

Comparison of Parallel Algorithms for the 0-1 Knapsack Problem on Networked
Computers

Rebecca A Hunt

A Thesis in the Field of Information Technology
for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

November 2004

Abstract

This thesis presents three unique algorithms for solving the 0-1 Knapsack Problem in parallel in a networked environment. For readers new to this area, this document presents background on the Knapsack Problem. Each of the three algorithms is described in detail in the following chapters. Two algorithms are from published literature, and one algorithm is an original contribution by the author. Finally, this thesis contains a comparison of the performance of the three algorithm, and discusses types of problem domains. In addition, the thesis compares the communication load of each algorithm and correlates the communication load with the performance.

Acknowledgments

I'd like to thank my advisor, Dr. Jeff Parker for his time, thoughts and ideas. I appreciate all the questions and suggestions. I also want to thank Forrest Glick and Gina Siesing for their insightful comments on my drafts. Finally, I am happy to acknowledge Jamaica Frederick for providing encouragement while I worked on the computer.

Table of Contents

Table of Contents.....	v
List of Figures.....	vii
List of Tables.....	viii
List of Equations.....	ix
Chapter 1 Introduction.....	1
Parallel Solutions.....	2
Networked Computers.....	3
Benefits of Implementing Selected Algorithms.....	5
Chapter 2 Overview of the Knapsack Problem.....	7
Relevance of the Knapsack Problem.....	7
Methods for Solving the Knapsack Problem.....	9
Chapter 3 Architecture and Design.....	21
Selection of Algorithms.....	21
Language and Tools.....	22
Roles of Client, Server, and Solver.....	23
Package Structure.....	26
Content Generator.....	31
Chapter 4 LEE Algorithm.....	33
Overview of the Algorithm.....	33
Exchange vectors in levels.....	37
Combine and History Calculations.....	40
Backtracking Steps.....	41
Differences Between Paper and Implementation.....	42
LEE's Performance.....	44
Chapter 5 CHEN Algorithm.....	46
Dynamic Programming with a Pipeline Model.....	46
Implementation of CHEN.....	47
Pipeline Processing with levels.....	49
Lists of Change Points.....	52
Implementation of CHEN.....	54
CHEN's Performance.....	56
Chapter 6 HUNTPL Algorithm.....	59
Similarity to CHEN.....	59
Levels as Implemented in HUNTPL.....	59
Communication in HUNTPL.....	62
HUNTPL's Performance.....	63
Chapter 7 Comparison of the Algorithms' Performance.....	65
Description of Tests.....	65
Performance for Multiple Solvers.....	66

Comparison of Network Communication	71
Networked vs. non-networked.....	73
Problem domains.....	75
Chapter 8 Summary and Conclusions.....	78
Chapter 9 References	81
Appendix 1 Client Package.....	83
Java.....	83
KnapsackClient.java.....	83
FileUtility.java	88
SortByWeightComparator.java.....	91
ClientException.java	92
Appendix 2 Server Package	93
Java.....	93
KnapsackClientServer.java.....	93
KnapsackClientServerImpl.java	95
SolverServerFactory.java	98
KnapsackSolverServer.java.....	100
KnapsackServerLeeImpl.java.....	102
KnapsackServerChenImpl.java.....	117
KnapsackServerHuntPL.java.....	123
ServerException.java.....	130
Appendix 3 Solver Package	131
Java.....	131
KnapsackSolverLeejava	131
SolverLeeImpl.java	134
KnapsackSolverChen.java.....	149
SolverChenImpl.java.....	151
KnapsackSolverHuntPL.java.....	166
SolverHuntPLImpl.java.....	168
HuntPLVectorObject.java	180
HuBackTrackImpl.java	181
SolverException.java.....	182
Appendix 4 Contents Package.....	184
Java.....	184
ContentsFile.java.....	184
ContentsFileTxtImpl.java.....	186
ContentsGenerator.java	188
ContentsGeneratorWrapperImpl.java	194
ContentsExceptionl.java.....	196
UserInputException.java	197

List of Figures

Figure 3-1 Client - Server - Solver Relationship	24
Figure 3-2 Client Package UML	27
Figure 3-3 Server Package UML	28
Figure 3-4 Solver Package UML	30
Figure 3-5 Contents Package UML.....	31
Figure 5-1 Pipeline with 3 levels and $c=5$	50
Figure 6-1 Levels and Ranges in HUNTPL.....	61
Figure 7-1 Comparison of Algorithms over Knapsack Size 50 in Pseudo-Network	67
Figure 7-2 Comparison of Algorithms over Knapsack Size 100 in Pseudo-Network	68
Figure 7-3 Comparison of Algorithms over Knapsack Size 200 in Pseudo-Network	69
Figure 7-4 Comparison of Algorithms over Knapsack Size 300 in Pseudo-Network	70
Figure 7-5 Comparison of HUNTPL and LEE on networked solvers	74

List of Tables

Table 2-1 Knapsack with 7 Objects and Capacity 9	11
Table 2-2 Greedy Algorithm Order.....	11
Table 2-3 Knapsack with 4 Objects and Capacity 10	15
Table 2-4 Profit Vectors for Dynamic Programming.....	16
Table 2-5 History Vectors for Backtracking.....	18
Table 4-1 Knapsack with 5 Objects and Capacity 5	35
Table 4-2 Example of Merging Lists	35
Table 4-3 Final Profit Vector.....	36
Table 4-4 Profit Vectors With Two Sub-problems.....	36
Table 4-5 Combined Profit and History Vectors	40
Table 4-6 Performance of LEE	44
Table 5-1 Knapsack With 5 objects and Capacity 5	51
Table 5-2 Profit Vectors Created in First 3 Solvers, in 3 Levels.....	51
Table 5-3 Lists of Change Points.....	53
Table 5-4 Performance of CHEN.....	56
Table 5-5 Comparison of Full Range vs. Change Points	57
Table 6-1 Performance of HUNTPL.....	63
Table 7-1 Percentage Slowdown in Networked Environment.....	73

List of Equations

Equation 2-1 Formal definition of 0-1 Knapsack Problem	7
Equation 2-2 Dynamic Programming Equation.....	14
Equation 2-3 Backtracking Conditions.....	17
Equation 2-4 Hu's Backtracking Algorithm.....	19
Equation 5-1 Dynamic Programming.....	46

Chapter 1 Introduction

As you pack your bags for a backpacking expedition, you consider the many items that would be useful on your journey. Some items would be very useful, while others might not have as much value on your journey. You have only one backpack, and it only holds so much. You carefully pack your bag to get the best combination of useful items that your bag can carry.

You have just solved what is known in computer science as the “Knapsack Problem”. A computer scientist would look at it this problem in a more formal manner. First she orders your items from 0 to $n-1$, where n is the number of items from which to choose. Instead of a nebulous idea of “usefulness”, she recognizes that each item has a profit that can be represented by an integer. She assigns these values to an array p of length n . Also, she measures each item’s weight, and assigns these to an array w of length n . The computer scientist considers the maximum amount of weight that can be carried in your backpack and calls this the capacity c . Now the problem above can be defined formally as:

$$\text{Maximize } \sum_{j=0}^n p_j x_j$$

$$\text{Subject To } \sum_{j=0}^n w_j x_j \leq c$$

In this case, each item may be included once or not at all. We can not choose multiple copies of the same item. This is what distinguishes the 0-1 Knapsack Problem

from the general Knapsack Problem. The 0-1 Knapsack Problem does not allow the user to put multiple copies of the same items in their knapsack.

The Knapsack Problem has applications in areas such as operations research and finance. It is used in areas such as cargo packing in the airline and shipping industry. It has been referred to in various contexts as the “bin packing problem”. The Knapsack problem is also well known in computer science, as it belongs to a class of problems which are NP-Complete. When a problem is "NP-Complete" there is no known algorithm to solve the problem in polynomial time. It also means that if there were a polynomial time solution, all other NP problems could be reduced to the knapsack problem in polynomial time, and therefore be solved in polynomial time themselves. Hence, this type of problem has been studied carefully because of its relationship to other important problems in computer science.

Parallel Solutions

Since the 0-1 Knapsack Problem is a well-studied problem, many potential solutions have arisen to improve performance. Several of these solutions have discussed the benefits of solving the problem in parallel with several processors. By distributing the problem to several processors, the solution to a given Knapsack Problem may be determined in less time.

Many of the proposed parallel algorithms have been successful in this regard. Some have resulted in improvements for the worst case scenario, and other algorithms have shown better average times for solving specific knapsack problems. Most of these solutions require special hardware or special networks. Some algorithms require multiple

processors on a single machine with shared memory. Other algorithms allow each processor to have a small amount of individual memory. Very few algorithms are designed specifically for a heterogeneous group of networked computers.

Networked Computers

All of the algorithms in this study were implemented to be run by a group of networked computers. The architecture is familiar to many people through projects like the Search for Extra Terrestrial Intelligence (SETI). In SETI and other networked parallel projects, various personal computers may register to solve a part of a large problem. The participating computers may include a variety of hardware architectures and operating systems. They do not have a specialized network with which to communicate to the server, instead relying on the standard TCP/IP protocols of the network.

This can be contrasted to other types of network architectures. For example, in a linear array, each processor has a direct link to two other processors. A ring network is similar to a linear array, except the links wrap around, so that the last processor is linked to the first. These types of architectures will commonly communicate by passing data in one direction.

Another type of specialized network architecture is a hypercube network. A hypercube network is described as a "mesh of processors with exactly two processors in each dimension... two processors are connected by a direct link if and only if the binary representation of their labels differ at exactly one bit point"(Kumar, Gramar & Gupta, 1994). A d -dimensional hypercube network has 2^d processors. Each processor in a d -

dimensional hypercube has a direct, dedicated connection to d other processors. For more discussion on specialized network architectures, see (Kumar, Gramar & Gupta, 1994).

In these specialized architectures, there are direct connections between some of the processors. This may not be the case in a networked environment. In a networked environment, the processors may not even be in the same sub-net. This may introduce extra latency as the message must pass routers to reach the next processor. In a networked environment, one must deal with higher message costs than in the architectures mentioned above. In addition, the protocols do not guarantee that messages arrive in the order sent. Therefore, a networked environment may have the additional burden of ordering messages received.

However, there are benefits to implementing algorithms on a networked architecture. As more and more personal computers are purchased and used, solving difficult problems using networked computers becomes more practical. Indeed, if a problem can be solved using several office desktops overnight while their users are at home, a specialized architecture for the same problem might seem less cost-effective.

For this reason, I examine the implementation of the Knapsack Problem over a set of heterogeneous networked personal computers. Several of the algorithms I implement are based on algorithms that were not originally designed for this type of network. In the following chapters, I will discuss the differences and difficulty of adapting these algorithms so they may be run without specialized hardware or networks.

Benefits of Implementing Selected Algorithms

Computer scientists have developed a method for analyzing and comparing algorithms that does not rely on comparing implementations. These theoretical results are extremely useful for comparing the worst case performance, and for comparing algorithms without comparing the implementation. As stated in one paper surveying various algorithms, “The quantitative results reported by researchers are largely based on independent tests carried out for unrelated problem instances on different generations of parallel architectures. Comparative assessments of the algorithms are unfair and/or difficult to make under these circumstances.” (Gerasch & Wang, 1993)

This thesis attempts to remedy the situation for a limited number of proposed algorithms. Three unique implementations, referred to as CHEN, LEE, and HUNTPL, are described and tested. The implementation of each algorithm should be of comparable quality, done by the same programmer in the same programming language. The tests use the same problem instances for each algorithm on the same hardware, and hence the results can be compared directly.

By examining the average running time of each algorithm, I will be able to compare the performance of each algorithm on different types of problem sets, which I refer to as problem domains. This will allow me to determine how the implementations can be expected to perform on real world problem sets. This information will add to what is already known about each algorithm’s worst case performance.

By implementing the unique algorithms we are able to discover the difficulties inherent in adapting theoretical works to applications. In addition, we will better understand the assumptions inherent in algorithms are written for a specific network or

hardware architecture. We can compare the difficulty of adapting algorithms intended for one network architecture to a more generalized architecture.

The following chapter contains an overview of the literature for the knapsack problem. The third chapter discusses the software architecture for the implementations, and the design of the overall solution. The following chapters explain each unique algorithm in greater depth. The final chapter contains a comparison of the algorithms over the different problem domains.

In completion, this body of work will demonstrate how algorithms for solving the 0-1 Knapsack Problem can be implemented on a system of distributed processors and memory. I discuss the differences between the theoretical papers and the implementations. I compare the performance of the three unique implementations on several different problem sets, creating a better understanding of the strengths of each algorithm. In total, this paper will compare and contrast parallel algorithms for the 0-1 Knapsack problem, allowing insights to be gained in parallel programming on a networked environment.

Chapter 2 Overview of the Knapsack Problem

As discussed in the introduction, the 0-1 Knapsack Problem can be given as a capacity c , an ordered list of weights w , and an ordered list of profits p . The weight and profit values are positive integers, and the capacity is also a positive integer. We shall denote the optimal solution for the maximum profit that can be obtained as an integer z . The corresponding solution of optimal items can be denoted as a vector x of length n , in which x_i is 1 if the i th object is included, and 0 if it is not included.

As presented in the introduction, the formal definition of the 0-1 Knapsack Problem is as follows.

Equation 2-1 Formal definition of 0-1 Knapsack Problem

$$\text{Maximize } \sum_{j=0}^n p_j x_j$$

$$\text{Subject To } \sum_{j=0}^n w_j x_j \leq c$$

Relevance of the Knapsack Problem

Since the Knapsack Problem only has one constraint, and the coefficients are integers, it is considered one of the simpler NP-Problems. It is easily understood by many people who have not studied other problems in theoretical computer science. Some computer scientists study the problem with the idea that techniques learned from the simple problems can be generalized to more complex domains. In addition, the Knapsack Problem is used in more complex problems as a sub-problem.

The Knapsack Problem is a popular algorithm for study. Skiena analyzed the requests to the Stony Brook Algorithm Repository (Skiena, 1998). As he explains, “the majority of visitors to the Algorithms Repository come seeking implementations of algorithms which solve the problem they are interested in”. The Knapsack algorithm was the 18th most requested type of algorithm out of 75 types in the repository. These requests appeared to be coming from both the academic and commercial communities. This demonstrates the strong interest in the Knapsack Problem. In addition, Skiena ranked the Knapsack Problem the fourth “most-needed” implementation, by comparing the number of requests for a solution to the number of solutions available.

There are several variants and extensions to the knapsack problem. For example, the “subset sum problem” is a Knapsack Problem in which an object’s profit is directly proportional to its weight. Another variant is the “multi-dimensional knapsack problem”. This problem considers a constraint in addition to weight on the problem set. For example, when you pack your knapsack, you must not only consider the weight of your items, but also their volume. A third, but by no means final, example of a Knapsack Problem variant is the “quadratic knapsack problem”. In this problem, every j th item has a profit p_{ij} that can only be redeemed if the i th item is also put in the knapsack. In our backpacking example, there is no use taking the camping stove unless you also take fuel. For more examples of the Knapsack Problem variants and applications, refer to the excellent book, titled “Knapsack Problems” by Kellerer, Pserchy and Pisinger (Kellerer, Pserchy & Pisinger, 2004).

The Knapsack Problem and many of its variants have been presented as a way to model applied problems. For example, Fréville (Fréville, 2003) lists some uses for a variant of the 0-1 Knapsack, called the multi-dimensional knapsack problem:

“The [multi-dimensional knapsack problem] has been introduced to model problems including cutting stock, loading problems, investment policy for the tourism sector of a developing country, allocation of databases and processors in a distributed data processing [sic], delivery of vehicles with multiple compartments and approval voting. More recently, the MKP has been used to model the daily management of a remote sensing satellite..., which consisted in deciding every day what photographs will be attempted the next day”.

Many applications for the Knapsack Problem can be in both finance and operations research. For more information on work done in this field, see the paper "A Survey of Parallel Algorithms for the One-Dimensional Integer Knapsack Problem" (Gerasch & Wang, 1993). The generic problem is intuitive and can be applied in a number of real-world problems.

Methods for Solving the Knapsack Problem

Having seen that the Knapsack Problem is useful and popular, let's consider how one finds a solution to a given problem. First we need to define what it means to solve a Knapsack Problem correctly to get the best profit z . The set of objects used to get the best profit are denoted x , and are part of the final solution.

There may be more than one correct answer. That is, the best profit z may have multiple corresponding correct solution vectors. For example, two unique items may have the same profit and weight, but it is possible to carry only one of them.

Additionally, two different subsets of items may add up to the same profit and use the

same weight. In either case, we may have several distinct, but correct, solution vectors. For our use, we will show no preference to either solution set. We shall consider a solution correct if it returns the best profit z and any corresponding solution vector x .

Several studies of the Knapsack Problem have shown the viability of producing an approximate solution, instead of the exact solution. For what one loses in accuracy, one gains in the improvement in total solution time and computational power required. These approximate solutions are worthwhile when a “good” solution is adequate and the computing resources are limited. Also, these algorithms are interesting in their attempts to determine the core of the problem. However, in this paper, we are interested in finding exact solutions using parallel processing.

The Greedy Algorithm

One method for solving the Knapsack Problem is intuitive, and represents the algorithm a non-expert might apply to this problem. Although it is not an exact method, I present this algorithm as an introduction to the notation of the Knapsack Problem, and to begin thinking about methods to solve the problem.

This method, called the “greedy algorithm”, builds on the notion that you would want the objects with the highest profit to weight ratio. First, the greedy algorithm orders the objects by the ratio of their profit to weight. It then takes all the objects with the highest ratios that will fit into the knapsack. For example, consider the Knapsack defined in the following table. This example is taken from (Kellerer, Pserchy & Pisinger 2004).

Table 2-1 Knapsack with 7 Objects and Capacity 9

index	0	1	2	3	4	5	6
weight	2	3	6	7	5	9	4
profit	6	5	8	9	6	7	3

Using the example above, we will apply the greedy algorithm to get a result. Take the ratios (profit / weight) for each item in the knapsack. Now re-order the items so that the items with the highest ratio are ahead of the items with lower ratios. We will see the results as shown in the following table.

Table 2-2 Greedy Algorithm Order

index	0	1	2	3	4	5	6
weight	2	3	6	7	5	9	4
profit	6	5	8	9	6	7	3
ratio	3	1.666667	1.333333	1.285714	1.2	0.777778	0.75

In this case, the objects are already in order of decreasing ratios. To get our solution using the greedy algorithm, we add items from left to right to our knapsack until our knapsack is full. This results in a knapsack containing object zero (with weight 3 and profit 6), object one (with weight 3 and profit 5), and object six (with weight 4 and profit 3). These three objects have a combined weight of 9. The profit obtained using the greedy algorithm is fourteen, the sum of the profits of the three items.

However, it is possible to get a better profit for the given knapsack. An exhaustive search of all possibilities would show that by including different objects, a profit of fifteen is possible. This profit by is obtained by including the objects with index 0 and 3. Using the notation defined earlier, the exact solution to the problem is:

$$z = 15$$

$$x = [1,0,0,1,0,0,0]$$

While it is worth understanding the formal definition of the solution set, we can present this information more clearly. My implementations return the solutions to the knapsack to the user in the following format.

Best Value is 15 ResultSet (2, 6) (7, 9)

The greedy algorithm is not appropriate if we are looking for an exact solution. The 0-1 Knapsack Problem is usually solved for exact results in one of two ways. The first method is called “branch and bound”. The second method is known as “dynamic programming”. There are additional possibilities for serial implementations for the Knapsack Problem. These include genetic algorithms and list splitting. Some of these, such as genetic algorithms, have been shown to perform worse than the traditional branch and bound or dynamic programming algorithms (Gordon, Boehm & Whitley 1994). Others are not applicable to parallel programming. Some algorithms, such as 2-list splitting, restrict how many sub-problems the problem can be divided into (Horowitz & Sahni, 1974). As such, they are not covered in this paper.

Branch and Bound

To understand the branch and bound approach, let us first consider a brute force method to solving the knapsack problem. One could find the solution by considering the problem a tree, and the decision to include each object a branch in the tree. As the tree is traversed down to each node, every possible combination of items is considered.

However, this method can become intractable as the number of objects grows larger and 2^n possible solutions must be considered.

The branch and bound method improves on the exhaustive search by examining a subset of the possible solutions. It does this by pruning the branches of the solution tree. However, it does not prune any branches that may contain a feasible solution to the problem.

The branches are pruned in one of two ways. First, if the current weight of the solution at a given node exceeds the capacity of the knapsack, there is no need to continue down the paths below the node. Nothing below that node can be added to the knapsack as the capacity has already been exceeded.

Second, branches can be pruned if it can be determined that no better solution can be reached. First, an upper bound of possible solutions is calculated at each node. It can then be compared to the optimal bound found up to this point. If the node's bound is less than the optimal bound, the branch is not worth pursuing and can be pruned. This implies that the objects are sorted by value or value/weight before the traversal begins.

However, for distributed computing, the complicated communication strategies required for the branch and bound algorithm can be difficult to solve. Compared to dynamic programming algorithms, the branch and bound technique is not as "amenable to systematic manipulation" (Alexandrov & Megson, 1999). Since the processors need to notify others in a timely manner when an upper bound is determined, the distributed solutions can be marred by communication issues. As stated in (Alexandrov & Megson, 1999), "for serial machines it is accepted that B&B [Branch and Bound] has better performance than Dynamic Programming for the Knapsack Problem, but this observation

has not been shown to translate to the parallel case”. Indeed, my research returned very few papers using the branch and bound approach for parallel solutions to the Knapsack Problem.

Dynamic Programming

Dynamic programming is an alternative solution to branch and bound programming for the Knapsack Problem. Dynamic programming is a method used for solving many types of problems; it is not specific to the Knapsack Problem. Using the dynamic programming methods, an entire problem can be split into interdependent sub-problems. Each sub-problem can be solved separately, and then the results are used to solve larger sub-problems successively. Eventually the entire problem is solved.

Using dynamic programming to solve the Knapsack Problem involves two stages. The first stage is the forward programming stage, in which the maximum profit for the Knapsack is computed. The second stage is the backtracking step. This step uses information from the forward stage to determine which objects were used to obtain the optimal profit.

For the forward stage of the 0-1 Knapsack Problem, we can use the following function to find the solution.

$$\begin{aligned}
 0 &\leq j \leq c \\
 0 &\leq k \leq n \\
 f_k(j) &= \max\{f_{k-1}(j), f_{k-1}(j - w_k) + p_k\}
 \end{aligned}$$

Equation 2-2 Dynamic Programming Equation

In the above equation, $f_k(j)$ is the maximum profit of the knapsack at capacity j if the first k objects are included. The value of $f_k(0)$ will be 0 for all k , since at capacity 0 the knapsack can not hold any objects. The profits possible when the k th object is calculated is called the profit vector, and equivalent to f_k .

The optimal solution to the knapsack problem can be obtained by calculating $f_0, f_1, f_2, \dots, f_n$ until $f_n(c)$ is computed for the correct answer. This allows us to build up a table of values to be accessed in each successive stage of computation. In other words, for each possible object, we consider whether to include it at each possible capacity (from 0 to the maximum capacity). We store the maximum profit we could reach for that object at that capacity in a table. The calculation continues on to determine whether including the next item is more profitable than not including it. The columns in the table are the profit vectors for the objects.

For example, consider the following knapsack definition with four objects.

Table 2-3 Knapsack with 4 Objects and Capacity 10

knapsack capacity=10				
object index	0	1	2	3
object weight	9	2	9	3
object profit	1	2	2	3

Since this example knapsack has four objects and capacity ten, we can expect to create a four by ten matrix of results from equation 2-2. The following table shows the resulting profit vectors for our example.

Table 2-4 Profit Vectors for Dynamic Programming

capacity\object	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	2	2	2
3	0	2	2	3
4	0	2	2	3
5	0	2	2	5
6	0	2	2	5
7	0	2	2	5
8	0	2	2	5
9	1	2	2	5
10	1	2	2	5

In the above table, we see all the results for j . For example, consider the first object, which has weight nine and profit one. Clearly, at any capacity less than nine, we can not include this object, and our profits will be zero. Indeed, if we look at the column for the object with index zero, we see that the profits are zero until the object can be included. At row nine, we can see the profit of including this object is one. This column represents j , where j is the capacities zero through ten.

To investigate this table further, notice that the profits at each object for a given capacity are always at least the profit for the previous capacity. Looking back at equation 2-2, we see this is because we are always using the profit at the previous object if the current object can not do any better.

Furthermore, we can see the final solution to our problem in the lower right hand corner of the table. The maximum value for this knapsack, found at $(10, 5)$, is five.

While building this table, we need to keep track of the objects that were used to create these profits. This information will be used in the backtracking stage when the forward processing is complete. We will create this information by building history vectors that contain the indices of the objects that were used. We first initialize our

vectors to a value which indicates that no object was used. I use “-1” to indicate an invalid object index. We then fill in the values for the matrix according to the following condition in equation 2-3.

In the following equation, $u_k(j)$ represents the value of the history vector at object k with the possible capacity j .

Equation 2-3 Backtracking Conditions

$$u_k(j) = \begin{cases} k & \text{if } f_k(j - w_k) + p_k > f_{k-1}(j) \\ u_{k-1}(j) & \text{otherwise} \end{cases}$$

If we examine Equation 2-3, we see that it is doing a similar operation to that done in the dynamic programming equation. Both equations are essentially testing whether the profit of including the object, represented by $f_k(j - w_k) + p_k$, is greater than the profit we got without the current object, represented by $f_{k-1}(j)$. The calculation for the dynamic programming equation and the backtracking conditions could be included in the same step in an implementation.

The following table demonstrates the values obtained when using equation 2-3 when calculating the Knapsack example from Table 2-3.

Table 2-5 History Vectors for Backtracking

capacity\object index	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	-1	-1
2	-1	1	1	1
3	-1	1	1	3
4	-1	1	1	3
5	-1	1	1	3
6	-1	1	1	3
7	-1	1	1	3
8	-1	1	1	3
9	0	1	1	3
10	0	1	1	3

Each cell indicates the index of the object that was used to get the maximum profit at this capacity. I used “-1” to indicate that no object was used. For example, at capacity 9, object zero was included. The vector from object two is exactly the same as the vector from object one, since object two is not used in the final solution. On the other hand, object one’s vector indicates that object one was used in all capacities greater than or equal to two, which is the weight of object one.

Once the forward process is completed and the profit vectors and history vectors have been computed, the backtracking process can compute the solution vector. It will use the history vector and the Hu backtracking algorithm. The definition of the Hu backtracking algorithm as presented in (Andonov, Raimbault, & Quinton, 1993) is shown here.

Equation 2-4 Hu's Backtracking Algorithm

```
 $j := c;$   
for  $k = m$  downto 1 do  
   $z_k = 0$   
  while  $u_m(j) = k$  do  
     $z_k = z_k + 1;$   
     $j = j - w_k;$   
  end (while)  
end(for)
```

The Hu backtracking algorithm starts from the last object and traces back. At each point it determines whether the given object was used to create the maximum profit. If so it subtracts the weight of the given object from the current capacity and continues. In doing so, it builds the solution vector for the problem.

A method such as dynamic programming can be applied to a distributed solution. Since each problem can be split into sub-problems, different processors can work on each sub-problem. These sub-problems can then be combined to solve the larger problem at hand. Therefore, variants of dynamic programming are used for the three implementations discussed in this paper.

The following chapters will discuss and analyze the implementations of the three algorithms. They will assume familiarity with the notation defined in this chapter. In addition they will build on concepts introduced in the section on dynamic programming

such as backtracking and profit vectors. The sections will discuss how each unique algorithm was implemented, and how it performed.

Chapter 3 Architecture and Design

The following chapter describes the design of the software for this thesis. This chapter is intended to address questions a software engineer might have about the architecture and design of the project. As such, this chapter assumes some knowledge of programming concepts. Details of the different algorithms, and the unique issues addressed in each one, are discussed separately in the following chapters. This chapter gives an overview of the features common to the three implementations.

Selection of Algorithms

Three unique algorithms have been implemented and tested. Each algorithm presents a unique method of solving the 0-1 Knapsack Problem in a parallel manner over the network. Two algorithms, referred to as CHEN and LEE, were selected from published literature. These algorithms were selected because of their promise to increase speed, and because of their unique attributes in relation to each other.

I use “CHEN” and “LEE” to refer to my implementations, which should be distinguished from the papers by the authors of the same name. The implementation named LEE is based on the paper [A Hypercube Algorithm for the 1/0 Knapsack Problem](#) (Lee, Shragowitz & Sahni, 1988). The implementation named CHEN was based on the algorithm described in [Pipeline Architectures for Dynamic Programming Algorithms](#)

(Chen, Chern, & Jang 1990). The algorithm presented in (Lee, Shragowitz & Sahni, 1988) has many steps, and uses several levels to combine profit and history vectors. The algorithm presented in (Chen, Chern, & Jang, 1990) seems at first more straightforward than the work by Lee, Shragowitz & Sahni, but suggests improvements that make the implementation more difficult. The two algorithms are similar in that they offer solutions specifically for the 0-1 Knapsack problem, and that they can be modified to run in a networked environment.

The third implementation is named HUNTPL, and is an original contribution by the author. Without having read all of the many contributions to the literature on the knapsack problem, I cannot guarantee that it has never been done before. The ideas behind HUNTPL were sparked while implementing CHEN, and are a significant modification of the algorithms described in (Chen, Chern, & Jang, 1990).

