VBnB- Visual Branch and Bound

Murat Gorguner
Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

May 1997

[illegible]

Professor Philip N. Klein
Advisor

Table of Contents

List of Figures

1. Introduction

1.1 NP-Complete and Combinatorial Optimization Problems:

A large set of problems in computer science fall into the NP computational class. Unfortunately, with the existing algorithms, solution of any NP-hard problem takes time exponential in the problem size in the worst case. This means that, even the small instances of these problems can take unreasonable amount of time, which makes these problems intractable. Let's consider an example of an NP-hard problem. Let's say a school has 50 courses and five days in which to schedule examinations. An optimal solution would be one where no student has to take two examinations on the same day. One way to solve this problem is by searching for the optimal solution through all the possible schedules. In the worst case we have to check for all the schedules, and there are $O(5^n)$ possible schedules. Although this problem seems quite simple, if we calculate the worst case it is not. So number of possible schedules equals:

5^{50} = 8.881784e+34 and in the worst case we have to check for all of these possibilities. Assuming that we have a computer which can check a million schedules each second, the time needed for 50 courses would be about

200,000,000,000,000,000,000 years!

Combinatorial optimization problems are optimization problems that require solutions to be integer valued. Although not all combinatorial optimization problems are NP-hard, many are. Combinatorial optimization problems have many practical uses. In many fields of industry and technology solving complex optimization problems is the key to increase productivity and product quality.

The objective of a combinatorial optimization method is to search for the best solution out of a very large number of possible solutions. Normally, the best solution means the solution with the lowest costs or the solution with the highest profit. Simply checking or enumerating all possible solutions of real-world problems would take thousands of years even using the fastest supercomputers of the world as shown in the above example.

Given so many problems are combinatorial optimization problems, there has been much research on the approximation algorithms and special techniques to speed the search for the optimal exact solution. There are two common approaches. Historically, the first method developed was based on *cutting planes* (adding constraints to force integrality). In the last twenty years or so, however, the most effective technique has been based on dividing the problem into a number of smaller problems in a method called *Branch-and-Bound*. Both of these approaches involve solving a series of linear programs. before we explain what a linear program is and what the relationship is between these problems and linear programming, lets look at more examples of combinatorial optimization problems.

Graph Coloring:

Given a graph G with vertices V and edges E, find the smallest number of colors needed to color G, in such a way that no adjacent vertices are assigned the same color.

Application: Exam Scheduling

Knapsack Problem:

The traditional story is that, a burglar has a knapsack with some finite capacity. There are a number of items in the house that he is trying to rob. Each item has a specific size and value. So the problem is which items should he take to make the maximum profit.

Applications: Economic Planning and Loading Problems, Cryptology

Traveling Salesman Problem:

A salesman needs to visit some finite number of cities and return to his home city. So the problem is in which order should he visit the cities so that he would have travelled the least distance.

1.2 Relationship to Linear Programming:

In the general linear-programming problem, we wish to optimize a linear function subject to a set of linear inequalities. So what is the relationship between an integer program and a linear program.

Given an integer program(IP)

Maximize Cx

subject to Ax = b

where x>=0 and integer,

there is an associated linear program called the linear relaxation(LR) formed by dropping the integrality restrictions. Thus we get:

Maximize Cx

subject to Ax = b

where x>=0.

Since linear relaxation is less constrained than the integer program. We can say that integer program is a subset of the linear relaxation: As seen in figure 1

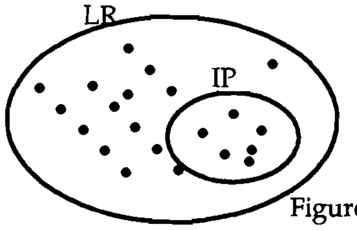


Figure 1

The following are immediate:

- ❍ The optimal objective value for LR is greater than to or equal to that of IP.
- ❍ If there is no solution for the LR, then there is no solution for the IP.
- ❍ If LR is optimized by integer variables, then that solution is feasible and optimal for IP.
- ❍ If the objective function coefficients are integer then the optimal objective for (IP) is less than or equal to the round down of the optimal objective for LR.

So solving for LR does give us some information. Especially, we can use the bound on the optimal value, which is a crucial information in the branch-and-bound algorithm and, if we are lucky it may give us the optimal solution to IP.

