Optimizations for Depth-First Searches in Discrete Spaces

Mr. Kevin Shurtz

Abilene Christian University  
624 Honeysuckle Hollow  
(214) 966-8439

kms14d@acu.edu

**ABSTRACT**

This paper investigates simple optimization methodologies for moderately optimized depth-first searches in discrete state spaces. In particular, this paper evaluates the impact of presorting datasets before searching. I also consider the improvement garnered from eliminating unnecessary overhead in the LIFO queue used to manage the frontier of unexplored states. I found that presorting by descending cost and highest ratio generally led to faster searches. I additionally found that eliminating overhead in the stack used to navigate the state space offered the starkest and most consistent improvements in performance.

**CCS Concepts**

• **Computing methodologies➝Artifical Intelligence➝Search methodologies➝Discrete space search**

**Keywords**

Artificial intelligence; depth-first; heuristic; discrete space; search; optimization; stack; Java

# INTRODUCTION

Depth-first searching techniques are the essential to a variety of algorithms for navigating large state-spaces within reasonable memory constraints. Consequently, it is in the interest of the computing industry to investigate opportunities for optimizing such algorithms.

To explore this subject, I explore various optimizations to the exhaustive depth-first search in the context of the knapsack problem, a well-known problem in the NP-hard set. Tests were performed on a diverse set of data, exploring the implications of a variety of presorts and a small optimization in stack structure used to manage the frontier. All tests and algorithms were implemented in Java 1.8.0\_112-b15. Runtime results were recorded in nanoseconds and each represent the average of one hundred trials, thereby ensuring reasonable accuracy between tests. Several tests were conducted, thereby generating several reports. Data for this paper will be taken from the report entitled report8.txt, one of two comprehensive reports generated by the test suite. Two input set proved problematic, and were consequently omitted from report8.txt – these will be discussed independently of the other results.

# BACKGROUND

Before addressing the implications of the generated reports, we must first discuss the challenge and parameters surrounding the knapsack problem, the basis for my tests.

The knapsack problem is a well-known NP-hard combinatorial problem. It is conceived as such: suppose there is a container of with limited capacity. Additionally, suppose there is some limited number of items of varying costs and values. The knapsack problem requires that the some set of items is returned such that the sum of item costs does not exceed the capacity of the “knapsack” container, while the sum of the items’ values is maximized [2].

There has been a wealth of research on this subject, as it represents an interesting problem in the mathematical subfield of combinatorics and has a variety of potential applications.

# METHODOLOGY

## Implementation

The metrics gathered for this research come from two separate tests, entitled “PhaseFour.java” and “PhaseFive.java”. As is likely evident from the naming conventions, the experiments were organized into several phases, the first three of which served as preliminary experiments before the tests conducted in phases four and five. Phase 1 sampled several imperfect, greedy solutions to the problem, and was tested against five, small test cases. Phase 2 implemented an exhaustive search, which was demonstrably correct in all cases but remarkably slow. Phase 3 demonstrated a simple tree-pruning technique. Using the greedy solutions generated in Phase 1, branches with insufficient value remaining in the tree were pruned, thus reducing the number of branches available in the tree. Heretofore, all test were conducted on a small set of five files, deemed the “original files”. Later tests were more robust, and were divided into “easy files” and “hard files”.

The implementations for the depth-first searching algorithms and the greedy algorithms used in this test were implemented in a public library and shared across the different testing modules. Additionally, the input data for each test is read from several directories of common input files shared across the different modules. This ensures that the results gathered from the tests across modules are reasonably comparable.

When choosing a unit of measurement for comparing runtimes, I felt that I should choose between measuring my findings in milliseconds and nanoseconds. Java has built-in support for these benchmarks. For the former operation, Java offers the “System.currentTimeMillis()” function. For the latter, Java provides “System.nanoTime()”. Because of the disparity between the runtimes of the pruned and unpruned searches, nanoseconds seemed excessively granular. Consequently, I decided that milliseconds would be more appropriate for the Phase 4 test. For the Phase 5 test, however, variations of the pruned search were compared against one another. Because the differences in these tests were more minute, it was sensible to use nanoseconds. This is reflected in the data to follow.

Lastly, for some graphs and analysis in this article, I made use of R version 3.3.1, the open-source statistical programming language.

