**November 2014**

CSC 342 – Exploration 6

James Francis/Sam Jentsch

Account Numbers: CSC342107/CSC342111

08

**Fall**

­­

Table of Contents

Problem Statement 3

Algorithm 3

Running our programs: 5

Implementation Similarities 5

Brute Force Algorithm Implementation 5

Dynamic Algorithm Implementation 5

Branch and Bound Algorithm Implementation 6

Experiment 6

Data sets 6

Issues 6

Machines 6

Timing Methods 7

Data Files and Sample Output 7

Charts and Timing Results 9

gprof 15

hprof 16

Conclusions 17

Future Work 18

Problems Encountered 18

Time Chart 18

Appendix A – C++ Implementation Listing: Brute Force 19

Appendix B – Java Implementation Listing: Brute Force 23

Appendix C – C++ Implementation Listing: Dynamic (No Memory Functions) 26

Appendix D – Java Implementation Listing: Dynamic (No Memory Functions) 29

Appendix E – C++ Implementation: Dynamic with Memory Functions 32

Appendix F – Java Implementation: Dynamic without Memory Functions 35

Appendix G – C++ Implementation: Branch and Bound 38

Appendix H – Java Implementation: Branch and Bound 43

Appendix I – References 47

# Problem Statement

The purpose of this assignment is to test several different methods for finding an optimal solution to the discrete version of the Knapsack problem. The methods examined include: Brute Force, Dynamic (with and without memory functions), and Branch & Bound approximation algorithms.

In the discrete version of the problem, there are no fractional items, so the entire item must be put in the sack to be included in the final solution.

# Algorithm

**Brute Force:**

The Brute Force algorithm works by generating all possible subsets of the set of items that are candidates for inclusion in the knapsack. The weight and value of each of these subsets are calculated. If a subset is feasible (the weight is less than the capacity of the sack) and the value is higher than the previously recorded highest value, that subset becomes the new optimal set. Each subset generated is processed in the manner just described and at the end of processing each subset the maximum value subset for the given capacity is returned.

*BruteForceKnapsack(weights, values, capacity)*

*Generate all permutations of values (Binary Reflected Gray Codes)*

*Process each permutation*

*If feasible (weight < capacity) and value > maxValue*

*maxValue = value*

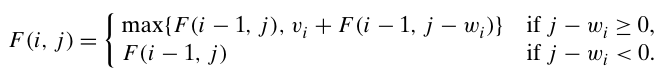
*maxSet = permutation*

*Now maxValue will contain the maximum value for the capacity and maxSet will be the optimal set of items*

**Dynamic:**

The Dynamic algorithms operate by recording smaller sub instances of the problem that are solved to find the solution in a table. These solutions can be used for later calculations so that the values will not need to be calculated again (instead the solution can be directly retrieved from the table).

An *nxW* (where n is the number of items and W is the overall capacity of the sack)table is constructed to hold intermediary solutions. The first row and column are filled with 0s. The value of each index in the table is found using the following formula:



where i and j correspond to the index of the table.

Both of the algorithms with and without memory functions use the formula and table described above.

**Without Memory Functions:**

*BottomUpKnapsack()*

*for each item in the knapsack*

*for each capacity from 1 to W*

*if j > Weights[i]*

*F[i, j] = max{F(i-1,j), Values[i] + F[i-1], j-Weights[i])}*

*else*

*F[i,j] = F[i-1,j]*

For the algorithm without memory functions, the solution is found using a bottom-up technique. All solutions preceding the final solution are generated to obtain the answer.

**With Memory Functions:**

*MFKnapsack(i, j)*

*If F[i,j] < 0*

*If j < Weights[i]*

*Value = MFKnapsack(i-1, j)*

*Else*

*Value = max(MFKnapsack(i-1, j), Values[i] + MFKnapsack(i-1, j-Weights[i]))*

*F[i,j] = Value*

*Return F[i,j]*

For the algorithm with memory functions, the solution is found using a top-down technique. Beginning with the index corresponding to the final solution, the solution is obtained by recursively calling only the sub-instances of the problem necessary to obtain the solution desired. These solutions are recorded in the table. Note that this table will likely have empty spaces (solutions not found during the algorithm’s process).

**Branch and Bound:**

The branch and bound technique works by generating a state space tree based on a best-first search. This best-first search is based on an upper bound that is determined for each node added to the tree.

The state space tree represents at each level a combination of items. At a level i, the tree’s left branch indicates the inclusion of item i, while its right branch indicates the exclusion of item i.

The upper bound for each node is determined using the following formula:

*ub = v + (W – w)(vi+1/wi+1)*

This upper bound dictates the nodes that are built first when constructing the tree.

The most promising node is built out at each level of the tree. When a leaf is reached, that leaf’s accumulated value along that path of the tree is returned. If this value is greater than any of the upper bounds for unexplored nodes, the algorithm has found the max value and ends. If the value is less than the upper bound of an unexplored node, that node is explored. This process is repeated until a maximum value is found that is greater than any other upper bound or the tree is exhausted.

### Running our programs:

**All files are submitted in account: cs342107**

The class files used for our exploration are as follows.

For C++:

To run the implementations using our data sets: run the following shell scripts:

knapsackBF.sh (For the Brute Force Implementation)

knapsackD.sh (For the Dynamic w/o Memory Function Implementation)

knapsackMF.sh (For the Dynamic w/ Memory Function Implementation)

knapsackBB.sh (For the Brand & Bound Implementation)

The generic form of executing any of the .cpp files is first compile by:

**g++ knapsack.cpp –o knapsack**

*Where knapsack.cpp is one of the knapsack c++ implementations submitted under the cs342107 account*

Then running the executable with your data file:

**./knapsack ../instr/datafile.dat**

Please ensure your data file is formatted in the following manner:

capacity

# of objects

weight value

weight value

weight value

weight value

An example with a capacity of 10 and 4 objects:

10

4

7 42

3 12

4 40

5 25

For JAVA:

To run the implementations using our data sets: run the following shell scripts:

knapsackBF\_java.sh (For the Brute Force Implementation)

knapsackD\_java.sh (For the Dynamic w/o Memory Function Implementation)

knapsackMF\_java.sh (For the Dynamic w/ Memory Function Implementation)

knapsackBB\_java.sh (For the Brand & Bound Implementation)

The generic form of executing any of the .java files is first compile by:

**javac knapsack.java**

*Where knapsack.java is one of the knapsack java implementations submitted under the cs342107 account*

Then running the executable with your data file:

**java knapsack ../instr/datafile.dat**

Please ensure your data file is formatted in the following manner:

capacity

# of objects

weight value

weight value

weight value

weight value

An example with a capacity of 10 and 4 objects:

10

4

7 42

3 12

4 40

5 25

### Implementation Similarities

All implementations run by passing a command line argument containing a path to a data file representing a knapsack. The data file matches the format described in the Data Sets section of the report.

### Brute Force Algorithm Implementation

The Brute Force algorithm uses the Binary Reflected Gray Code algorithm described in the text to generate all subsets of a set of items. In the C++ implementation, a vector of strings contains all of the subsets, while in the Java implementation an ArrayList of strings contains all of the subsets.

Strings are used to represent a subset (each string represents a possible subset of the items). A special append method was created to easily append to the beginning of a string in Java, while an iterator was used to insert at the beginning of the string in C++.