2.0 Branch-and-Bound

2.1 What is Branch-and-Bound?

Branch-and-Bound makes use of a tree formulation of a given problem. Each leaf of this tree can be viewed as a possible solution to the problem and each internal node represents a partial solution. For example, in the case of the knapsack problem, for each object in the problem we need to decide whether to take it or not. We can assign one to an object if we decide to take it and zero if we decide not to take it. If we did not decide whether to take an object or not, then we assign it a question mark. A partial solution to this problem would be one, where we didn't decide about all the items, which means our decision has some question marks. A complete solution to this problem would be one, where for every object a 1 or 0 is assigned. A leaf in the tree would represent a solution, and an internal node would represent a partial solution. B&B algorithm tries to speed up the process of searching this enumeration tree by using a bounding strategy. B&B maintains a single best lower bound and computes an upper bound at each node of the enumeration tree. If the upper bound that is computed exceeds the lower bound, then we know that the subtree that is rooted at this node cannot contain the optimal solution, therefore we do not need to further explore this subtree. This is known as pruning the tree. In practice, for most problems, large portions of the tree can be pruned this way, thus making the search space much smaller and reducing the calculation time significantly.

Lets say we have a knapsack problem, where the capacity of the knapsack is 14 and we have 4 items in a house. These items (x_1, x_2, x_3, x_4) weigh 5, 7, 4, and 3 respectively and their values are 8,11, 6, and 4 respectively. So which items should we choose to take in order to maximize the profit? An integer program for this problem would be:

Maximize: $8x_1 + 11x_2 + 6x_3 + 4x_4$

Subject to: $5x_1 + 7x_2 + 4x_3 + 3x_4 <= 14$

where $x_j = \{0,1\}$ for $j = 1, 2, 3, 4$

we can get the linear relaxation by relaxing the condition on x so we get:

Maximize: $8x_1 + 11x_2 + 6x_3 + 4x_4$

Subject to: $5x_1 + 7x_2 + 4x_3 + 3x_4 <= 14$

where $0 >= x_j >= 1$ for $j = 1, 2, 3, 4$

Solving the linear relaxation we get $x_1 = 1$ $x_2 = 1$, $x_3 = 0.5$, and $x_4 = 0$ with a total value of 22. Therefore we know that no integer solution will have value more than 22. But since we don't have an integer solution yet, we branch on one of the variables say $x1$.

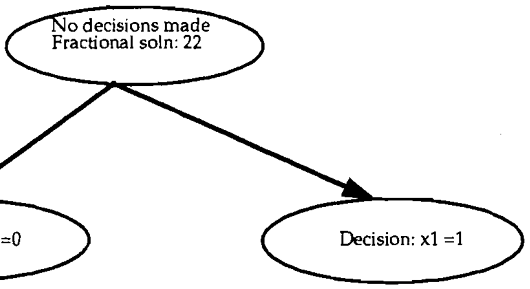


Figure 2

At this point, for the left child we solve LR:

Maximize: $8x0 + 11x_2 + 6x_3 + 4x_4$

Subject to: $5x0+7x_2 + 4x_3 + 3x_4 <= 14$

where $0>= x_j >= 1$ for j= 2, 3, 4

When we solve this we get $x_1= 0$, $x_2 =1$, $x_3 = 1$, and $x_4 = 1$ with a total value of 21. And since this is an integer solution, we can eliminate searching the subtree below this node, because we know that this is the best solution that we can obtain from any of the leafs of this subtree. So we update our lower bound to 21 and continue with the remaining active nodes (in this case the right child of the root).

For the right child we have to solve:

Maximize: $8x1 + 11x_2 + 6x_3 + 4x_4$

Subject to: $5x1+7x_2 + 4x_3 + 3x_4 <= 14$

where $0>= x_j >= 1$ for j= 2, 3, 4

Solving the linear relaxation we get $x_1= 1$, $x_2 =1$, $x_3 = 0.5$, and $x_4 = 0$ with a total value of 22. So the upper bound of this node is 22, which is greater than the current lower bound so we have to branch on this node, and we get:

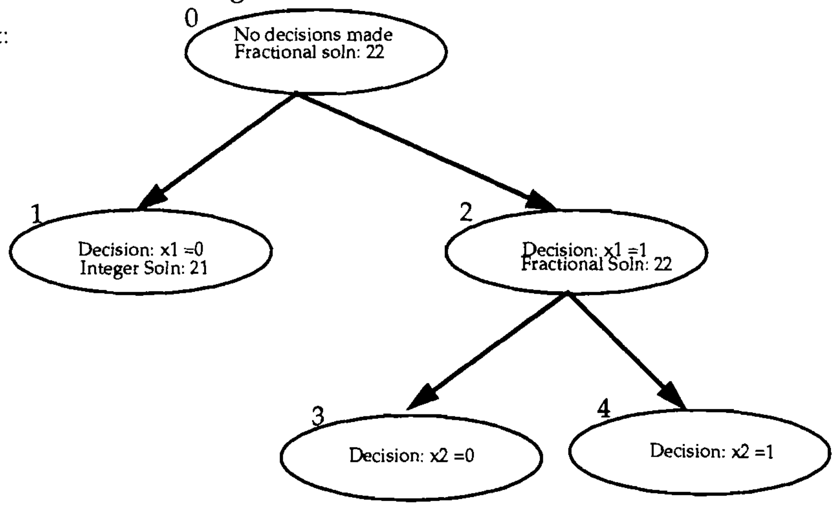


Figure 3

At node 3, the decisions we already made are: $x_1= 1$, $x_2 =0$. So solving the LR of this problem we get $x_3 = 1$, and $x_4 = 1$, with an upper bound of 18. Since the upper bound for this node is less than the current lower bound we don't have to go further down this part of the tree and we can safely prune the subtree rooted at this node.

At node 4, the decisions we already made are: $x_1= 1$, $x_2 =1$. So solving the LR of this problem we get $x_3= 0.5$, $x_4 =0$, with an upper bound of 22 so we have to further branch on this node. so we get:

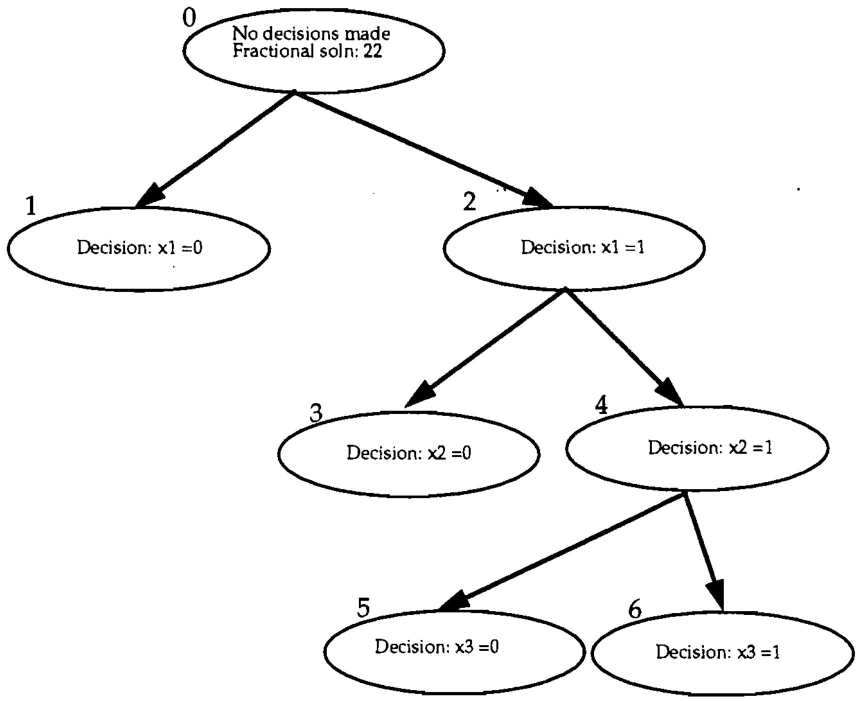


Figure 4

At node 5, the decisions we already made are: $x_1= 1$, $x_2 =1$, and $x_3= 0$. So solving the LR of this problem we get $x_4 = 0$, with an upper bound of 19. Since the upper bound for this node is less than the current lower bound we don't have to go further down this part of the tree and we can safely prune the subtree rooted at this node.

At node 6, the decisions we already made are: $x_1= 1$, $x_2 =1$, $x_3= 1$. So solving the LR of this problem we find that it is unfeasible because if $x_1= 1$, $x_2 =1$, $x_3= 1$, the total weight of these objects is 16 which already exceeds our limit of 14. So we can safely prune the subtree rooted at this node.
And since we don't have any other active nodes we are done and the solution is:

$x_1= 0, x_2 =1, x_3= 1, x_4= 1$

2.2 We can summarize the B&B Algorithm as follows:

1. For each active node solve the linear relaxation problem. If the solution is integer, then we are done with that node. Otherwise if the upper bound on this node is greater then the current lower bound then

create two new subproblems by branching on the next variable.