## Sampling Techniques

Typically, when comparing the pruned and unpruned searches (as was done in Phase 4), the pruned search was several orders of magnitude faster than its unpruned counterpart. Additionally, when running the Phase 4 test repeatedly, the output was remarkably consistent. Therefore, it would have been superfluous to average several tests during Phase 4. Additionally, the variation between the pruned and unpruned techniques was not a primary item of interest, and thus was less worth of additional inquiry.

During Phase 5, the variation between tests became much less perceptible. Between tests, it was frequent that tends from one test would appear to dissipate in the next, leading to unhelpful and inconsistent data. Clearly, a more robust technique was necessary to ensure reliable results.

To achieve this, for Phase 5, I took the average of one hundred samples for each test. For tests dependent on a presort, I decided that the sorting process was pertinent to the total runtime of the modified algorithm, and included it in the runtime sample. I passed a fresh copy of the list of nodes on each iteration, thereby avoiding the possibility of passing a list of items sorted from a previous iteration.

After implementing these changes, the Phase 5 tests became significantly more stable and consistent. It is these averaged values that are used in the analysis later in this document.

# RESULTS

## Pruned vs Unpruned Search

The transition from an unpruned exhaustive search to a pruned search improved search time by several orders of magnitude. While the general result was not surprising, the size of the improvement was a bit unexpected.

The scale of this improvement is best demonstrated in the original test files, of which there are five. They are entitled “c01.csv”, “k05.csv”, “k10.csv”, “k24.csv”, and “k30.csv”. The first of these files I created myself, primarily to test the functionality of my system. The others, however, test larger and more interesting datasets; this is reflected in the naming conventions. There are five items in the second file, ten in the third, twenty-four in the fourth, and thirty in the fifth.

In the smallest of these files, the improvement is negligible. As seen in Figure I, for the datasets of length 5, the runtime improvement of the pruned search is nearly imperceptible. In other tests conducted while refining the output, it was actually observed that the unpruned search often ran a millisecond faster that its pruned counterpart. This is likely due to the extra branching logic necessitated by the pruning function. For the larger datasets, however, it become immediately evident that the pruned search is significantly faster.

For the list of length ten, the unpruned search runs for eight seconds while the pruned search runs for merely five. In some earlier tests, the pruned search ran for three seconds, suggesting that the recorded figure may be high. By the time the list reaches length twenty-four, the pruned search becomes many times quicker than the alternative. In this instance, the pruned search is 17.513 times faster than its unpruned counterpart. Depending on the ordering of the inputs and the quality of the greedy estimates, the improvement can be even starker. For “k30.csv”, the unpruned search is particularly slow, taking in 515,555 milliseconds in the selected test, and only 33 milliseconds in the pruned test. This represents 1,562,200% improvement for this test. This is – needless to say – remarkable.

Given the fact that this was a larger dataset, and yet the pruned time for the dataset of twenty-four items is actually fifteen times slower than that of the larger dataset, I am led to believe that the ordering of the data and the quality of the greedy estimates that prune the exhaustive search are dominant in shaping the outcome of enhanced search.

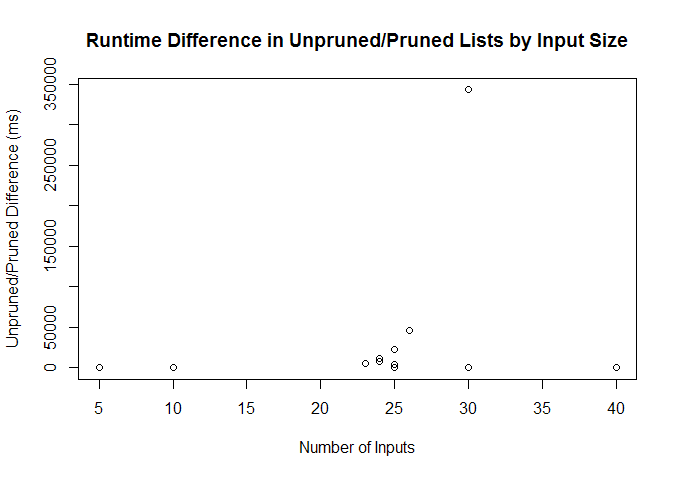
Figure I. Table Depicting Improvement Between Pruned and Unpruned Searches for Original Input Set

|  |  |  |  |
| --- | --- | --- | --- |
| **Length** | **Capacity** | **Unpruned Time (ms)** | **Pruned Time (ms)** |
| 5 | 10 | 3 | 3 |
| 5 | 13 | 2 | 2 |
| 10 | 30 | 8 | 5 |
| 24 | 60 | 8,704 | 497 |
| 30 | 96 | 515,555 | 33 |