The container holding all of the subsets is iterated through and the weight/sum of each subset is calculated. The maximum feasible subset and value is returned.

### Dynamic Algorithm Implementation

We used a two dimensional integer array to represent the table for the dynamic algorithms. The implementations closely resemble the algorithms with and without memory functions. The array is simply processed from the first to last index for the bottom-up approach, or recursively from the last index down for the approach with memory functions. The Weights and Values of the items in the sack are stored in arrays for use in calculating the index value in both algorithms.

### Branch and Bound Algorithm Implementation

Used binary tree to represent the solution tree for branching and bounding. The Binary tree was created using a struct in C++ and a class in Java as a TreeNode. Additionally, an Item struct and class (for C++ and Java respectively) was created to hold all of the data associated with an item (weight, value, and weight-value ratio). These Items are stored in an array. The array is sorted by descending order of the Item’s weight-value ratios and then processed and built into a binary tree based on the algorithm described in the Algorithms section of the report.

# Experiment

The experiment was run on the Unix server and a Mac OS X machine for each of the implementations for analysis.

### Data sets

To test the algorithms, we constructed a set of data with increasing numbers of items. Beginning with 4, we increased by orders of 2 up to 128 items for testing. All of the algorithms we tested see a direct increase in runtime when the number of items is increased, so we believe this will allow for a comparison of the algorithms’ efficiencies. The capacities for all of these sets remained the same so that the affect of increasing the items input parameter could be measured.

The next set of data sets for our experiment is based on a hypothesis we had about the behavior of the algorithms. We decided that the Brute Force and Branch and Bound algorithms would be entirely unaffected by increasing the capacity of the sack (a maximum would be reached based on the available items), while the Dynamic algorithms would see an increase in running time as the capacity increased. This is due to the table size increasing with the size of the capacity.

To test this, we constructed a set of data with 4 identical sets of items and increasing capacities ranging from 10 to 100,000.

### Issues

### Machines

The machines used for testing include:

* **15” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2GHz quad-core Intel core i7
* **13” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2.4 GHz Intel Dual Core i5
* **SFA Unix Server**
  + OS: Linux 2.6.18-371.12.1.el5PAE i686
  + MEMORY: Undetermined (required sudo access)
  + PROCESSOR: eight-core Intel(R) Xeon(TM) MP CPU 3.66GHz

### Timing Methods

Due to The differences with time granularity between Java and C++ we tried to be as precise as possible while maintaining some level of uniformity between the data produced. Given that C++ can only report time in milliseconds using the tools provided for us in the handouts, and Java can use a nanosecond timer, giving us 1E06 more precision, we needed to convert the reported Java times and round to the nearest millisecond, to have results similar to what we were getting with C++. All the results reported herein are the result of either output from a C++ timer method provided or conversion to milliseconds from nanoseconds and rounding to two decimal places.

Timing was achieved through first preparing the algorithm for execution (reading in appropriate test data from the file and initializing variables), starting the timer, executing the methods implementing the algorithm, and stopping the timer.

### Data Files and Sample Output

The smaller data files (for 4 items) were selected from the text for verification of the program’s correctness. The remainder of the sets were created by hand with varying number’s of items for the first set, and increasing capacities for the second set.

Both of our data sets follow the following format and only differ in which input parameter was increased (items or capacity). The first item in the file is the capacity of the sack, and the second is the number of items in the sack. The values following are the weight/value pairs for each item:

4\_Items.dat

10

4

7 42

3 12

4 40

5 25

8\_Items.dat

10

8

7 42

3 12

4 40

5 25

6 39

2 17

9 20

8 3

For example, using 4\_Items.dat as an input file would generate the following output:

**MAX VALUE: 65**

This output matches the output produced in the text for the maximum value set of items.

### Charts and Timing Results

UNIX COMPARISONS

Charts

MAC COMPARISONS

### gprof

GPROF

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

100.06 0.01 0.01 1 10.01 10.01 BubbleSort(std::vector<Item, std::allocator<Item> >&)

0.00 0.01 0.00 61732 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item const\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 61732 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item const\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 47218 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 30866 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item const\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 30866 0.00 0.00 std::vector<Item, std::allocator<Item> >::end() const

0.00 0.01 0.00 30866 0.00 0.00 std::vector<Item, std::allocator<Item> >::size() const

0.00 0.01 0.00 30866 0.00 0.00 std::vector<Item, std::allocator<Item> >::begin() const

0.00 0.01 0.00 23966 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 23593 0.00 0.00 std::vector<Item, std::allocator<Item> >::begin()

0.00 0.01 0.00 23585 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> > >::operator+

0.00 0.01 0.00 23585 0.00 0.00 std::vector<Item, std::allocator<Item> >::\_M\_range\_check(unsigned int) const

0.00 0.01 0.00 23585 0.00 0.00 std::vector<Item, std::allocator<Item> >::at(unsigned int)

0.00 0.01 0.00 23585 0.00 0.00 std::vector<Item, std::allocator<Item> >::operator[](unsigned int)

0.00 0.01 0.00 968 0.00 0.00 TreeNode::TreeNode(double, int, int)

0.00 0.01 0.00 588 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> > >::base()

0.00 0.01 0.00 389 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> > >::operator+

0.00 0.01 0.00 278 0.00 0.00 bool \_\_gnu\_cxx::operator!=<Item\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 254 0.00 0.00 Item::~Item()

0.00 0.01 0.00 254 0.00 0.00 void std::\_Destroy<Item>(Item\*)

0.00 0.01 0.00 254 0.00 0.00 operator new(unsigned int, void\*)

0.00 0.01 0.00 127 0.00 0.00 \_\_gnu\_cxx::new\_allocator<Item>::construct(Item\*, Item const&)

0.00 0.01 0.00 127 0.00 0.00 std::vector<Item, std::allocator<Item> >::push\_back(Item const&)

0.00 0.01 0.00 127 0.00 0.00 void std::\_Construct<Item, Item>(Item\*, Item const&)

0.00 0.01 0.00 27 0.00 0.00 \_\_gnu\_cxx::new\_allocator<Item>::~new\_allocator()

0.00 0.01 0.00 26 0.00 0.00 \_\_gnu\_cxx::new\_allocator<Item>::new\_allocator(\_\_gnu\_cxx::new\_allocator<Item> const&)

0.00 0.01 0.00 26 0.00 0.00 std::allocator<Item>::~allocator()

0.00 0.01 0.00 25 0.00 0.00 std::allocator<Item>::allocator(std::allocator<Item> const&)

0.00 0.01 0.00 25 0.00 0.00 std::\_Vector\_base<Item, std::allocator<Item> >::\_M\_get\_Tp\_allocator()

0.00 0.01 0.00 16 0.00 0.00 std::vector<Item, std::allocator<Item> >::end()

0.00 0.01 0.00 16 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> > >

0.00 0.01 0.00 16 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 16 0.00 0.00 \_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item, std::allocator<Item> >

0.00 0.01 0.00 9 0.00 0.00 std::\_Vector\_base<Item, std::allocator<Item> >::\_M\_deallocate(Item\*, unsigned int)