2. A subproblem is defined as not active when any of the following occurs:

    a. You used the problem to branch on.

    b. All the variables in the solution are integer,

    c. The sub-problem is infeasible.

3. Choose an active sub-problem and branch on the fractional variable. Repeat until there are no active sub-problems.

3.0 Implementation

3.1 Implementation of Branch-and-Bound:

Implementation of B&B on a single machine is pretty straight forward. Before you start the B&B, using a heuristic you calculate a solution. Then you start your algorithm at the root of the tree and calculate the upper bound on the LR and then you branch (which means create the children of this node and place them into a priority queue) if the upper bound is greater than the current best solution. Then you take the next node in the priority queue and do the same thing again, until there are no more jobs in the priority queue.

Here is the pseudocode:

```
Find a solution using a greedy algorithm
Place the root node in the Priority Queue
while (Priority Queue is not empty )
  Get the Current Description from the Priority Queue
  Solve the LR of the Current Description
  If (an integer solution is found)
      If (better than the current best solution)
          Update the best solution
  else If (upper bound > current best solution)
        Create the children of this node and place
        them in the Priority Queue
```

Figure 5

When implementing B&B on a distributed network of computers, further things have to be taken into consideration. The purpose of a distributed implementation is to exploit parallelism. There are several characteristics that we want our system to have:

O Scalable: This program should run on a single computer as well as running on hundreds of computers connected by a network.

O Fault Tolerance: If a computer crashes in the network we don't want to lose all the computation done so far, so recovering from crashes is important.

O Modularity: Although we have implemented the knapsack problem, we want to be able to easily modify the program to solve other applications of B&B.

In order to achieve these goals B&B is implemented in two separate parts. One is the master process which keeps track of all the computations, current best solution and other information needed. It also starts the second part of B&B which is the slave part. Slave processes receive all the information they

need from the master process and after solving the Linear Relaxation (LR) form of the current description they send their result back to the master process.
Master process makes use of the following tables.

Job Table

Vir Job ID	Real Job ID	Description	Children	Parent	Status	Priority

Figure 6

where:

Vir Job ID	: The Job ID number assigned by Central Process when creating the job
Real Job ID	: The Job ID assigned by pvm when the job is spawned.
Description	: Description of the current node (i.e. which choices are already made)
Children	: Number of children that this process has.
Parent	: Virtual Job ID of the parent process
Status	: Status of this job (in the priority queue, done, waiting for children, etc...)
Priority	: Priority of this job

A job is placed in the Job Table as soon as it is created.

When we are spawning a job to a worker we place it in the Running Jobs Table which keeps the following information:

Running Jobs Table

Vir Job ID	Start Time	Worker

Figure 7

where:

Vir Job ID	: The Job ID number assigned by Central Process when creating the job
Start Time	: The time that the job is spawned.
Worker	: Name of the worker that this process is running on.

Using the information from the running jobs table we can calculate the amount of time taken to complete a job. Another use of running jobs table is for fault tolerance. If a process dies in pvm, pvm reports this to the parent process. So if we get a pvm_exit signal for a slave process while the job is still in the running jobs table, this means that the process died so using the virtual job id of the process we can create a new process and place it in the priority queue using the information from the Job Table.

Priority Queue

Vir Job ID	Priority

Figure 8

where:

Vir Job ID	: The Job ID number assigned by Central Process when creating the job
Priority	: Priority of this job

When a process is created it is placed in the priority queue, and when that process is spawned on to a machine than it is removed from the priority queue. In this implementation priority is based on the depth of the node. Root node has the priority 0, its children have the priority 1, etc. The Central Process assigns jobs starting at the job with the highest priority.