Language and Tools

The programming language chosen for this project is Java. Java is an object oriented programming language developed by Sun. Java allows programs to be written in a machine independent manner. In addition, Java has a component called “Remote Method Invocation”, or RMI. RMI is a native Java distributed object system which allows programmers to pass object references and run methods on remote clients as if they were local machines. RMI allows the programmer to write objects that can be communicated across the network without focusing on the details of the network communication. For more information on RMI and other alternatives for writing networked software in Java, please refer to the book “Java Network Programming” (Harold, 2000).

When messages are sent using RMI, the thread sending the message will stop until the message has been sent and processed. In my implementation this is usually undesirable, as we would like calculation to continue while messages are being sent. Therefore, the components start threads to send messages, and these threads are independent of the calculation. This allows calculation to continue as necessary while messages are being sent.

In my implementations, all the network communication between the processors uses RMI. By using RMI, I was able to concentrate on algorithms and not on the network packets being sent.

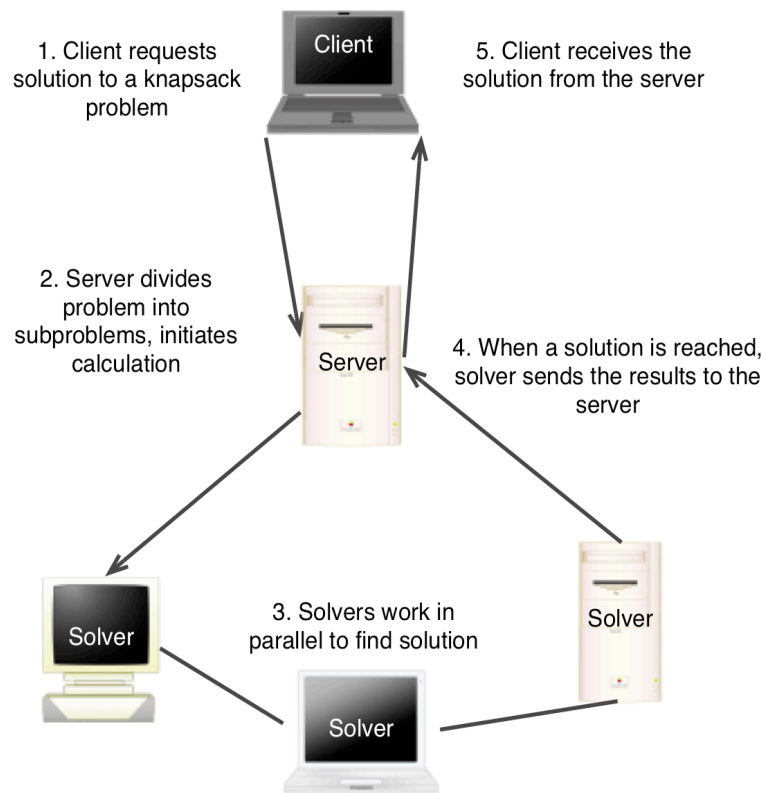
Roles of Client, Server, and Solver

The software presented has three main components. These components are the client, server, and solver. Each of these components can run separately on different processors, and has a different role in calculating the solution. The client acts to take a request to solve a knapsack from the user. The server accepts the knapsack definition from the client and farms the work out to the solvers. The solvers do most of the work in determining the solution. When a solver reaches the solution, it is returned to the server. The server then sends the solution to the client.

As mentioned earlier, we have a choice of three algorithms. When the user starts the server, the user determines which algorithm will be used. Solvers written specifically for a given algorithm may then register with the server. The client may request a solution once the server has started and the minimum number of solvers has registered with the server.

The following diagram illustrates the relationship between the components during the calculation of a knapsack problem. The lines connecting each computer in the diagram represent network communication. The computers are represented with heterogeneous icons, as the computers themselves may be heterogeneous in terms of hardware and software such as the operating system. The steps taken by the components are shown in counter-clockwise order.

Figure 3-1 Client - Server - Solver Relationship



The step numbered 1 in the above diagram occurs after a user gives the client a knapsack to be solved. The client sends a request to the server to solve the given knapsack problem. In step 2, the server divides the problem into sub-problems and calculates the parameters that will be needed during the calculations. The server then

sends this information to the solvers and initiates the calculations. Next, in step 3, the solvers begin computing the solution to their sub-problem, using their specific algorithm. The fourth step occurs when all the values have been calculated and a solution has been found. At this point, a solver sends the results to the server. For the fifth step, the server sends the results to the client. The client will display the results to the user and exit. The server and the clients will wait to receive the next knapsack.

The server acts as an intermediary between the solvers and the client. Only the server communicates directly with the client. The solvers may communicate with other solvers and with the server, but they never communicate directly with the client. The client only needs to make a request to a server. The client does not need any information on the algorithm being used, or the number of solvers available. It is simply an interface between the user and the server.

In the above diagram, there are three solvers. These three solvers will contain identical code for the given algorithm. The number of solvers can vary. There can be more than three solvers. Some algorithms allow as few as one solver, although this is not very useful for a distributed, parallel solution. All of the parallel work is done by the solvers. There is only one client and one server involved in the computation of a single knapsack problem.

Solvers may communicate with the server, and with other solvers. At various stages of computation, the solvers may make requests to the server. For example, in one algorithm, a solver notifies the server when they have completed a task and requests the next step from the server. In other algorithms, the server subdivides the solvers, but the server does not play an active role in the computation of the result, once the calculation

has started. In all three implementations, the server calculates some parameters, controls the initiation of computation, and determines the ordering of the solvers and with whom each solver communicates.

We have seen how the client, server, and solver encapsulate different concerns. The client accepts a knapsack definition and sends it to the server. The server controls the overall flow of information. The solvers work in parallel to find the solution. The next section discusses how these different roles are separated into packages.

Package Structure

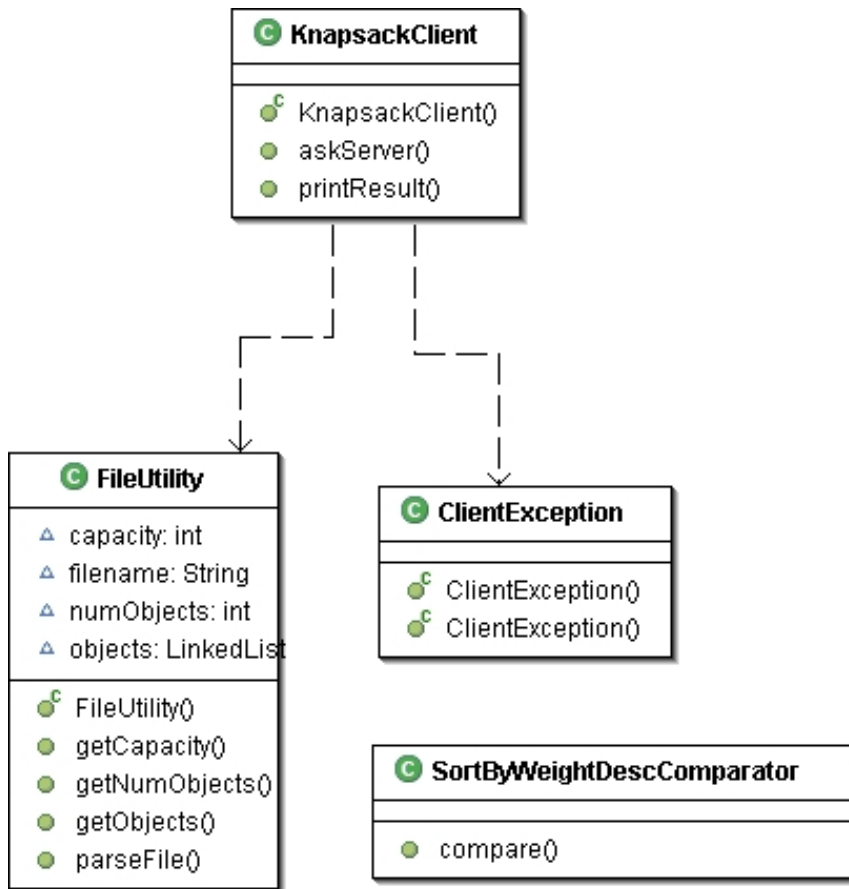
There are three packages involved in the solution of a knapsack: the client, server, and solver. Each package contains the implementations of the classes that will play the role of the networked client, solver, or server as described above. In addition, the packages contain the associated interfaces, exceptions and helper modules.

Client Package

The client package contains one class which performs the client's role. This class accepts a knapsack definition, as defined in a text file, and returns the resulting solution to the user via standard output. This package also contains the class which parses the knapsack definition file. In addition, it contains a class which extends the "Comparator" class. The comparator defines how objects in the knapsack can be sorted by weight.

The following UML diagram illustrates the Client Package structure.

Figure 3-2 Client Package UML



Server Package

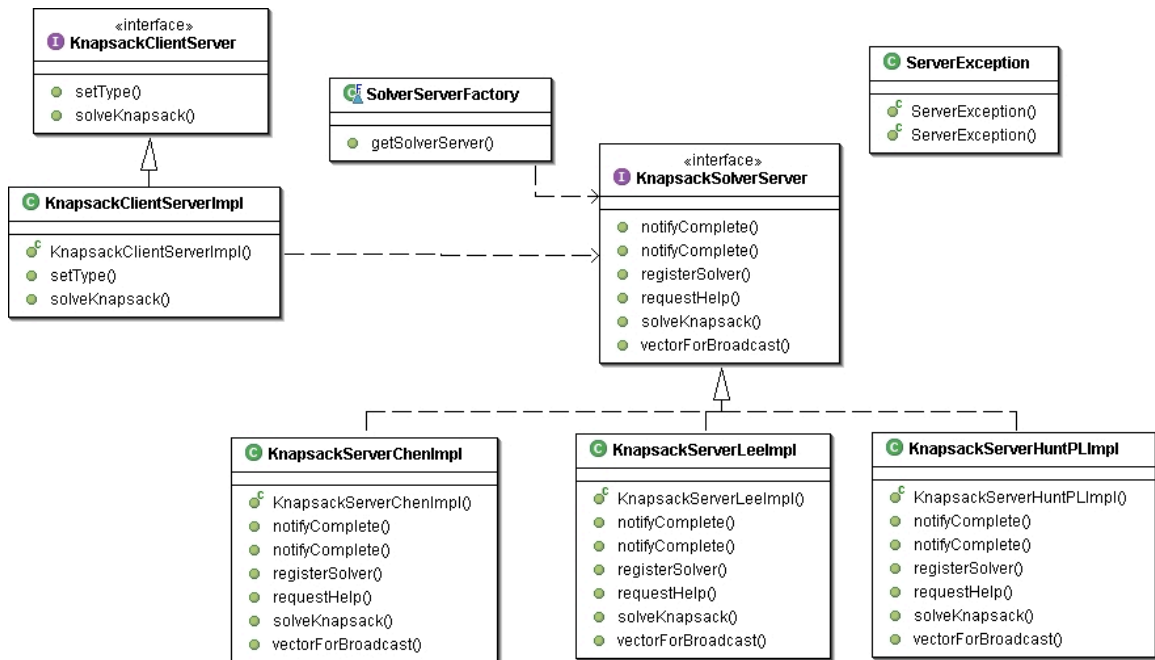
The server package contains two interfaces for the server. One defines the interface between the client and the server, which I call the `ClientServer`. The other interface defines the allowed interactions between the server and the solver, which I call the `SolverServer`. The `ClientServer` interface and interaction remain the same, regardless of the algorithm, while the solver will have different requests for the server, depending on the algorithm. As such, there are multiple implementations of the `SolverServer` interface.

When the user starts the server, he or she specifies the type of algorithm that will be used. The different algorithm implementations use different methods, and interact with the solvers for each implementation.

In addition to the two server interfaces and the corresponding implementations, the server package also contains an abstract factory. For a description of abstract factories, and other design patterns used in the code see (Gamma, Helm, Johnson & Vlissides 2002). This factory class allows the creation of the related ServerSolver implementations. The package also contains an exception class for handling errors that occur in the server's execution.

The following UML diagram illustrates the Server package.

Figure 3-3 Server Package UML



Solver Package

The solver package contains the interfaces and implementations for the solvers for each type of algorithm. In general, the interfaces for solvers have names of the form “KnapsackSolver” concatenated with the algorithm name. The implementations of these interfaces are named “Solver” followed by the algorithm name, followed by “Impl”.

In addition, this package contains an implementation of the Hu Backtracking algorithm. This class is shared by two of the three algorithms. There is also a remote object in this package. This object is used by the HUNTPL solvers to facilitate the passing of vectors between the solvers.

The following UML diagram illustrates the inheritance of the classes in the Solver Package.

Figure 3-4 Solver Package UML

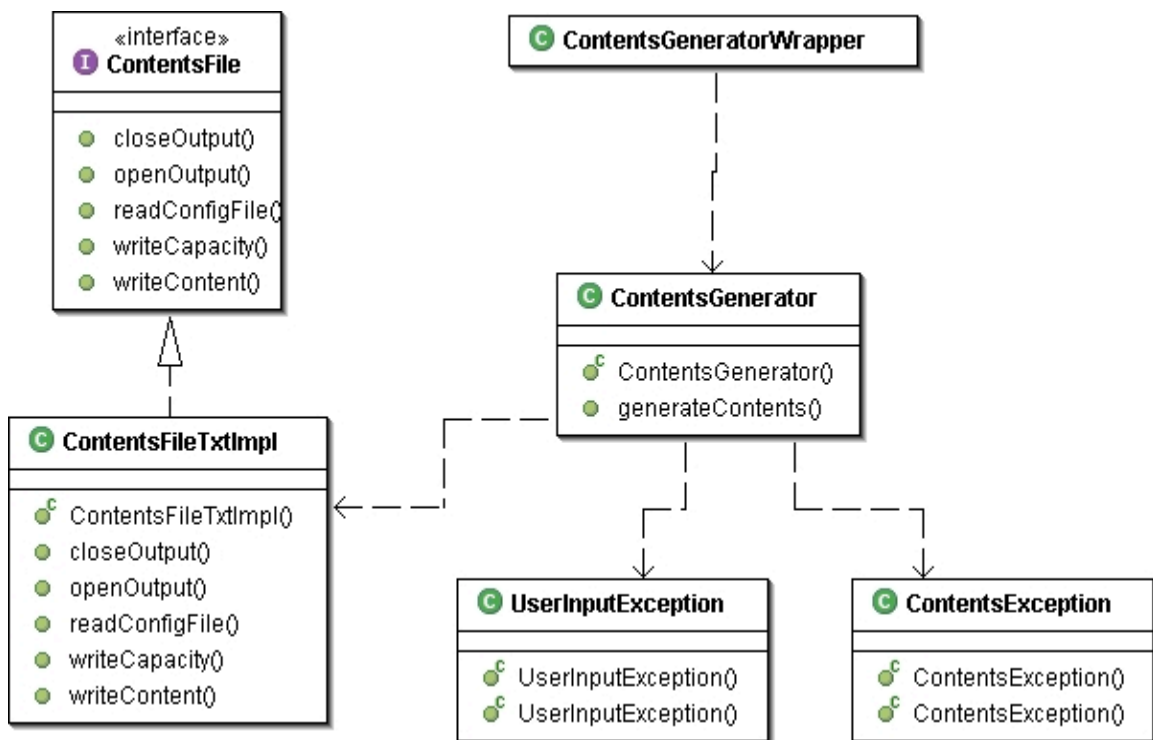


Content Generator

There is an additional package, named “contents”, which is not used in the calculation of the knapsack. This package contains a small application that calculates different types of Knapsack Problem sets, and writes them to a file. Chapter 9 will describe the types of problem sets in more details. Here, I will discuss the software that generates the possible contents of a knapsack for a knapsack problem.

The following UML diagram shows the classes in the contents package and the relationships among them.

Figure 3-5 Contents Package UML



The **ContentsGenerator** class in the contents package is responsible for creating a random set of knapsack items. It can read from a configuration file that defines the maximum number of objects, the highest possible weight and profit for each item, and the

problem set type. It then generates random weights and profits less than the maximum value for each object. The capacity should be some number which would be reasonable given the number of items and their weights. Therefore, the capacity is set to a random number between 0 and the number of objects multiplied by the maximum weight.

A class named `ContentsGeneratorWrapper` simply calls `ContentsGenerator` for each possible type of problem set. This allows the user to create a full range of test sets for a given number of objects.

An interface named `ContentsFile` allows multiple implementations to write a problem set file. Only one implementation exists. In this implementation, the contents are defined in a simple text file. The first line of the text file contains a single digit that represents the knapsack's capacity. The following lines each represent one object available to be added to the knapsack. Each line contains a weight and profit value, separated by a comma. Extensions to this software system would allow the `ContentsFile` to be represented differently. This would require parallel extensions in the client package.

In summary, this chapter has described the main components of the software. In addition, it has covered the overall structure of the packages in the solution. The following chapters explain the implementations of each of the three algorithms in greater detail.

Chapter 4 LEE Algorithm

The LEE algorithm is a dynamic programming algorithm for solving a 0-1 Knapsack Problem. Presented in (Lee, Shragowitz & Sahni, 1988), the algorithm is designed for a hypercube network. In addition, this algorithm utilizes concepts for solving the knapsack such as merging lists, dividing the problem into sub-problems, and using groups of solvers and levels of calculations to combine the results.

Overview of the Algorithm

The algorithm has five main steps to calculate the results of a 0-1 Knapsack. Some steps are repeated multiple times, while other steps are performed only once. The steps can be summarized as follows.

Assume p solvers are available and we have n objects. Our knapsack has a maximum capacity of c .

1. (partition) The original problem is divided into p sub-problems, and the sub-problems are assigned to the solvers.
2. (forward part of dynamic programming) Each solver creates a profit vector for capacity c , given the objects in the assigned sub-problems.
3. (forward part of combining procedure) The solvers distribute their profit vectors and combine them to get profit and history vectors for the whole problem.
4. (backtracking part of combining procedure) The p solvers trace back the combining history to get a value which is the optimal solution to the original problem.

5. (backtracking part of dynamic programming) Each processor traces back its dynamic programming history to get an optimal solution vector.

Originally, the server receives a request from the client to solve a given problem. The server confirms that an acceptable number of solvers have registered. The algorithm in (Lee, Shragowitz & Sahni, 1988) requires that the number of processors be some power of two, due to the level method used in steps 3 and 4. The server computes how many levels will be needed to compute the problem, based on the number of processors.

The server begins partitioning the problem, so that each processor has a sub-problem of fairly equal size to compute. Each sub-problem consists of a unique set of sequential objects available for the knapsack. The distribution of objects is nearly equal, so that each sub-problem will contain about n/p objects. The server divides the processor into groups, and communicates to each processor which other processors are in its group. The server then forks requests to the processors to solve the given sub-problem.

Upon receiving the weights and profits for the objects in the sub-problem, and capacity c for the knapsack, each solver begins the second step in parallel. Each solver begins computing a list of optimal states for each object. The idea is to create a list of tuples (\bar{w}, \bar{p}) . The value \bar{w} denotes a given capacity, and \bar{p} denotes the maximum profit at capacity \bar{w} considering the sub-problem defined up to the current object. At each stage of computation, the solver merges the list for the current objects with the previous list. A tuple (\bar{w}, \bar{p}) is said to dominate another tuple (\bar{w}', \bar{p}') if $\bar{p} \leq \bar{p}'$ and $\bar{w} \geq \bar{w}'$. If a tuple is dominated by another tuple, it is not included in the merged list. By merging

the lists to create only dominating states, we can reduce the total number of states, which in turn reduces the running time.

For example, consider the following knapsack with capacity $c=5$:

Table 4-1 Knapsack with 5 Objects and Capacity 5

Knapsack Capacity=5					
object index	0	1	2	3	4
object weight	1	1	1	1	1
object profit	1	2	3	4	5

Merging lists as described above creates the following lists of tuples for the knapsack in the previous example.

Table 4-2 Example of Merging Lists

Objects included	List of Tuples (p,w)
0	(0,0) (1,1)
0,1	(0,0) (2,1)(3,2)
0,1,2	(0,0) (3,1)(5,2)(6,3)
0,1,2,3	(0,0) (4,1)(7,2)(9,3)(10,4)
0,1,2,3,4	(0,0) (5,1)(9,2)(12,3)(14,4)(15,5)

In this example, when all objects are considered, there are 2^5 possible states. Merging reduces the number of possible states to six possible states. These six are shown in the last list in the above table. Although this technique of merging lists “does not change the worst-case complexity, it may considerable improve the computation time for

practical instances” (Kellerer, Pferschy & Pisinger, 2004). That is, problems may exist for which merging does not help, but it will reduce the number of states in many cases.

Once this merging is complete, the processors can create a profit vector from the list. The goal is to compute the final profit vector. Although I have not yet described how the final profit vector is computed, I will show the final profit vector the algorithm should compute for the above example. I include it to clarify the direction in which the algorithm is headed in the following steps.

Table 4-3 Final Profit Vector

Weight of object	Profit
1	5
2	9
3	12
4	14
5	15

The profit vector shown is the maximum profit possible at each possible weight given the objects and the capacity. The algorithm should compute the above profit vector once all the sub-problems have been combined.

If this problem were divided into two sub-problems, two unique profit vectors would be created in step 2. An example of the division into sub-problems and the resulting profit vectors are illustrated in the following problem.

Table 4-4 Profit Vectors With Two Sub-problems

Processor	Objects (w_j, p_j)	Profit vector (1..5)
1	(1,1) (1,2)	[2,3,3,3,3]
2	(1,3) (1,4) (1,5)	[5,9,12,12,12]

As expected, the two example resulting profit vectors are unique and must be combined to get the profit vector for the entire problem.

This combination is done in step 3. The vectors are combined in levels. At each level the solvers are paired. Each pair exchanges vectors. The vectors are combined and new profit and history vectors are created. This is done over a series of levels until all the profit vectors have been exchanged and combined. First I describe how the solvers determine how to exchange vectors. In the next section, we discuss how the combine and history vectors are created.

Exchange vectors in levels

In the first step, the server communicates group information to each solver so that each solver knows to which group it belongs. This group information includes the RMI network names so that the solvers can contact other solvers in the group. It also includes an ordering of the solvers in the group. The solvers use this information to determine who their neighbor is and how to contact it.

In (Lee, Shragowitz & Sahni, 1988), the solvers are assumed to be on a hypercube. In this case, each processor has a direct link to its neighbor. In a networked architecture, as implemented, a direct link might occur but is not guaranteed. No information about the distance between processors on the network is used to determine the neighbors. Instead, the implementation assures that each processor has one unique neighbor at each level. Each processor exchanges vectors with a different processor at each level.

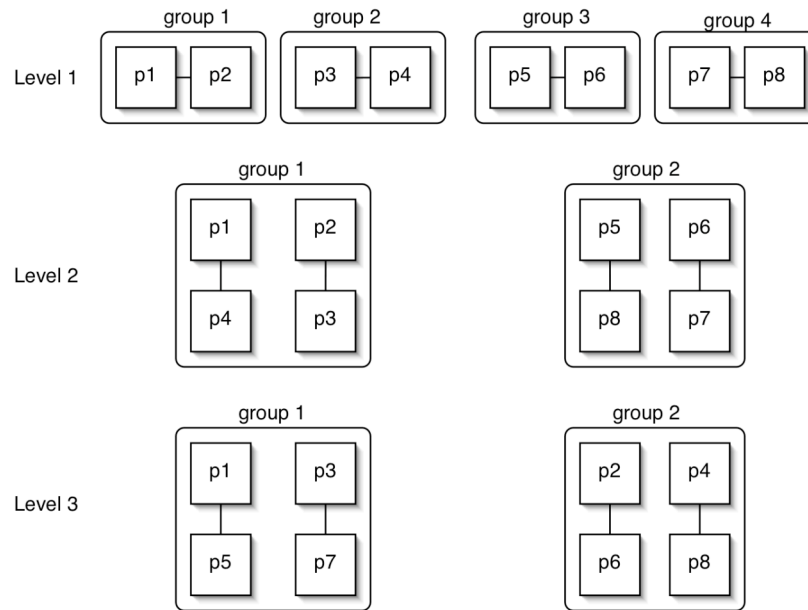
After exchanging vectors, and then combining to create a new profit vector and a history vector, each processor notifies the server that it has completed its task. Once the

server receives a notification from all the processors, it creates new groups of processors and notifies them of the new groups. The processors can then begin the process of determining their neighbor, exchanging the profit vector, and then combining to create a new profit and history vector.

Given that there are 2^k processors for some integer k , there will be n levels. The server divides the processors into n groups of 2 at the first level. The number of solvers per group progressively doubles until the second-to-last level is reached. At the final level, there will be two groups of $k/2$ processors in each group. At the final (k th) level, the server uses a different methodology to divide the processors, ensuring that the neighbors for each processor remain unique.

Following is a diagram of the division into levels and groups if there are 8 processors.

Figure 5.1 Levels in LEE



In the above diagram, the first solver is labeled “p1”; the second solver is labeled “p2” and so on. This example has $8 = 2^3$ solvers, so there are 3 levels in which the combine and history vectors are exchanged. The first level has four groups, and each group has two solvers. In this diagram, groups are represented by rounded boxes. Each solver exchanges vectors with one other solver in the group. The exchange is represented by a line connecting two solvers.

In the second level, there are now two groups of four processors each. The last level shows how the processors are again grouped into two groups. In this manner, every processor will exchange vectors with a processor that was in a different group at each stage. The vectors will, in the end, be exchanged such that the final combined profit vector represents the profit vector of the entire knapsack.

Combine and History Calculations

The paper (Lee, Shragowitz & Sahni, 1988) defines the combine and history operations and shows that the combine method is both commutative and associative, allowing us to combine the profit vectors in any order to compute the final profit vector for the given knapsack. Each solver combines its own profit vector with the neighbor's vector to create a new profit vector. At the same time, the solvers create a history of the combine procedure, which is used to trace back the steps later. The solvers do not store the profit vectors created at each level, but they do store the history vectors that result from the combining process.

Continuing with the knapsack from the previous example, and using the profit vectors as described above, the following table illustrates the new combined profit vector and the new history vectors.

Table 4-5 Combined Profit and History Vectors

Processor	Profit vector	history vector
1	[5,9,12,14,15]	[0,0,1,2,3]
2	[5,9,12,14,15]	[1,2,2,2,2]

The combined profit vectors are the same for the two neighbors. The history vectors, however, are not the same. At each possible weight they specify the index of the original profit vector that was used to get the value in the profit vector. A "0" value in the profit vector indicates that this processor did not contribute to the best profit at this capacity.

Once the final level is reached, the server has the final profit vector, which is the same for all solvers. Each solver has stored the history vectors that have been created at each level. The solvers communicate back to the server that they have finished this level.

The server recognizes that the maximum level has been reached. At this point, the server initiates step 4, the process of backtracking the combine procedure.

Backtracking Steps

The goal in the backtracking steps is to determine the best combination of profits and weights from each of the sub-problem to create the maximum profit. The solvers already have the best overall profit for the entire knapsack, which has been determined in the previous step by the final profit vector. The server begins with the profit and steps through the levels from the maximum level to the first level and requests the best contribution from each processor at each level.

When the first level is reached, there is a weight for each solver. This weight represents the state at which the sub-problems can contribute most profitably to the final knapsack. Once this weight is computed, the solvers begin step 5 in parallel. The backtracking process is similar to that described for other implementations. However, instead of using the knapsack capacity c , the backtracking uses the weight w determined in the previous set. It then determines which objects contribute to a knapsack with weight w .

The winning objects are returned to the server by the solver. Once the server has received the objects from all processors, it has the complete solution. In addition to the maximum profit for the knapsack, it has the objects that can fill the knapsack up to the maximum capacity and while obtaining the maximum profit. The server returns objects to the client. The server and the solvers await the next request from the client.

Differences Between Paper and Implementation

My implementation is different in several ways from the implementation by the authors of (Lee, Shragowitz & Sahni, 1988). In some cases, I had to modify their proposed algorithms to work in a grid situation. In other situations, the paper did not specify the implementation details, so I made decisions based on the best situation for my implementation.

Existence of a server

There is nothing described in (Lee, Shragowitz & Sahni, 1988) that takes on the role and performs the functions that KnapsackLeeServer performs. Yet, the server performs two large functions that are crucial to the algorithm described.

One task of the server is to organize the division of labor between the processors. The processors do not have full knowledge of the other processors involved in solving the problem. Only the server has the full list of processors who have registered to solve the problem. The server may distribute any information needed by the processors. For example, the server receives the initial request from the client about the full knapsack that must be solved. It then evenly divides the problem into sub-problems based on the processors that have registered. Also, the server divides the processors into groups in preparation for the forward combine phase. The server sends the individual processors only the information they need to find and contact their neighbor.

Another key task performed by the server is to organize the sequences of steps among the processors. Unlike the algorithm described in (Lee, Shragowitz & Sahni, 1988), my implementation could not rely on having identical processors that return their

results with the identical speed. Yet, the levels and regrouping must be done in parallel in this algorithm, and a single processor can not continue to the next level until all the other processors are finished with the current step. The processors therefore notify the server when they have completed the task at hand. The server stores the information, and when all the solvers have completed the task, prepares them for the next step.

Combine Method Not Parceled

My tests show that the network communication between the processors can be quite expensive. For this reason, I eliminated one step in the algorithm described in (Lee, Shragowitz & Sahni, 1988). This step involved dividing the profit vectors into smaller units in the forward combine phase. This would decrease the number of integer additions made in the combine and history phase. However, it also required a heavy additional communication load. The server would have to communicate to each processor not only which part of the vector it is to solve, but also which part all the other processors were to solve. Once the combining was complete, the processors would need to broadcast their results to all the other processors in their group, which would then have to process the broadcasts and modify their profit vectors at the appropriate location.

In my implementation, each processor performs about twice as many integer additions at each combine phase than the implementation described in (Lee, Shragowitz & Sahni, 1988). However, the calculations described above do not need to be performed. An additional benefit of my implementation is that there is significantly less communication between the processors when the combine method is not parceled.