0.00 0.01 0.00 8 0.00 0.00 \_\_gnu\_cxx::new\_allocator<Item>::deallocate(Item\*, unsigned int)

0.00 0.01 0.00 8 0.00 0.00 \_\_gnu\_cxx::new\_allocator<Item>::allocate(unsigned int, void const\*)

0.00 0.01 0.00 8 0.00 0.00 \_\_gnu\_cxx::new\_allocator<Item>::max\_size() const

0.00 0.01 0.00 8 0.00 0.00 std::vector<Item, std::allocator<Item> >::max\_size() const

0.00 0.01 0.00 8 0.00 0.00 std::\_Vector\_base<Item, std::allocator<Item> >::\_M\_allocate(unsigned int)

0.00 0.01 0.00 8 0.00 0.00 std::vector<Item, std::allocator<Item>

0.00 0.01 0.00 8 0.00 0.00 void std::\_\_destroy\_aux<\_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item,

0.00 0.01 0.00 8 0.00 0.00 void std::\_Destroy<\_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item,

0.00 0.01 0.00 8 0.00 0.00 void std::\_Destroy<\_\_gnu\_cxx::\_\_normal\_iterator<Item\*, std::vector<Item,

0.00 0.01 0.00 3 0.00 0.00 TreeNode::TreeNode()

0.00 0.01 0.00 2 0.00 0.00 Time1(int)

0.00 0.01 0.00 1 0.00 0.00 global constructors keyed to TreeNode

0.00 0.01 0.00 1 0.00 0.00 populateVariables(std::basic\_fstream<char, std::char\_traits<char> >&)

0.00 0.01 0.00 1 0.00 0.00 branchBoundKnapsack(TreeNode\*, int)

0.00 0.01 0.00 1 0.00 0.00 \_\_static\_initialization\_and\_destruction\_0(int, int)

0.00 0.01 0.00 1 0.00 0.00 \_\_gnu\_cxx::new\_allocator<Item>::new\_allocator()

0.00 0.01 0.00 1 0.00 0.00 std::allocator<Item>::allocator()

0.00 0.01 0.00 1 0.00 0.00 std::allocator<Item>::allocator(std::allocator<Item> const&)

C++ profiling, of Branch and Bound algorithm for 128 items. All of the CPU time was actually taken up by the BubbleSort used to sort the array of items (this sort isn’t included in the timing). The majority of calls associated with running the program are recursive calls to the BranchBound method for constructing the tree..

### hprof

HPROF

CPU SAMPLES BEGIN (total = 63) Mon Nov 24 21:28:20 2014

rank self accum count trace method

1 25.40% 25.40% 16 300068 java.util.zip.ZipFile.read

2 4.76% 30.16% 3 300180 com.sun.tools.javac.file.ZipFileIndex$ZipDirectory.readEntry

3 4.76% 34.92% 3 300179 java.lang.StringCoding$StringDecoder.decode

4 3.17% 38.10% 2 300032 java.util.zip.ZipFile.open

5 3.17% 41.27% 2 300060 java.lang.ClassLoader.defineClass1

6 3.17% 44.44% 2 300084 java.util.Arrays.copyOf

7 3.17% 47.62% 2 300144 java.lang.Thread.currentThread

8 3.17% 50.79% 2 300187 java.io.RandomAccessFile.readBytes0

9 1.59% 52.38% 1 300093 java.util.zip.ZipFile.freeEntry

10 1.59% 53.97% 1 300094 com.sun.tools.javac.main.JavaCompiler.<init>

11 1.59% 55.56% 1 300172 com.sun.tools.javac.parser.ParserFactory.newParser

12 1.59% 57.14% 1 300053 sun.net.www.URLConnection.<init>

13 1.59% 58.73% 1 300174 com.sun.tools.javac.file.ZipFileIndex.get4ByteLittleEndian

14 1.59% 60.32% 1 300061 java.lang.ClassLoader.findBootstrapClass

15 1.59% 61.90% 1 300178 java.util.HashMap.getEntry

16 1.59% 63.49% 1 300181 java.lang.ref.Reference$ReferenceHandler.run

17 1.59% 65.08% 1 300182 java.nio.charset.CharsetDecoder.maxCharsPerByte

18 1.59% 66.67% 1 300183 java.util.ComparableTimSort.binarySort

19 1.59% 68.25% 1 300184 java.lang.AbstractStringBuilder.<init>

20 1.59% 69.84% 1 300185 com.sun.tools.javac.util.Name.append

21 1.59% 71.43% 1 300186 java.util.ArrayList$Itr.next

22 1.59% 73.02% 1 300188 com.sun.tools.javac.util.Name.append

23 1.59% 74.60% 1 300189 com.sun.tools.javac.util.List.reverse

24 1.59% 76.19% 1 300190 com.sun.tools.javac.comp.Resolve.rawInstantiate

25 1.59% 77.78% 1 300191 com.sun.tools.javac.comp.Resolve.checkRawArgumentsAcceptable

26 1.59% 79.37% 1 300075 java.util.zip.ZipFile$ZipFileInputStream.read

27 1.59% 80.95% 1 300157 sun.net.www.ParseUtil.decode

28 1.59% 82.54% 1 300195 com.sun.tools.javac.jvm.ClassWriter.writeFields

29 1.59% 84.13% 1 300083 com.sun.tools.javac.util.Names.<init>

30 1.59% 85.71% 1 300139 java.util.HashMap.getEntry

31 1.59% 87.30% 1 300063 java.util.zip.ZipFile.getEntry

32 1.59% 88.89% 1 300199 java.lang.Shutdown.halt0

33 1.59% 90.48% 1 300086 com.sun.tools.javac.code.Type$ErrorType.<init>

34 1.59% 92.06% 1 300087 sun.misc.Unsafe.getObjectVolatile

35 1.59% 93.65% 1 300088 java.net.URL.equals

36 1.59% 95.24% 1 300089 java.util.zip.ZipFile.freeEntry

37 1.59% 96.83% 1 300090 java.lang.String.toCharArray

38 1.59% 98.41% 1 300091 java.lang.AbstractStringBuilder.<init>

39 1.59% 100.00% 1 300092 java.net.URL.getPort

CPU SAMPLES END

We ran hprof on the Branch and Bound algorithm for 128 items.

# Conclusions

**Brute Force:**

Time efficiency of the Brute Force Algorithm is dependent on the amount of subsets that must be generated to obtain the optimal solution. 2n subsets must always be generated to find the solution, so the algorithm is O(2n).

We found that the time required for the Brute Force algorithm seemed to increase exponentially for both the Java and C++ implementations on both machines. On both the C++ and Java implementations, on both machines tested, the Brute Force algorithm ran out of heap space for data sets with 32 or more items. We believe this is do to the enormous amount of data sets required to obtain the solution to the knapsack problem for 32 items. The number of permutations required for a set of 32 is 232 = 4,294,967,296. A

**Dynamic Algorithms:**

Before we began the experiment, we noted that the efficiency of the dynamic algorithms is affected by increased capacity while the Brute Force and Branch and Bound algorithms are not. This is because the size of the table required for the dynamic algorithms increase as the capacity increases. In our tests, we saw this hypothesis mostly confirmed. It was especially evident in the results for the Dynamic algorithm without Memory Functions, where we reached up to 225ms for a capacity of 100,000 on the Unix machine (Java), compared to 0ms for the Branch and Bound algorithm for the same capacity and data set on the same machine (Java).