Here is the pseudocode of the central process.

```
Determine the number of machines to use

Determine a solution using some method

Create the root and place it in the Priority Queue

forever do {

   while (there is a worker(W) available &&
          Priority Queue is not empty)
       Place the job in the running jobs table
       Start the job on machine W

   if there is no job in the running jobs table then we
   are done


   Wait for a reply from any of the children
   Process the reply
       •update the Job Table and Running Jobs Table
       • Create the children and place them in the
            priority queue if branch is requested
       • Update the current best solution if a better
           best solution is returned from the process
   If there are more replies from the children process
   them else continue;
}
```

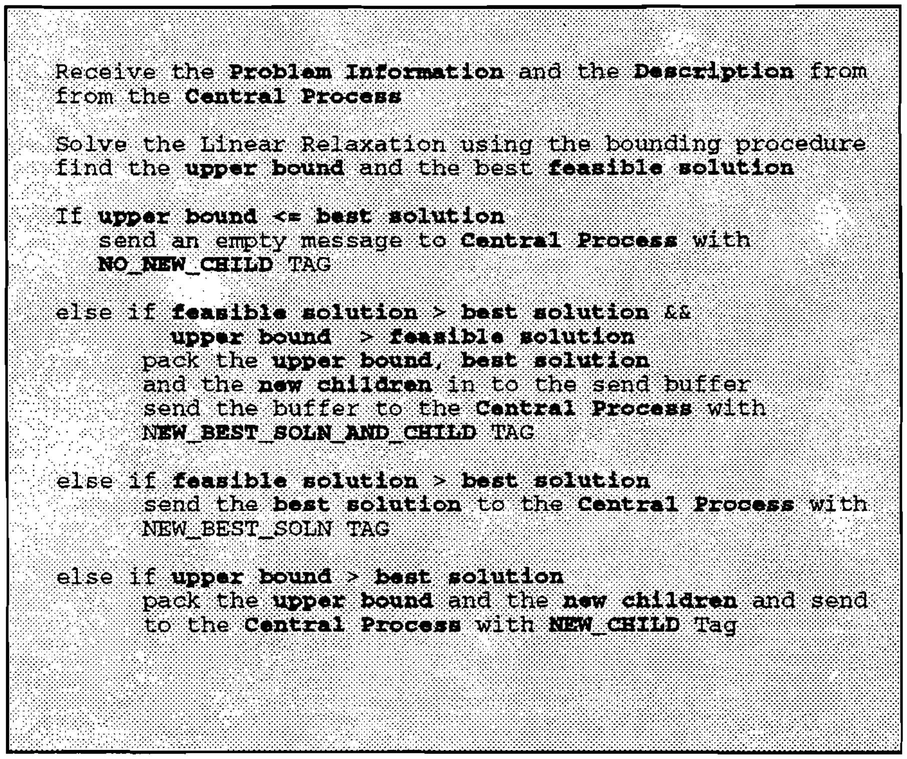
Figure 9

Central process keeps track of all the available workers, all the jobs, best solution so far, etc. However, the linear relaxation is solved in slave processes. Slave processes are started by the master process. When a slave process is first started it waits for the central process (master program) to send it all the information needed about the problem and the current description. Once it receives all the necessary information it solves the linear relaxation of the current description. In the case of knapsack problem the information that is send to the slave process include.

- Number of objects in the room
- Weight of each object
- Value of each object

- ❍ Capacity of the knapsack
- ❍ Value of the current best solution
- ❍ Current Description (0, 1, or -1 for each object)

Once the slave process receives all this information, it calls its bounding procedure which calculates the solution for a linear relaxation of the problem which in the case of knapsack problems just means that you can take a fraction of an object. It then returns this result to the central process and if the upper bound found is greater than the current best solution than the information needed to create the children of the current node is calculated and send to the central process.
Pseudocode for the slave process:



```
Receive the Problem Information and the Description from
from the Central Process

Solve the Linear Relaxation using the bounding procedure
find the upper bound and the best feasible solution

If upper bound <= best solution
   send an empty message to Central Process with
   NO_NEW_CHILD TAG

else if feasible solution > best solution &&
        upper bound  > feasible solution
      pack the upper bound, best solution
      and the new children in to the send buffer
      send the buffer to the Central Process with
      NEW_BEST_SOLN_AND_CHILD TAG

else if feasible solution > best solution
      send the best solution to the Central Process with
      NEW_BEST_SOLN TAG

else if upper bound > best solution
      pack the upper bound and the new children and send
      to the Central Process with NEW_CHILD Tag
```

Figure 10

We choose the knapsack problem because of its easy implementation. However, we designed our program in such a way that it would be possible to solve other applications by making some changes in the code. First, lets talk about the parts that need to be changed in the Central Process.