LEE's Performance

This algorithm has impressive performance. It is able to solve a knapsack problem with up to 300 objects in less than one minute. The following table shows the results of running the algorithm on knapsacks of different sizes with different numbers of solvers. This table contains the averages times required to calculate six different knapsacks, and the standard deviation of the times. The format of each cell is a unit of time, and the times shown are in minutes, seconds, and a tenth of a second.

Table 4-6 Performance of LEE

solvers objects	2		4		8	
	avg	std dev	avg	std dev	avg	std dev
50	00:00.9	00:00.0	00:01.1	00:00.1	00:02.3	00:00.3
100	00:05.9	00:02.1	00:02.2	00:00.2	00:03.8	00:00.2
200	00:52.4	00:15.9	00:08.8	00:02.8	00:03.7	00:00.2
300	03:32.3	00:49.3	00:40.3	00:10.3	00:20.1	00:01.0

This table shows that the larger the knapsack, the more beneficial extra processors are to the solution time. The paper (Lee, Shragowitz & Sahni, 1988) provides experimental results in line with my results. The authors found that as the number of processors increased, this algorithm “is expected to result in substantially higher speedup... when instances of larger m [number of objects] are solved”. The original paper also points out that there is an optimal number of processors based on the problem size. A formula for determining this number is not provided and apparently can be decided experimentally.

The previous table allows us to begin to determine what this optimal number may be. The results show that the communication costs are more important for smaller knapsacks. At small knapsack sizes, it is better to have fewer processors. For knapsacks

with fewer than 100 objects, the system with two solvers performed better than those with four or eight solvers. This relationship begins to change as the knapsack size approaches 200 objects. At this point, the communication time between processors becomes a smaller percentage of the elapsed time. When knapsacks with 300 objects are examined, the average performance of the 8-processor solution is half that of the 4-processor option, and one sixth that of the 2-processor option.

LEE demonstrated good performance. In addition, the trends show that it will continue to perform well for large knapsacks if more solvers are added. I would recommend LEE for further testing and study for networked solutions of the 0-1 Knapsack problem.

Chapter 5 CHEN Algorithm

The parallel solution proposed in Pipeline Architectures for Dynamic Programming Algorithms (Chen, Chern, & Jang, 1990) for the 0-1 Knapsack Problem also takes a dynamic programming approach. However, instead of considering a hypercube of processors, this paper presents a solution that is computed in a pipeline-like fashion. In this algorithm, the processors pass data to their neighbor to the right, and receive data from their left neighbor. Computed data flows through the ring of processors like oil through a pipeline.

Dynamic Programming with a Pipeline Model

Recall from chapter 2 that the following equation can be used to solve the 0-1 knapsack problem.

Equation 5-1 Dynamic Programming

$$0 \leq j \leq c$$

$$0 \leq k \leq n$$

$$f_k(j) = \max\{f_{k-1}(j), f_{k-1}(j - w_k) + p_k\}$$

This function is amenable to a pipeline solution because data dependency only occurs between adjacent stages and within the same stage. For a pipeline architecture, in which each solver calculates f_k , this means that the calculation of f_k is only dependent on two things. The first dependency is on the previous processor calculating $f_{k-1}(j)$. This

means that data can be passed from left to right, and at each point the processor will only need the data received from its left neighbor to begin calculating a new object. The second dependency is that the current solver itself must have calculated up to $f_k(j-1)$. The calculations are not dependent on any other processors or calculations.

Implementation of CHEN

A representation of the ring network sending data like a pipeline is created as the solvers register with the server at startup. References to each solver are added to a doubly linked list in the server. This linked list will represent the ring of processors; a processor's "left neighbor" is represented by the previous processor in the list, and the "right neighbor" is the next processor in the list. By connecting the first and last processors, a ring is formed. For example, the last processor in the list considers the first processor its "right neighbor". In this way, we can treat a group of networked processors as if they were a ring of processors.

When the client sends a request to the solver, the server saves the number of total solvers available at the time of the request. The server calculates performance parameters based on the total solvers available and the capacity of the knapsack. These performance parameters, such as the number of levels to use in the calculation, are covered later in this section.

The server sends the knapsack definition to each processor. In addition, the server sends the performance parameters and information on how to contact each solver's right neighbor. The server then initiates the calculation by sending some empty data to

the first solver. The data sent indicates that the solver is to begin processing the profits for the first object.

In a pipeline implementation, the first solver would receive the data and calculate the profits for the first object. When it has processed these profits, it would send the information to its right neighbor. The first solver would then wait to receive the data sent by its left neighbor, which is also the last solver in the server's linked list. In the meantime, the second solver would begin calculating the data for the next object. When it finishes the calculation, it would send the data to the right processor and await more data from the left processor. In this way, the processors would calculate the values for each object, and pass the results to the next processor until the final object has been calculated. When the final object has been calculated, the processor runs the Hu backtracking algorithm and returns the results.

This process could allow multiple different knapsacks to be calculated in parallel. For example, once the second processor has begun processing the data for the first knapsack problem, the first processor could receive data on a new problem and begin processing that data. The processors would be solving different knapsacks at the same time.

However, we are focused on improving the parallel processing of a single knapsack. If only one knapsack is being calculated, no work is being done in parallel. Each processor will wait until the previous processor is finished to begin its calculations. A distributed implementation does not have any benefits over a single processor implementation for solving a single knapsack.

Pipeline Processing with levels

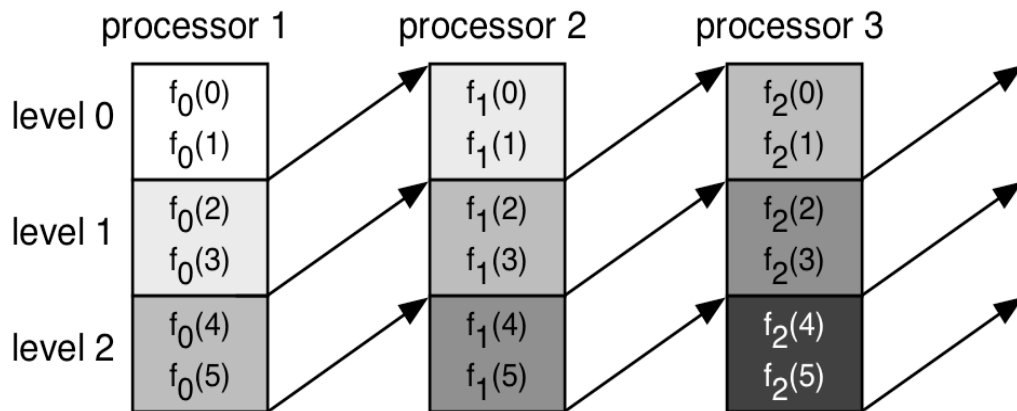
The paper (Chen, Chern, & Jang, 1990) suggests an improvement that allows for parallel processing even when a single knapsack is being solved. This improvement involves the use of levels. The levels used in CHEN are not analogous to those used in LEE, as we do not combine vectors as described in LEE.

Levels in CHEN are based on the observation made earlier that, when calculating the profit for an object k at capacity j , we need the values for object $k-1$ at capacities 0 through j . Therefore, a processor can calculate the profits for capacities 0-- j for object k , and then send these profits to the next processor. It can then continue calculating the rest of the profits for that object. The next processor can begin calculating the profits for object $k+1$ for capacities up to j . At the same time, the original processor will continue calculating the profits for object k at capacities $j+1$ and onward. The two processors will be doing the calculations in parallel.

We call this division of calculation a level. If the calculation used two levels, the processor would send its values on to the next processor twice. The first values would be sent when it had calculated the values for the object up to one half of the capacity. The second set of value would be sent when the entire object had been calculated.

The following diagram illustrates how data is calculated and passed to the next processor. The example shows a system with three levels and a capacity of five.

Figure 5-1 Pipeline with 3 levels and $c=5$



The diagram shows three solvers calculating the first three objects. Each processor calculates the profit for the object at two capacities, and sends the values to the next processor. In this example, we assume there are more than three objects to calculate, and the third processor is passing its calculated values on to the next solver. The shades of gray show what is being calculated in a given time period. For example, processor 1 is calculating level 1 at the same time processor 2 is calculating level 0, and they have the same shade of gray.

The solver sends the profit vector only for the current object and the history vectors. Since all the history vectors are needed for backtracking, the processor will send all the history vectors to the next solver. It sends only the profit vector for the current object. This profit vector contains only the best profits that have been calculated up to and including the current level.

I will give an example of profit vectors being passed in levels. The next table defines the Knapsack Problem that will be used in this example.

Table 5-1 Knapsack With 5 objects and Capacity 5

	Solver 1	Solver 2	Solver 3
Level 0	[<u>0,1</u> ,0,0,0,0]	[<u>0,2</u> ,1,1,1,1]	[<u>0,3</u> ,3,3,3,3]
Level 1	[0,1, <u>1,1</u> ,0,0]	[0,2, <u>3,3</u> ,1,1]	[0,3, <u>5,5</u> ,3,3]
Level 2	[0,1,1,1, <u>1,1</u>]	[0,2,3,3, <u>3,3</u>]	[0,3,5,6, <u>6,6</u>]

The next example shows the profit vectors that would be sent by each processor if there were three levels. It shows only the profit vectors sent by the first three processors, to be consistent with our previous figure. Profit vectors are discussed in chapters 2 and 4, and are not reviewed here.

Table 5-2 Profit Vectors Created in First 3 Solvers, in 3 Levels

	Solver 1	Solver 2	Solver 3
Level 0	[<u>0,1</u> ,0,0,0,0]	[<u>0,2</u> ,1,1,1,1]	[<u>0,3</u> ,3,3,3,3]
Level 1	[0,1, <u>1,1</u> ,0,0]	[0,2, <u>3,3</u> ,1,1]	[0,3, <u>5,5</u> ,3,3]
Level 2	[0,1,1,1, <u>1,1</u>]	[0,2,3,3, <u>3,3</u>]	[0,3,5,6, <u>6,6</u>]

In the table above, each cell represents the profit vector sent by the given processor at each level. The values that were calculated at that level are underlined. As we see, the first processor is gradually calculating the entire profit vector for the first object. While doing so, it is sending on the values to the next processor so that the next processor can begin calculating the profit vector for the next object.

The server determines the number of levels and sends this as a parameter to each of the solvers before calculations begin. The server also sends the capacities to be calculated in each level. The solvers receive these performance parameters from the server before they begin calculating the knapsack.

Lists of Change Points

Adding levels allows the processors to calculate a Knapsack Problem in parallel. Chen, Chern, & Jang suggest an additional method which should increase the speed at which the knapsack is calculated.

The suggested improvement tries to reduce the amount of computation by creating a list of capacity points at which the profit might change. I will refer to these points as "change points". Instead of calculating the profit for an object from 0 up to the capacity, only the profits at the change points need to be calculated. The list of change points will usually be smaller than the capacity, so fewer calculations would be required.

As suggested, the change points are the points in the profit vector of an object in which the profit might change. That is, the profit of an object at capacity j is a change point if it is greater than the profit at capacity $j-1$. Intuition tells us that the change points for an object are related to the weight of that object. The profit can change only because we have added the current object to a previous combination.

We will be calculating new change points for each object based on the change points of the previous object. The list of change points, denoted S , is initialized with zero and the maximum capacity for the first object. For every following object, S consists of the list resulting from the previous stages' calculations. The weight of the current object is added to each capacity point in S to create a new list T . A new list is formed from the union of the points in S and T . Now that we have the list of change points, the profit and history can be calculated for the current object at each point in the new list.

For an example of change point lists, I re-introduce the Knapsack Problem we examined in chapter 2, when examining the Greedy algorithm. The object's weights and profits can be seen in the second column. This knapsack has a capacity of 9.

Table 5-3 Lists of Change Points

Object index	(w,p)	S	S U T
0	(2,6)	0,	0,2
1	(3,5)	0,2	0,2,3,5
2	(6,8)	0,2,3,5	0,2,3,5,6,8,9
3	(7,9)	0,2,3,5,6,9	0,2,3,5,6,7,8,9
4	(5,6)	0,2,3,5,6,7,9	0,2,3,5,6,7,8,9
5	(9,7)	0,2,3,5,6,7,8,9	0,2,3,5,6,7,8,9
6	(4,3)	0,2,3,5,6,7,8,9	0,2,3,4,5,6,7,8,9

The table above shows the original list S with which an object starts. The column titled S U T shows the union of S and T. This column shows the list of unique points resulting from adding the weight of the current object to every point in S.

The change point list will grow in size with each object, as more possible change points are calculated. The authors suggest a method to minimize the growth of these lists. At each stage, the points are tested for dominance, and the points that are dominated are dropped from the list. Dominance can occur in two ways. First, if the point x came from original list S, it is considered dominated if the profit of including the current object is greater than not including the object. Secondly, if the point came from the new list T, then the point is dominated when the profit of including the current object is less than or equal to the profit of not including the object.

For an example of dominance, consider the change points for the object with index 1 in our example above. In the table, the change point "3" was added to represent adding the weight of the object to 0, one of the values from S. However, this point is an

example of the second kind of dominated point. The point 3 comes from our new list T. However, the profit of object 1 at capacity 3 is 5. This is less than the profit if we do not include object 1. If we do not include object 1 at this point, we will include object 0 with a higher profit of 6. Therefore, the point 3 is dominated, and could be merged out of the change point lists.

Calculating the list of change points and then checking the dominance of each point incurs some additional overhead. However, according to (Chen, Chern, & Jang, 1990), “the overhead occurred by checking the dominance [of points in S and T] is negligible when it is compared with the gains of efficiency from reducing computation and communication steps”. The last section in this chapter will examine the results of my implementation using capacity points. We will compare this to the results if the entire range of points is calculated, but the capacity lists do not have to be created.

Implementation of CHEN

In my implementation, I combine the two improvements that are suggested by (Chen, Chern, & Jang, 1990). I use levels and I create lists of capacity points. Each solver uses the level scheme to determine when to send data to the right neighbor. In addition the solver calculates the capacity points in an attempt to reduce the calculation required. Each solver must send the capacity point list as well as the profit and history vectors to its right neighbor. The right solver then performs calculations only at the capacity points that are within the range of the current level. By doing so, we get the benefits of parallel processing from using levels, and we also get the reduction in computation steps from using the lists of capacity points.

Since we are communicating over the network, we cannot rely on messages from the left neighbor arriving in order. Although a solver may send the values for level 1 and then the values for level 2, the right neighbor may receive the values for level 2 before it receives the values for level 1. In LEE we were able to rely on the server for synchronization of data messages. However, CHEN does not communicate with the server during computation, so an individual solver must be sure to perform extra checks to ensure that it does the calculations in the correct order. When the CHEN solver receives a message from its left neighbor, it must not try to begin calculation on data it does not yet have. Secondly, if the messages are received in the incorrect order, the solver must not overwrite data for a higher level with data from a lower level.

The order of calculations is ensured by adding some extra checks into each CHEN solver. First, many of the methods use Java synchronization. Synchronization prevents the same method from being executed more than once at a given time. This prevents a second message from interrupting the first one. However, synchronization is not enough. Messages might still be received in the incorrect order.

When the solver receives information about a new object, the solver begins the calculations for all levels of the object. As the calculation reaches the capacity of a new level, it confirms that it has already received the information for this level from its left neighbor. If it has not, it waits for the data to be sent. Meanwhile, the left neighbor is sending new values. If the solver receives data for a level greater than the level it already has, it will copy the new data into memory. Otherwise, if the level is less than that previously received, the solver has data that supersedes the received data. The solver does nothing with the values.

Therefore, if a solver receives a message with data for level L, it can perform computation on all data up to and including L. If the messages are out of order and the solver next receives a message with data for L-1, it will discard this data. When it receives the data for L+1, it will begin calculations for L+1 only when it has finished calculations for L. In this manner, the solver will not perform calculations on data that is too old, and will not calculate profits using data that it has not yet received.

CHEN's Performance

The CHEN algorithm performs decently for knapsacks with a small number of objects. It took over an hour to calculate any knapsack problem with more than 300 objects. Also, there was a fairly high variance in the performance for each test set.

The following table shows the average results across six attempts of the CHEN algorithm. The same test sets were used in the LEE algorithm. The performance of each test differed greatly. Hence, I have included both the averages of the six tests and the standard deviations at each point. The dashed line indicates that tests were cut off when they required more than one hour.

Table 5-4 Performance of CHEN

Objects	2 solvers		4 solvers		8 solvers	
	avg	std dev	avg	std dev	avg	Std dev
50	00:59.9	00:36.9	00:58.1	00:15.1	00:53.3	00:15.6
100	03:01.4	01:24.0	02:29.0	00:59.3	04:43.2	02:12.2
200	07:09.0	03:59.5	06:49.4	03:06.2	20:41.3	08:17.4
300	-	-	-	-	-	-

The results in the table above are disappointing for a number of reasons. Most importantly, the communication costs of this algorithm are never outweighed by the

amount of work done. The tests with eight solvers consistently performed worse than the tests with two or four solvers. I do not believe larger tests would have yielded different results. In this algorithm, the number of messages grows proportionally with the number of solvers. The number of calculations being done before a message is sent is the capacity divided by the number of solvers. Therefore, as the number of processors grows when solving knapsacks of similar sizes, each processor will perform fewer calculations before sending a message. In addition, each processor will send more messages.

While implementing this algorithm, I felt that the extra calculations added by calculating the change points might not make up for the amount of calculation saved. I added an option to run the solvers without using the changes lists. If this option were used, the solvers would calculate the full profit vector for each object instead of calculating the merge points. The following table compares the results from calculating the change points versus calculating the full range of capacities.

Table 5-5 Comparison of Full Range vs. Change Points

Objects	2 solvers		4 solvers	
	chnge pnts	full range	chnge pnts	full range
50	00:59.9	00:41.8	00:58.1	00:42.5
100	03:01.4	02:46.2	02:29.0	02:03.3
200	07:09.0	13:37.0	06:49.4	09:52.9
300	-	-	-	-

It looks as though the change points might begin to show some benefits with the larger knapsacks. However, this is certainly not conclusive, since the differences between using change points or not falls within the standard deviation of the averages. Nevertheless, I continue to use change points in all the tests. All results for CHEN outside of the above table were calculated using lists of change points.

Overall, the results for CHEN are disappointing. Knapsacks of only 300 objects take over one hour to calculate. Extra solvers are not seen to improve the performance time. Additionally, there is a large variance in expected performance time. I would not recommend CHEN for solving a single large knapsack in a networked environment.

Chapter 6 HUNTPL Algorithm

The HUNTPL algorithm is an original idea. However, it was inspired by the algorithm described in (Chen, Chern, & Jang, 1990). HUNTPL, named after “Hunt” and “pipeline”, could be considered a modification of the algorithm presented in the previous chapter.

Similarity to CHEN

HUNTPL is very similar to CHEN. As in CHEN, the server orders the solvers into a list and tells each solver how to contact its neighbor. The server sends some performance parameters to the solvers before the calculation begins. When the server receives a request from the client, it initiates the calculation by sending a message to the first solver. Once the communication begins, the server is not an active participant in the calculation.

The HUNTPL solvers communicate in a method similar to the CHEN solvers. The HUNTPL solvers send messages only to their right neighbor, and receive data from the left neighbor. The calculated data moves from left to right through the solvers until a solution to the knapsack has been found.

Levels as Implemented in HUNTPL

This algorithm attempts to reduce the amount of communication required in a pipeline implementation. Compared to CHEN, it performs more calculations before

sending a message to the next solver. It also distinguishes itself from CHEN in that it does not act like a ring of processors. It is instead mimics a linear array, as described in Chapter 1. The last solver does not send a message on to the first solver. Instead, all the calculation is completed in a single left-to-right pass through the list of processors. In order for the entire knapsack to be calculated in one pass, each solver must process a larger chunk of data.

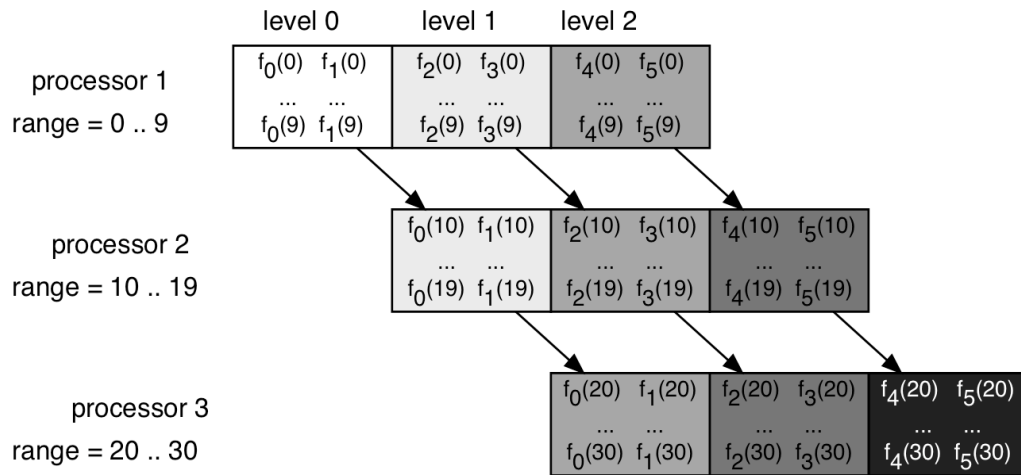
In CHEN, the solvers calculated all the profits for a single object. It would not begin calculating profits for another object until each processor on the ring had calculated an object, and it was the solver's turn again. The solvers in HUNTPL do not calculate the profits for a single object at a time. Instead, each solver calculates a fixed range of capacities for all objects. This next processor then begins calculation on the same objects for the next higher range of capacities.

The range of capacities is distributed among the solvers as evenly as possible. If there are p solvers working to solve a knapsack with capacity c , each solver will calculate a range of approximately p/c consecutive capacities. Each solver will calculate the first object, then the second object and so on until it has calculated the range of capacities for all the objects.

At intermediate stages in the calculation of objects, it will communicate the calculated values to the next processor. This will allow the right solver to begin working on one portion of the problem while the left solver continues its calculations. These periodic messages are sent in levels of objects. That is, a level consists of some number of objects that are to be calculated before the message is sent.

This is illustrated in the following diagram. This diagram shows an example knapsack with six objects and a capacity of 30. There are three solvers in this example. Each solver is shown calculating about 1/3 of the capacities.

Figure 6-1 Levels and Ranges in HUNTPL



The diagram illustrates what is being processed in parallel by the shades of gray. Blocks that share the same shade of gray are being processed at the same time by the separate solvers.

This example shows the 6 objects being calculated in 3 levels. Each level has 2 objects. The first processor will calculate the profits obtained at capacities 0 through 9 of the first object. Then it will calculate the profits obtained at capacities 0 through 9 for the second object. It has now completed a level and will send a message to the next processor. While the first processor continues to calculate the profits for the following object at capacities 0 through 9, the second processor begins to calculate the profits for the first object at 10 through 19. The second processor and the first processor will be

processing profits in parallel. This will continue until the last solver has calculated the profit for the final object at the knapsack capacity.

Once the final profit has been calculated, the final solver will perform the Hu backtracking algorithm to determine which object should be included in the knapsack. It will then send the results to the server.

Communication in HUNTPL

Like in CHEN, there is no guarantee that messages from the left processor reach the right processor in the order they were sent. In fact, my tests showed that is likely that messages are received in the incorrect order. Therefore, each solver must be able to store data it receives until it is ready to use it to perform calculations. Also, it must be able to wait for the data it needs to continue its calculations. Third, it must be prepared to ignore data if it has received more recent data.

While the solver is performing the calculations, it receives messages from the left neighbor. These messages contain the history vectors and the partial profit vectors for all the objects. In addition, the message includes the highest index of the object calculated by the left neighbor. Therefore, the solver knows which object's data to store for future calculations. It will copy the relevant data to a local array to be used for further calculations.

When a HUNTPL solver receives the first message from its left neighbor, it begins calculating the profit ranges of the objects. It keeps track of the highest indexed object it has received from the left neighbor, and will not calculate beyond that object. If

it gets to that point in the calculation and has not received the data, it waits until the left solver sends the necessary information.

If the solver receives a message with an index lower than an index from a previous message, it will already have all of the data sent. It can disregard the message that apparently came out of order.

As mentioned above, each message includes the history vectors for all objects, and the partial profit vectors for each object. The solvers do not send a profit vector with length capacity for each object. Instead, they send profit vectors only for the capacities it has calculated. Therefore, the size of the message will increase with each solver. Only the last object will calculate the profit vectors to the full capacity, but it does not have a right neighbor. It will not send the profit vectors on. Therefore, the size of the profit vectors being sent in the messages will increase to the highest capacity calculated by the second to last solver. Therefore, these messages can grow quite large.

HUNTPL's Performance

The performance of HUNTPL is much better than that of CHEN, but it has similar weaknesses. The following table shows the results from running HUNTPL. Just as for CHEN, this table shows the averages and standard deviations of six tests.

Table 6-1 Performance of HUNTPL

<i>Solvers</i> objects	2		4		8	
	avg	std dev	avg	std dev	avg	std dev
50	00:00.6	00:00.1	00:02.1	00:01.0	00:06.0	00:02.9
100	00:01.1	00:00.1	00:03.3	00:00.6	00:15.5	00:03.2
200	00:01.3	00:00.1	00:05.9	00:02.9	00:25.6	00:06.6
300	00:03.5	00:01.1	00:15.4	00:01.6	04:34.9	01:30.1

HUNTPL was able to calculate of a 300-object knapsack in less than four seconds. Unfortunately, this result was processed with two processors. The same calculations with eight processors took over four minutes. Like CHEN, HUNTPL does not benefit from the additional processors.

HUNTPL also suffers from a great variability in the calculation time. As the number of objects grows, it becomes more difficult to predict the total time it will take to calculate a knapsack. The variability also appears to increase with more processors. The more network messages are sent, the more likely it is that network latency can significantly delay a message.

HUNTPL offers an improvement on CHEN. The time to calculate the same Knapsack problems is definitely improved with HUNTPL. However, it does not give us the benefits we are looking for in a good parallel algorithm. Namely, we want additional solvers to be a benefit to the calculation. Although the HUNTPL algorithm performs remarkable well, it does not shine as a potential for parallel processing.

Chapter 7 Comparison of the Algorithms' Performance

This chapter compares and contrasts the performance of the three algorithms implemented for this thesis. I will discuss different types of problem domains, and how the algorithms perform in each of them. I will also discuss the affect of implementing the algorithms in a networked environment, and the costs of the networked communication.

The goal for each of these algorithms is to see a linear speedup for a single knapsack based on the number of processors. The total time for each calculation was determined by running a “time” function on the client side, which calculates the total elapsed seconds. The results show the time from when the client started, sent a knapsack definition to the server, and finally received the result. If any single test knapsack took more than one hour to compute in a test, the data is not included in the results or averages. It is marked with a "-" sign in the figures.

Description of Tests

The tests were run in two environments. The first environment is what I call a “pseudo networked” environment. In this environment, the client, server and solvers were all started as separate instances on the same machine. The packages were still sent using RMI, and hence relied on the TCP/IP protocol. The tests in this environment give us a comparison of how the algorithms compare in a controlled environment, without network variability. All tests were performed in this environment unless stated otherwise. The

hardware used was an IBM with an Intel Pentium M, 1600 MHz processor running Microsoft Windows XP.

The second environment is more realistic for this problem. In this environment, the calculations are performed over a networked environment using four different systems on two different operating systems. The four machines were on the same subnet of a wireless network. An additional fifth machine on the subnet was not involved with the calculations. However, this machine was performing network communication to purposely introduce variability and latency into the network. The hardware used for calculation includes the following four machines:

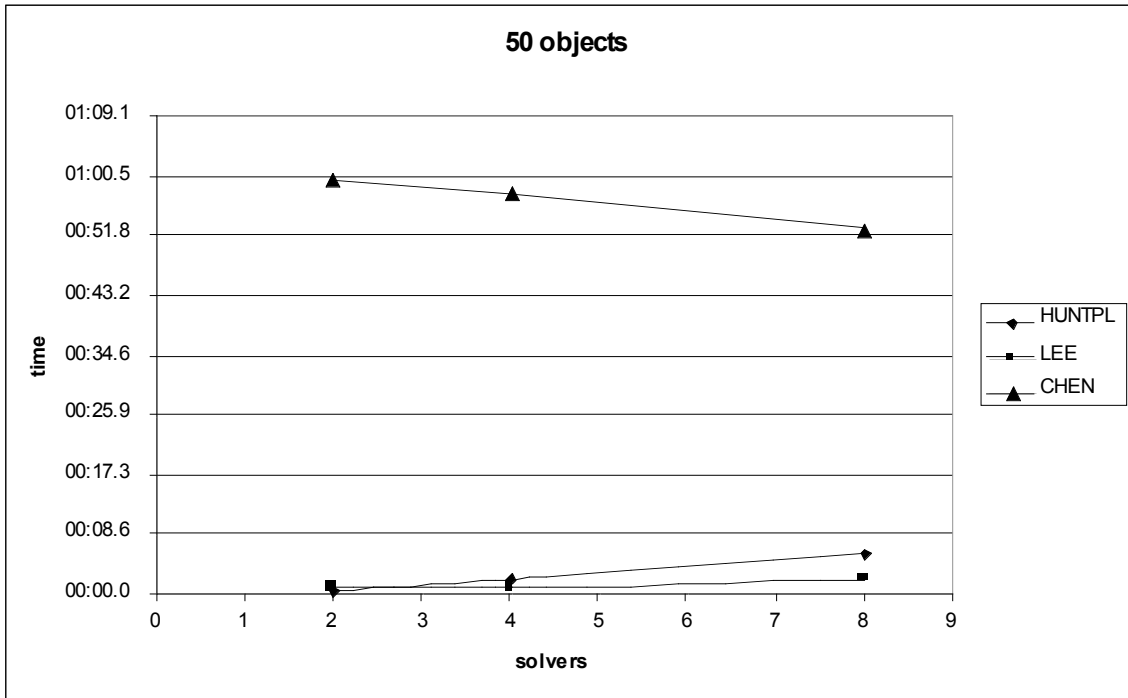
1. IBM with an Intel Pentium M, 1600 MHz processor running Microsoft Windows XP
2. Apple 1.33 GHz PowerPC G4 running Mac OS 10.3.3
3. Apple 1.33 GHz PowerPC G4 running Mac OS 10.3.5
4. Apple 667 MHz PowerPC G4 running Mac OS 10.3.5

Performance for Multiple Solvers

The following graphs compare the performance of each of the algorithms over the different on knapsacks of different sizes. These graphs allow us to compare how each algorithm performed on knapsacks with 50, 100, 200, or 300 objects. These tests were run in the pseudo-networked environment; the results between runs can be compared without trying to factor in network variability. Each point on the graphs represents the average time required to solve six knapsack problems. The six knapsack problems were the same for each algorithm for each knapsack size.