The Dynamic algorithm with Memory Functions performed exceptionally well, even for the increased capacities. On the Unix machine, the times recorded were reported as 0ms for all the capacities tested. In this case, the Dynamic Algorithm with Memory functions seemed to have a clear edge on the algorithm without memory functions.

Time and space efficiency of the dynamic algorithm without memory functions is O(nW) and the time needed to find the composition of the optimal solution is O(n)

The Time and space efficiency of the dynamic algorithm with memory functions is also O(nW), and generally more than a constant-factor efficiency gain cannot be expected, although in our case when testing the increased capacities the algorithm with memory functions *seemed* to exhibit more than a constant factor efficiency increase.

One note we made about the dynamic algorithms is that both also require additional computation to find the items comprising the optimal set, while both the Brute Force and Branch & Bound algorithms determine the composition of the optimal set when the highest value is found.

**Branch and Bound:**

For both the OSX and Java implementations, the Branch and bound algorithm was clearly more time efficient than the Brute Force implementation of the algorithms. On Unix in C++, the Branch and Bound algorithm required only 6ms for a set of 128 items, while the equivalent Brute Force algorithm required 195ms for a set of just 16 items. For a direct comparison, the set of 16 items required only 1ms, so it was 128 times faster in this case than the Brute Force algorithm.

The difference in time efficiency was not as drastic when comparing the Branch and Bound algorithm to the dynamic algorithms. Somewhat surprisingly, the Dynamic algorithm without Memory Functions was actually faster than the Branch & Bound algorithm on the Unix machine for the C++ implementation (6ms for the Branch & Bound, 0ms for the Dynamic). We believe this is due to the amount of recursive calls required for the Branch and Bound implementation, compared to the relatively simple process of iterating through an array required for the dynamic algorithm. Unsurprisingly, the same result holds for the Dynamic algorithm with Memory Functions.

### Future Work

We would like to put some work into fine-tuning the efficiency of the Branch and Bound algorithm implementation. While researching, we found a couple of implementations that used a priority queue to implement the algorithm. We decided to use a binary tree, but would be interested in seeing the difference in efficiencies between the different methods of implementation.

### Problems Encountered

Crashing in Brute Force (running out of heap space). We found that the Brute Force algorithm crashed for any data sets consisting of 32 or more items. This became a slight issue in collecting and examining our data.

The Brute Force algorithm was at one point using 41GB of virtual memory on the OSX machine before crashing.

We were hoping to see more clear trends in our data for the Dynamic algorithms and the Branch and Bound algorithms. Both algorithms seemed to run extremely quickly for the data sets we tested.

### Time Chart

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sunday | Monday | Tuesday |
| James | 3.5 hrs | 6 hrs | 3 hrs |
| Sam | 3.5 hrs | 6 hrs | 3 hrs |

# Appendix A – C++ Implementation Listing: Brute Force

//

// Knapsack\_BF.cpp

// Knapsack\_BruteForce

//

// Created by Sam Jentsch on 11/20/14.

// Copyright (c) 2014 Sam Jentsch. All rights reserved.

//

#include <iostream>

#include <vector>

#include <fstream>

#include <iomanip>

#include <sys/timeb.h>

#include <time.h>

#define START 0

#define END 1

int capacity;

int items;

std::vector<int> weights;

std::vector<int> values;

using namespace std;

string knapsack(vector<int> weights, vector<int> values, int capacity);

std::vector<string> BRGC(int n);

void printVect(vector<string> vect);

void populateVariables(fstream& file);

int Time1(int);

int main(int argc, const char \* argv[]) {

//Given n items of known weights and values and a knapsack of capapcity W,

//find the most valuable subset of items that fit into the knapsack.

//Generate all subsets of the set of n items given, computing the total weight

//of each subset to identify feasible subsets and find the subset with the largest

//value among them.

//Number of subsets of an n element set is 2^n

fstream infile(argv[1], ios::in);

if (!infile.is\_open()) {

cout<<"File not found."<<endl;

return -1;

}

populateVariables(infile);

//-----TIMING-----//

int time = 0;

time = Time1(START);

string solution = knapsack(weights, values, capacity);

time = Time1(END);

cout << "\nTIME: " << time << endl;

cout << solution << endl;

return 0;

}

string knapsack(vector<int> weights, vector<int> values, int capacity) {

string bestSet;

int bestValue = 0;

vector<string> subsets = BRGC((int)weights.size());

printVect(subsets);

for (int i = 0; i < subsets.size(); i++) {

string set = subsets.at(i);

int setValue = 0;

int setWeight = 0;

bool isFeasible = true;

//cout << "Checking set: " << set << endl;

for (int j = 0; j < set.size() && isFeasible; j++) {

if (set.at(j) == '1') {

//Value at position j is in the subset.

setValue += values.at(j);

setWeight += weights.at(j);

}

if (setWeight > capacity) {

isFeasible = false;

}

}

if (setValue > bestValue && isFeasible) {

bestValue = setValue;

bestSet = set;

}

}//end go through all subsets.

cout << "Best Value: " << bestValue << endl;

return bestSet;

}

vector<string> BRGC(int n) {

//Generates the binary reflected gray code of order n

//Input: positive integer n

//Output: A list of all bit strings of length n composing the gray code.

vector<string> L;

if (n == 1) {

//make list L containing bit strings 0 and 1 in this order.

L.push\_back("0");

L.push\_back("1");

//printVect(L);

} else {

//Generate list L1 of bit strings of size n-1 by calling BRGC(n-1)

vector<string> L1 = BRGC(n-1);

//cout << "N = " << n << " L1 size: " << L1.size() << "\n";

//Copy list L1 to List L2 in reverse order.

vector<string> L2;

for (int i = (int)L1.size() - 1; i >= 0; i--) {

//cout << "i = " << i << "\n";

L2.push\_back(L1.at(i));

}

//Add 0 in front of each bit string in List L1 and 1 in front of each

//bit string in L2.

for (int i = 0; i < L1.size(); i++) {

L1.at(i).insert(0, "0");

//cout << "L1." << i << " is now " << L1.at(i) << endl;

L2.at(i).insert(0, "1");

//cout << "L2." << i << " is now " << L2.at(i) << endl;

}

//Append L2 to L1 to get list L.

for (int i = 0; i < L2.size(); i++) {

L1.push\_back(L2.at(i));

}

//L is the combination of L2 and L1.

