source code of all comments. It does this by tracking a Boolean variable ($isComment) that represents whether the current character being analyzed is part of a comment. This variable is flipped whenever the compiler reads a double quote. If the variable is “false,” the character is added to a buffer, which is what gets processed in the second pass.

The second pass over the code relies upon a for-loop with an index variable ($i) to keep track of which character is being parsed at any given time. At any point during parsing, the current character in question is accessible with $codeIn[$i]. At the beginning of each iteration though the loop, the relevant character is fed through a switch statement, which has case blocks corresponding to Bur’s tokens. Within each case, an inner loop is run to read all characters leading up to the start of the next token. These characters are processed to determine their meaning, and the corresponding c code is added to an output buffer. $i is incremented for each character read in the inner loop so that the outer loop will resume at the next token.

After parsing and translating the program, the output buffer is written to

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

**1.2 Greedy Heuristic 1 (Most Valuable First)**

The first greedy algorithm is very simple. First, it sorts the list of available items in order of decreasing item value. Second, it steps through the sorted list and picks items sequentially until taking any more items would exceed the weight limit of the knapsack. In other words, it starts with the most valuable items first, and grabs as many as possible.

Function greedy1($items){

$sortedItems = sortByDecreasingValue $items

$accumulatedWeight = 0

$accumulatedValue = 0

$sortedItems | ForEach-Object -process {

if ($accumulatedWeight + $\_.weight -le $w){

$accumulatedValue += $\_.value

$accumulatedWeight += $\_.weight

}

}

return $accumulatedValue

}

There are two parts of this algorithm that may be significant in determining its time complexity. The first part is the sorting algorithm used. In this case, I chose to implement a radix sort, because all values being sorted are integers. In my implementation, this sort only has one for-loop that varies with problem size. This loops runs log10(**n**) times, and the entire sort is included in the timing. The second significant part of this algorithm is the for-each loop, which will always run a maximum of **n** times. In both of these cases, **n** is the number of item elements in the list being analyzed. By examination, the complexity of **n** will win out, yielding a worst case time complexity of **O(n)**.

**1.3 Greedy Heuristic 2 (Lightest First)**

The second greedy algorithm is identical in every way to the first, except that it sorts items by increasing value, and chooses the lightest ones first. This way, the greatest number of items can be chosen.

Function greedy2 ($items){

$sortedItems = sortByIncreasingWeight $items

$accumulatedWeight = 0

$accumulatedValue = 0

$sortedItems | ForEach-Object -process {

if ($accumulatedWeight + $\_.weight -le $w){

$accumulatedValue += $\_.value

$accumulatedWeight += $\_.weight

}

}

return $accumulatedValue

}

Only the sorting conditions are different for this algorithm than for the first greedy algorithm. Therefore, it has an identical time complexity of **O(n)**.

**2 Experimental Setup**

This project was done entirely using Windows PowerShell. PowerShell was chosen for its ability to prototype software rapidly, and for the lack of needing a complicated workspace or collection of project files. All calculations and tests were run on a Core i5 processor at 2.71 GHz on a system with 8 GB of RAM.