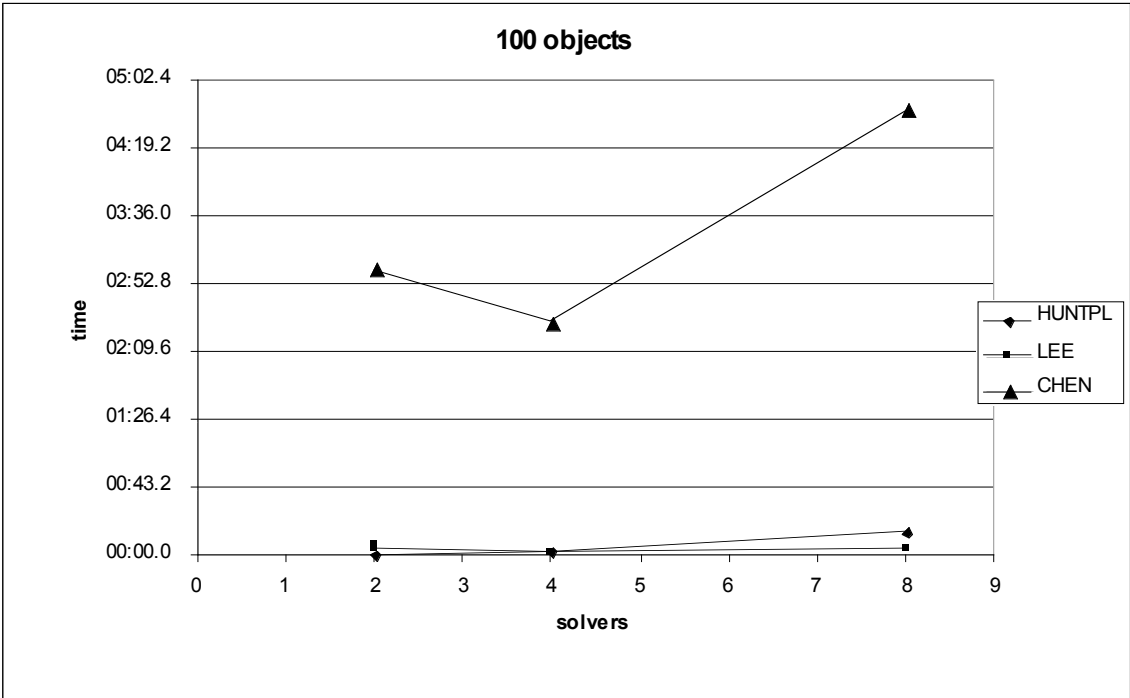
Each algorithm was tested using two, four and eight solvers. These numbers were chosen so that they would meet LEE's requirement that the number of processors be some power of two.

Figure 7-1 Comparison of Algorithms over Knapsack Size 50 in Pseudo-Network



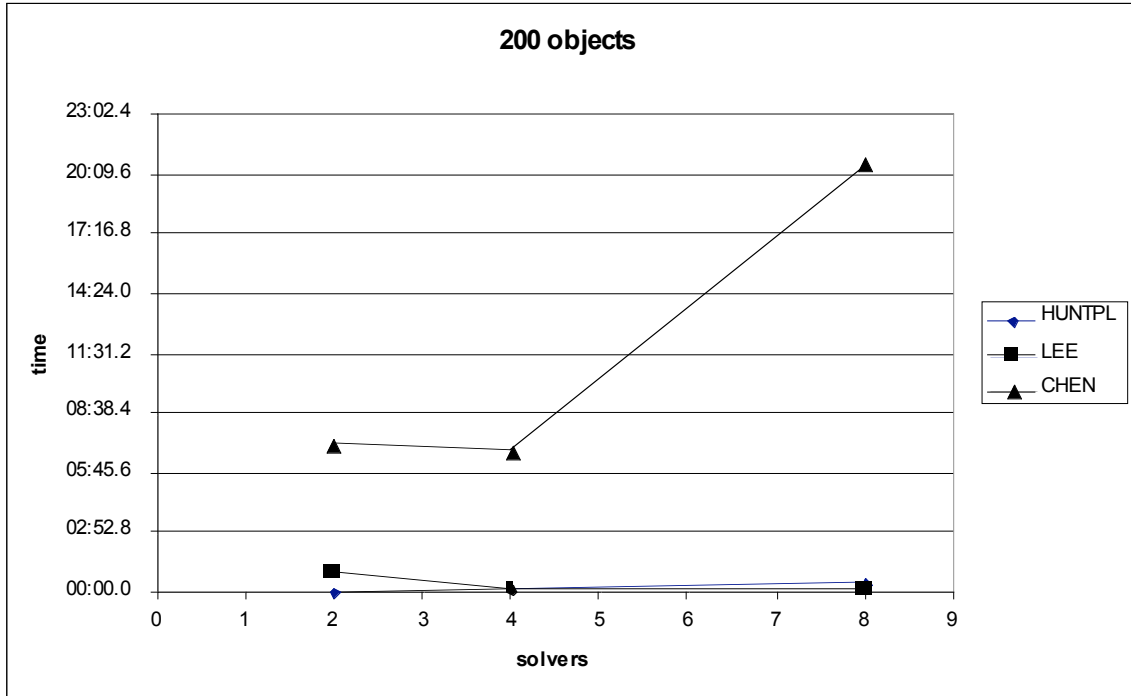
This diagram compares how the algorithms perform for knapsacks with 50 objects. With knapsacks this small, I expect the network communication to take a large percentage of the total time required to solve the problem. In this diagram, CHEN clearly has the worst performance for small knapsacks. In addition, both HUNTPL and LEE show a trend towards longer times as the number of solvers is increased. For small number of solvers, HUNTPL and LEE both perform well.

Figure 7-2 Comparison of Algorithms over Knapsack Size 100 in Pseudo-Network



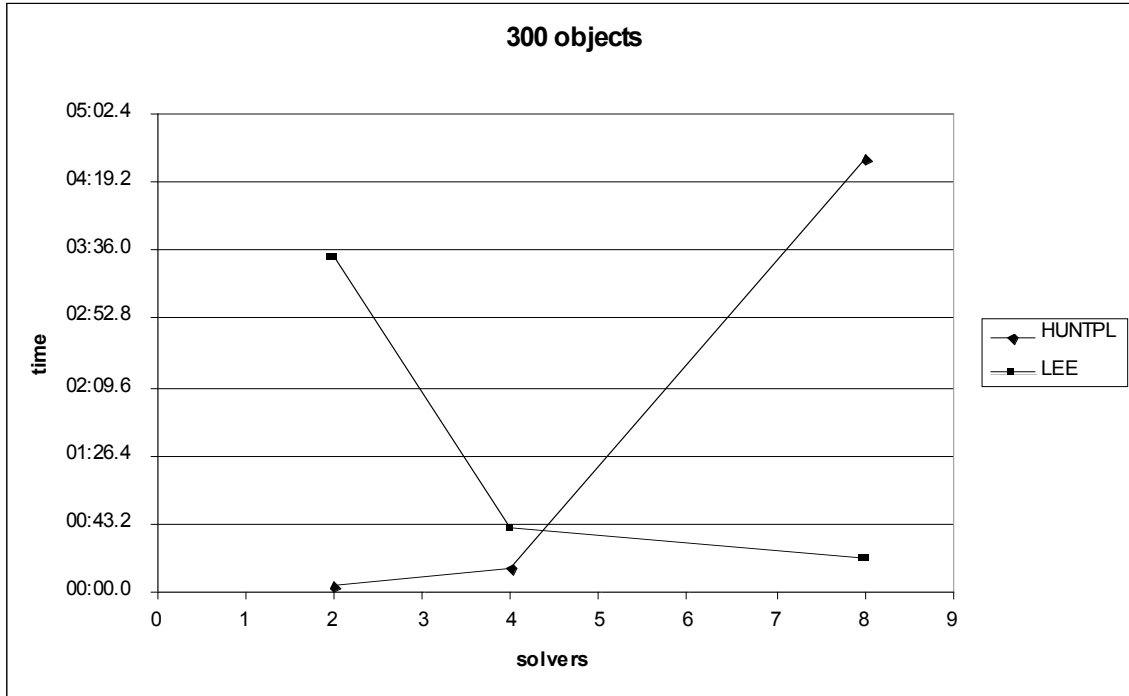
In the above diagram, I compare how the solvers perform on knapsacks with 100 objects. The best calculation times are from HUNTPL and LEE when run on four solvers. Once again, CHEN is shown to have much worse performance in all cases. CHEN also shows a drastic increase in time required as the number of solvers increases from four to eight.

Figure 7-3 Comparison of Algorithms over Knapsack Size 200 in Pseudo-Network



This diagram shows the average performance of the different algorithms on knapsacks with 200 objects. At 200 objects, LEE's best performance is when eight solvers are used. CHEN is displaying the same trend as in the previous graph, in that the time required is rocketing at eight solvers. HUNTPL also shows a decrease in performance when the calculations go from four solvers to eight solvers.

Figure 7-4 Comparison of Algorithms over Knapsack Size 300 in Pseudo-Network



The above graph shows how HUNTPL and LEE perform on knapsacks with 300 objects. The results for CHEN at 300 objects are not shown, as these calculations took over sixty minutes. The trends for HUNTPL and LEE are seen more clearly in this graph, since the y-axis showing time is on a smaller scale than the previous graph. HUNTPL seems to perform more slowly as more solvers are added, while LEE is improving. Overall, HUNTPL shows the best performance for 300 objects. With two solvers, HUNTPL takes 9 seconds. LEE takes 20 seconds with eight solvers. The difference between these two times is statistically significant. However, it is possible that LEE's time would improve enough with more solvers to be in the same range as HUNTPL.

As stated in previous chapters, the number of messages increases as the number of processors increase. I believe the number and size of messages is the culprit for the

decreasing performance with additional solvers in HUNTPL and CHEN. In the next section, I compare the messages sent in each algorithm.

Comparison of Network Communication

In each algorithm, data is communicated between the client, the server and the solver. Several types of communication are performed by each algorithm with exactly the same message sizes. One such type of communication is that between server and client in which the knapsack definition is sent and the result is returned. Another such communication is executed when each solver connects to the server, and the server must register each of the solvers. Finally, the server must send each solver some performance parameters before calculations begin. These messages are roughly the same size across algorithms. Since all of these forms of communications amount to roughly equal network traffic, they are not considered in the following comparisons.

I use the notation p for the number of solvers involved in the calculations. The symbol n represents the number of objects in the knapsack.

The CHEN algorithm sends a large number of messages. At the beginning, each solver connects to its neighbor, which amounts to p messages. The server sends a message to one solver to begin calculation. If there are ν levels, each solver sends ν messages per object. If we total these, we see that CHEN sends $p + n\nu p + 1$ messages per knapsack calculation. In my testing, I used two levels, so the number of messages is $p + 2np + 1$. The size of the messages sent by CHEN is fairly large. The messages sent in between solvers contain all the history vectors for the knapsack, and one profit vector, in addition to an integer for the level.

The goal in designing the HUNTPL algorithm was to reduce the total number of messages sent, and this goal was attained. Each solver must connect to its neighbor, and the server sends one message to a solver to initiate the calculation. HUNTPL uses n/p levels, so each of the p solvers will send n/p messages. This results in $p + p * n/p + 1 = p + n + 1$ messages sent per calculation of the knapsack. However, one drawback of this algorithm is that the messages can grow quite large. HUNTPL sends the history and profit vectors for all the knapsack items in each message. Therefore, HUNTPL is sending one table of length capacity and width n in all of the messages, and another table with a length that increases up to $n - c/p$ and width n .

The number of messages sent in the LEE algorithm is also dependent on the number of solvers. The LEE algorithm requires 2^j solvers for some j , where $2^j = p$. There are j levels in the calculations. First the server sends a message to each solver requesting that it begin calculation. At each level, the solver will send one message to the server. Each solver also connects to one solver and sends a message at every level, resulting in $2pj$ messages. At each level, the server will send one message to every solver with information about the groups, again resulting in pj messages. The total is therefore $p + 4jp$ message per knapsack computation. However, many of the messages sent between the server and the solvers are relatively small, containing a single string and an integer. Other messages contain the entire profit vector. Therefore, about half of the messages are roughly the same size as the messages sent in CHEN, and the rest of the messages are significantly smaller.

The number of messages sent in all the algorithms increases based on the number of solvers. CHEN is the only algorithm in which the number of messages also increases

based on the number of objects. HUNTPL tends to send the fewest messages, but they are also the largest messages by the end of the computation. The number of messages sent in LEE grows by a factor of $4j$ for each solver added, but the messages are smaller. Therefore, CHEN has the heaviest message burden.

Networked vs. non-networked

I expected that algorithms with the most and largest messages would be impacted more by moving to a networked environment. I determined how much each algorithm was affected by being run on separate computers over a network. The following comparisons show how much the average time for each algorithm increased when run on the same tests with the same number of solvers over the network.

Table 7-1 Percentage Slowdown in Networked Environment

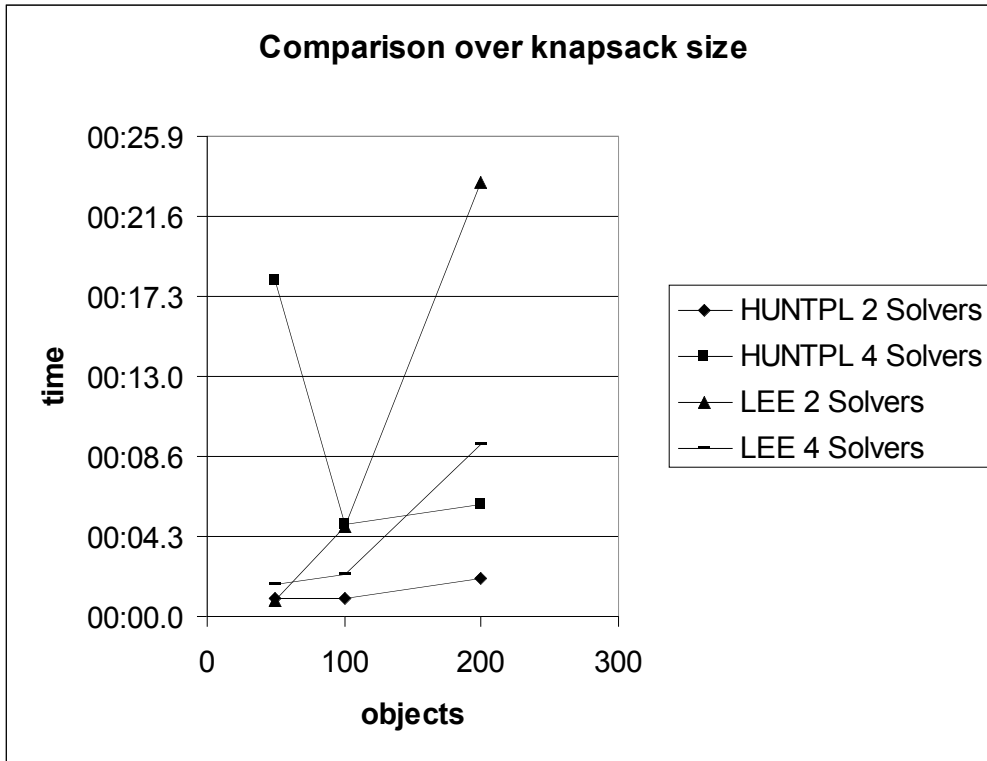
CHEN	50	2	15%
CHEN	50	4	252%
LEE	100	2	25%
LEE	100	4	530%
HUNTPL	100	2	26%
HUNTPL	100	4	27%

The percentages represent the increase in the average speed from the non-networked environment to the networked environment. For example, CHEN on 2 processors with 50 objects took 15% longer to complete in a networked environment than in the pseudo-networked environment. The algorithm most greatly affected by the network appears to be LEE with 4 solvers.

The following table compares HUNTPL's networked performance with LEE's networked performance on 50, 100 and 200 objects. It shows the performance at 2

solvers and at 4 solvers. CHEN's performance is not shown, as it remains much worse than HUNTPL and LEE.

Figure 7-5 Comparison of HUNTPL and LEE on networked solvers



This chart is intended to compare overall performance at the different knapsack sizes. HUNTPL, the algorithm which sent the fewest messages, consistently has the best performance when two solvers are used. However, this observation is tempered with two other remarks. First, these tests were not run on large knapsacks with many solvers. This was precisely the environment in which LEE showed the most improvement. Second, the results for HUNTPL with four solvers at 50 objects were biased by a single run which took about one minute. This demonstrates HUNTPL's propensity to have

irregular run times; the results for HUNTPL are less predictable, and the averages are less reliable. The larger messages in HUNTPL may cause unpredictable delays.

This section has demonstrated that algorithms for networked environments should aim to reduce not only the number of messages but also to reduce the size of messages sent in the course of calculation.

Problem domains

I tested each of the algorithms on six types of problem domains. These problem sets are originally defined in the paper titled "Where are the hard knapsack problems" (Pisinger, 2003). This paper defines several types of problem domains that were selected "to reflect special properties that may influence the solution process" (Pisinger, 2003). Each domain defines data within a range of values. In my case, the range is from 1 to some maximum value R . My problem sets used 100, and 1000 for R . The weight of a given object is denoted w , and the profit of a given object is denoted p . These problem domains are defined as follows:

Uncorrelated: The weights and profits are uniformly distributed within the range, with no correlation between weight and profit.

Weakly correlated: The weights are distributed in $[1, R]$ and the profits are distributed in $[w - R/10, w + R/10]$ such that the profit is greater than 0.

Strongly correlated: The weights are distributed in $[1, R]$, and the profits are set to $p = w + R/10$.

Inverse strongly correlated: The profits are distributed in $[1, R]$, and the weights are set to $w = p + R/10$.

Almost strongly correlated: The weights are distributed in $[1,R]$, and the profits are distributed in $[w + R/10 - R/500, w + R/10 + R/500]$.

Subset-sum instances: The weights and profits for each item is the same. The values are randomly distributed in $[1,R]$.

Uncorrelated instances with similar weights: The weights are distributed in $[1000000, 100100]$ and the profits are distributed in $[1,1000]$.

This Pisinger paper describes the results of tests of several algorithms to compare which are the best for each data domain. Pisinger tests algorithms with quite different methods such as dynamic programming, and branch and bound, and combination algorithms. It is easier to see that these different types of algorithms would fare differently in different domains. For example, properties of the subset sum data make it more difficult for branch and bound, since the bounding does not eliminate branches effectively.

However, it is not clear that the data domains show much effect when all the algorithms to be compared use dynamic programming. In dynamic programming, the amount of calculation is based on the number of objects and the capacity of the knapsack, and is not dependent on other properties of the items.

I created three copies of each type of data set, and ran LEE, CHEN, and HUNTPL on them. I then averaged the results to determine if any of the algorithms performed particularly well on any of the data sets. I did not test uncorrelated instances, as the suggested ranges were out of the bounds of my testing. I performed two tests like this, one on data sets of size 100, and one on data sets of size 200. I did not find any statistically relevant differences between the data types for any given algorithm. As the

algorithms had a high variability in any case, it would need to be a large difference to be statically significant.

In this section, I have compared and contrasted the three different algorithms presented. HUNTPL has the best performance overall on the network. LEE can solve knapsacks quickly but is affected by network latency. CHEN has the worst overall performance.

I investigated two possible differences that might affect the algorithms' performance. One was the amount and size of messages passed in an algorithm. The other possible difference was the type of data set. I found a correlation between performance and the number of messages. I did not find any correlation between data domains and performance for these three algorithms.

Chapter 8 Summary and Conclusions

This paper has presented an overview of the 0-1 Knapsack Problem. There are many suggested parallel algorithms for the 0-1 Knapsack Problem. I implemented and presented three algorithms to solve a 0-1 Knapsack Problem in parallel with networked computers. All three of the algorithms discussed in this paper use the dynamic programming method. They varied in their approach to dividing the problem into sub-problems among the solvers, communicating the calculations, and recombining the sub-problems.

Algorithms for parallel processing that are written for specific network or hardware architectures benefit from controlled and reliable communication methods. Algorithms for specialized hardware can assume that the processors solving each sub-problem are identical, and will therefore perform equal amounts of work in the same amount of time. Second, they can assume a consistent and ordered communication system.

My implementations were not able to make such assumptions about their networked solvers and the communication between them. Not only might the solvers perform calculations at different speeds, but messages sent over the network might arrive late or not at all. In fact, one of the more interesting aspects of implementing these algorithms was allowing for such differences. In the algorithms implemented based on

published papers, I had to make significant changes to allow for the variability in communication.

The communication costs clearly affected the performance of each algorithm. The CHEN algorithm sends many messages and does not perform well in a networked environment. The HUNTPL algorithm sent fewer message, but each message was very large. HUNTPL's performance degraded significantly when tested in a real networked environment. My results showed that an algorithm's performance could degrade if too many messages or messages that are too large must be sent over the network.

More work could be done to study these three algorithms. The tests could be extended to larger knapsacks and more solvers to see if there is a limit to the improvement gained by adding solvers in LEE. Additionally, it would be interesting to test to see if there is a large knapsack size for which additional solvers would begin to help HUNTPL.

In addition to testing, the three algorithms could be studied for improvements. For example, it might be possible to reduce the communication costs in HUNTPL, perhaps by calculating change points and sending only the data needed for the change points. Another possibility would be to modify LEE so that it does not require the number of solvers to be some power of two. Also, it could be worth studying whether these algorithms could be extended to solve other variants of the knapsack problem.

Additional algorithms could be studied for possible implementation in a parallel manner. For example, some algorithms involve a combination of dynamic programming

and branch and bound. Further exploration could determine whether these combination algorithms could be modified for parallel implementations.

The lessons learned in this thesis could be extended to other problem types. Although the algorithms implemented in this paper are specific to the 0-1 Knapsack Problem, the lessons learned about message and network communications are important when creating parallel algorithms to be run on networked computers. The work in this paper demonstrates important considerations for modifying algorithms to run on networked solutions.

Chapter 9 References

Alexandrov, V., Megson G., (1999) Parallel Algorithms for Knapsack Type Problems, World Scientific Publishing Co. Singapore

Andonov, R., Raimbault, F., Quinton, P., (1993) Dynamic Programming Parallel Implementations for the Knapsack Problem, submitted for review to the Journal of Parallel and Distributed Computers

Chen, G., Chern, M., Jang, J., (1990) Pipeline Architectures for Dynamic Programming Algorithms, *Parallel Computing* 13 pp 111-117

Fréville, A. (2003) The multidimensional 0-1 knapsack problem: An overview, *European Journal of Operational Research* 155 pp 1-21

Gamma E., Helm R., Johnson, R., Vlissides, J., (2002) Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Indianapolis, IN

Gerasch, T. , Wang, P., (1993) A Survey of Parallel Algorithms For One Dimensional Architectures *INFOR*, 32, 3 pp. 163-186

Gordon, V., Boehm, A. P., Whitley. D., (1994) A note on the performance of genetic algorithms on zero-one knapsack problems *Proceedings of the 1994 ACM symposium on Applied Computing* pp 194-195

Harold, E. R., (2000) *Java Network Programming* O'Reilly & Associates, Inc Sebastol, CA

Horowitz, E., Sahni, S., (1974) Computing Partitions with Applications to the Knapsack Problem, *Journal of the ACM (JACM)*, 21, 2 pp 277-292

Kellerer, H., Pferschy, U., Pisinger, D. (2004) *Knapsack Problems* Springer-Verlag Heidelberg

Kumar, V., Grama, A., Gupta, A., Karypis., (1994) *Introduction to Parallel Computing: Design and Analysis of Algorithms* The Benjamin/Cummings Publishing Company, Inc. Redwood City CA.

Lee J., Shragowitz, E., Sahni, S. (1988) A Hypercube Algorithm for the 1/0 Knapsack Problem, *Journal of Parallel and Distributed Computing* 5, pp 438-456

Pisinger, D., (2003) Where are the hard knapsack problems, Technical Report, DIKU, University of Copenhagen, Denmark

Skiena, S.S. (1998) Who is interested in algorithms and why? – lessons from the Stony Brook algorithms repository. In *Second Workshop on Algorithms Engineering (WAE 1998)* pp 204-212 Saarbruecken, Germany

Appendix 1 Client Package

Java

KnapsackClient.java

```
package client;

/**
 * Reads in a text file with knapsack information
 * Contact KnapsackClientServer an request a result
 * Display result to user
 */

import server.KnapsackClientServer;
import server.ServerException;

import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.io.FileNotFoundException;
import java.io.IOException;

public class KnapsackClient {
    private String serverHost;           //server host
    private int capacity;                // knapsack problem arguments
    private LinkedList objects;         //knapsack object argument

    // constructor
    public KnapsackClient() {
        serverHost = null;
        capacity = 0;
        objects = null;
    }

    /*
     * main method
     * expects a knapsack definition file
     * and a ip address for the server
     */
}
```

```

public static void main(String[] args) {
    if (args.length != 2) {
        usage();
    }
    HashMap result;
    String knapsackFile = args[0];

    KnapsackClient ks = new KnapsackClient();
    ks.serverHost = args[1];
    try {

        // get the problem info from the file
        ks.getKnapsackInfo(knapsackFile);

        if ((ks.objects).size() <= 0) {
            throw new ClientException(
                "There are no valid objects in this
                Knapsack.");
        }
        // ask the server to solve it
        result = ks.askServer(ks.capacity, ks.objects);
        if (result == null) {
            throw new ClientException(
                "Result not recieved from server");
        }

        // let user know we are waiting for response
        ks.printResult(result);
    } catch (ClientException e) {
        System.out.println("Fatal Error: " + e.toString());
        System.exit(1);
    } catch (Exception e) {
        System.out.println("Unexpected error: "
            + e.toString());
    }
}

/**
 * Connects to server and gets the result
 * @param capacity
 * @param objects
 * @throws Exception
 */
public HashMap askServer(int capacity, LinkedList objects)
    throws ClientException {
    // establish connection
    // ask server for results
    HashMap result = null;

    // format the arguments correctly to send to server
    int i = 0; // index
    int[] w = new int[objects.size()]; // array of weights
    int[] v = new int[objects.size()]; // array of values

    Iterator it = objects.iterator();

```

```

while (it.hasNext()) {
    HashMap object = (HashMap) it.next();
    w[i] = ((Integer) object.get("weight")).intValue();
    v[i] = ((Integer) object.get("value")).intValue();
    i++;
}

// get KnapSack connections
KnapSackClientServer server = getServer();

// ask the server to solve problem
try {
    result = server.solveKnapsack(capacity,
objects.size(), w, v);
} catch (RemoteException e) {
    throw new ClientException(
        "Error in remote communication "
        + " requesting solution."
        + e.getMessage());
} catch (ServerException e) {
    throw new ClientException(
        "Unable to request solution from server: " +
        e.getMessage());
}

return result;
}

/**
 * Encapsulates how results will be displayed to user
 * currently stdout basic printout
 */
public void printResult(HashMap results) {
    // print the results
    System.out.println("\nBest Value is " +
results.get("result"));
    System.out.println("ResultSet");
    int[] rs = (int[]) results.get("resultSet");
    for (int i = 0; i < rs.length; i++) {
        if (rs[i] == 1) {
            HashMap o = (HashMap) objects.get(i);
            System.out.println(
                " ( " + o.get("weight") + ", " +
                o.get("value") + " ) ");
        }
    }
}

}

////////// private methods //////////

/*

```

```

    * in charge of establishing connection with the server
    */
private KnapsackClientServer getServer() throws ClientException {
    String serverName = KnapsackClientServer.LOOKUPNAME;
    String lookupName = "rmi://" + serverHost + "/" +
        serverName;
    KnapsackClientServer server = null;

    try {
        System.out.print("Connecting to " + serverName +
            "... ");
        server = (KnapsackClientServer)
            Naming.lookup(serverName);
        if (server == null) {
            throw new ClientException(
                "Unable to find server " + serverName);
        }
        System.out.println("Connected to server");
    } catch (NotBoundException e) {
        throw new ClientException("Server not registered ");
    } catch (java.io.IOException e) {
        throw new ClientException(
            "I/O error or bad host name with "
                + serverName
                + " "
                + e.getMessage());
    }
    return server;
}

/*
 * reads the file using FileUtility class
 * stores the information about the problem set
 * in capacity and objects
 */
private void getKnapsackInfo(String file) throws ClientException
{
    // read info from file
    FileUtility fu = new FileUtility(file);
    try {
        fu.parseFile();
        capacity = fu.getCapacity();
        objects = fu.getObjects();
    } catch (FileNotFoundException e) {
        throw new ClientException(
            "The file "
                + file
                + " does not exist or could not be
found:"
                + e.toString());
    } catch (IOException e) {
        throw new ClientException(
            "Problem reading input file" + ": " +
            e.toString());
    }
}