L = L1;

}

return L;

}

void printVect(vector<string> vect) {

cout << "PRINT VECT\n";

for (int i = 0; i < vect.size(); i++) {

cout << vect.at(i);

cout << endl;

}

}

void populateVariables(fstream& file){

int count = 0;

int i = 0;

int value = 0;

while(file >> value){

if (count==0){

capacity= value;

cout<<"Capacity = "<<capacity<<endl;

//System.out.println("Capacity is:"+capacity);

count++;

file>>value;

}

if (count==1){

items = value;

cout<<"Items = "<<items<<endl;

//System.out.println(items+" items");

count++;

}

else if(count>1){

weights.push\_back(value);

file >>value;

values.push\_back(value);

cout<<weights[i]<<" "<<values[i]<<endl;

count++;

i++;

}

}

file.close();

}

int Time1(int flag) {

//Timing method described in assignment handout.

static struct timeb time1;

struct timeb time2;

int sec;

unsigned short mil;

if (flag == START) {

ftime(&time1);

return 0;

} else {

ftime(&time2);

if (time2.millitm < time1.millitm) {

time2.time--;

mil = time2.millitm + 1000 - time1.millitm;

} else {

mil = time2.millitm - time1.millitm;

}

sec = (int)time2.time - (int)time1.time;

return sec \* 1000 + (int)mil;

}

}

# Appendix B – Java Implementation Listing: Brute Force

//Java\_Knapsack\_BF.java

import java.util.\*;

import java.lang.\*;

import java.io.\*;

public class Java\_Knapsack\_BF {

static int[] weights;

static int[] values;

static int items;

static int capacity;

public static void main(String[] args) throws FileNotFoundException {

Scanner infile = new Scanner(new File(args[0]));

populateVariables(infile);

int weights2[] = weights;

int values2[] = values;

//start time

long startTime = System.nanoTime();

long stopTime = System.nanoTime();

long overTime = stopTime - startTime;

startTime = System.nanoTime();

String solution = knapsack(weights2, values2);

System.out.println(solution);

stopTime = System.nanoTime();

//Stop time for the algorithm

long time = stopTime - startTime - overTime;

System.out.println("Total time is: " +(time/1000000));

}

private static String knapsack(int[] weights2, int[] values2) {

String bestSet = "";

int bestValue = 0;

ArrayList<String> subsets = BRGC((int)weights2.length);

//printVect(subsets);

for (int i = 0; i < subsets.size(); i++) {

String set = subsets.get(i).toString();

int setValue = 0;

int setWeight = 0;

boolean isFeasible = true;

//cout << "Checking set: " << set << endl;

for (int j = 0; j < set.length() && isFeasible; j++) {

if (set.charAt(j) == '1') {

//Value at position j is in the subset.

setValue += values2[j];

setWeight += weights2[j];

}

if (setWeight > capacity) {

isFeasible = false;

}

}

if (setValue > bestValue && isFeasible) {

bestValue = setValue;

bestSet = set;

}

}//end go through all subsets.

System.out.println("Best Value: "+bestValue);

return bestSet;

}

static ArrayList<String> BRGC(int n) {

//Generates the binary reflected gray code of order n

//Input: positive integer n

//Output: A list of all bit strings of length n composing the gray code.

ArrayList<String> L = new ArrayList();

if (n == 1) {

//make list L containing bit strings 0 and 1 in this order.

L.add("0");

L.add("1");

//printVect(L);

} else {

//Generate list L1 of bit strings of size n-1 by calling BRGC(n-1)

ArrayList<String> L1 = BRGC(n-1);

//Copy list L1 to List L2 in reverse order.

ArrayList<String> L2 = new ArrayList();

for (int i = (int)L1.size() - 1; i >= 0; i--) {

L2.add(L1.get(i));

}

//Add 0 in front of each bit string in List L1 and 1 in front of each

//bit string in L2.

for (int i = 0; i < L1.size(); i++) {

L1.set(i,append("0",L1.get(i)));

L2.set(i,append("1",L2.get(i)));

}

//Append L2 to L1 to get list L.

for (int i = 0; i < L2.size(); i++) {

L1.add(L2.get(i));

}

//L is the combination of L2 and L1.

L = L1;

}

return L;

}

static void printVect(ArrayList<String> vect) {

System.out.println("PRINT LIST");

for (int i = 0; i < vect.size(); i++) {

System.out.println(vect.get(i));

}

System.out.println();

}

static String append(String value, String str){

value += str;

return value;

}

static void populateVariables(Scanner file){

int count = 0;

int i = 0;

while(file.hasNext()){

if (count==0){

capacity = file.nextInt();

//System.out.println("Capacity is:"+capacity);

count++;

}

if (count==1){

items = file.nextInt();

//System.out.println(items+" items");

values = new int[items];

weights = new int[items];

count++;

}

else{

weights[i] = file.nextInt();

values[i] = file.nextInt();

count++;

i++;

}

}

file.close();

}

}

# Appendix C – C++ Implementation Listing: Dynamic (No Memory Functions)

//

// Knapsack\_Dynamic.cpp

// Knapsack\_Dynamic

//

// Created by Sam Jentsch on 11/21/14.

// Copyright (c) 2014 Sam Jentsch. All rights reserved.

//

#include <iostream>

#include <iomanip>

#include <vector>

#include <fstream>

#include <sys/timeb.h>

#include <time.h>

#define START 0

#define END 1

using namespace std;

int \*\*dynamicTable;

int capacity;

int items;

vector<int> weights;

vector<int> values;

void printTable();

int max(int num1, int num2);

void bottomUpKnapsack(vector<int> weights, vector<int> values);

void populateVariables(fstream& file);

int Time1(int);

int main(int argc, const char \* argv[]) {

fstream infile(argv[1], ios::in);

if (!infile.is\_open()) {

cout<<"File not found."<<endl;

return -1;

}

populateVariables(infile);

//---INITIALIZE TABLE---//

dynamicTable = new int \*[items + 1];

for (int i = 0; i <= items; i++) {

dynamicTable[i] = new int[capacity + 1];

}

for (int i = 0; i <= items; i++) {

for (int j = 0; j <= capacity; j++) {

dynamicTable[i][j] = 0;

}

}

//printTable();

//-----TIMING-----//

int time = 0;

time = Time1(START);

//warshall(adjMatrix, n);

time = Time1(END);

cout << "\nTIME: " << time << endl;

bottomUpKnapsack(weights, values);

printTable();

return 0;

}

void bottomUpKnapsack(vector<int> weights, vector<int> values) {

//if j - iWeight >= 0, F(i, j) = max{F(i-1,j), iValue + F(i-1), j-iWeight)}

//else F(i,j) = F(i-1,j)

for (int i = 1; i <= items; i++) {

for (int j = 1; j <= capacity; j++) {

if (j - weights.at(i-1) >= 0) {

//F(i, j) = max{F(i-1,j), iValue + F(i-1), j-iWeight)}

dynamicTable[i][j] = max(dynamicTable[i-1][j], values.at(i-1) + dynamicTable[i-1][j-weights.at(i-1)]);

} else {

//F(i,j) = F(i-1,j)

dynamicTable[i][j] = dynamicTable[i-1][j];

}

}

}

}

int max(int num1, int num2) {

if (num1 >= num2) {

return num1;

} else {

return num2;

}

}

void printTable() {

cout << "PRINT TABLE\n";

for (int iCount = 0; iCount <= items; iCount++) {

for (int jCount = 0; jCount <= capacity; jCount++) {

cout<<setw(3)<< dynamicTable[iCount][jCount];

}

cout << endl;

}

}

void populateVariables(fstream& file){

int count = 0;

int i = 0;

int value = 0;

while(file >> value){

if (count==0){

capacity= value;

cout<<"Capacity = "<<capacity<<endl;

//System.out.println("Capacity is:"+capacity);

count++;

file>>value;

}

if (count==1){

items = value;

cout<<"Items = "<<items<<endl;

//System.out.println(items+" items");

count++;