First of all, we need to change the `Read Problem` Procedure, because depending on the application the format of the problem file will most probably change. We also need to change the `Description Class`, because `description` is also application-specific. Other than these two changes, the only other thing that needs to be changed is the part where we create the children of a node by using the information received from another process.

Most of the changes that need to be done are in the slave process since solving the LR of a problem is application-specific. The first thing that needs to be changed is the structure which holds the problem, so that the new problem description can be received. The main part that needs modification is the `Bounding Procedure` which is the place that we solve LR of the current description. This procedure need to be completely changed because it is application dependent. `Create Children` procedure also needs modification so that it will work with the new application.

4 Visual Tool

4.1 Visualization of Branch-and-Bound Algorithm:

It is often difficult to determine a parallel program's behavior due to the many concurrently executing threads of control. However, an evaluation of whether a program is running as expected or to see the efficiency of the implemented algorithm can be very useful to the programmer. One useful approach to this problem is the use of computer graphics to visually display the program's execution.

We have implemented a visualization tool that can be used specifically with the B&B algorithm. Using this tool one can have a better understanding of their algorithms, and can compare different approaches to solving a problem. There are several issues one has to think about when implementing such a tool. First of all we want it to be almost real-time. The graph that you see on the screen should be a fairly up do date representation of the computation that is going on. We also want to have some control on the representation of our graph. So we should be able to change the preferences or change the way the graph is displayed while the program is actually running. We also want to display as much information as we can about the computation.

Displaying the whole enumeration tree, would not be a very useful approach since our screen will get so crowded that, it would be impossible to get any information out of that graph. So we decided to only display the nodes which are already processed or at least waiting to be processed. Besides that we gave the user the control over which nodes should be displayed. So when we first start the only thing that is displayed is the root node. Then once it is processed and it creates its children we display the children of this parent. We have several different modes about adding and removing the nodes from the enumeration tree. When the program is in its initial state, the only thing that is displayed is the root node. Once the root node is processed, and if it has children, you can click on the root node and the children of the root will be added to the display. Now the children are processed, if you want to see the children of the left node you can click on the left node, if you want to see the children of the right node you can click on the right node. You can also remove parts of the tree by clicking on any of the internal nodes. Clicking on an internal node removes all the children of the selected node.

A node at any possible time has to be in one of the following states.

❍ Processing (A slave process is solving the LR of this node) : green

❍ Waiting (The node is in the priority queue waiting to be processed) : blue

❍ Already being processed: We already calculated the upper bound for this process:

If we have already processed this node then there can be several results:

❍Upper Bound for this node is less than the current lower bound: don't do anything : red

❍The solution to the LR is integer so no need to create children : black

❍Upper Bound is greater than the current lower bound then branch

❍All the nodes that are in the subtree rooted at this node is done : pink

○ There are still some nodes in the subtree which are not processed : yellow
See Figure 11 for a summary of the colors.

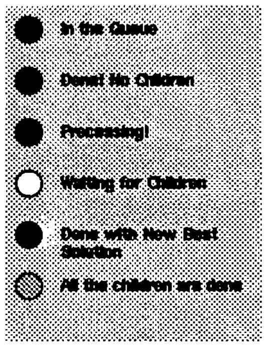


Figure 11

In addition to the enumeration tree, we have a display of bar graphs right below each leaf. As explained above a leaf node can either be a regular node, or it could be representing the subtree that is rooted at that point. The bars represent the time spend on that particular node or if it is representing a subtree then the bar which corresponds to that node represents the total time that is being spend on that subtree. If the subtree is still active then the bar increases as the time spend on the subtree increases. This information can be very helpful, because by comparing different bars we can figure out where our program is spending most of its time. Other than the main display we also have an other window which displays the current best upper bound and current best lower bound at every half second.

We also have several different modes. If we are in "Auto Add" mode then the program automatically adds the children to the display as they are created. We can also choose "Auto Remove" and this removes everything below a node once all the calculations on that subtree is completed. We also give a choice on where to place the leaf node. You can either place all the leaf nodes to the bottom which might make the association of the leaf node with the bar that is representing it easier or if you don't choose this then only the nodes that are currently being processed are put to the bottom of the display.