```

```
        }  
    }  
  
    /**  
     * usage - if no filename supplied  
     */  
    public static void usage() {  
        System.out.println(  
            "USAGE: KnapsackClient [knapsackdefinition.txt]  
            [server]");  
        System.exit(0);  
    }  
}
```

FileUtility.java

```
package client;

/**
 * This class encapsulates the process of reading input from the
 user/file
 * this implementation reads knapsack definition from a text file
 * assumed to be of the format
 * >capacity
 * >weight1,value1
 * >...
 * >weightn,valuen
 */

import java.util.LinkedList;
import java.util.HashMap;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.StringTokenizer;

public class FileUtility {
    String filename;          // name of input file
    int capacity;            // knapsack problem argument
    LinkedList objects;      // list of Hashmaps of weight and value
    int numObjects;         // numObjects found

    /**
     * Constructor for FileUtility.
     */
    public FileUtility(String file) {
        filename = file;
        capacity = 0;
        numObjects = 0;
        objects = new LinkedList();
    }

    /**
     * Method parseFile.
     * reads filename, line by line, calling appropriate functions
     * to parse each part
     * @throws FileNotFoundException
     * @throws IOException
     */
    public void parseFile() throws FileNotFoundException, IOException
    {
        FileReader fr = new FileReader(new File(filename));
        BufferedReader in = new BufferedReader(fr);
    }
}
```

```

        String line = in.readLine();
        if (line == null) {
            throw new IOException("Input file formatted
            incorrectly: First line must contain capacity");
        }
        parseCapacity(line);
        while ((line = in.readLine()) != null) {
            parseWeightAndValue(line);
        }
    }

    /**
     * Method getCapacity.
     * @return int
     */
    public int getCapacity() {
        return capacity;
    }

    /**
     * Method getValues.
     * @return LinkedList
     * @throws IOException
     */
    public LinkedList getObjects() throws IOException {
        return objects;
    }

    /**
     * Method getNumObjects.
     * @return int
     */
    public int getNumObjects() {
        return numObjects;
    }

    //////////////// private functions ////////////////////////

    /**
     * expects capacity to be only integer on line
     */
    private void parseCapacity(String line) {
        capacity = Integer.parseInt(line.trim());
    }

    /**
     * expects "weight, profit"
     */
    private void parseWeightAndValue(String line) throws IOException
{
        StringTokenizer st = new StringTokenizer(line, ",");
        if (st.countTokens() != 2) {

```

```
        throw new IOException(
"File line formatted incorrectly: Need Weight and Value");
    }
    Integer w = new Integer(st.nextToken());
    Integer v = new Integer(st.nextToken());
    // automatically parse out objects whose weight is greater
    //than the capacity
    if (w.intValue() > capacity) {
        return;
    }
    HashMap obj = new HashMap();
    obj.put("weight", w);
    obj.put("value", v);
    objects.add(obj);
    numObjects++;
}
}
```

SortByWeightComparator.java

```
package client;

import java.util.HashMap;

/**
 *
 * provide a comparator for Arrays.sort,
 * allows data structure of objects to be sorted by weight
 * used in initial testing to determine if it made a difference
 * it did not.
 */

public class SortByWeightDescComparator implements java.util.Comparator
{
    public int compare(Object o1, Object o2) {
        Integer i1 = (Integer)((HashMap)o1).get("weight");
        Integer i2 = (Integer)((HashMap)o2).get("weight");

        return i2.compareTo(i1);
    }
}
```

ClientException.java

```
package client;

/**
 * definition of Client Exception
 */

public class ClientException extends Exception {
    public ClientException() {
        super();
    }
    public ClientException(String msg) {
        super(msg);
    }
}
```

Appendix 2 Server Package

Java

KnapsackClientServer.java

```
package server;
/**
 * This interface defines the interface the Server will have with the
 Client
 * It is implemented by KnapsackClientServerImpl.java
 */

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.HashMap;
import java.net.MalformedURLException;

public interface KnapsackClientServer extends Remote {

    // define static lookupnames for consistency
    public final static String LOOKUPNAME = "KnapsackClientServer";

    /**
     * Method solveKnapsack.
     * sets information about knapsack
     * @param capacity
     * @param numObjects
     * @param weights
     * @param values
     * @return HashMap
     * @throws RemoteException
     * @throws ServerException
     */
    public HashMap solveKnapsack(
        int capacity,
        int numObjects,
        int[] weights,
        int[] values)
        throws RemoteException, ServerException;
```

```
/**
 * specify the Server type requested
 * will attempt to instantiate the server
 * @param solverType
 * @throws RemoteException
 * @throws MalformedURLException
 */
public void setType(String solverType)
    throws RemoteException, MalformedURLException;
}
```

KnapsackClientServerImpl.java

```
/*
 * KnapsackClientServerImpl.java
 */
package server;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.HashMap;

/**
 * interface between the server and the client
 * takes their request for a Solver type
 * instantiate that type of solver, and start the solve when requested
 */
public class KnapsackClientServerImpl
    extends java.rmi.server.UnicastRemoteObject
    implements KnapsackClientServer {

    private KnapsackSolverServer server; // server based on client
                                         //request

    /**
     * constructor
     */
    public KnapsackClientServerImpl() throws RemoteException {
        server = null;
    }

    /** (non-Javadoc)
     * @see server.KnapsackClientServer
     */
    public HashMap solveKnapsack (
        int capacity,
        int numObjects,
        int[] weights,
        int[] values)
        throws RemoteException, ServerException {
        HashMap result = server.solveKnapsack(capacity,
            numObjects,
            weights,
            values);
        return result;
    }

    /** (non-Javadoc)
```

```

    * @see server.KnapsackClientServer#setType(java.lang.String)
    */
public void setType(String solverType) throws RemoteException,
    MalformedURLException {
    if (solverType.equals("LEE") ) {
        server =
        SolverServerFactory.newInstance().getSolverServer("LEE");
    }
    else if (solverType.equals("CHEN") ) {
        server =
        SolverServerFactory.newInstance().getSolverServer("CHEN");
    }
    else if (solverType.equals("HUNTPL") ) {
        server =
        SolverServerFactory.newInstance().getSolverServer("HUNTPL")
        ;
    }
    else {
        throw new RemoteException("requested solver type is not
        valid " + solverType);
    }
    Naming.rebind(KnapsackSolverServer.SOLVERLOOKUPNAME,
    server);
}

/**
 * starts and registers the servers
 */
public static void main(String[] args) {
    if (args.length != 2 ) {
        usage();
    }
    String algorithm = args[0];
    String serverhost = args[1];
    System.setSecurityManager(new RMISecurityManager());

    try {

        KnapsackClientServerImpl server = new
        KnapsackClientServerImpl();

        Registry r = LocateRegistry.getRegistry( serverhost );
        Naming.rebind(KnapsackClientServer.LOOKUPNAME, server);

        System.out.println(KnapsackClientServer.LOOKUPNAME
        +" ready and waiting...");
        server.setType(algorithm);
    }
    catch (java.io.IOException e) {
        System.out.println("Could not register Server "
        + e.getMessage());
        e.printStackTrace();
        System.exit(1);
    }
    catch (Exception e) {

```

```
        System.out.println("Unexpected error starting ClientServer
" + e.getMessage());
        System.exit(1);
    }
}

/*
 * instruct user how to start this class
 */
private static void usage() {
    System.out.println("USAGE: KnapsackClientServer [algorithm]
[servername]");
    System.out.println("\t side effect: will start the solver server
for the given algorithm.");
    System.exit(0);
}
}
```

SolverServerFactory.java

```
package server;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;

/**
 * PURPOSE: Factory which allows different readers and writers to be
 * created
 * based on the string passed to getClientServer
 * class follows Singleton Pattern
 */

final class SolverServerFactory {
    // one private static instance of this class allowed
    static private SolverServerFactory factory
        = new SolverServerFactory();

    //private constructor - placeholder
    private SolverServerFactory() {
    }

    /**
     * @return the single Factory instance
     */
    public static SolverServerFactory newInstance() {
        // singleton pattern, so we only have one factory instance
        return factory;
    }

    /**
     * @param type
     * @return an instance of a SolverServer implementation
     * based on the String type parameter
     */
    public KnapsackSolverServer getSolverServer(String type)
        throws RemoteException, MalformedURLException {
        KnapsackSolverServer server = null;
        if (type.equals("HUNTPL") ) {
            server = new KnapsackServerHuntPLImpl();
            Naming.rebind(
                KnapsackSolverServer.SOLVERLOOKUPNAME, server);
        }
        if (type.equals("LEE")){
            server = new KnapsackServerLeeImpl();
            Naming.rebind(
                KnapsackSolverServer.SOLVERLOOKUPNAME, server);
        }
        if (type.equals("CHEN")){
            server = new KnapsackServerChenImpl();
            Naming.rebind(
```

```
        KnapsackSolverServer.SOLVERLOOKUPNAME, server);  
    }  
    return (KnapsackSolverServer) server;  
}  
  
}
```

KnapsackSolverServer.java

```
package server;

import java.rmi.*;
import java.util.HashMap;

/**
 * defines the interface the Server provides to Solver components
 * also provides a starting point (solveKnapsack) so
 * that a method from a local class can
 * trigger the solve process
 */
public interface KnapsackSolverServer extends Remote {

    // define static lookupnames for consistency
    public final static String SOLVERLOOKUPNAME =
        "KnapsackSolverServer";

    /**
     * @param solverNam
     * @throws ServerException
     * @throws RemoteException
     * Solver may register itself as being available through this
     * method
     */
    public void registerSolver(String solverNam) throws
        ServerException, RemoteException;

    /**
     * request from solver to have another solver help them with
     * the branch with these proprties
     * will return the best value for those values
     * @param depth
     * @param cap
     * @param value
     */
    public int requestHelp(int depth, int cap, int value) throws
        RemoteException, ServerException;

    /**
     * Method solveKnapsack.
     * sets information about knapsack
     * @param capacity
     * @param numObjects
     * @param weights
     * @param values
     * @return HashMap
     * @throws RemoteException
     */
}
```

```

    * @throws ServerException
    */
    public HashMap solveKnapsack(int capacity, int numObjects, int[]
weights, int[] values)
throws RemoteException, ServerException;

/**
 * @param name
 * @param pindex
 * @throws RemoteException
 * allows solver to notify the
 * Server when it has completed a stage
 */
public void notifyComplete(String name, int pindex) throws
RemoteException;

/**
 * @param solverindex
 * @param array of vectors to be broadcast back to all
 * solvers in group
 * @throws RemoteException
 * allows a solver to send server a vector
 * which it would like the server to broadcast
 */
public void vectorForBroadcast(int solverindex, int [][] vectors
)
throws RemoteException;

/**
 * @param result
 */
public void notifyComplete(HashMap result) throws
RemoteException;
}

```

KnapsackServerLeeImpl.java

```
package server;

import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.HashMap;
import java.util.LinkedList;

import solver.KnapsackSolverLee;

/**
 *
 * Implements the LEE algorithm for solving the 0-1 Knapsack algorithm
 * Takes on role of server
 *
 * The algorithm can be briefly explained as follows.
 * Assume p processors are available.
 * 1. (partition) original problem was divided into p subproblems, as
 * evenly as possible.
 * Each processor is assigned a subproblem.
 * 2. (forward part of dynamic programming) Each processor solves
 * subproblem, creating a profit vector.
 * 3. (forward part of combining procedure) The processors combine
 their
 * resulting profit vectors by distributing the vectors to get a
 * resulting profit vector for the
 * whole problem
 * 4. (backtracking part of combining procedure)
 * The p processors trace back the
 * combining history to get a value which is the optimal solution to
 the
 * original problem.
 * 5. (backtracking part of dynamic programming)
 * Each processor traces back its
 * dynamic programming history to get an optimal solution vector
 *
 *
 */
public class KnapsackServerLeeImpl extends
    java.rmi.server.UnicastRemoteObject implements
    KnapsackSolverServer {

    private int capacity; // knapsack capacity
    private int [] weights; // weights[i] = weight of ith object
    private int [] values; // values of objects private
    int numObjects; // total number of objects to be used

    /* list of solvers registered as available */
}
```

```

private LinkedList solvers;

/* list of names of solvers */
private LinkedList solverNames;

/* contains the indices of weights/values for used for
 * each subproblem
 * used for piecing the problem back together at the end */
private HashMap subproblemIndices;

/* number of processors = 2^processorsN for some n>= 0
 * dimension of the hypercube, also the maximum level */
private int dimension;

/* total number of processors, defined when
 * solveKnapsack called */
private int totalProcessors;

/* the most processors we want in a group, based on capacity */
private int maxGroupSize;

/* used to keep track of which processors have finished the
 * current step */
private HashMap finished;

/* vectors to be broadcast to solvers */
private HashMap broadcastVectors;

/* level of combining procedure */
private int level;

/* number of processors per group at
 * this level */
private int groupSize;

/* number of groups at this level */
private int numGroups;

/* final result, solved solution */
private HashMap result;

/* final sum profit vector */
private int[] profitVector;

/**
 * constructor required to throw RemoteException
 */
public KnapsackServerLeeImpl() throws RemoteException {
    numObjects = 0;
    capacity = 0;
    solvers = new LinkedList();
    solverNames = new LinkedList();
    dimension = 0;
    totalProcessors = 0;
    maxGroupSize = 0;
}

```

```

        level = 1; // level counting is one-based
        finished = new HashMap();
        broadcastVectors = new HashMap();
        subproblemIndices = new HashMap();
        result = null;
    }

    /**
     *given a solver name, look it up
     *establish that we can use it as a server
     *and add it to our stack
     */
    public void registerSolver(String solverName)
        throws ServerException, RemoteException {

        KnapsackSolverLee solver = null;
        try {
            System.out.print("Connecting to " + solverName
                + "... ");
            solver =
                (KnapsackSolverLee)Naming.lookup(solverName);
            System.out.println("Connected to solver");
        }
        catch (NotBoundException e) {
            throw new ServerException("Server not
                registered. " + e.getMessage());
        }
        catch (java.io.IOException e) {
            throw new ServerException("I/O error or bad
                host name with " + solverName + " " +
                e.getMessage());
        }
        addSolver(solverName, solver);
    }

    /* (non-Javadoc)
     * @see server.KnapsackSolverServer#requestHelp(int, int, int)
     */
    public int requestHelp(int depth, int cap, int value)
        throws RemoteException, ServerException {
        return 0;
    }

    /**
     * fork out the requests to begin solving the knapsack
     * so they can run in parallel
     * @param knapsacksolver to which request should be made
     * @param hashmap definition of portion of knapsack to be solved
     * @param index
     * @return thread
     */
    private Thread forkSolveRequest(
        final KnapsackSolverLee solver,

```

```

final HashMap knapsack,
final int index) {
Thread t = null; // the thread to ask the server
t = new Thread("PartitionThread") {
    public void run() {
        try {
            solver.setValue("pindex", index);
            solver.solve(capacity,
                ((Integer)knapsack.get("numObjects")).
                intValue(),
                (int[])knapsack.get("weights"),
                (int[])knapsack.get("values"));
        } // otherwise will notify user and continue
        catch (RemoteException e) {
            System.out.println(
                "Error with requesting solver start "
                + e.getMessage());
        }
    }
};
t.start();
return t;
}

/* (non-Javadoc)
 * @see server.KnapsackSolverServer#solveKnapsack
 * assigns the subproblems to each solver and
 * initiates calculation
 */
public HashMap solveKnapsack(int capacity,
    int numObjects,
    int[] weights,
    int[] values)
    throws RemoteException, ServerException {
    clearValues();
    this.capacity = capacity;
    this.numObjects = numObjects;
    this.weights = weights;
    this.values = values;
    this.totalProcessors = solvers.size();
    this.dimension = check2ToNProcessors();
    this.maxGroupSize = computeMaxGroupSize();
    this.result = null;

    // let each solver know the names of other solvers

    broadcastSolverNames();

    // partition the problem, store results in subProblem
    HashMap subproblems[] = partition();

    // call solve for solvers on their assigned subProblems

    assignAndSolve(subproblems);
}

```

```

        while (this.result == null) {
            /* wait for result*/
        }

        return this.result;
    }

/**
 * @param stepName
 * @param processor
 * used to notify server that a processor or group
 * has finished a specific task
 */
synchronized public void notifyComplete(String stepName,
    int pindex)
    throws RemoteException
{
    boolean allFinished = false;
    if (stepName.equals("forwardDynamic")) {
        finished.put(new Integer(pindex), stepName) ;
        // add check that all have finished
        if (finished.size() == totalProcessors ){

            finished.clear();

            try {
                forwardCombine(level);
            } catch (Exception e) {
                System.err.println(
                    "forwardCombine failed");
            }

        }
    }
    // all groups have finished a level of the combineForward
    else if (stepName.equals("combineLevel")) {
        finished.put(new Integer(pindex), stepName) ;
        if (level >= dimension) {
            // if all levels complete:
            try {
                backtrackCombine(level);
            } catch (Exception e) {
                System.err.println(
                    "error in backtrack combine");
            }

        } else {
            finished.clear();
            level++; // move to next level
            forwardCombine(level);
        }
    }
    else if (stepName.equals("backtrackDynamic")) {
        if (finished.size() == totalProcessors ){
            HashMap res = new HashMap();

```

```

        int [] allResults = new int[numObjects];
        for (int i =0;i<totalProcessors;i++) {
            int []partialResults = (int[])
            finished.get(new Integer(i));
            HashMap indices =
            (HashMap) subproblemIndices.get(new
Integer(i));
            int destinationPosition =
            ((Integer)indices.get("start")).intValue();

            System.arraycopy(partialResults,
            0,
            allResults,
            destinationPosition,
            partialResults.length-1);
        }
        res.put("resultSet", allResults);
        int profit = profitVector[capacity];
        res.put("result", new Integer( profit) );
        this.result = res;
        return;
    }
}

return;
}

/*
 * accept a vector from the solver which
 * should be broadcast to all other solvers
 */
public void vectorForBroadcast(int solverindex, int [][] vectors)
throws RemoteException {
    broadcastVectors.put(new Integer(solverindex), vectors);

    if (broadcastVectors.size() == totalProcessors ) {
        // broadcast to all groups
        //broadcastAllVectors();
        if (level == dimension) {
            profitVector = vectors[0];
        };
        broadcastVectors.clear();
        notifyComplete("combineLevel", totalProcessors);
    }
}

/* //////////////////////////////////////////////////// private methods //////////////////////////////////// */

/*
 * send the vectors to the correct groups
 */
private void broadcastAllVectors() {

```

```

int pindex = 0;
for (int grp=0;grp<numGroups;grp++) {
    // create the packets of vectors which will
    // be sent to each group
    int start = groupSize * grp ;
    int end = start + groupSize -1;
    for (int p=0;p<groupSize;p++) {

        // send all the profitvectors
        KnapsackSolverLee solver = (KnapsackSolverLee)
        solvers.get(pindex);
        for (int i =start;i<=end;i++){
            if (i != pindex ){
                try {
                    int [][] vectors = (int[][])
                    broadcastVectors.get(new
                    Integer(i));
                    solver.broadcastRecieve(i,
                    vectors);

                    } catch (RemoteException e) {
                        System.err.println(
                            "Failed to broadcast to
                            " + i);
                    }
                }
            }
            pindex++;
        }
    }
    broadcastVectors.clear();
}

```

```

/* Partition Knap(G,c) into p subproblems KNAP(Gi, c),
* i=0,1...p such that
* G is the union of all subproblems Gi
* Gi \cup Gj is disjoint for i!=j
* and |Gi| = |G|/p for all i
* HashMap [] subproblems;
* data structure for how problem is subdivided
* index is the processor
* value is a Hashmap with
* key "numObjects" = numObjects for subproblem
* key "weights"    = array of object weights for subproblem
* key "values"     = array of object values for subproblem
*/
protected HashMap [] partition() {
    int i =0; // counter
    HashMap [] subproblems = new HashMap [totalProcessors];

    // numobjects in each subproblem = numObjects / p
    double x = (double)numObjects/(double)totalProcessors;

```

```

int Gi = (int) Math.round(x);

// assign the problem as equally as possible
// create array specifying where each subproblem should end
// always begin at previous plus one
int endIndex = 0;
int startIndex = 0;
for (i=0;i<totalProcessors;i++){
    if (startIndex + Gi < numObjects) {
        endIndex = startIndex + Gi;
    } else {
        endIndex = numObjects;
    }
    // but if we are on the last processor,
    // give it everything left
    if (i+1==totalProcessors){
        endIndex=numObjects;
    }
    HashMap indices = new HashMap();
    indices.put("start", new Integer(startIndex));
    indices.put("end", new Integer(endIndex));
    subproblemIndices.put(new Integer(i), indices);
    subproblems[i]
    = createSubproblem(startIndex, endIndex);
    startIndex = endIndex;
}
return subproblems;
}

/*
 * creates a Hashmap fully describing the subproblem
 * key "numObjects" = numObjects for subproblem
 * key "weights" = array of object weights for subproblem
 * key "values" = array of object values for subproblem
 */
private HashMap createSubproblem(int startIndex, int endIndex) {
    int i=0; //counter for point within main problem
    int j=0; // counter for point within subproblem
    int size = endIndex-startIndex; // size of subproblem
    int [] subweights = new int[size];
        // subproblem weight array

    int [] subvalues = new int[size];
        // subproblem value array
    HashMap subproblem = new HashMap();
        // hashmap describing subproblem

    // fill the new arrays
    for (i = startIndex; i<endIndex; i++) {
        subweights[j] = weights[i];
        subvalues[j] = values[i];
        j++;
    }
    subproblem.put("numObjects", new Integer(size));
}

```

```

        subproblem.put("weights", subweights );
        subproblem.put("values", subvalues );
        return subproblem;
    }

    /*
     * assign KNAP(Gi, c) to PRi, for i=0,1...p-1
     * PR is the ith processor
     */
    private void assignAndSolve(HashMap subproblems []) throws
    RemoteException, ServerException {
        int p = totalProcessors;
        int i = 0;
        for (i=0;i<p; i++) {
            KnapsackSolverLee solver =
                (KnapsackSolverLee)solvers.get(i);
            HashMap knapi = (HashMap)subproblems[i];

            if (knapi == null ) {
                throw new ServerException(
                    " error partitioning: no partition for " + i);
            }
            forkSolveRequest(solver, knapi, i);
        }
    }

    /*
     * check that we have 2^n processors
     * return n
     */
    private int check2ToNProcessors() throws ServerException {
        int p = totalProcessors;
        if (p == 0) {
            throw new ServerException("0 solvers are registered."
                + "Can not solve problem.");
        }
        double nDouble = ( Math.log((double)p)/Math.log(2) );
        double rounded = Math rint(nDouble);
        if (rounded != nDouble) {
            throw new ServerException("the number of solvers " +
                p + " is not 2^n for some n. "
                + " Closest is " + nDouble + " not " + rounded );
        }

        int n = (int)rounded;
        return n;
    }

    /*
     * backtracking part of combining process
     * after combining p profit vectors, it is necessary to

```

```

* trace back the combining history to get a set of components
* x_0, x_1, ...x_{p-1} of a constraint c assigned to each processor
* s.t
*   the sum of all x_i for (i=0...p-1) = c
* and the sum of all a[i][x_i] for (i=0...p-1) = a[(0,n)][c]
* a[(0,n)][c] is the optimal solution for the original problem
*
* x_0 = h_c[0,n]
* x_1 = c-x_0
* foreach h(k) where k is a level
*   t = x_i
*   x_{2i} = h_t(k)
*   x_{2i+1} = t - x_{2i}
* end
*/
private void backtrackCombine(int level) {
    int x[] = new int[totalProcessors];
    int maxLevel = level;
    int t = capacity;
    while (level>=1) {
        computeGroupSize();
        for (int i=0;i<totalProcessors;i++) {
            if (level != maxLevel) {
                //t = x[level];
                t = x[i];
            }
            int pindex = i;
            KnapsackSolverLee solver
            = (KnapsackSolverLee) solvers.get(i);
            int h_t_ik = 0;
            try {
                h_t_ik
                = solver.backtrackCombine(level, t);
            } catch (RemoteException e) {
                e.printStackTrace();
                System.exit(1);
            }

            x[i] = h_t_ik;
        }
        level--;
    }
    backtrackDynamic(x);
}

/*
* each processor i traces back the history of
* the dynamic programming procedure using backtrackDynamic
* with (P,W) = (a^i_xi, xi)
*/
private void backtrackDynamic(int [] x) {

    finished.clear();

```

```

        for (int i=0;i<totalProcessors;i++) {
            KnapsackSolverLee solver
            = (KnapsackSolverLee) solvers.get(i);
            forkBacktrackDynamic(solver, x[i], i);
        }
    }

    /*
    * @param solver
    * @param x_i
    * @return threaf
    */
    private Thread forkBacktrackDynamic(
        final KnapsackSolverLee solver,
        final int x_i,
        final int pindex) {

        Thread t = null; // the thread to ask the server
        t = new Thread("backtrackThread") {
            int [] z = null;
            public void run() {
                try {

                    z = solver.backtrackDynamic(x_i);
                    finished.put(new Integer(pindex), z);
                    notifyComplete("backtrackDynamic", pindex);
                } catch (RemoteException e) {
                    System.err.println("Error running backtrack "
                        +" dynamic with " + numObjects);
                    e.printStackTrace();
                }
            }
        };
        t.start();
        return t;
    }

    /* forward part of the combining procedure
    * each level called when all groups report completion of a level
    *
    * k = current level in combining tree
    * l = a level from which the group size is not increasing
    * a[i][k] is the profit vector
    * h[i][k] is the history vector
    *
    * l= log2c -1
    * a[i][0] = a[i] for i=0...p-1
    * for k = 1 to n do
    *     g = min(k,l); g is maxGrouops
    *     r= p/(2^g) r is groupSize
    */

```

```

* partition p processors into r groups of size 2^g
* for each group i(0<-i<=r-1) in parallel do
*   All processors in the group i compute
*   a[i][k] = combine(a[2i][k-1], a[2i+1,k-1]);
*   h[i][k] = history(a[2i][k-1], a[2i+1,k-1]);
* end for each
* end for
*/
private void forwardCombine(int level) {
    int groups[][] = computeGroupSize();

    // partition the set of p processor into r groups of size
2g
    int solverCounter = 0;
    int vectorStart = 0;
    int vectorFinish = 0;
    for (int groupCounter=0;
        groupCounter<numGroups;groupCounter++ ) {
        for (int i=0;i<groupSize;i++){

            // groups are parceled differently in top level
            int groupIndex = groupCounter;
            if (level == dimension && numGroups > 1) {
                groupIndex = solverCounter % 2;
            }
            KnapsackSolverLee s =
                (KnapsackSolverLee)solvers.get(solverCounter);

            try {
                s.setValue("level", level );
                startCombine(s, groups[groupIndex]);
            } catch (RemoteException e) {
                System.err.println("remote exception
                thrown setting values or creating thread
                " + e.getMessage() );
            }

            solverCounter++;
        }
    }

    return;
}

/*
* starting the combine process on each solver is it's own thread
*/
private Thread startCombine(final KnapsackSolverLee solver, final int[]
group ) {
    Thread t = null; // the thread to ask the server

    t = new Thread("HelpThread") {
        public void run() {

```

```

        try {
            solver.combine(group);
        } catch (RemoteException e) {
            System.err.println(
                "Error calling combine on processor");
            System.exit(1);
        }
    }
};
t.start();
return t;
}

```

```

/*
 * sets the instances vars numGroups and groupSize
 * should be called before forwardCombine
 */
private int[][] computeGroupSize() {
    double g = Math.min((double)level,maxGroupSize);
    double groupsize = Math.pow((double)2,g);
    // how many processors in each group

    if ((int)groupsize > maxGroupSize) {
        groupsize = maxGroupSize;
    }
    double r = Math.floor(
        (double)totalProcessors/(double)groupsize );
    // how many groups of processors

    this.groupSize = (int) groupsize;
    this.numGroups = (int) r;

    int p=0;
    int [][] groups = new int[numGroups][groupSize];
    if (level == dimension && numGroups > 1) {
        int []groupEven = new int[groupSize];
        int []groupOdd = new int[groupSize];
        int e=0;
        int o=0;
        for (int i=0;i<totalProcessors;i++) {
            if (i%2==0){
                groupEven[e] = i;
                e++;
            }else {
                groupOdd[o] = i;
                o++;
            }
        }
        groups[0] = groupEven;
        groups[1] = groupOdd;
    }
    else {
        for (int i=0;i<numGroups;i++) {

```

```

        int[] groupProcessors = new int[groupSize];
        for (int j=0;j<groupSize;j++) {
            groupProcessors[j] = p;
            p++;
        }
        groups[i] = groupProcessors;
    }
}

return groups;
}

/*
 * let each solver know the names of the other solvers
 */
private void broadcastSolverNames() throws RemoteException {
    int i = 0;
    for (i=0;i<totalProcessors; i++) {
        KnapsackSolverLee solver =
            (KnapsackSolverLee) solvers.get(i);
        solver.setNames(solverNames);
    }
}

/*
 * @return max number processors in group
 * ( Math.log(capacity)/Math.log(2) ) -1;
 */
private int computeMaxGroupSize() {
    double size = totalProcessors/2;
    size = Math.max(2, size); //must be more than one per
group
    return (int)size;
}

/*
 * puts a solver on the stack of valid solvers
 * Method registerSolver.
 */
protected void addSolver(String solverName,
KnapsackSolverLee solver) {
    solvers.add(solver);
    solverNames.add(solverName);
}

/* reset all values for problem
 * so we can start a new problem with the same
 * server and solver setup
 */