}

else if(count>1){

weights.push\_back(value);

file >>value;

values.push\_back(value);

cout<<weights[i]<<" "<<values[i]<<endl;

count++;

i++;

}

}

file.close();

}

int Time1(int flag) {

//Timing method described in assignment handout.

static struct timeb time1;

struct timeb time2;

int sec;

unsigned short mil;

if (flag == START) {

ftime(&time1);

return 0;

} else {

ftime(&time2);

if (time2.millitm < time1.millitm) {

time2.time--;

mil = time2.millitm + 1000 - time1.millitm;

} else {

mil = time2.millitm - time1.millitm;

}

sec = (int)time2.time - (int)time1.time;

return sec \* 1000 + (int)mil;

}

}

# Appendix D – Java Implementation Listing: Dynamic (No Memory Functions)

//Java\_Knapsack\_Dynamic.java

import java.io.\*;

import java.util.\*;

public class Java\_Knapsack\_Dynamic {

static int[] weights;

static int[] values;

static int items;

static int capacity;

static int dynamicTable[][];

public static void main(String[] args) throws FileNotFoundException {

Scanner infile = new Scanner(new File(args[0]));

populateVariables(infile);

//---INITIALIZE TABLE---//

for (int i = 0; i <= items; i++) {

dynamicTable[i] = new int[capacity + 1];

}

for (int i = 0; i <= items; i++) {

for (int j = 0; j <= capacity; j++) {

if (i == 0 || j == 0) {

dynamicTable[i][j] = 0;

} else {

dynamicTable[i][j] = -1;

}

}

}

//printTable();

long startTime = System.nanoTime();

long stopTime = System.nanoTime();

long overTime = stopTime - startTime;

startTime = System.nanoTime();

bottomUpKnapsack(dynamicTable,weights, values, items, capacity);

stopTime = System.nanoTime();

//Stop time for the algorithm

long time = stopTime - startTime - overTime;

// printTable(dynamicTable,items,capacity);

System.out.println("Total time is: " +(time/1000000));

}

static void bottomUpKnapsack(int dynamicTable[][],int weights[], int values[],int items,int capacity) {

//if j - iWeight >= 0, F(i, j) = max{F(i-1,j), iValue + F(i-1), j-iWeight)}

//else F(i,j) = F(i-1,j)

for (int i = 1; i <= items; i++) {

for (int j = 1; j <= capacity; j++) {

if (j - weights[i-1] >= 0) {

//F(i, j) = max{F(i-1,j), iValue + F(i-1), j-iWeight)}

dynamicTable[i][j] = max(dynamicTable[i-1][j], values[i-1] + dynamicTable[i-1][j-weights[i-1]]);

} else {

//F(i,j) = F(i-1,j)

dynamicTable[i][j] = dynamicTable[i-1][j];

}

}

}

}

static int max(int num1, int num2) {

if (num1 >= num2) {

return num1;

} else {

return num2;

}

}

static void printTable(int dynamicTable[][], int items, int capacity) {

System.out.println("PRINT TABLE");

for (int iCount = 0; iCount <= items; iCount++) {

for (int jCount = 0; jCount <= capacity; jCount++) {

System.out.printf("%3d",dynamicTable[iCount][jCount]);

}

System.out.println();

}

}

static void populateVariables(Scanner file){

int count = 0;

int i = 0;

while(file.hasNext()){

if (count==0){

capacity = file.nextInt();

//System.out.println("Capacity is:"+capacity);

count++;

}

if (count==1){

items = file.nextInt();

values = new int[items];

weights = new int[items];

dynamicTable = new int [items+1][capacity+1];

count++;

}

else{

weights[i] = file.nextInt();

values[i] = file.nextInt();

count++;

i++;

}

}

file.close();

}

}

# Appendix E – C++ Implementation: Dynamic with Memory Functions

//

// Knapsack\_MF.cpp

// Knapsack\_Dynamic

//

// Created by Sam Jentsch on 11/21/14.

// Copyright (c) 2014 Sam Jentsch. All rights reserved.

//

#include <iostream>

#include <vector>

#include <iomanip>

#include <fstream>

#include <sys/timeb.h>

#include <time.h>

#define START 0

#define END 1

using namespace std;

int \*\*dynamicTable;

int capacity;

int items;

vector<int> weights;

vector<int> values;

void printTable();

int max(int num1, int num2);

int MFKnapsack(int i, int j);

void populateVariables(fstream& file);

int Time1(int);

int main(int argc, const char \* argv[]) {

fstream infile(argv[1], ios::in);

if (!infile.is\_open()) {

cout<<"File not found."<<endl;

return -1;

}

populateVariables(infile);

//---INITIALIZE TABLE---//

dynamicTable = new int \*[items + 1];

for (int i = 0; i <= items; i++) {

dynamicTable[i] = new int[capacity + 1];

}

for (int i = 0; i <= items; i++) {

for (int j = 0; j <= capacity; j++) {

if (i == 0 || j == 0) {

dynamicTable[i][j] = 0;

} else {

dynamicTable[i][j] = -1;

}

}

}

//printTable();

//-----TIMING-----//

int time = 0;

time = Time1(START);

//warshall(adjMatrix, n);

time = Time1(END);

cout << "\nTIME: " << time << endl;

cout << MFKnapsack(items, capacity) << endl;

printTable();

return 0;

}

int MFKnapsack(int i, int j) {

int value = 0;

if (dynamicTable[i][j] < 0) {

if (j < weights[i-1]) {

value = MFKnapsack(i-1, j);

} else {

value = max(MFKnapsack(i-1, j), values[i-1] + MFKnapsack(i-1, j-weights[i-1]));

}

dynamicTable[i][j] = value;

}

return dynamicTable[i][j];

}

int max(int num1, int num2) {

if (num1 >= num2) {

return num1;

} else {

return num2;

}

}

void printTable() {

cout << "PRINT TABLE\n";

for (int iCount = 0; iCount <= items; iCount++) {

for (int jCount = 0; jCount <= capacity; jCount++) {

cout <<setw(3)<< dynamicTable[iCount][jCount];

}

cout << endl;

}

}

void populateVariables(fstream& file){

int count = 0;

int i = 0;

int value = 0;

while(file >> value){

if (count==0){

capacity= value;

cout<<"Capacity = "<<capacity<<endl;

//System.out.println("Capacity is:"+capacity);

count++;

file>>value;

}

if (count==1){

items = value;

cout<<"Items = "<<items<<endl;

//System.out.println(items+" items");

count++;

}

else if(count>1){

weights.push\_back(value);

file >>value;

values.push\_back(value);

cout<<weights[i]<<" "<<values[i]<<endl;

count++;

i++;

}

}

file.close();

}

int Time1(int flag) {

//Timing method described in assignment handout.

static struct timeb time1;

struct timeb time2;

int sec;

unsigned short mil;

if (flag == START) {

ftime(&time1);

return 0;

} else {

ftime(&time2);

if (time2.millitm < time1.millitm) {

time2.time--;

mil = time2.millitm + 1000 - time1.millitm;

} else {

mil = time2.millitm - time1.millitm;

}

sec = (int)time2.time - (int)time1.time;

return sec \* 1000 + (int)mil;