Other than this we have an options window where you can change the size of the nodes and the bars, while the program is running. And finally we have a "Show Solutions" button which simply displays the solution once all the calculations are completed.
Figure 12 show the example given in the Section 2.1 solved using the visual display:

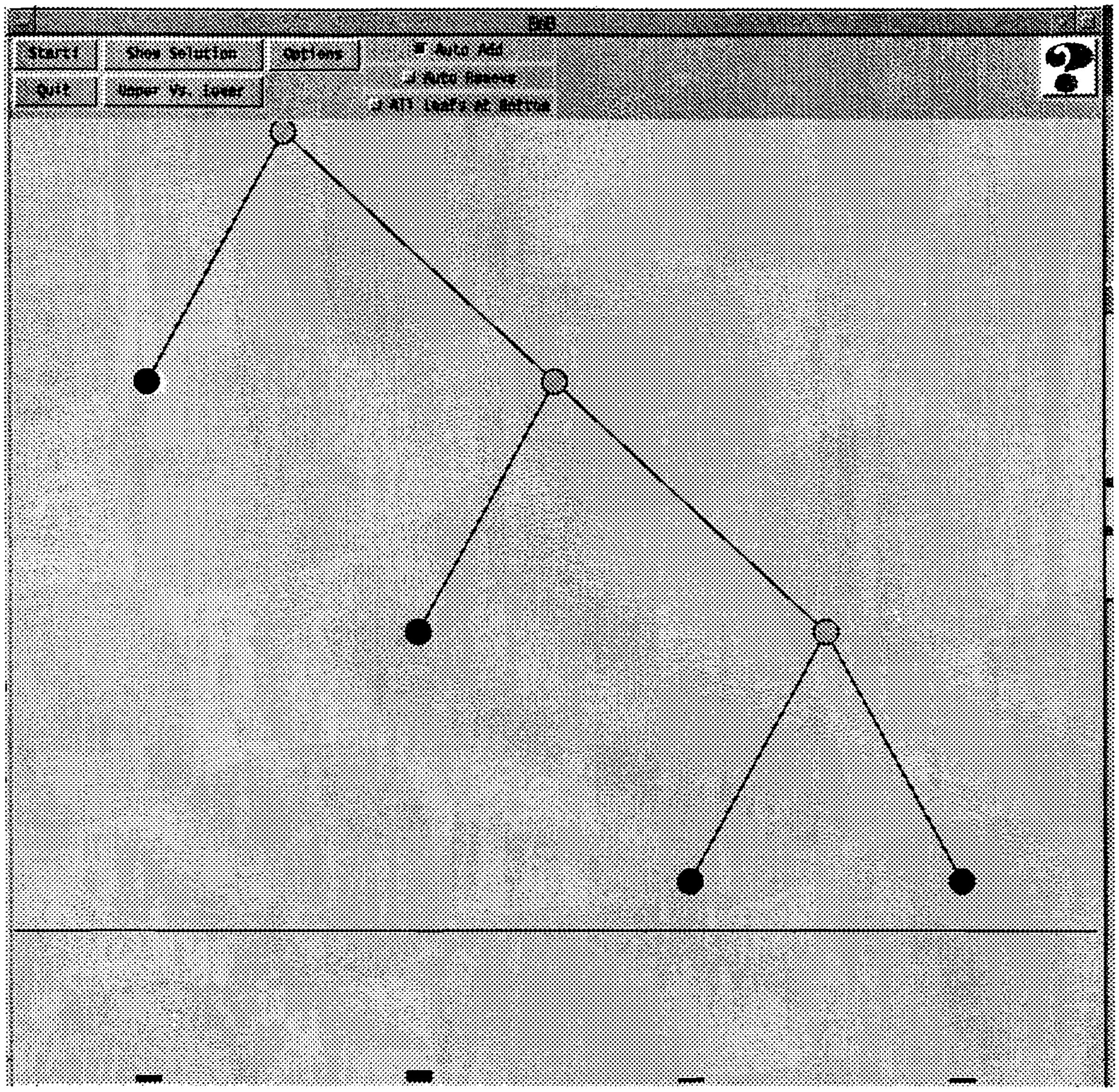


Figure 12

The next figure (Figure 13) shows the program while it is solving a problem which has 45 objects using 50 computers.