```

```

private void clearValues() {
    numObjects = 0;
    numGroups = 0;
    capacity = 0;
    maxGroupSize = 0;
    level = 1; // level counting is one-based
    finished.clear();
    broadcastVectors.clear();
    subproblemIndices.clear();
    values = null;
    weights = null;
    profitVector = null;
}

////////////////////////////////////
// empty methods required to implement interface

/* (non-Javadoc)
 *
 */
public void notifyComplete(HashMap result)
    throws RemoteException
{}

}

```

KnapsackServerChenImpl.java

```
/*
 * KnapsackServerChenImpl.java
 */
package server;

import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;

import solver.KnapsackSolverChen;

/**
 * Implements the CHEN algorithm for solving the 0-1 Knapsack
 algorithm
 * Takes on role of server
 * 1. accept request from client and set variables
 * 2. partition and allocate data
 * 3. communicate to solvers who they will be communicating with
 * 4. initiate calculation
 * 5. recieve solution from solvers and send back to client
 */

public class KnapsackServerChenImpl
    extends java.rmi.server.UnicastRemoteObject
    implements KnapsackSolverServer
{

    private int [] weights;        // weights[i] = weight of ith object
    private int [] profits;        // values of objects
    private int numObjects;        // total number of objects to be used
    private int capacity;          // knapsack capacity      private
    int numLevels;                // number of Levels to be used in
                                // calculations
    private HashMap result;        // result for solved knapsack
    private LinkedList solvers;    // list of solvers registered
    private int totalProcessors;   // total number of processors

    /*
     * constructor, nulls instance variables
     */
    public KnapsackServerChenImpl() throws RemoteException {
        numObjects = 0;
        capacity = 0;
        totalProcessors = 0;
        weights = null;
        profits = null;
        solvers = null;
    }
}
```

```

        result = null;
    }

    /**
     *given a solver name, look it up
     *establish that we can use it as a server
     *and add it to our stack
     */
    public void registerSolver(String solverName)
        throws ServerException, RemoteException
    {

        KnapsackSolverChen solver = null;
        try {
            System.out.print(
                "Connecting to " + solverName + "... ");
            solver =
                (KnapsackSolverChen) Naming.lookup(solverName);
            System.out.println("Connected to solver");
        }
        catch (NotBoundException e) {
            throw new ServerException(
                "Server not registered. " +
                e.getMessage());
        }
        catch (java.io.IOException e) {
            throw new ServerException(
                "I/O error or bad host name with "
                + solverName + " " + e.getMessage());
        }
        addSolver(solverName, solver);
    }

    public HashMap solveKnapsack(int capacity,
        int numObjects,
        int[] weights,
        int[] values)
        throws RemoteException, ServerException
    {
        if (solvers==null || solvers.size() == 0) {
            throw new ServerException(
                "Can not solve knapsack: "
                +"no solvers have registered.");
        }

        clearValues();
        this.capacity = capacity;
        this.numObjects = numObjects;
        this.weights = weights;
        this.profits = values;
        this.totalProcessors = solvers.size();
        this.numLevels = 2;
    }

```

```

        // order processors, and tell each one about the neighbor
        // call solve for each processor to tell them about
knapsack
    try {
        sendSolverInformation();
        startProcessing();
    } catch (ServerException e) {
        String message = "Failed to solve the knapsack "
            + e.getMessage();
        System.err.println(message);
        throw new ServerException(message );
    }

    while (this.result == null) {
        /* wait for result*/
    }

    return this.result;
}

/* (non-Javadoc)
 *
 */
public void notifyComplete(HashMap result)
    throws RemoteException
{
    this.result = result;
}

/**
 * @param solverName
 * @param solver
 * puts a solver on the stack of valid solvers
 */
protected void addSolver(String solverName,
    KnapsackSolverChen solver) {
    if ( solvers == null ) {
        solvers = new LinkedList();
    }
    solvers.add(solver);
}

/***** private functions *****/

/*
 * for each solver,
 * send solver information about its neighbor
 * and the knapsack to be solved
 */
private void sendSolverInformation() throws ServerException {

```

```

// get the name of the last solver for the first solver
String name = null;
KnapsackSolverChen solver =
(KnapsackSolverChen) solvers.getLast();

try {
    name = solver.getSolverName();
} catch (RemoteException e1) {
    throw new ServerException(
        "Failed to communicate with last solver");
}

// get the range information
HashMap [] levelRanges = calculateRanges(numLevels);

// iterate through the solvers and set the name and the
Iterator solverit = solvers.iterator();
int objectOffset = totalProcessors;

while ( solverit.hasNext() ) {
    solver = (KnapsackSolverChen) solverit.next();

    try {
        solver.receiveNeighborName(name);
        solver.recieveKnapsackDefinition(profits,
            weights, capacity, numLevels,
            objectOffset, levelRanges);
        name = solver.getSolverName();
    } catch (RemoteException e) {
        throw new ServerException(
            "Failed to communicate with solver: "
            + e.getMessage());
    }
}
}

/*
 * return an array Hashmaps with keys startRange and endRange
 * array is indexed by the level in which to compute given ranges
 * want to calculate this only once
 */
private HashMap[] calculateRanges(int totalLevels) {
    HashMap [] levelRanges = new HashMap[totalLevels];
    for (int level=0;level<totalLevels;level++ ) {
        int range = (int) Math.ceil(
            (double)capacity/(double)totalLevels );
        int startRange = range * ( level ) ;
        int endRange    = range * ( level +1 ) -1 ;
        if ( level == totalLevels -1) {
            endRange = capacity;
        }
        HashMap ranges = new HashMap();
        ranges.put("startRange", new Integer(startRange));
        ranges.put("endRange", new Integer(endRange));
    }
}

```

```

        levelRanges[level] = ranges;
    }
    return levelRanges;
}

/*
 * create an intial vector of blank profits,
 * and a blank bactrack array
 * and kick off the process by sending them to
 * the first processor
 */
private void startProcessing() throws ServerException {
    int [] profits = new int[capacity+1];
    int [][] backtrack = new int[numObjects][capacity + 1];

    // initialize the backtrack vector with -1
    for (int j=0;j<numObjects;j++) {
        for (int i=0;i<=capacity;i++) {
            backtrack[j][i]=-1;
        }
    }
    int objectIndex = -1;

    LinkedList capacityPoints = new LinkedList();
    capacityPoints.add( new Integer(0 ) );
    capacityPoints.add( new Integer(capacity));

    // start each level of processing on the first processor
    threadReceiveMessage(profits, backtrack, objectIndex,
        numLevels -1, capacityPoints);
}

/*
 * send some blank vectors to first processor to kick off the
 * process
 */
private void threadReceiveMessage( final int[] profits,
    final int[][] backtrack,
    final int objectIndex,
    final int level,
    final LinkedList capacityPoints) {
    // get the first processor
    final KnapsackSolverChen solver =
        (KnapsackSolverChen) solvers.getFirst();

    // thread the message sending
    Thread t = null;
    t = new Thread("beginCalculationThread") {
        public void run()    {
            try {

```

```

        solver.receiveMessage(profits,
                               backtrack, objectIndex,
                               level,
                               capacityPoints );
    } catch (RemoteException e) {
        System.err.println("Server can not start:
Failed to send profits to processor at "
+level);
        System.exit(0);
    }
    }
};
t.start();
}

/*
 * clear out the cache so the next knapsack can be requested
 */
private void clearValues() {
    numObjects = 0;
    capacity = 0;
    weights = null;
    profits = null;
    result = null;
}

////////////////////////////////////
// functions required to fulfill interface requirements for
// KnapsackSolverServer

public int requestHelp(int depth, int cap, int value) throws
RemoteException, ServerException {
    return 0;
}

/* (non-Javadoc)
 *
 */
public void notifyComplete(String name, int pindex) throws
RemoteException {
}

/* (non-Javadoc)
 *
 */
public void vectorForBroadcast(int solverindex, int[][] vectors) throws
RemoteException {
}

}

```

KnapsackServerHuntPL.java

```
package server;

import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.HashMap;
import java.util.LinkedList;

import solver.HuntPLVectorObject;
import solver.KnapsackSolverHuntPL;

/**
 * Implements the HUNTPL algorithm for solving the 0-1 Knapsack
 * algorithm
 * Takes on role of server
 * PURPOSE: register, and organize solvers
 *      1. accept request from client and set variables
 *      2. assign a range of objects and capacities for each solver
 *      3. communicate to solvers who their right neighbor is
 *      4. request solution from solvers
 */

public class KnapsackServerHuntPLImpl
    extends java.rmi.server.UnicastRemoteObject
    implements KnapsackSolverServer
{
    private int [] weights;          // weights[i] = weight of ith
object
    private int [] profits;         // values of objects
    private int numObjects;         // total number of objects
    private int capacity;          // knapsack capacity
    private int numLevels;         // number of Levels to be
// used in calculations
    private HashMap result;        // result for solved knapsack
    private int numObjectsToCalc; // the number of objects to
// calculate before passing data
// to next solver

    private LinkedList solvers;    // list of solvers registered as
// being available
    private int totalProcessors;   // total number of processors
//available

    /**
     * constructor, nulls instance variables
     */
    public KnapsackServerHuntPLImpl() throws RemoteException {
        numObjects = 0;
    }
}
```

```

        capacity = 0;
        totalProcessors = 0;
        weights = null;
        profits = null;
        solvers = null;
        result = null;
    }

    /**
     *given a solver name, look it up
     *establish that we can use it as a server
     *and add it to our stack
     */
    public void registerSolver(String solverName)
        throws ServerException, RemoteException
    {

        KnapsackSolverHuntPL solver = null;
        try {
            System.out.print(
                "Connecting to " + solverName + "... ");
            solver =
                (KnapsackSolverHuntPL)Naming.lookup(solverName)
                ;
            System.out.println("Connected to solver");
        }
        catch (NotBoundException e) {
            throw new ServerException(
                "Server not registered. " + e.getMessage());
        }
        catch (java.io.IOException e) {
            throw new ServerException(
                "I/O error or bad host name with "
                + solverName + " " + e.getMessage());
        }
        addSolver(solverName, solver);
    }

    /* (non-Javadoc)
     * @see server.KnapsackSolverServer#solveKnapsack()
     */
    public HashMap solveKnapsack(int capacity, int numObjects,
        int[] weights, int[] values)
        throws RemoteException, ServerException
    {
        if (solvers==null || solvers.size() == 0) {
            throw new ServerException(
                "Can not solve knapsack: no solvers have
                registered.");
        }

        clearValues();
        this.capacity = capacity;
    }

```

```

        this.numObjects = numObjects;
        this.weights = weights;
        this.profits = values;
        this.totalProcessors = solvers.size();

        // order processors, and tell each one about the neighbor
        // call solve for each processor to tell them about
knapsack
    try {
        sendSolverInformation();
        startProcessing();
    } catch (ServerException e) {
        String message = "Failed to solve the knapsack " +
            e.getMessage();
        System.err.println(message);
        throw new ServerException(message );
    }
    catch (Exception e) {
        e.printStackTrace();
    }

    while (this.result == null) {
        /* wait for result*/
    }

    return this.result;
}

/* (non-Javadoc)
 * @see server.KnapsackSolverServer#notifyComplete
 */
public void notifyComplete(HashMap result)
    throws RemoteException
{
    this.result = result;
}

/**
 * puts a solver on the stack of valid solvers
 * Method registerSolver.
 * @param solver
 */
protected void addSolver(String solverName,
    KnapsackSolverHuntPL solver) {
    if ( solvers == null ) {
        solvers = new LinkedList();
    }
    solvers.add(solver);
}

```

```

/***** private functions *****/

/*
 * for each solver,
 * send solver information about its neighbor
 * and the knapsack to be solved
 */
private void sendSolverInformation() throws ServerException {

    //objects to be calculated by this solver before sending
    numObjectsToCalc = (int)Math.ceil(
        (double) numObjects / (double) totalProcessors +1) ;
    if (numObjects < 1) {
        numObjectsToCalc = 1;
    }
    // number of levels to be processed
    this.numLevels = totalProcessors;

    // iterate through the solvers and
    // set the name and the range
    for ( int i=0;i<solvers.size();i++ ) {
        KnapsackSolverHuntPL solver =
            (KnapsackSolverHuntPL) solvers.get(i);
        HashMap range = calculateRange(i, numLevels);
        try {

            if (i +1 < solvers.size() ) {
                // last solver is processed differently
                // because it does not have a rightNeighbor
                KnapsackSolverHuntPL solverNext =
                    (KnapsackSolverHuntPL) solvers.get(i+1);
                String name = solverNext.getSolverName();
                solver.receiveNeighborName(name);
            }
            solver.receiveCalculationParameters(
                numLevels,
                numObjectsToCalc,
                i,
                range);
            solver.receiveKnapsackDefinition(profits,
                weights,
                capacity);
        } catch (RemoteException e) {
            throw new ServerException(
                "Failed to communicate with solver: "
                + e.getMessage());
        }
    }
}

```

```

/*
 * create an initial vector of blank profits,
 * and a blank backtrack array
 * and kick off the process by sending them to
 * the first processor
 */
private void startProcessing() throws ServerException {
    int [][] profits = new int[numObjects][2];
    int [][] backtrack = new int[numObjects][capacity +
1];

    // initialize the backtrack vector with -1
    for (int j=0;j<numObjects;j++) {
        for (int i=0;i<=capacity;i++) {
            backtrack[j][i]=-1;
        }
    }
    int objectStartIndex = 0;
    int objectEndIndex = numObjects -1;

    // start processing on the first processor
    threadReceiveMessage(
        profits,
        backtrack,
        objectStartIndex,
        objectEndIndex);
}

/*
 * send some blank vectors to first processor
 * to kick off the process
 */
private void threadReceiveMessage( final int[][] profits,
    final int[][] backtrack,
    final int objectStartIndex,
    final int objectEndIndex) {
    // get the first processor
    final KnapsackSolverHuntPL solver =
        (KnapsackSolverHuntPL) solvers.getFirst();

    final HuntPLVectorObject vo = new HuntPLVectorObject();
    vo.setProfitVectors(profits);
    vo.setHistoryVectors(backtrack);

    // thread the message sending
    Thread t = null;
    t = new Thread("beginCalculationThread") {
        public void run() {

```

```

        try {
            solver.receiveMessage(vo,
                objectStartIndex, objectEndIndex );
        } catch (RemoteException e) {
            System.err.println(
                "Server can not start calculation: "
                + " Failed to send profits to processor:
            "
                + e.getMessage());
            System.exit(0);
        }
    }
};
t.start();
}

/*
 * return an HashMap of the startRange and endRange
 * of capacities which the processor will
 * calculate at the given level
 */
synchronized private HashMap calculateRange(int level,
int totalLevels) {
    int range = (int) Math.ceil(
        (double)capacity/(double)totalLevels );
    int startRange = range * ( level ) ;
    int endRange = range * ( level +1 ) -1 ;
    if ( level == totalLevels -1) {
        endRange = capacity;
    }
    HashMap ranges = new HashMap();
    ranges.put("startRange", new Integer(startRange));
    ranges.put("endRange", new Integer(endRange));
    return ranges;
}

/**
 * clear out the cache so the next knapsack can be requested
 */
private void clearValues() {
    numObjects = 0;
    capacity = 0;
    weights = null;
    profits = null;
    result = null;
}

////////////////////////////////////
// functions required to fulfill interface requirements

/* (non-Javadoc)
 * @see server.KnapsackSolverServer#requestHelp(int, int, int)
 */

```

```

public int requestHelp(int depth, int cap, int value)
throws RemoteException, ServerException {
    return 0;
}

/* (non-Javadoc)
 * @see server.KnapsackSolverServer#notifyComplete(
 */
public void notifyComplete(String name, int pindex)
throws RemoteException {
}

/* (non-Javadoc)
 * @see server.KnapsackSolverServer#vectorForBroadcast
 */
public void vectorForBroadcast(int solverindex, int[][] vectors)
throws RemoteException {
}
}

```

ServerException.java

```
/*
 * ServerException.java
 */

package server;

/*****
 * Identify ServerExceptions
 * *****/

public class ServerException extends Exception {
    public ServerException() {
        super();
    }
    public ServerException(String msg) {
        super(msg);
    }
}
```

Appendix 3 Solver Package

Java

KnapsackSolverLeejava

```
package solver;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.LinkedList;

/**
 * defines a remote interface for the SolverLeeImpl Class
 * allows interaction between SolverLeeImpl and KnapsackServerLeeImpl
 */
public interface KnapsackSolverLee extends Remote {
    /**
     * function allow parameters to be added for different solver
     * types
     * this instance sets the processr's index
     */
    public abstract void setValue(String name, int value)
        throws RemoteException;

    /**
     * @param capacity
     * @param numObjects
     * @param objectweights
     * @param objectvalues
     * @throws RemoteException
     */
    public abstract void solve(
        int capacity,
        int numObjects,
        int[] objectweights,
        int[] objectvalues)
        throws RemoteException;

    /**
     * @param grp
     * @throws RemoteException
     */
    public abstract void combine(int[] grp) throws RemoteException;
}
```

```

/**
 * @param neighborsVector
 * @param level
 * @throws SolverException
 * @throws RemoteException
 */
public abstract void vectorExchange(int[] neighborsVector,
    int level)
    throws SolverException, RemoteException;

/**
 * @param index
 * @param recievedProfitVector
 * step 4 of algorithm 6
 * every node in the group recieves all other parts
 * of profit vectors
 */
public abstract void broadcastRecieve(int index, int[][]
    recievedVectors)
    throws RemoteException;

/**
 * called by server for backtrackcombine phase
 * returns the integer for the level's history vector
 * at index t
 */
public abstract int backtrackCombine(int level, int t)
    throws RemoteException;

/**
 * @param xi (weight at which total maximum capacity found)
 * @throws RemoteException
 */
public abstract int[] backtrackDynamic(int xi) throws
    RemoteException;

/**
 * @param capacity
 * @param numObjects
 * @param objectweights
 * @param objectvalues
 * @throws RemoteException
 */
public abstract void setValues(
    int capacity,
    int numObjects,
    int[] objectweights,
    int[] objectvalues)
    throws RemoteException;

/**
 * @param solverNames
 */

```

```
public abstract void setNames(LinkedList solverNames)
    throws RemoteException;

/**
 * the RMI lookup name of the solver
 * @return the Solvername
 * @throws RemoteException
 */
public abstract String getSolverName() throws RemoteException;
}
```

SolverLeeImpl.java

```
/*
 * SolverLeeImpl.java
 */
package solver;

import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.HashSet;

import server.KnapsackSolverServer;
import server.ServerException;

/**
 *
 */
/**
 *
 * PURPOSE: Implement the LEE Algorithm for solving 0-1 Knapsack
 problem
 *
 * performs role of solver
 *
 * must perform the following functions
 *
 * 1. accept the knapsack sub-problem from the server
 *
 * 2. create a list of possible states possible from the
 * sub-problem
 *
 * 3. create a profit vector
 *
 * 4. communicate completion of each stage with server
 *
 * 5. accept group information from server
 *
 * 6. exchange profit vector with solver who is
 * neighbor in group
 *
 * 7. create combined profit vector and history
 *
 * 8. give server requested information for combined
 * backtracking
 *
 * 9. backtrack dynamic based on results from 2 and 8
 */
public class SolverLeeImpl
    extends java.rmi.server.UnicastRemoteObject
    implements KnapsackSolverLee {

    public static String SOLVERNAME = "LEE";
    private int[] weights; // weigth of object i
    private int[] values; // value of object i

    private int capacity; // starting capacity of the knapsack
    private int numObjects; // number of objects in the knapsack

    // profit vector - will change at various stages of computation
    private int[] profitVector;
```

```

//    profit vector received from neighbor solver
private int[] neighborsVector;

//    which level the neighbors profit vector represents
private int levelVectorSent;

//    reference to Server that requested service
private KnapsackSolverServer server;

//    step point results from forward dynamic
private HashSet[] stepPoints;

//    used in backtrack processes, one int[] for each level
private HashMap historyVectors;

//    this processor's unique index
private int pindex;

//    which level are we in the forward combine state
private int combineLevel;

//    start point for subvector
private int subvectorStart;

//    list of connection names for otherSolvers
private LinkedList otherSolvers;

/**
 * constructor
 */
public SolverLeeImpl(boolean debug)
    throws RemoteException
{
    server = null;
    historyVectors = new HashMap();
    capacity = 0;
    numObjects = 0;
    pindex = 0;
    combineLevel = 0;
    neighborsVector = null;
    levelVectorSent = 0;
}

/**
 * function allow parameters to be added for different
 * solver types
 * this instance sets the processr's index
 */
public void setValue(String name, int value)
    throws RemoteException
{
    if (name.equals("pindex")) {
        this.pindex = value;
    } else if (name.equals("level")) {

```

```

        combineLevel = value;
    } else if (name.equals("subvectorStart")) {
        subvectorStart = value;
    }
}

/**
 * get names of all solvers that have registered with the server
 */

public void setNames(LinkedList solverNames)
    throws RemoteException
{
    otherSolvers = solverNames;
}

/**
 * called by server to initiate the forward dynamic process
 */
public void solve(
    int capacity,
    int numObjects,
    int[] objectweights,
    int[] objectvalues)
    throws RemoteException
{
    clearValues();

    setValues(capacity, numObjects, objectweights,
        objectvalues);

    HashSet[] S = null;
    S = forwardDynamic();
    try {
        profitVector = getProfitVector(S);
    } catch (SolverException e) {
        log("Error getting profitVector in solve: " +
            e.getMessage());
    }

    stepPoints = S;
    server.notifyComplete("forwardDynamic", pindex);

    // server should call combine()
    // when all solvers have
    // notified they have finished this stage
}

```

```

/**
 * called for forwardDynamic stages
 * error handling - don't want to send errors back to Server,
 * so catch all SolverExceptions
 */
public void combine(int[] grp)
    throws RemoteException
{
    // each node exchanges its profit vector with
    // opposite node in the kth direction

    try {
        exchangeVectorWithNeighbor(grp);
    } catch (SolverException e) {
        log("failed to connect to neighbor " +
            e.getMessage());
        System.exit(1);
    }

    int[][] ch = null;
    ch = combineAndHistory(profitVector, neighborsVector);

    profitVector = ch[0];
    historyVectors.put(new Integer(combineLevel), ch[1]);
    broadcastSend(ch);
}

/**
 * returns this solvers current profitVector
 */
public void vectorExchange(int[] neighborsV, int level)
    throws RemoteException {
    neighborsVector = neighborsV;
    levelVectorSent = level;
}

/**
 * @param index
 * @param recievedProfitVector
 * step 4 of algorithm 6
 * every node in the group recieves all
 * other parts of profit vectors
 */
synchronized public void broadcastRecieve(
    int index,
    int[][] recievedVectors)
    throws RemoteException
{
    int[] recievedProfitVector = recievedVectors[0];
    int[] recievedHistoryVector = recievedVectors[1];
    if (
        !historyVectors.containsKey(new
            Integer(combineLevel))
    ) {

```

```

        log(" history Vector claims it does not have " +
            combineLevel);
    }

    int[] historyVector =
        (int[]) historyVectors.get(new
            Integer(combineLevel));
}

/**
 * returns the integer requested for backtrackCombine
 * @param level
 * @param t
 * @return value of history at level for index t
 */
public int backtrackCombine(int level, int t) {
    int[] history = (int[])
        historyVectors.get(new Integer(level));
    if (history == null || t >= history.length) {
        return 0;
    }
    return history[t];
}

/**
 * @return Solvername (RMI lookup name)
 */
public String getSolverName()
throws RemoteException
{
    return SOLVERNAME;
}

/* (non-Javadoc)
 * @see solver.KnapsackSolverLee#backtrackDynamic(int)
 */
public int[] backtrackDynamic(int xi) throws RemoteException {

    // we stored impossible values
    // in the profit vector as 10* capacity
    if (xi > capacity) {
        int z[] = new int[numObjects + 1];
        return z;
    }

    int z[] = null;
    try {
        profitVector = getProfitVector(stepPoints);
        int P = profitVector[xi];
        int W = xi;

        z = backtrackDynamic(stepPoints, P, W);
    } catch (Exception e) {

```

```

        log("error running backtrack dynamic " +
            e.getMessage());
    }

    return z;
}

/***** private functions *****/

/*
 * backtracking part of dynamic programming
 * (P,W) <= last tuple in SnumObjects
 * for <= m downto 1 do
 * begin
 *   if (P - pi, W-wi) in S_(i-1) then
 *     zi<=1; P <= P- pi; W <= W-wi
 *   else
 *     zi<=0
 *   endif
 * end
 * returns the solution vector
 */
private int[] backtrackDynamic(HashSet[] S, int P, int W) {
    int[] z = new int[numObjects + 1]; // solution vector

    for (int i = numObjects - 1; i >= 0; i--) {
        HashSet s_last = (HashSet) S[i];

        int[] checkTuple = createTuple(P - values[i],
                                        W - weights[i]);

        if (P == 0 && W == 0) {
            return z;
        }

        if (hashSetContainsTuple(s_last, checkTuple)) {
            z[i] = 1;
            P = P - values[i];
            W = W - weights[i];
        } else {
            z[i] = 0;
        }
    }

    return z;
}

/*
 * get the neighbor,
 * and request their vector
 * since neighbor will be calling this too, end results is
 * both vectors are exchanged
 */
private void exchangeVectorWithNeighbor(int[] grp)
    throws RemoteException, SolverException {
    int opposite = getOpposite(grp);

```

```

KnapsackSolverLee oppositeSolver =
connectToSolver(opposite);

oppositeSolver.vectorExchange(profitVector, combineLevel);
while (levelVectorSent != combineLevel) {
    // wait for neighborVector to be sent
}
}

/*
 * get all solvers in our group,
 * and send them the piece of the vector we computed
 * threads so that calling function can return
 */
private void broadcastSend(final int[][] vectors) {

    Thread t = new Thread("broadcastThread") {
        public void run() {
            try {
                server.vectorForBroadcast(pindex,
                    vectors);
            } catch (RemoteException e) {
                log(
                    "Failed to broadcast vector to server: "
                    + e.getMessage());
            }
        }
    };
    t.start();
}

/*
 * combine operation combines the profit vectors
 * it is commutative and associative
 * history operation provides information for
 * tracing back the combining procedure later
 * this function returns an array two arrays
 * the first array returned is the profit vector,
 * the second is the history vector
 * values sent in first vector are what is used in history
 */
private int[][] combineAndHistory(int[] bfull, int[] dfull) {
    int[] h = new int[capacity + 1]; // new history vectory
    int[] e = new int[capacity + 1]; // new profit vector

    for (int cap = 0; cap <= capacity; cap++) {
        int max = 0;
        int j = cap;
        int i = 0;
        h[cap] = capacity * 10;
        while (i <= cap && j >= 0) {
            int tempmax = bfull[i] + dfull[j];
            if (tempmax > max) {
                e[cap] = tempmax;
                h[cap] = i;
            }
        }
    }
}

```

```

        max = tempmax;
    }
    i++;
    j--;
}

int[][] results = new int[2][capacity + 1];
results[0] = e;
results[1] = h;
return results;
}

/*
 * fm(c) is the optimal solution for the problem
 * fi(x) is max up to i
 * Si is list of tuples from the coordinates of the step points
 * the size of the list Si is not greater than capacity + 1
 * tuples are listed in increasing order of x and fi(x)
 * The sequence fo lists S0, S1...Si...Sm is a
 * history of the dynamic programming
 *
 */
private HashSet[] forwardDynamic() {
    int i = 0; //counter
    int W = 0; //max weight
    int P = 0; //max profit
    HashSet[] S = new HashSet[numObjects + 1];
        // array where each value
    // is list of tuples for dynamic programming

    //-- initialize S0 = { (0,0) }
    S[0] = new HashSet();
    S[0].add(createTuple(0, 0));

    for (i = 0; i < numObjects; i++) {
        HashSet tempS = new HashSet();
        int pi = values[i];
        int wi = weights[i];

        //-- (P,W) \elementof S
        //Si is result of previous iteration
        Iterator it = S[i].iterator();
        while (it.hasNext()) {
            int[] preTuple = (int[]) it.next();
            P = preTuple[0];
            W = preTuple[1];

            //-- W + wi \leq c
            if (W + wi <= capacity) {
                //-- tempS is { (P+pi, W+wi) }
                tempS.add(createTuple(P + pi, W + wi));
            }
        }
    }
}

```

```

        S[i + 1] = (HashSet) merge(S[i], tempS);

    }
    log("forwardDynamic ending");
    return S;
}

/*
 * The merge procedure
 * merges two lists Si-1 and tempS to create list Si
 * During the merging, if tempS \Union Si-1 contains two tuples
 * (Pj, Wj) and (Pk, Wk) such that
 * Pj <= Pk and Wj >=Wk
 * i.e. (Pk, Wk) dominates (Pj, Wj),
 * then (Pj, Wj) is discarded
 * in tuple[0] = profit
 *     tuple[1] = weight
 * @param lastS HashSet
 * @param tuple tempS
 * @return merged LinkedList
 */
private HashSet merge(HashSet lastS, HashSet tempS) {
    HashSet Si = new HashSet();
    Si.addAll(lastS);
    Si.addAll(tempS);
    Iterator lastIt = lastS.iterator();
    while (lastIt.hasNext()) {

        int[] lastTuple = (int[]) lastIt.next();
        Iterator tempIt = tempS.iterator();
        while (tempIt.hasNext()) {

            int[] nextTuple = (int[]) tempIt.next();

            // if nextTuple dominates last Tuple do not
            // include lastTuple
            if ((lastTuple[0] <= nextTuple[0])
                && (lastTuple[1] >= nextTuple[1])) {
                // do not include lastTuple
                Si.remove(lastTuple);
            }
            // if lastTuple dominates nextTuple,
            // do not include nextTuple
            else if (
                (nextTuple[0] <= lastTuple[0])
                && (nextTuple[1] >= lastTuple[1])) {
                Si.remove(nextTuple);
            }
        }
    }
    return Si;
}

/*