}

}

# Appendix F – Java Implementation: Dynamic without Memory Functions

//Java\_Knapsack\_MF.java

import java.io.\*;

import java.util.\*;

public class Java\_Knapsack\_MF {

static int[] weights;

static int[] values;

static int items;

static int capacity;

static int dynamicTable[][];

public static void main(String[] args) throws FileNotFoundException {

Scanner infile = new Scanner(new File(args[0]));

populateVariables(infile);

//---INITIALIZE TABLE---//

for (int i = 0; i <= items; i++) {

dynamicTable[i] = new int[capacity + 1];

}

for (int i = 0; i <= items; i++) {

for (int j = 0; j <= capacity; j++) {

if (i == 0 || j == 0) {

dynamicTable[i][j] = 0;

} else {

dynamicTable[i][j] = -1;

}

}

}

//printTable();

long startTime = System.nanoTime();

long stopTime = System.nanoTime();

long overTime = stopTime - startTime;

startTime = System.nanoTime();

System.out.println(MFKnapsack(dynamicTable,weights,values,items, capacity));

stopTime = System.nanoTime();

//Stop time for the algorithm

long time = stopTime - startTime - overTime;

System.out.println("Total time is: " +(time/1000000));

printTable(dynamicTable,items,capacity);

}

static int MFKnapsack(int dynamicTable[][],int weights[],int values[],int i, int j) {

int value = 0;

if (dynamicTable[i][j] < 0) {

if (j < weights[i-1]) {

value = MFKnapsack(dynamicTable,weights,values,i-1, j);

} else {

value = max(MFKnapsack(dynamicTable,weights,values,i-1, j), values[i-1] + MFKnapsack(dynamicTable,weights,values,i-1, j-weights[i-1]));

}

dynamicTable[i][j] = value;

}

return dynamicTable[i][j];

}

static int max(int num1, int num2) {

if (num1 >= num2) {

return num1;

} else {

return num2;

}

}

static void printTable(int dynamicTable[][], int items, int capacity) {

System.out.println("PRINT TABLE");

for (int iCount = 0; iCount <= items; iCount++) {

for (int jCount = 0; jCount <= capacity; jCount++) {

System.out.printf(" %3d",dynamicTable[iCount][jCount]);

}

System.out.println();

}

}

static void populateVariables(Scanner file){

int count = 0;

int i = 0;

while(file.hasNext()){

if (count==0){

capacity = file.nextInt();

count++;

}

if (count==1){

items = file.nextInt();

values = new int[items];

weights = new int[items];

dynamicTable = new int [items+1][capacity+1];

count++;

}

else{

weights[i] = file.nextInt();

values[i] = file.nextInt();

count++;

i++;

}

}

file.close();

}

}

# Appendix G – C++ Implementation: Branch and Bound

//

// Knapsack\_BranchBound.cpp

// Knapsack\_BranchBound

//

// Created by Sam Jentsch on 11/23/14.

// Copyright (c) 2014 Sam Jentsch. All rights reserved.

//

#include <iostream>

#include <vector>

#include <iomanip>

#include <fstream>

#include <sys/timeb.h>

#include <time.h>

#define START 0

#define END 1

using namespace std;

struct TreeNode {

TreeNode() {

upperBound = 0;

totalWeight = 0;

totalValue = 0;

leftChild = NULL;

rightChild = NULL;

}

TreeNode(double ub, int tw, int tv) {

upperBound = ub;

totalWeight = tw;

totalValue = tv;

leftChild = NULL;

rightChild = NULL;

}

double upperBound;

int totalWeight;

int totalValue;

TreeNode \*leftChild;

TreeNode \*rightChild;

} TreeNode;

struct Item {

int weight;

int value;

double WVRatio;

} Item;

struct TreeNode root;

vector<struct Item> items;

int capacity;

int branchBoundKnapsack(struct TreeNode \*currentNode, int nodeLevel);

void BubbleSort(vector<struct Item> &vect);

void populateVariables(fstream& file);

int Time1(int);

int main(int argc, const char \* argv[]) {

//Sort items in descending order by their given value-to-weight

//ratios.

//First item gives best payoff per wight unit and last gives the

//worst payoff per weight unit.

//A branch going to the left indicates the inclusion of the next item,

//a branch going to the right indicates the exclusion of the next item.

//ub = v + (W - w)(nextI\_V / nexti\_W)

fstream infile("../instr/list4.dat", ios::in);

if (!infile.is\_open()) {

cout<<"File not found."<<endl;

return -1;

}

populateVariables(infile);

//items = {{3, 12}, {7, 42}, {5, 25}, {4, 40}};

//capacity = 10;

for (int i = 0; i < items.size(); i++) {

items.at(i).WVRatio = (double)items.at(i).value/items.at(i).weight;

//cout << items.at(i).WVRatio << endl;

}

BubbleSort(items);

root = \*new struct TreeNode();

//-----TIMING-----//

int time = 0;

time = Time1(START);

int max = branchBoundKnapsack(&root, 0);

cout << "FINAL VALUE: " << max << endl;

time = Time1(END);

cout << "\nTIME: " << time << endl;

return 0;

}

int branchBoundKnapsack(struct TreeNode \*currentNode, int nodeLevel) {

int value = 0;

//ub = v + (W-w)(vi+1/wi+1)

int valueIncludeI = (currentNode->totalValue+items.at(nodeLevel).value);

int weightIncludeI = (currentNode->totalWeight+items.at(nodeLevel).weight);

if (nodeLevel+1 >= items.size()) {

//Base case

//Current Node is a leaf.

int maxValue;

if (valueIncludeI > currentNode->totalValue && (weightIncludeI <= capacity)) {

maxValue = valueIncludeI;

} else {

maxValue = currentNode->totalValue;

}

//cout << "MAX VALUE: " << maxValue << endl;

return maxValue;

} else {

//Current node is not a leaf.

double upperBoundLeft = valueIncludeI + (capacity - weightIncludeI)\*(items.at(nodeLevel+1).WVRatio);

double upperBoundRight = currentNode->totalValue + (capacity - currentNode->totalWeight)\*(items.at(nodeLevel+1).WVRatio);

cout << "Upperbounds: " << upperBoundLeft << " " << upperBoundRight << endl;

struct TreeNode \*leftChild = new struct TreeNode(upperBoundLeft, weightIncludeI, valueIncludeI);

struct TreeNode \*rightChild = new struct TreeNode(upperBoundRight, currentNode->totalWeight, currentNode->totalValue);

currentNode->leftChild = leftChild;

currentNode->rightChild = rightChild;

//----Select Next Path----//

int leftValue = -1;

int rightValue = -1;

if ((upperBoundLeft > upperBoundRight) && weightIncludeI <= capacity) {

leftValue = branchBoundKnapsack(currentNode->leftChild, nodeLevel + 1);

//Make sure it was the best

if (leftValue < upperBoundRight) {

rightValue = branchBoundKnapsack(currentNode->rightChild, nodeLevel + 1);

}

} else {

rightValue = branchBoundKnapsack(currentNode->rightChild, nodeLevel + 1);

//Make sure it was the best

if (rightValue < upperBoundLeft && weightIncludeI <= capacity) {

leftValue = branchBoundKnapsack(currentNode->leftChild, nodeLevel + 1);

}

}

//Get the max from the 2 branches

if (leftValue >= rightValue) {

value = leftValue;

} else {

value = rightValue;

}

}

return value;

}

void BubbleSort(vector<struct Item> &vect) {

int i, j, flag = 1; // set flag to 1 to start first pass

struct Item temp; // holding variable

for(i = 1; i <= vect.size() && flag; i++) {

flag = 0;

for (j=0; j < (vect.size() - 1); j++)

{

if (vect.at(j+1).WVRatio > vect.at(j).WVRatio) {

temp = vect.at(j);

vect.at(j) = vect.at(j+1);

vect.at(j+1) = temp;

flag = 1; // indicates that a swap occurred.