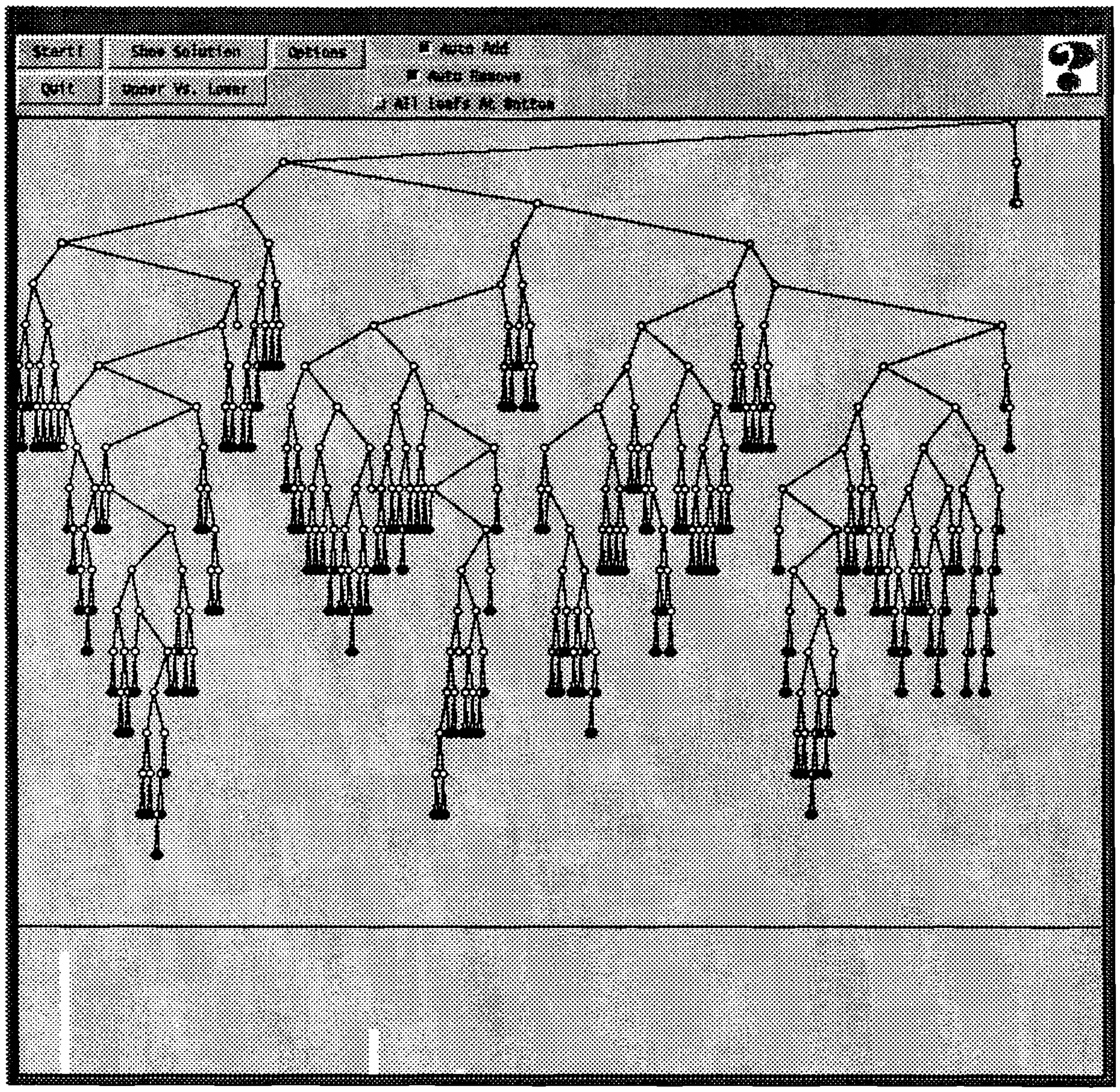

Start!
Show Solution
Options
Quit
Upper Vs. Lower
Auto Add
Auto Remove
All Leafs At Bottom

Figure 13

5.0 Conclusion:

In this paper, we have presented a distributed implementation of *Branch-and-Bound* Algorithm which is used to solve the knapsack problem, and a description and implementation of a visual tool which can help a programmer to see the process of B&B. There are many ways one can change the B&B algorithm, for instance we choose to select the priority of our nodes depending on their depth. However, we could have made the priority depend on the upper bound of the parent, and there is no theoretical answer which can claim that one is better than the other for all applications of B&B. In our implementation, we are sending only one job at a time to a worker, however if we send a lot of jobs to a machine at one time we can reduce the communication overhead. However, reducing the communication overhead might not mean that the overall program will be more efficient and productive.

By visually seeing the performance of different approaches, one can understand these computations better and can make further improvements in the algorithms.