```

```

    * create a 2-tuple represented as array with 2 indices
    */
private int[] createTuple(int x, int y) {
    int[] tuple = new int[2];
    tuple[0] = x;
    tuple[1] = y;
    return tuple;
}

/*
 * figure out with whom this processor
 * should be communicating with
 * for a given level, have nodes in
 * opposite direction with whom to exchange
 * and have a group for broadcastin
 */
private int getOpposite(int[] group) {

    if (group.length == 1) {
        log("failed system status, can not have a"
            + "group of one");
        System.exit(1);
    }
    int opposite_index = 0;

    for (int i = 0; i < group.length; i++) {
        if (group[i] == pindex) {
            opposite_index = i;
        }
    }

    opposite_index = group.length - 1 - opposite_index;
    return group[opposite_index];
}

/* (non-Javadoc)
 * @see solver.KnapsackSolverLee#setValues
 */
public void setValues(
    int capacity,
    int numObjects,
    int[] objectweights,
    int[] objectvalues)
    throws RemoteException {
    this.capacity = capacity;
    this.numObjects = numObjects;
    this.weights = objectweights;
    this.values = objectvalues;
}

/* reset all variables
 * so we can call this solver again with a new problem
 */
private void clearValues() {

```

```

        this.capacity = 0;
        this.numObjects = 0;
        this.weights = null;
        this.values = null;
        combineLevel = 0;
        levelVectorSent = 0;
        subvectorStart = 0;
        profitVector = null;
        neighborsVector = null;

        historyVectors.clear();
        stepPoints = null;
    }

    /* return a profit vector
     * a[i] = f_numObjects[i]
     */
    private int[] getProfitVector(HashSet[] S) throws SolverException
    {
        // start with list created from all the objects
        if (S == null || S.length < 1) {
            throw new SolverException("getProfitVector called
            with a null HashSet Array S");
        }
        if (S[S.length - 1] == null) {
            throw new SolverException(
                "getProfitVector found a null hashset at "
                + "the end of S "
                + S.length);
        }
        HashSet s_last = (HashSet) S[S.length - 1];
        int[] a = new int[capacity + 1];
        a[0] = 0;

        Iterator it = s_last.iterator();
        while (it.hasNext()) {
            int[] tuple = (int[]) it.next();
            int weight = tuple[1];
            int value = tuple[0];
            if (value >= a[weight]) {
                a[weight] = value;
            }
        }
        for (int i = 1; i <= capacity; i++) {
            if (a[i - 1] >= a[i]) {
                a[i] = a[i - 1];
            }
        }
        return a;
    }
}

```

```

/*
 * connect to another solver with index i
 */
private KnapsackSolverLee connectToSolver(int i)
throws SolverException {
    KnapsackSolverLee solver = null;
    String solverName = (String) otherSolvers.get(i);
    try {
        solver = (KnapsackSolverLee)
            Naming.lookup(solverName);
    } catch (NotBoundException e) {
        throw new SolverException(
            "Could not connect to "
                + solverName
                + " to broadcast "
                + e.getMessage());
    } catch (java.io.IOException e) {
        throw new SolverException(
            "I/O error or bad host name with "
                + solverName
                + " "
                + e.getMessage());
    }
    return solver;
}

/*
 * setup connection with the server
 * and let that server know this server is open to requests
 */
private KnapsackSolverServer registerWithServer(String
lookupName, String serverHost)
throws SolverException {
    String serverName = "rmi://" + serverHost + "/" +
KnapsackSolverServer.SOLVERLOOKUPNAME;
    KnapsackSolverServer server = null;
    try {
        System.out.print("Connecting to "
            + serverName + "... ");
        server = (KnapsackSolverServer)
            Naming.lookup(serverName);
        System.out.println("Connected to server: "
            + serverName);
    } catch (NotBoundException e) {
        throw new SolverException("Server not registered ");
    } catch (java.io.IOException e) {
        throw new SolverException(
            "I/O error or bad host name with "
                + serverName
                + " "
                + e.getMessage());
    }
    try {
        server.registerSolver(lookupName);
    } catch (RemoteException e) {

```

```

        throw new SolverException(
            "Failed to register self with server."
            + e.getMessage());
    } catch (ServerException e) {
        throw new SolverException(
            "Failed to add self to server "
            + e.getMessage());
    }
    return server;
}

/*
 * since "HashSet.contains()" method compares object reference,
 * no values of array,
 * we need to write a special function
 * that compares the values in the tuple
 */
private boolean hashSetContainsTuple(HashSet s, int[] tuple) {
    Iterator it = s.iterator();
    while (it.hasNext()) {
        int[] testtuple = (int[]) it.next();
        if (testtuple[0] == tuple[0]
            && testtuple[1] == tuple[1]) {
            return true;
        }
    }
    return false;
}

/*
 * single place for all logging done in this class
 */
private void log(String message) {
    System.err.println(message);
}

/*
 * return an array as a string
 */
private String stringArray(int[] a) {
    StringBuffer sb = new StringBuffer();
    sb.append("\n");
    for (int i = 0; i < a.length; i++) {
        sb.append(a[i] + " - ");
    }
    sb.append("\n");
    return sb.toString();
}

/*
 * return a hashset as a string
 */
private String stringHashSet(HashSet s) {
    StringBuffer sb = new StringBuffer();

```

```

        Iterator it = s.iterator();
        while (it.hasNext()) {
            int[] tuple = (int[]) it.next();
            sb.append(" - " + stringArray(tuple));
        }
        return sb.toString();
    }

    ////////////////////////////////////////////////// user interface methods
    //////////////////////////////////////////////////

    /**
     * usage - if no filename supplied
     */
    public static void usage() {
        System.out.println("USAGE: SolverLeeImpl [bindName] [debug]
            [ipaddress] [serveripaddress]");
        System.out.println(
            "\t bindname - unique name for client to register with ");
        System.out.println(
            "\t debug - 0 = do not show debugginginfo");
        System.exit(0);
    }

    /**
     * starts and registers itself as the servers
     * contacts KnapsackSolverServer to register self
     * as being ready to take requests
     */
    public static void main(String[] args) {
        if (args.length != 4) {
            usage();
        }

        // user provides a name at command line
        String name = args[0];
        boolean dbug = (args[1].equals("0")) ? false : true;
        String solverhostname = args[2];
        String serverhostname = args[3];

        SOLVERNAME = "rmi://" + solverhostname +
            "/" + SOLVERNAME + name;

        // register self as server
        SolverLeeImpl self = null;
        try {

            self = new SolverLeeImpl(dbug);
            Naming.rebind(SOLVERNAME, self);
            System.out.println(
                "This solver " + SOLVERNAME
                + " ready and waiting...");
        } catch (java.io.IOException e) {
            System.out.println("Could not register Solver: " +
                e.getMessage());
        }
    }

```

```

        System.exit(1);
    } catch (Exception e) {
        System.out.println(
            "Unexpected error starting Solver: " +
            e.getMessage());
        e.printStackTrace();
        System.exit(1);
    }

    // lookup KnapsackSolverServer
    try {
        self.server = self.registerWithServer(SOLVERNAME,
            serverhostname);
        System.out.println("Registered with Server ");
    } catch (SolverException e) {
        System.out.println(
            "Error registering with SolverServer: " +
            e.getMessage());
        e.printStackTrace();
        System.exit(1);
    }
}
}
}

```

KnapsackSolverChen.java

```
/*
 * KnapsackSolverChen.java
 */
package solver;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.HashMap;
import java.util.LinkedList;

/**
 * defines a remote interface for the SolverChenImpl Class
 * allows interaction between SolverChenImpl and KnapsackServerChenImpl
 */
public interface KnapsackSolverChen extends Remote {

    /**
     * receive the previous profit vector,
     * the backtrack vector,
     * and the current index
     */
    void receiveMessage(
        int[] prevProfits,
        int[][] backtrack,
        int index,
        int level,
        LinkedList capacityPoints)
        throws RemoteException;

    /**
     * Server calls this to set the neighbors name
     * @param name rmi lookup name for neighbor
     */
    public void receiveNeighborName(String name) throws
        RemoteException;

    /**
     * called by server to specify the knapsack which needs
     * to be solved
     * @param profits
     * @param weights
     * @param capacity
     * @param levels
     * @param objectOffset
     * @throws RemoteException
     */
    public void recieveKnapsackDefinition(
        int[] profits,
        int[] weights,
        int capacity,
        int levels,
```

```
        int objectOffset,  
        HashMap[] levelRanges)  
        throws RemoteException;  
  
    /**  
     * @return name of solver  
     */  
    public String getSolverName() throws RemoteException;  
  
}
```

SolverChenImpl.java

```
/*
 * SolverChenImpl.java
 */
package solver;

import java.rmi.MarshalException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import server.KnapsackSolverServer;
import server.ServerException;

/**
 * PURPOSE: Act as one processor which solves the
 * 0-1 Knapsack in the manner described
 * in [CHEN]
 * Must perform the following functions
 * 1. accept parameters from server
 * 2. accept information about neighbors from group
 * 3. calculate values for object i
 * 4. at each level, send values to right neighbor
 * 5. receive results from left neighbor
 * 6. return results to solver
 */
public class SolverChenImpl
    extends java.rmi.server.UnicastRemoteObject
    implements KnapsackSolverChen {

    public static String SOLVERNAME = "CHEN";

    // server registration
    private KnapsackSolverServer server = null;

    // solver to the right
    private KnapsackSolverChen rightNeighbor;

    //options to use capacity points
    private boolean useCapacityPoints;

    // weights of all objects
    private int[] weights;

    // profits of all objects
```

```

private int[] profits;

// total number of objects
private int numObjects;

// capacity of knapsack
private int capacity;

// total number of levels for computation
private int totalLevels;

/* expected offset of objects
 * in between calculations */
private int indexOffset;

// profits for f(k,g) where k is constant
private int[][] profitVectors;

// backtrack values sent from left neighbor
private int[][] backtrack;

// the last level we received for each object
private int[] lastLevelReceived;

// current range of computation based on level
private int startRange;

// current range of computation based on level
private int endRange;

// calculate all the ranges for each level at the beginning
private HashMap[] levelRanges;

/**
 * constructor for SolverChenImpl
 */
public SolverChenImpl() throws RemoteException {
    super();
    rightNeighbor = null;
}

/**
 * @return Solvername (RMI lookup name)
 */
public String getSolverName() throws RemoteException {
    return SOLVERNAME;
}

/**
 * Server calls this to set the neighbors name
 * @param name rmi lookup name for neighbor
 */
public void receiveNeighborName(String name) throws
RemoteException {

```

```

    try {
        if (rightNeighbor == null) {
            connectToNeighbor(name);
        }
    } catch (SolverException e) {
        System.err.println(
            "Failed to establish connection with "
                + name
                + ": "
                + e.getMessage());
        throw new RemoteException(e.getMessage());
    }
}

/**
 * called by server to specify the knapsack which needs to
 * be solved
 * initializes variables
 */
synchronized public void recieveKnapsackDefinition(
    int[] profits,
    int[] weights,
    int capacity,
    int levels,
    int objectOffset,
    HashMap[] levelRanges)
    throws RemoteException {
    if (rightNeighbor == null) {
        System.err.println(
            "Can not begin to solve knapsack until neighbor " +
            " is set.");
        throw new RemoteException("Can not begin to solve "+
            " knapsack until neighbor is set.");
    }

    // set the variables required to process this knapsack
    this.profits = profits;
    this.weights = weights;
    this.capacity = capacity;
    this.totalLevels = levels;
    this.numObjects = weights.length;
    this.indexOffset = objectOffset;
    this.levelRanges = levelRanges;

    // initialize other variables
    this.profitVectors = new int[numObjects][capacity + 1];
    this.backtrack = new int[numObjects][capacity + 1];
    this.lastLevelReceived = new int[numObjects];
}

/**
 * receive a message from the left neighbor
 * this triggers the processor to begin the process
 * @throws SolverException

```

```

*/
public void receiveMessage(

    int[] prevProfs,
    int[][] prevBacktrack,
    int index,
    int level,
    LinkedList capacityPoints)
    throws RemoteException {

    // the object we will want to calculate if we are ready
    int currentIndex = index + 1;
    int prevIndex = index >= 0 ? index : 0;

    log("received message for index " + index + " at level "
        + level);

    // if we have not received a level greater
    // than the one sent
    if (lastLevelReceived[currentIndex] <= level) {

        // copy the recieved profit vector into the array
        profitVectors[prevIndex] = prevProfs;
        // copy bactrack received into our array
        copyBackTrack(prevBacktrack, currentIndex, level);

        lastLevelReceived[currentIndex] = level;
    }

    // if this is a new object, begin processing it
    if (level == 0 || index == -1) {
        try {
            //determine the next level/object to calculate
            calculateObjectAtAllLevels(currentIndex,
                capacityPoints);
        } catch (SolverException e) {
            String message =
                ("Failed to calculate object"
                    + currentIndex
                    + " at level "
                    + level
                    + ": "
                    + e.getMessage());
            log(message);
            throw new RemoteException(message);
        }
    }
}
}

```

```

//////////////////////////////// private methods //////////////////////////////////

/*
 * continue to the next step in our calculation
 * calculate the object at all levels
 * wait if the levels have not yet been received
 */
private void calculateObjectAtAllLevels(
    int index,
    LinkedList capacityPoints)
    throws SolverException {

    // get the capacitypoints for this object
    LinkedList objectCapacityPoints = null;
    if (useCapacityPoints) {
        objectCapacityPoints =
            createCapacityPointsCurrent(capacityPoints,
                index);
    }

    for (int level = 0; level < totalLevels; level++) {
        while (lastLevelReceived[index] < level) {
            // wait for previous processor to send
            // this level
        }

        // set the ranges for this level based on previously
        // calculated data struct
        calculateRange(level);

        // perform the calculation
        LinkedList capacityPointsNext =
            calculateProfits(index, capacityPoints,
                objectCapacityPoints);

        // if we have calculated all objects, do not send //
        profits to next processor
        if (index == numObjects - 1) {
            // get final solution if we are at last level
            if (level == totalLevels - 1) {
                HashMap result = calculateResults();
                notifyServerWithResults(result);
                return;
            }
        } else {
            //send the results to neighbor
            sendProfits(
                index,
                level,

                capacityPointsNext,
                profitVectors[index]);
        }
    }
}

```

```

    }

    return;
}

/*
 * start the process of calculating f(k,g)
 * where k is index of current object (fixed)
 * and g is all possible capacities
 */
synchronized private LinkedList calculateProfits(
    int index,
    LinkedList capacityPointsPrevious,
    LinkedList objectCapacityPoints)
    throws SolverException
{
    //           previous object   index
    int prevIndex = (index - 1 >= 0) ? index - 1 : 0;

    final int[] previousProfits = profitVectors[prevIndex];

    // objectCapacityPoints is the copy that we will modify
    //while iterating through capacityPointsCurrent
    LinkedList capacityPointsCurrent = null;
    if (useCapacityPoints) {
        capacityPointsCurrent = new LinkedList();
        capacityPointsCurrent.addAll(objectCapacityPoints);
        if (capacityPointsCurrent.size() == 0) {
            throw new SolverException("reached
            calculate profits with no
            objectCapacityPoints "
                + objectCapacityPoints.size());
        }
    } else {
        capacityPointsCurrent =
            createCapacityListFromRange(startRange, endRange);
    }

    // move up to starting point
    Integer startRangeObj = new Integer(startRange);
    int nextPoint = ((Integer)
        capacityPointsCurrent.getFirst()).intValue();
    int i = 0;

    while (nextPoint < startRange
        && i < capacityPointsCurrent.size() - 1) {
        i++;
        nextPoint = ((Integer)
            capacityPointsCurrent.get(i)).intValue();
    }

    // i is now the index of the first capacityPoint
    // greater than the startRange
    // nextPoint is the value of the first point that is
    // greater than the start range

```

```

// profit at next point has not yet been calculated

// begining calculating capacity points in our range
while (nextPoint <= endRange
&& i < capacityPointsCurrent.size()) {
    Integer currentPointObj = (Integer)
    capacityPointsCurrent.get(i);
    int currentPoint = currentPointObj.intValue();
    int[] currentProfits = profitVectors[index];

    // if there is room for this object, determine
    // whether to adding it is beneficial
    int capacityRemaining =
        currentPoint - weights[index];
    if (capacityRemaining >= 0) {

        int currentObjectProfit
        = profits[index] +
        previousProfits[capacityRemaining];
        currentProfits[currentPoint] =
        Math.max(previousProfits[currentPoint],
            currentObjectProfit);

        if (currentObjectProfit >=
        previousProfits[currentPoint]) {
            if (useCapacityPoints) {

                capacityPointsPrevious.remove(
                    currentPointObj);
            }
            backtrack[index][currentPoint] = index;
        } else {
            if (useCapacityPoints) {
                objectCapacityPoints.remove(
                    currentPointObj);
            }

            backtrack[index][currentPoint] =
                backtrack[prevIndex][currentPoint];
        }
    }
}

// there is no room for this object
// use profit of previous object
else {
    currentProfits[currentPoint] =
    previousProfits[currentPoint];
    backtrack[index][currentPoint] =
    backtrack[prevIndex][currentPoint];
    if (useCapacityPoints) {

        objectCapacityPoints.remove(
            currentPointObj);
    }
}

```

```

    }

    i++;
    if (i < capacityPointsCurrent.size()) {
        nextPoint =
            ((Integer)capacityPointsCurrent.get(i)).
                intValue();
    } // else will fail on next loop

}

LinkedList capacityPointsNext = null;
if (useCapacityPoints) {
    //          fill in blanks for last set of indices
    updateVectorsAtBetweenPoints(index);

    // create a list of all capacity points which
    // will be merge/trimmed for next processor
    capacityPointsNext =
        createCapacityPointsAll(
            capacityPointsPrevious,
            objectCapacityPoints);
}

return capacityPointsNext;
}

/*
 * if the profit has been modified, we need to modify
 * all capacitys in between the points of change
 */
synchronized private void updateVectorsAtBetweenPoints(int index)
{

    int prevIndex = index - 1 < 0 ? 0 : index - 1;
    for (int i = startRange; i <= endRange; i++) {
        int prevCapacity = i - 1 < 0 ? 0 : i - 1;
        if (profitVectors[index][prevCapacity] >
            profitVectors[index][i]) {
            profitVectors[index][i] =
                profitVectors[index][prevCapacity];
            backtrack[index][i] =
                backtrack[index][prevCapacity];
        }
        if (profitVectors[prevIndex][i] >
            profitVectors[index][i]) {
            profitVectors[index][i] =
                profitVectors[prevIndex][i];
        }
    }
}

/*
 * set the member variables for ranges

```

```

    * for this level based on previously calculated data struct
    */
private void calculateRange(int currentLevel) {
    Integer startRangeInt =
        (Integer) ((HashMap)
            levelRanges[currentLevel]).get("startRange");

    Integer endRangeInt =
        (Integer) ((HashMap)
            levelRanges[currentLevel]).get("endRange");

    // set the instance variables
    startRange = startRangeInt.intValue();
    endRange = endRangeInt.intValue();

}

/*
 * create a sorted list of the union of all Capacity Points
 */
synchronized private LinkedList createCapacityPointsAll(
    LinkedList capacityPointsOld,
    LinkedList capacityPointsNew) {
    // set so values will be unique
    HashSet allSet = new HashSet();
    allSet.addAll(capacityPointsNew);
    allSet.addAll(capacityPointsOld);
    allSet.add(new Integer(0));

    // linked list since order matters
    LinkedList capacityPointsAll = new LinkedList();
    capacityPointsAll.addAll(allSet);

    Collections.sort((List) capacityPointsAll);
    return capacityPointsAll;

}

/*
 * create list of new capacity Points
 * capacity points should all be in the range of the level
 */
synchronized private LinkedList createCapacityPointsCurrent(
    LinkedList capacityPointsOld,
    int index) {
    LinkedList capacityPointsNew = new LinkedList();
    Iterator it = capacityPointsOld.iterator();

    int newPoint = 0;

    // all lists are initialized with 0
    capacityPointsNew.add(new Integer(newPoint));

    // add the weight of the current object to each
    // point in the old list

```

```

while (it.hasNext() && newPoint <= capacity) {
    Integer val = (Integer) it.next();
    newPoint = weights[index] + val.intValue();
    if (newPoint <= capacity) {
        capacityPointsNew.add(new Integer(newPoint));
    }
}

// add the final capacity
capacityPointsNew.add(new Integer(capacity));
return capacityPointsNew;
}

/*
 * to test whether the capacity points list is an improvement,
 * we have an option not to use it
 * however, we still want a linkedlist of points through which
 * to iterate when calculating profits
 * so create this linked list of points in our range
 */
private LinkedList createCapacityListFromRange(
    int startRange,
    int endRange) {
    LinkedList capacityPoints = new LinkedList();
    for (int i = startRange; i <= endRange; i++) {
        capacityPoints.add(new Integer(i));
    }
    return capacityPoints;
}

/*
 * copy the backtrack vectors sent to current backtrack
 * up to the given object
 * start at the previous object calculated by this processor
 */
synchronized private void copyBackTrack(
    int[][] prevBacktrack,
    int objectIndex,
    int level) {
    // if we are on first object, just copy whole thing
    if (objectIndex == 0) {
        backtrack = prevBacktrack;
        return;
    }

    // just copy what has been calculated since our last object
    int lastIndexSeen =
        objectIndex - indexOffset > 0
        ? objectIndex - indexOffset
        : 0;
    for (int i = lastIndexSeen; i < objectIndex; i++) {
        backtrack[i] = prevBacktrack[i];
    }
}

```

```

// if we are just beginning to look at object
// copy the whole backtrack vector
if (level == 0) {
    System.arraycopy(
        prevBacktrack[objectIndex - 1],
        0,
        backtrack[objectIndex],
        0,
        capacity);
}
}

/*
 * send currentProfits to the neighbor to our right
 */
synchronized private void sendProfits(
    final int index,
    final int level,
    final LinkedList capacityPoints,
    final int[] profits) {
    final int[][] backtrck = backtrack;

    Thread t = null; // the thread to ask the server
    t = new Thread("sendProfitsThread") {
        public void run() {
            boolean sent = false;
            while (sent == false) {
                try {
                    rightNeighbor.receiveMessage(
                        profits,
                        backtrck,
                        index,
                        level,
                        capacityPoints);

                } catch (RemoteException e) {
                    sent = false;
                    log(
                        "Failed to send profits to "
                        +" processor at "
                        + index
                        + " "
                        + e.getMessage());
                }
                sent = true;
            }
        }
    };
    t.start();
}

/*
 * calculateResults
 * called when the profits have been calculated

```

```

    * runs the backtrack algorithm and then returns the
    * result to the server
    */
    synchronized private HashMap calculateResults() {
        int maxProfit = profitVectors[numObjects - 1][capacity];

        int[] bag = null;
        try {
            bag =
                HuBackTrackImpl.backtrack(
                    capacity,
                    weights.length,
                    backtrack,
                    weights);

        } catch (SolverException e) {
            System.err.println(
                "Failed to compute the knapsack solution: " +
                e.getMessage());
            System.exit(-1);
        }

        //store the result in data structure and return it to
        // server
        HashMap result = new HashMap();
        result.put("result", new Integer(maxProfit));
        result.put("resultSet", bag);
        return result;
    }

    /*
    * notify Server of completed results
    * @param result
    */
    synchronized private void notifyServerWithResults(HashMap result)
        throws SolverException {
        try {
            server.notifyComplete(result);
        } catch (RemoteException e) {
            throw new SolverException(
                "reached result but failed to send to server "
                + e.getMessage());
        }
    }

    /*
    * establish the connection with the neighbor
    * and set the instance variable
    */
    private void connectToNeighbor(String solverName)
        throws SolverException {
        try {
            rightNeighbor = (KnapsackSolverChen)
                Naming.lookup(solverName);
        } catch (NotBoundException e) {

```

```

        throw new SolverException(
            "Could not connect to "
                + solverName
                + " to broadcast "
                + e.getMessage());
    } catch (java.io.IOException e) {
        throw new SolverException(
            "I/O error or bad host name with "
                + solverName
                + " "
                + e.getMessage());
    }
}

/*
 * setup connection with the server
 * and let that server know this server is open to requests
 */
private KnapsackSolverServer registerWithServer(
    String serverhostname,
    String lookupName)
    throws SolverException {
    String serverName =
        "rmi://"
            + serverhostname
            + "/"
            + KnapsackSolverServer.SOLVERLOOKUPNAME;

    KnapsackSolverServer server = null;
    try {
        System.out.print("Connecting to "
            + serverName + "... ");
        server = (KnapsackSolverServer)
            Naming.lookup(serverName);
        System.out.println("Connected to server: "
            + serverName);
    } catch (NotBoundException e) {
        throw new SolverException("Server not registered ");
    } catch (java.io.IOException e) {
        throw new SolverException(
            "I/O error or bad host name with "
                + serverName
                + " "
                + e.getMessage());
    }
    try {
        server.registerSolver(lookupName);
    } catch (RemoteException e) {
        throw new SolverException(
            "Failed to register self with server."
                + e.getMessage());
    } catch (ServerException e) {
        throw new SolverException(
            "Failed to add self to server "

```

```

        + e.getMessage());
    }
    return server;
}

private void log(String message) {
    System.out.println(message);
}

////////// main and usage
//////////
/**
 * starts and registers itself as the servers
 * contacts KnapsackSolverServer to register self
 * as being ready to take requests
 */
public static void main(String[] args) {
    if (args.length != 4) {
        usage();
    }

    // user provides a name at command line
    String name = args[0];
    boolean tempUseCapacityPoints = (args[1].equals("0"))
        ? false
        : true;
    String solverhostname = args[2];
    String serverhostname = args[3];

    SOLVERNAME = "rmi://" + solverhostname + "/"
        + SOLVERNAME + name;

    // register self as server
    SolverChenImpl self = null;
    try {
        self = new SolverChenImpl();
        Naming.rebind(SOLVERNAME, self);
        System.out.println(
            "This solver " + SOLVERNAME
            + " ready and waiting...");
    } catch (java.io.IOException e) {
        System.out.println(
            "Could not register Solver: " +
            e.getMessage());
        System.exit(1);
    }

    // lookup KnapsackSolverServer
    try {
        self.server =
            self.registerWithServer(serverhostname, SOLVERNAME);
        System.out.println("Registered with Server ");
    } catch (SolverException e) {

```

```

        System.out.println(
            "Error registering with SolverServer: " +
            e.getMessage());
        e.printStackTrace();
        System.exit(1);
    }
    self.useCapacityPoints = tempUseCapacityPoints;
}

/**
 * usage - if no filename supplied
 */
public static void usage() {
    System.out.println(
        "USAGE: SolverChenImpl [bindName]
        [useCapacityPints] [solveripaddress]
        [serveripaddress]");
    System.out.println(
        "\t bindname - unique name for client to register
        with ");
    System.out.println(
        "\t useCapacityPoints: whether to calculate capacity
        points or calculate all profits");
    System.exit(0);
}
}

```

KnapsackSolverHuntPL.java

```
/*
 * KnapsackSolverHuntPL.java
 */
package solver;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.HashMap;

/**
 * defines a remote interface for the SolverHuntPLImpl Class
 * allows interaction between SolverHuntPLImpl and
 * KnapsackServerHuntPLImpl
 */
public interface KnapsackSolverHuntPL extends Remote {

    /**
     * receive the previous profit vector,
     * the backtrack vector,
     * and the current index
     */
    public void receiveMessage(
        HuntPLVectorObject vo,
        int startIndex,
        int endIndex)
        throws RemoteException;

    /**
     * Server calls this to set the neighbors name
     * @param name rmi lookup name for neighbor
     */
    public void receiveNeighborName(String name) throws
    RemoteException;

    /**
     * server calls this to set the required knapsack information
     */
    public void receiveKnapsackDefinition(
        int[] profits,
        int[] weights,
        int capacity)
        throws RemoteException;

    /**
     * send the specific information about this configuration
     * depends on both the knapsack and the number of other
    processors
     */
    public void receiveCalculationParameters(
        int totalLevels,
        int numObjectsToCalc,
```

```
        int solverIndex,  
        HashMap range)  
        throws RemoteException;  
  
    /**  
     * @return name of solver  
     */  
    public String getSolverName() throws RemoteException;  
}
```