}

}

}

return; //arrays are passed to functions by address; nothing is returned

}

void populateVariables(fstream& file){

int count = 0;

int value = 0;

while(file >> value){

if (count == 0){

capacity= value;

//cout << "Capacity = " << capacity << endl;

count++;

} else if (count==1){

//int numItems = value;

count++;

} else if(count>1){

int weight;

weight = value;

file >> value;

struct Item newItem = {weight, value, 0};

//cout << "Adding new Item. Weight: " << newItem.weight << " Value: " << newItem.value << endl;

items.push\_back(newItem);

count++;

}

}

file.close();

}

int Time1(int flag) {

//Timing method described in assignment handout.

static struct timeb time1;

struct timeb time2;

int sec;

unsigned short mil;

if (flag == START) {

ftime(&time1);

return 0;

} else {

ftime(&time2);

if (time2.millitm < time1.millitm) {

time2.time--;

mil = time2.millitm + 1000 - time1.millitm;

} else {

mil = time2.millitm - time1.millitm;

}

sec = (int)time2.time - (int)time1.time;

return sec \* 1000 + (int)mil;

}

}

# Appendix H – Java Implementation: Branch and Bound

//Java\_Knapsack\_BranchBound.java

import java.util.\*;

import java.io.\*;

public class Java\_Knapsack\_BranchBound {

static TreeNode root;

static ArrayList<Item> items;

static int capacity;

public static void main(String args[]) throws FileNotFoundException {

//Sort items in descending order by their given value-to-weight

//ratios.

//First item gives best payoff per weight unit and last gives the

//worst payoff per weight unit.

//A branch going to the left indicates the inclusion of the next item,

//a branch going to the right indicates the exclusion of the next item.

//ub = v + (W - w)(nextI\_V / nexti\_W)

items = new ArrayList<Item>();

Scanner infile = new Scanner(new File(args[0]));

populateVariables(infile);

for (int i = 0; i < items.size(); i++) {

items.get(i).WVRatio = (double)items.get(i).value/items.get(i).weight;

//cout << items.at(i).WVRatio << endl;

}

BubbleSort(items);

for (int i = 0; i < items.size(); i++) {

System.out.println(items.get(i).WVRatio);

}

root = new TreeNode(0,0,0);

//start time

long startTime = System.nanoTime();

long stopTime = System.nanoTime();

long overTime = stopTime - startTime;

startTime = System.nanoTime();

int max = branchBoundKnapsack(root, 0);

System.out.println("FINAL VALUE: " + max);

stopTime = System.nanoTime();

//Stop time for the algorithm

long time = stopTime - startTime - overTime;

System.out.println("Total time is: " +(time/1000000));

}

static int branchBoundKnapsack(TreeNode currentNode, int nodeLevel) {

int value = 0;

//ub = v + (W-w)(vi+1/wi+1)

int valueIncludeI = (currentNode.totalValue+items.get(nodeLevel).value);

int weightIncludeI = (currentNode.totalWeight+items.get(nodeLevel).weight);

if (nodeLevel+1 >= items.size()) {

//Base case

//Current Node is a leaf.

int maxValue;

if (valueIncludeI > currentNode.totalValue && (weightIncludeI <= capacity)) {

maxValue = valueIncludeI;

} else {

maxValue = currentNode.totalValue;

}

//cout << "MAX VALUE: " << maxValue << endl;

return maxValue;

} else {

//Current node is not a leaf.

int upperBoundLeft = (int)(valueIncludeI + (capacity - weightIncludeI)\*(items.get(nodeLevel+1).WVRatio));

int upperBoundRight = (int)(currentNode.totalValue + (capacity - currentNode.totalWeight)\*(items.get(nodeLevel+1).WVRatio));

//cout << upperBoundLeft << " " << upperBoundRight << endl;

TreeNode leftChild = new TreeNode(upperBoundLeft, weightIncludeI, valueIncludeI);

TreeNode rightChild = new TreeNode(upperBoundRight, currentNode.totalWeight, currentNode.totalValue);

currentNode.leftChild = leftChild;

currentNode.rightChild = rightChild;

//----Select Next Path----//

int leftValue = -1;

int rightValue = -1;

if ((upperBoundLeft > upperBoundRight) && weightIncludeI <= capacity) {

leftValue = branchBoundKnapsack(currentNode.leftChild, nodeLevel + 1);

//Make sure it was the best

if (leftValue < upperBoundRight) {

rightValue = branchBoundKnapsack(currentNode.rightChild, nodeLevel + 1);

}

} else {

rightValue = branchBoundKnapsack(currentNode.rightChild, nodeLevel + 1);

//Make sure it was the best

if (rightValue < upperBoundLeft && weightIncludeI <= capacity) {

leftValue = branchBoundKnapsack(currentNode.leftChild, nodeLevel + 1);

}

}

//Get the max from the 2 branches

if (leftValue >= rightValue) {

value = leftValue;

} else {

value = rightValue;

}

}

return value;

}

static void BubbleSort(ArrayList<Item> vect) {

int i, j;

boolean flag = true; // set flag to 1 to start first pass

Item temp; // holding variable

for(i = 1; i <= vect.size() && flag; i++) {

flag = false;

for (j=0; j < (vect.size() - 1); j++)

{

if (vect.get(j+1).WVRatio > vect.get(j).WVRatio) {

temp = vect.get(j);

vect.set(j, vect.get(j+1));

vect.set(j+1, temp);

flag = true; // indicates that a swap occurred.

}

}

}

return; //arrays are passed to functions by address; nothing is returned

}

static void populateVariables(Scanner file){

int count = 0;

while(file.hasNext()){

if (count==0){

capacity = file.nextInt();

//System.out.println("Capacity is:"+capacity);

count++;

}

if (count==1){

int numItems = file.nextInt();

count++;

}

else{

Item newItem = new Item(file.nextInt(), file.nextInt());

items.add(newItem);

count++;

}

}

file.close();

}

}

class TreeNode {

int upperBound;

int totalWeight;

int totalValue;

TreeNode leftChild;

TreeNode rightChild;

TreeNode() {

upperBound = 0;

totalWeight = 0;

totalValue = 0;

leftChild = null;

rightChild = null;

}

TreeNode(int ub, int tw, int tv) {

upperBound = ub;

totalWeight = tw;

totalValue = tv;

leftChild = null;

rightChild = null;

}

}

class Item {

int weight;

int value;

double WVRatio;

Item(int w, int v) {

weight = w;

value = v;

}

}

# Appendix I – References

Introduction to the Design and Analysis of Algorithms by Anany Levitin

Assignment sheet – Dr. Robert Strader