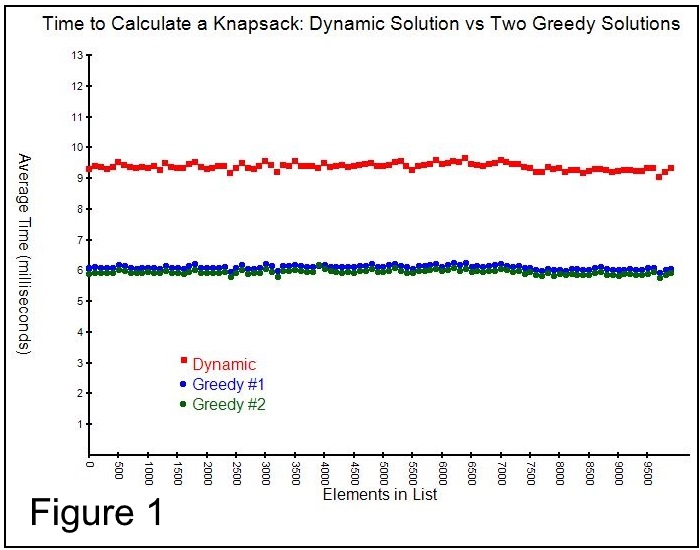
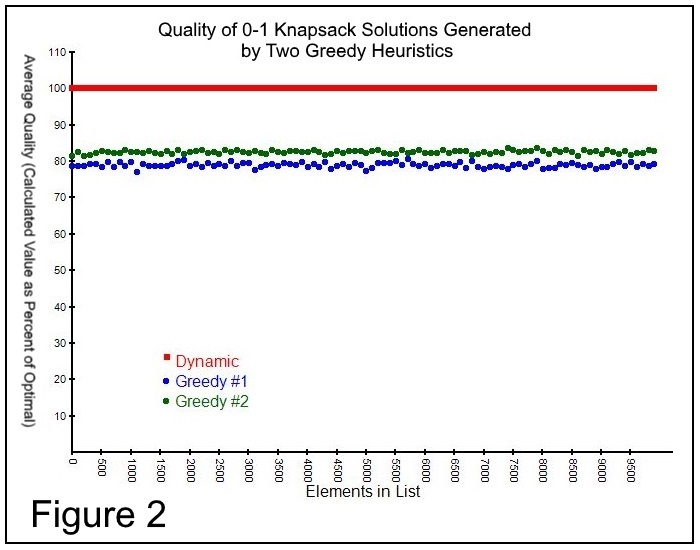
The program was divided into several parts.

The first part measures execution time (in milliseconds) and result quality (value returned, as a percentage of the optimal value) of all three algorithms for parameters provided by the user. Specifically, the user is prompted to provide values for **n** (the number of items to analyze) and **w** (the maximum weight any given item can have (minimum weight is 1), and the maximum weight that can be held by the bag). For this part, the maximum item value (**v**, minimum value is 0) and the number of iterations (used for averaging results) are both held constant at 100. Execution time does not include the time needed to generate the pseudorandom list of items, but does include the time needed to run a radix sort for the greedy algorithms.

The second part of the program generates data points for **v**=100, **w**=20, and various values of **n**. Values used for **n** were 0, 100, 200, 300, …, 9700, 9800, 9900, for a total of 100 data points. Every data point is an average of 1000 trials. As in part one, the minimum weight and value are 1 and 0, respectively. This part is run to calculate the average execution time of all 100 values of **n**, and also to calculate the average result quality of both greedy heuristics for all 100 values of **n**.

The final part of the program used PowerShell to manipulate Windows Forms to generate scatterplots of the data points generated in part two. The code in this section renders the data as three series of data points, color coded, representing the results of running the three different algorithms. In the case of displaying the result quality measurements, the quality of the dynamic solution is rendered as a solid red line at y=100. This serves as a benchmark for the greedy solutions (green and blue).

**3 Results**

Figure 1 (left) shows the execution time of all three algorithms under the conditions described in the Experimental Setup section. This plot makes it clear that both greedy algorithms (green and blue) are considerably (and consistently) faster than the dynamic solution (red). It also shows that the first greedy algorithm (blue, “most valuable items first”) tends to be slightly faster than the second greedy algorithm (green, “lightest items first”). Figure 2 (right) demonstrates that the knapsacks calculated by both greedy algorithms are about 20% less valuable than the optimal knapsacks. It also shows that the second greedy algorithm tends to perform slightly better than the first greedy algorithm.

**4 Conclusion**

When examined together, the two scatterplots indicate that the greedy algorithms are both faster and of lower quality than the dynamic algorithm. They also suggest that collecting the lightest items first is superior to collecting the most valuable items first, in terms of both speed and quality. One interesting observation is that neither the run time nor the result quality appear to be affected by the size of **n**. This is concerning, and may be indicative of a logic error in the timing methods. Assuming the results are accurate, however, it would be interesting to explore the effects of varying other variables besides **n**, or perhaps using a different sorting algorithm.