This same procedure was applied systematically to both the “original” and “easy” datasets and converted into a CSV table for further processing, the results of which are seen in Figure II. Excepting two outlying data values, mentioned above, it appears that the difference in pruned and unpruned search times is likely exponential, hooking upward sharply between list lengths twenty-three and twenty-six.

Graphics for this data were generating using R, and open-source statistical programming language.

Figure II. Scatterplot Depicting Improvement Between Pruned and Unpruned Searches



## Elimination of Stack Overhead

The improvements garnered from the pruned search are relatively unsurprising. Consequently, I spent significant time implementing and analyzing additional optimizations for the pruned search.

The first of these optimizations was a small optimization in the stack structure used to manage the search tree. Clearly, when expanding a depth-first search, after each iteration, the deepest node in the frontier is expanded before any neighboring nodes are reached [1]. This is typically managed with a FIFO queue, or “stack”, which manages all nodes along the frontier. The first node added to the frontier is the first to be removed during frontier expansion, meaning that the memory burden of a search is a factor of the search space’s depth.

When expanding the search space, there are many ways to handle the stack. In my original implementation, once reaching a state stored on the stack, I expand all branching nodes. After the children are explored, the parent is removed from the stack. While this is functional, is unnecessarily inefficient. Once a parent stack has been reached, assuming that all children are expanded along the frontier, there is no utility in preserving that state in memory. This also necessitates more logic for determining if a state has been previously visited, leading to more branching logic in the body of the function.

In an effort to improve the performance of the function, I simplify my frontier maintenance, popping parent nodes off of the frontier stack immediately reaching them. This led to notable and consistent gains in performance, although these gains appear more poignant when the capacity for a particular input is lower. There also does not appear to be an exponential improvement with that accompanies the length of the input list, as with the previously discussed optimization.

Figure III. Table Depicting Improvements for Stack Optimization in Frontier Maintenance for A and B input files

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File** | **Length** | **Capacity** | **Control (ns)** | **Optimized (ns)** |
| A1.csv | 23 | 149 | 680,793 | 295,886 |
| B1.csv | 23 | 300 | 2,764,291 | 2,566,182 |
| A2.csv | 24 | 107 | 547,969 | 308,376 |
| B2.csv | 24 | 200 | 5,483,275 | 4,170,788 |
| A3.csv | 25 | 1116 | 171,727 | 126,618 |
| B3.csv | 25 | 1,637,333 | 713,829,142 | 617,150,726 |
| A4.csv | 26 | 900 | 3,133,731 | 2,609,037 |

Files with smaller capacities, such as “A1.csv” and “A2.csv” run 230.09% and 131.47% faster, respectively, when optimized. Inputs with larger capacities, such as “B3.csv”, however, do not demonstrate such an improvement. There are outliers to this generalization, however. The driving phenomenon behind these discrepancies could be a subject of later research.

# FUTURE WORK

Virtual compilers are enormously complex, and understanding the subtle optimizations conducted by the Java 1.8 compiler could be instrumental to understanding the performance of future tests. It is probable that the Java compiler makes optimization decisions that obfuscate the source of performance gains, thereby limiting the utility of tests such as these. This, in turn, could lead to more conclusive results.

# ACKNOWLEDGMENTS

I would like to thank Stuart Russell and Peter Norvig for their 2010 work, “Artificial Intelligence: A Modern Approach Third Edition”, which was invaluable in examining this subject. REFERENCES

1. Norvig, Peter, and Russel, Stuart 2010. Artificial Intelligence A Modern Approach Third Edition.

[2] <Https://Www.Utdallas.Edu/~Scniu/OPRE->

6201/Documents/DP3-Knapsack.Pdf. 1st ed. 2017. Print.