SolverHuntPLImpl.java

```
/*
 * SolverHuntPLImpl.java
 */
package solver;

import java.rmi.MarshalException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RMIException;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.RemoteServer;

import java.util.HashMap;
import java.util.LinkedList;

import server.KnapsackSolverServer;
import server.ServerException;

/**
 *
 * PURPOSE: Act as one processor which solves the 0-1 Knapsack
 *          Must perform the following functions
 *          1. accept parameters from server
 *          2. accept information about neighbor
 *          3. compute all objects for a given capacity
 *          4. every n objects, send message to right processor
 *          5. at same time, receive messages from left processor
 *          6. return results to solver
 */
public class SolverHuntPLImpl
    extends java.rmi.server.UnicastRemoteObject
    implements KnapsackSolverHuntPL
{

    public static String SOLVERNAME = "HUNTPL";

    // server registration
    private KnapsackSolverServer server = null;

    // solver to the right
    private KnapsackSolverHuntPL rightNeighbor;

    // weights of all objects
    private int [] weights;

    // profits of all objects
```

```

private int [] profits;

// total number of objects
private int numObjects;

// capacity of knapsack
private int capacity;

// total number of levels for computation
private int totalLevels;

// number of Objects to calculate
private int objectsToCalc;

// range of capacities for computation
private int startRange;
private int endRange;

// what is this solver's place among all solvers
private int solverIndex;

// the last end index received from the previous solver
private int lastIndexReceived;

private int [][] profitVectors;

// backtrack values (sent from left neighbor and then modified)
private int [][] backtrackVectors;

// capacityPoints for calculation that span the level
private LinkedList objectCapacityPoints;

/**
 * constructor for SolverChenImpl
 */
public SolverHuntPLImpl() throws RemoteException {
    super();
    rightNeighbor = null;
}

/**
 * @return Solvername (RMI lookup name)
 */
public String getSolverName() throws RemoteException {
    return SOLVERNAME;
}

/**
 * Server calls this to set the neighbors name
 * @param name rmi lookup name for neighbor
 */

```

```

    public void receiveNeighborName(String name) throws
RemoteException {
        try {
            if (rightNeighbor == null) {
                connectToNeighbor(name) ;
            }
        } catch (SolverException e) {
            System.err.println(
                "Failed to establish connection with "
                + name + ": " + e.getMessage());
            throw new RemoteException(e.getMessage());
        }
    }

/**
 * accept the configuration values
 * for how much should be calculated and communicated
 */
public void receiveCalculationParameters(
    int totalLevels,
    int numObjectsToCalc,
    int solverIndex,
    HashMap range)
throws RemoteException
{
    this.objectsToCalc = numObjectsToCalc;
    this.totalLevels = totalLevels;
    this.solverIndex = solverIndex;
    endRange = ((Integer)range.get("endRange")).intValue();
    startRange = ((Integer)range.get("startRange")).intValue();
}

/**
 * called by server to specify the knapsack
 * contains information about the specific Knapsack
 */
public void receiveKnapsackDefinition(
    int [] profits,
    int [] weights,
    int capacity)
throws RemoteException
{
    // set the variables required to process this knapsack
    this.profits = profits;
    this.weights = weights;
    this.capacity = capacity;
    this.numObjects = weights.length ;

    // initialize other variables
    this.lastIndexReceived = -1;
    this.backtrackVectors = new int[numObjects][capacity + 1];
    this.profitVectors = new int[numObjects][endRange + 1];
}

```

```

/**
 * @param int[] profit vector sent by neighbor
 * receive a message from the left neighbor
 * this triggers the processor to begin the process
 * index is the index on which to start,
 * going up to index + objectToCalc
 * @throws SolverException
 */
public void receiveMessage(
    HuntPLVectorObject vo,
    int startIndex,
    int endIndex)
    throws RemoteException
{

    // index is highest object that has been calculated for
    //previous set of capacities

    if (lastIndexReceived <= endIndex ) {

        // copy the vectors sent into memory
        copyVectors(vo.getProfitVectors(),
            vo.getHistoryVectors(), startIndex, endIndex);

        // set parameter so we can begin the next object
        lastIndexReceived = endIndex;
    }
    // start the calculation if we are at the
    // beginning of the knapsack
    if (startIndex == 0) {
        startCalculation(startIndex);
    }

    vo = null;
    return;
}

////////// private methods //////////

/*
 * continue to the next step in our calculation
 * startIndex is the index on which this level began
 */
private void calculateObjectsInLevel(int startIndex)
throws SolverException {

    int endIndex = objectsToCalc -1;
    for (
    int currentIndex = 0;

```

```

currentIndex<numObjects;
currentIndex++) {

    while (currentIndex > lastIndexReceived ) {
        // wait to receive the values from previous processor
    }

    // perform the calculation
    calculateProfits(currentIndex, startRange, endRange);

    // check whether we are done
    if ( isFinished(currentIndex) ) {
        return;
    }

    // endIndex is marker that we should sendMessage
    if (currentIndex == endIndex) {
        sendProfits( startIndex, endIndex );

        // set endIndex for the next set
        startIndex = currentIndex + 1;
        endIndex = currentIndex + objectsToCalc + 1;
        if (endIndex >= numObjects -1) {
            endIndex = numObjects -1;
        }
    }
}

return;
}

/*
 * checks whether we have reached the last point of calculation
 * if so, has side affect of calculating results
 * and notifying server
 */
boolean isFinished(int index) throws SolverException {

    // if we have calculated all objects
    // backtrack for results and send them to server
    if ( index == numObjects-1 ) {
        if ( endRange ==capacity) {
            HashMap result = calculateResults();
            notifyServerWithResults(result);
            return true;
        }
    }
    return false;
}

```

```

/*
 * copy the vectors from what is sent to the in memory vectors
 * but only copy as far as the previous neighbor sent
 */
private void copyVectors(
    int [][]prevProfits,
    int[][] prevBackTrack,
    int startIndex,
    int endIndex) {
    int profitRangeToCopy
        = startIndex -1 > 0 ? startIndex -1 : 1;
    int backtrackRangeToCopy = startIndex;
    // if we are on the first object
    // initialize all of the backtrack to -1
    if (solverIndex == 0 || startIndex == 0) {
        profitRangeToCopy = 0;
        backtrackRangeToCopy = capacity;
    }

    for (int i=0;i<=endIndex;i++) {
        System.arraycopy(prevBackTrack[i],
                        0,
                        backtrackVectors[i],
                        0,
                        backtrackRangeToCopy );
        System.arraycopy(prevProfits[i],
                        0,
                        profitVectors[i],
                        0,
                        profitRangeToCopy );
    }
    return;
}

/*
 * start the process of calculating f(k,g)
 * where k is index of current object (fixed)
 * and g is all possible capacities
 */
private void calculateProfits(int index,
                             int startRange,
                             int endRange ) {
    int prevIndex = ( index-1 >= 0 )? index-1: 0;

    int [] currentProfits;
    int [] previousProfits;

    if (index == 0) {
        // want new copy of currentVectors for first index
        //(when index==prevIndex)
        // otherwise as we modify it we will modify prevIndex
        previousProfits = new int[endRange+1];
        currentProfits = new int[endRange+1];
        System.arraycopy(profitVectors[index],

```

```

        0,
        currentProfits,
        0,
        endRange );
    } else {
        //create pointers to make code more readable
        currentProfits = profitVectors[index];
        previousProfits = profitVectors[prevIndex];
    }

    // perform the calculations
    for (int g=startRange ;g<=endRange ;g++) {
        int capacityRemaining = g - weights[index];
        if (capacityRemaining >= 0 ){
            currentProfits[g] =
                Math.max(previousProfits[g], profits[index] +
                    previousProfits[capacityRemaining]);
            if (profits[index] +
                previousProfits[capacityRemaining] >
                previousProfits[g]) {
                backtrackVectors[index][g] = index;
            } else {
                backtrackVectors[index][g] =
                    backtrackVectors[prevIndex][g];
            }
        } else {
            currentProfits[g] = previousProfits[g];
            backtrackVectors[index][g] =
                backtrackVectors[prevIndex][g];
        }
    }

    profitVectors[index]=currentProfits;
}

/*
 * calculateResults
 * called when the profits have been calculated
 * runs the backtrack algorithm and then returns the result to
the
 * server
 */
private HashMap calculateResults() {
    int maxProfit = profitVectors[numObjects - 1][capacity];

    int[] bag = null;
    try {
        bag =
            HuBackTrackImpl.backtrack(
                capacity,
                weights.length,
                backtrackVectors,

```

```

        weights);

    } catch (SolverException e) {
        System.err.println(
            "Failed to compute the knapsack solution: " +
            e.getMessage());
        System.exit(-1);
    }

//store the result in data structure and return it to
server
HashMap result = new HashMap();
result.put("result", new Integer(maxProfit));
result.put("resultSet", bag);
return result;
}

/*
 * send currentProfits to the neighbor to our right
 */
private void sendProfits(final int startIndex, final int
endIndex) {

    if (rightNeighbor == null) {
        return;
    }

    Thread t = null; // the thread to ask the server
    t = new Thread("sendProfitsThread") {
        public void run() {
            try {
                HuntPLVectorObject vo = new
                HuntPLVectorObject();
                vo.setProfitVectors(profitVectors);
                vo.setHistoryVectors(backtrackVectors);

                rightNeighbor.receiveMessage(vo,
                startIndex, endIndex);
                vo = null;
            } catch (MarshalException e) {
                log(
                    "error writing to socket at: "
                    + startIndex
                    + "\n "
                    + e.getMessage());
                // if rightNeighbor failed to recieve
                //profits, let it go
                // unless we are sending for first time

                //last time
                if (startIndex == 0
                    || endIndex == numObjects - 1) {
                    sendProfits(startIndex, endIndex);
                }
            } catch (RemoteException e) {

```

```

        System.err.println(
            "Failed to send profits at "
                + startIndex
                + " "
                + e.getMessage());
        System.exit(0);
    }
}
};
t.start();
}

/*
 * create a thread to begin calculation
 * so that the receiveMessage function can unlock
 */
private void startCalculation(final int startIndex) {

    // define the thread that starts the calculation
    Thread t = null;
    t = new Thread("startCalcThread") {
        public void run() {
            try {
                calculateObjectsInLevel(startIndex);
            } catch (SolverException e) {
                System.err.println(
                    "Failed to start Calculation: " +
                    e.getMessage());
                System.exit(0);
            }
        }
    };

    t.start();

    return;
}

/*
 * notify Server of completed results
 * @param result
 */
private void notifyServerWithResults(HashMap result)
    throws SolverException {
    try {
        server.notifyComplete(result);
    } catch (RemoteException e) {
        throw new SolverException(
            "reached result but failed to send to server "
                + e.getMessage());
    }
}

/*
 * establish the connection with the neighbor

```

```

    * and set the instance variable
    */
private void connectToNeighbor(String solverName) throws
SolverException {
    if (solverName == null) {
        rightNeighbor = null;
        return;
    }
    try {
        log("Connecting to neighbor " + solverName + "...");
        rightNeighbor = (KnapsackSolverHuntPL)
Naming.lookup(solverName);
        log("Connection successful");
    } catch (NotBoundException e) {
        throw new SolverException(
            "Could not connect to " + solverName
            + " " + e.getMessage());
    } catch (java.io.IOException e) {
        throw new SolverException(
            "I/O error or bad host name with "
            + solverName
            + " "
            + e.getMessage());
    }
}

}

/*
 * setup connection with the server
 * and let that server know this server is open to requests
 */
private KnapsackSolverServer registerWithServer(
String serverhost,
String lookupname)
throws SolverException {

KnapsackSolverServer server = null;
try {
    System.out.print("Connecting to "
        + serverhost + "... ");

    Registry r = LocateRegistry.getRegistry(serverhost);
    server =
        (KnapsackSolverServer) r.lookup(
            KnapsackSolverServer.SOLVERLOOKUPNAME);

    System.out.println(
        "Connected to server: "
        + KnapsackSolverServer.SOLVERLOOKUPNAME);
} catch (NotBoundException e) {
    throw new SolverException("Server not registered ");
} catch (java.io.IOException e) {
    throw new SolverException(
        "I/O error or bad host name with "
        + KnapsackSolverServer.SOLVERLOOKUPNAME

```

```

        + " "
        + e.getMessage());
    }
    try {
        server.registerSolver(lookupname);
    } catch (RemoteException e) {
        throw new SolverException(
            "Failed to register self with server." +
            e.getMessage());
    } catch (ServerException e) {
        throw new SolverException(
            "Failed to add self to server " +
            e.getMessage());
    }
    return server;
}

private void log(String message) {
    System.out.println(message);
}

//////////////////////////////////// main and usage
////////////////////////////////////
/**
 * starts and registers itself as the servers
 * contacts KnapsackSolverServer to register self
 * as being ready to take requests
 */
public static void main(String[] args) {
    if (args.length != 3) {
        usage();
    }

    // user provides a name at command line
    String name = args[0];
    String solverhostname = args[1];
    String serverhostname = args[2];

    SOLVERNAME = "rmi://"
        + solverhostname + "/" + SOLVERNAME + name;

    // set security manager
    System.setSecurityManager(new RMISecurityManager());

    // register self as server
    SolverHuntPLImpl self = null;
    try {
        self = new SolverHuntPLImpl();
        Naming.rebind(SOLVERNAME, self);
        System.out.println(
            "Solver: " + SOLVERNAME
            + " ready and waiting...");
    } catch (java.io.IOException e) {
        System.out.println("Could not register Solver: "

```

```

        + e.getMessage());
        System.exit(1);
    }

    // lookup KnapsackSolverServer
    try {
        self.server = self.registerWithServer(
            serverhostname, SOLVERNAME);
        System.out.println("Registered with Server ");
    } catch (SolverException e) {
        System.out.println(
            "Error registering with SolverServer: " +
            e.getMessage());
        System.exit(1);
    }
}

/**
 * usage - if no filename supplied
 */
public static void usage() {
    System.out.println(
        "USAGE: SolverHuntPLImpl [bindName] [solverhost]
        [serverhost]");
    System.out.println(
        "\t bindname - "
        +" unique name for client to register with ");
    System.out.println(
        "\t solverhost "
        +" - TCP/IP network name of the host where the RMI
        +" registry for this solver is running");
    System.out.println(
        "\t serverhost - TCP/IP network name of the host "
        +" where the RMI registry for the remote server "
        +" is running");
    System.exit(0);
}
}

```

HuntPLVectorObject.java

```
package solver;

import java.io.Serializable;
import java.rmi.Remote;

/**
 *
 * PURPOSE: simple javabean to hold the vectors that must be send in
 HUNTPL
 */
public class HuntPLVectorObject extends Object implements Remote,
Serializable {

    private int [][] profitVectors;
    private int [][] historyVectors;

    /**
     * constructor
     */
    public HuntPLVectorObject() {
    }

    /**
     * @return history vectors
     */
    public int[][] getHistoryVectors() {
        return historyVectors;
    }

    /**
     * @return
     */
    public int[][] getProfitVectors() {
        return profitVectors;
    }

    /**
     * @param is
     */
    synchronized public void setHistoryVectors(int[][] is) {
        historyVectors = is;
    }

    /**
     * @param is
     */
    synchronized public void setProfitVectors(int[][] is) {
        profitVectors = is;
    }
}
```

HuBackTrackImpl.java

```
/*
 * HuBackTrackImpl.java
 */
package solver;

/**
 * PURPOSE: defines the classic Hu backtracking algorithm
 * during the computation in the forward phase,
 * set history[k][j] = index of the last type of object used in
 f[k][j]
 * therefore if history[k][j] = r, then the profitvector[r] >= 1
 * that is , the rth object was used to obtain the maximum profit
 */
public class HuBackTrackImpl {

    static public int[] backtrack(int maxValue, int numObjects,
int[][]history, int weights[]) throws SolverException {
        if ( history == null || weights == null) {
            throw new SolverException(
                "Backtrack requires u and weights to be defined");
        }
        else if ( history.length < numObjects ) {
            throw new SolverException(
                " u is less than the number of objects " +
                history.length);
        }

        int z[] = new int[numObjects];
        int k;

        for (k=numObjects-1;k>=0;k--) {
            z[k] = 0;
            if ( ( history[k][maxValue] == k) && (maxValue>0) ) {
                z[k]=1;
                maxValue=maxValue-weights[k];
            }
        }

        return z;
    }
}
```

SolverException.java

```
/*
 * SolverException.java
 */
package solver;

/*****
 * Identify SolverExceptions
 * *****/

public class SolverException extends Exception {
    public SolverException() {
        super();
    }
    public SolverException(String msg) {
        super(msg);
    }
}
```


Appendix 4 Contents Package

Java

ContentsFile.java

```
/*
 * ContentsFile.java
 */

package contents;

/**
 * interface for reading of config file
 * and generation of the contents test files
 *
 */

import java.util.HashMap;
import java.io.FileNotFoundException;
import java.io.IOException;

public interface ContentsFile {

    /**
     * @param filename
     * @return HashMap of defined values for configuration
     * @throws FileNotFoundException
     * @throws IOException
     * reads in the given file and returns the configuration values
     */
    public HashMap readConfigFile(String filename) throws
FileNotFoundException, IOException;

    /**
     * @param filename
     * @throws IOException
     * opens the output file for writing
     */
    public void openOutput(String filename) throws IOException;

    /**
     * @throws IOException
     * closes the output file
     */
}
```

```
public void closeOutput() throws IOException;

/**
 * @param content
 * @throws IOException
 * writes a single content weight value pair to output
 */
public void writeContent(int [] content) throws IOException;

/**
 * @param capacity
 * @throws IOException
 * writes the knapsack capacity to output file
 */
public void writeCapacity(int capacity) throws IOException;
}
```

ContentsFileTxtImpl.java

```
/*
 * ContentsFileTxtImpl.java
 */
package contents;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.util.HashMap;
import java.util.StringTokenizer;

/*
 * parse a text config file,
 * output a text contents file
 */
public class ContentsFileTxtImpl implements ContentsFile {

    private HashMap obj = null;
    private PrintWriter output = null;

    public ContentsFileTxtImpl() {
        this.obj = new HashMap();
    }

    /*
     * @see contents.ContentsFile#readFile(java.lang.String)
     */
    public HashMap readConfigFile(String filename)
        throws FileNotFoundException, IOException {

        FileReader fr = new FileReader( new File(filename) );
        BufferedReader in = new BufferedReader( fr );
        String line = null;
        while ( (line = in.readLine()) != null ) {
            parseConfigString(line);
        }
        fr.close();
        return obj;
    }

    /* (non-Javadoc)
     * @see contents.ContentsFile#openOutput(java.lang.String)
     */
    public void openOutput(String filename) throws IOException {
        output = new PrintWriter( new FileWriter(filename) );
    }
}
```

```

/* (non-Javadoc)
 * @see contents.ContentsFile#closeOutput()
 */
public void closeOutput() throws IOException {
    output.close();
}

/* (non-Javadoc)
 * @see contents.ContentsFile#writeContent(int[])
 */
public void writeContent(int[] content) throws IOException {
    output.println(content[0] + "," +content[1]);
}

/**
 * @param capacity
 * @throws IOException
 * writes the knapsack capacity to output file
 */
public void writeCapacity(int capacity) throws IOException {
    output.println(capacity);
}

/***** private methods *****/

/*
 * each line in the text file is something=else
 * put all something as keys in Hash, and else as value
 */
private void parseConfigString(String line) throws IOException {

    StringTokenizer st = new StringTokenizer(line, "=");
    if (st.countTokens() != 2 ) {
        throw new IOException(
            "File line formatted incorrectly: Need name and
            value.");
    }
    String name = new String( st.nextToken() );
    String value = new String( st.nextToken() );
    obj.put(name, value);
}
}

```

ContentsGenerator.java

```
/*
 * ContentsGenerator.java
 */

package contents;

/**
 * Generates a file of contents
 * each object in the contents has a weight and a value
 * the weight and values may be related in a number of ways
 * This class allows this relation to be specified
 */

import contents.UserInputException;
import java.util.Random;
import java.util.HashMap;
import java.io.IOException;

public class ContentsGenerator {

    private int maxRange;          //max value for weight and value

    private int numContents;      //knapsack object argument
    private String contentType;  // relationship of weight-value
    private Random randNum;      // random number generator

    // default values for type "similar"
    private static int similarWeight = 100000;
    private static int similarValue = 1000;

    /*
     * set class variables dependant on input HashMap
     */
    public ContentsGenerator(HashMap config) {
        maxRange = (new Integer((String)
            config.get("maxRange"))).intValue();
        numContents =
            (new Integer((String)
                config.get("numContents"))).intValue();
        contentType = (String) config.get("contentType");

        // instantiate once, not every time makeRandom is called
        randNum = new Random();
    }

    /**
     * main method
     * so user can call this from command line
     */
    public static void main(String[] args) {
        HashMap config = null;          // values for configuration
    }
}
```

```

        if (args.length != 2) {
            usage("Not enough arguments.");
        }

        //open file handler
        ContentsFile cfile = new ContentsFileTxtImpl();

        // open and parse the input config file
        try {
            config = cfile.readConfigFile(args[0]);
        } catch (IOException e) {
            usage(e.getMessage());
        }
        try {
            checkConfigArguments(config);
        } catch (UserInputException ex) {
            usage(ex.getMessage() + config.toString());
        }

        // create instance of this class and run generateContents
        ContentsGenerator gc = new ContentsGenerator(config);
        try {
            gc.generateContents(args[0]);

        } catch (ContentsException e) {
            usage(
                "Fatal error generating content. "
                + e.getMessage()
                + " \n"
                + " Perhaps your range is not high
                enough.");
        } catch (IOException e) {
            System.out.println(e.getMessage());
            System.exit(0);
        }
    }

    /**
     * @param outputfile
     * @throws IOException
     * @throws UserInputException
     * @throws ContentsException
     * opens the output file, writes to it, closes output file
     */
    public void generateContents(String outputfile)
        throws IOException, ContentsException {

        //open file handler
        ContentsFile cfile = new ContentsFileTxtImpl();
        // open output file
        cfile.openOutput(outputfile);

        // write knapsack capacity
        cfile.writeCapacity(makeRandom(0, maxRange * numContents));
        int i;

```

```

        for (i = 0; i < numContents; i++) {
            cfile.writeContent(createContent());
        }
        cfile.closeOutput();
    }

    /***** private methods *****/

    /* confirm that configuration input Hash
     * contains the required keys
     */
    static private void checkConfigArguments(HashMap config)
        throws UserInputException {
        if (!config.containsKey("maxRange")) {
            throw new UserInputException(
                "Configuration must contain a maxRange");
        }
        if (!config.containsKey("numContents")) {
            throw new UserInputException(
                "Configuration must contain a numContents");
        }
        if (!config.containsKey("contentType")) {
            throw new UserInputException(
                "Configuration must contain a contentType");
        }
    }

    /*
     * instruct user how to run this class appropriately
     */
    static private void usage(String message) {
        System.out.println(message);
        System.out.println(
            "USAGE: ContentsGenerator
            <config file> <output file>");
        System.out.println(
            "config file must define maxRange, numContents and
            contentType");
        System.out.println("contentType must be one of the
            following: ");
        System.out.println(
            "\t uncorrelated \n"
            + "\t weaklyCorrelated \n"
            + "\t stronglyCorrelated \n"
            + "\t inverseStronglyCorrelated \n"
            + "\t almostStronglyCorrelated \n"
            + "\t SubsetSum \n"
            + "\t similarWeights");
        System.exit(0);
    }
}

```

```

/*
 * knowing the contentType requested,
 * create a single content pair of that type
 * returns int [2] representing the ordered pair (weight, value)
 */
private int[] createContent() throws ContentsException {
    int[] content = new int[2]; // array of weight, value
    if (contentType.equals("uncorrelated")) {
        content = uncorrelated(content);
    } else if (contentType.equals("weaklyCorrelated")) {
        content = weaklyCorrelated(content);
    } else if (contentType.equals("stronglyCorrelated")) {
        content = stronglyCorrelated(content);
    } else if (contentType.equals("inverseStronglyCorrelated"))
    {
        content = inverseStronglyCorrelated(content);
    } else if (contentType.equals("almostStronglyCorrelated"))
    {
        content = almostStronglyCorrelated(content);
    } else if (contentType.equals("subsetSum")) {
        content = subsetSum(content);
    } else if (contentType.equals("similarWeights")) {
        content = similarWeights(content);
    } else {
        throw new ContentsException(
            "did not find contentType " + contentType);
    }
    return content;
}

/**
 * uncorrelated
 * weight and profit are chosen randomly
 * in [1,R]
 */
private int[] uncorrelated(int[] content) throws
ContentsException
{
    content[0] = makeRandom(1, maxRange);
    content[1] = makeRandom(1, maxRange);
    return content;
}

/**
 * weaklyCorrelated
 * weights are chosen randomly in [1,R]
 * values are in [w - R/10, w+R/10]
 */
private int[] weaklyCorrelated(int[] content) throws
ContentsException {
    int weight = makeRandom(1, maxRange);
    int profit = makeRandom(weight - maxRange / 10,
        weight + maxRange / 10);
    content[0] = weight;
    content[1] = profit;
}

```

```

        return content;
    }

    /**
     * stronglyCorrelated
     * weights are distributed in [1,R]
     * values = w + R/10
     */
    private int[] stronglyCorrelated(int[] content) throws
    ContentsException {
        int weight = makeRandom(1, maxRange);
        int profit = weight + maxRange / 10;
        content[0] = weight;
        content[1] = profit;
        return content;
    }

    /**
     * inverseStronglyCorrelated
     * values are distributed in [1,R]
     * weights = v + R/10
     */
    private int[] inverseStronglyCorrelated(int[] content)
    throws ContentsException {
        int profit = makeRandom(1, maxRange);
        int weight = profit + maxRange / 10;
        content[0] = weight;
        content[1] = profit;
        return content;
    }

    /**
     * almostStronglyCorrelated
     * weights are distributed in [1,R]
     * values are in [w + R/10 - R/500, w+R/10+R/500]
     */
    private int[] almostStronglyCorrelated(int[] content)
    throws ContentsException {
        int topDiff = 500;
        if (maxRange < topDiff) {
            topDiff = maxRange * numContents;
        }
        int weight = makeRandom(1, maxRange);
        int profit =
            makeRandom(
                weight + maxRange / 10 - maxRange / topDiff,
                weight + maxRange / 10 + maxRange / topDiff);
        content[0] = weight;
        content[1] = profit;
        return content;
    }

    /**
     * subsetSum

```

```

    * weights are distributed in [1,R]
    * value = weight
*/
private int[] subsetSum(int[] content) throws ContentsException {
    int weight = makeRandom(1, maxRange);
    int value = weight;
    content[0] = weight;
    content[1] = value;
    return content;
}

/**
 * similarWeights
 * weights are distributed in [100 000,100 100]
 * value are in [1,1000]
*/
private int[] similarWeights(int[] content) throws
ContentsException {
    int maxSim = (int) ((similarWeight * .001) +
similarWeight);
    int weight = makeRandom(similarWeight, maxSim);
    int profit = makeRandom(1, similarValue);
    content[0] = weight;
    content[1] = profit;
    return content;
}

/*
 * returns a random number in the given range
 */
private int makeRandom(int min, int max) throws ContentsException
{
    if (min > max) {
        throw new ContentsException(
            "min " + min + " must be less than max " +
max);
    }
    if (min == max) {
        return min;
    }

    // get the range, casting to long to avoid overflow
problems
    long range = (long) max - (long) min + 1;
    // compute a fraction of the range, 0 <= frac < range
    long fraction = (long) (range * randNum.nextDouble());
    return (int) (fraction + min);
}
}

```

ContentsGeneratorWrapperImpl.java

```
/*
 * ContentsGeneratorWrapper.java
 */
package contents;

import java.util.HashMap;
import java.io.IOException;

/*
 * Wrapper around ContentsGenerator:
 * runs all possible types of Contents
 * outputting fixed file names for each type
 */
public class ContentsGeneratorWrapper {

    /*
     * takes user arguments from command line
     * and generate all types
     */
    public static void main(String[] args) {
        if (args.length != 2) {
            usage("Not enough arguments");
        }
        try {
            generateAllContents(args[0], args[1]);
        } catch (Exception e) {
            e.printStackTrace();
            usage(e.getMessage());
        }
    }

    /*
     * instruct user how to run this class appropriately
     */
    static private void usage(String message) {
        System.out.println(message);
        System.out.println(
            "USAGE: ContentsGeneratorWrapper <numContents> "
            + "<maxRange>");
        System.out.println(
            "Will create all types of contents files "
            + "with numContents contents, and all with a "
            + "maximum of maxRange. "
            + "Outputs files named <contentType.txt>, "
            + "for example, \"uncorrelated.txt\". ");
        System.exit(0);
    }
}
```

```

/**
 * @param numContents
 * @param maxRange
 * @throws ContentsException
 * @throws IOException
 * Calls ContentsGenerator for each type of content
 * all with the same numContents and maxRange
 */
static void generateAllContents(String numContents, String
maxRange)
    throws ContentsException, IOException {

    // all values for generateContent
    HashMap config = new HashMap();
    int i = 0; //counter
        String[] contentTypes = // all contentTypes
{
    "uncorrelated",
    "stronglyCorrelated",
    "almostStronglyCorrelated",
    "inverseStronglyCorrelated",
    "weaklyCorrelated",
    "subsetSum",
    "similarWeights" };

    // set up the configuration
    config.put("maxRange", maxRange);
    config.put("numContents", numContents);

    // call generate contents for each type
    for (i = 0; i < contentTypes.length; i++) {
        String type = contentTypes[i];
        String filename = type + ".txt";
        config.put("contentType", type);
        ContentsGenerator gc = new ContentsGenerator(config);
        gc.generateContents(filename);
        System.out.println(
            "Generated "
                + filename
                + " with range "
                + maxRange
                + " objects "
                + numContents);
    }
}
}

```

ContentsExceptionI.java

```
/*
 * ContentsException.java
 */

package contents;

/*****
 * Identify Content Exceptions
 *****/

public class ContentsException extends Exception {
    public ContentsException() {
        super();
    }
    public ContentsException(String msg) {
        super(msg);
    }
}
```

UserInputException.java

```
/*
 * ContentsException.java
 */
package contents;

/*****
 *Identify user input Exceptions
 *****/

public class UserInputException extends Exception {

    public UserInputException() {
        super();
    }

    public UserInputException(String msg) {
        super(msg);
    }

}
```