Optimisation Coursework

A Solution To The Knapsack Problem

Introduction

Imagine that you are going to go on a long hike. You know that this will require you to take several different kinds of equipment, but you do not want to overburden yourself. To this end, you decide that you wish to take the most valuable set of items, without carrying more than a certain amount of weight.

This can be formulated as an integer programming problem. Suppose that we have a set, $\mathcal{W}$, of n items of weight w_i; and a set of values, $\mathcal{P}$, of n items of value p_i. Writing M for the maximum weight of items that can be carried and letting x_i be the decision variables the problem can be written as:

$$\begin{aligned} \max & \sum_{i=1}^{n} p_i x_i \\ \text{s.t.} & \sum_{i=1}^{n} w_i x_i \leq \mathrm{M} \qquad \forall i \in \{1, \ldots, n\} \\ & x_i \in \{0, 1\} \qquad \forall i \in \{1, \ldots, n\} \end{aligned}$$

This problem, however, is NP-hard and can be, when n is large, very difficult to solve efficiently. [3] In order to solve the knapsack problem several methods have been proposed including those using Dynamic Programming technique, Branch and Bound methods and Approximation Algorithms. [6] It is a Branch and Bound method that is applied here to solve the problem. In particular, the algorithm applied is a combination of the Horowitz-Sahni and Martello-Toth Algorithms.

Methodology

In order to construct an effective Branch and Bound algorithm three areas must be considered. These are how the branching will be carried out, what value the bound will take on a branch, and how the algorithm will choose to search for new nodes to branch on.

Before any branches are carried out, however, an algorithm is used to decide where to start. This algorithm is then also used to create the optimal solutions for any future branches.

Whilst alternative methods do exist, a greedy algorithm is used to generate the initial value. In doing so, the problem is relaxed from an Integer Programme to a Linear Programme. This is done by creating a set of fractional values calculated as a ratio, r_i of the profit and weight values.

$$r_i = \frac{p_i}{w_i}$$

The greedy algorithm then orders these ratios from highest to lowest and fills the knapsack with each item in turn until there is insufficient weight to add in the next item. It can be seen that this is a viable approach to the problem because of the fact that the items that provide the best profit for their weight are added first. This

should then allow for some of the best value items to be inserted into the knapsack giving an excellent start. To further reinforce this argument, it can be shown that for some small problems, for instance with $n = 4$, the greedy algorithm can take the correct items without any need for a branch and bound application.

It should be noted that the bounds generated are based on solutions to the fractional, or continuous, knapsack problem. This is the relaxation of the condition that x_i must be a binary variable and instead $x_i \in [0, 1]$ is permitted. The branch and bound method is then applied to find a solution to the relaxed linear program that is also a solution to the integer program. This is then the optimal solution to the knapsack problem.

Branching Method

In a branch and bound method the algorithm must decide, once an initial solution has been found, what item to branch on next. By branching, it is meant that the algorithm selects an item, then creates two sets of data; one with the item: the first branch; and one without the item: the second branch. The greedy algorithm is then run on these two branches and different bound values are compared. The algorithm then selects the greatest upper bound before moving on and branching on this next item. How a bound is selected will be discussed below.

When choosing which item to branch on, two options exist. The first option is to branch on the critical item. This is the item that the greedy algorithm cannot fit into the knapsack. What happens is the branch then tries to fit this critical item into the knapsack instead of adding in the last item that was placed into the sack. If the bound is improved, then this is a better choice and this item is added to the solution instead of the last item to be added.

An alternative to this approach is referred to as branching on the best item. This method branches on each item in turn, starting from the one with the greatest profit to weight ratio. So, at the first node, the first item is removed and bounds are again calculated. If the bound is improved by removing the item, again it is removed from further solutions.

At first it can seem counter intuitive to remove the item with the greatest value of r_i, however what this branching approach does is that it verifies which items can be left in the optimal solution. This means that after the first branch it can be concluded if the optimal solution contains the first item, and then after the second branch it can be concluded whether or not the optimal solution contains the second item and so on. As a result of this the algorithm is often able to search deeper, and faster, than when branching on the critical item. [4]

For the algorithm presented the branching approach selected is to branch on the best item. The reason for this selection is twofold: firstly, the optimal solution was found faster than with branching on the critical item. This is in agreement with literature that attributes this speed down to the ability to search deeper faster, thus requiring fewer branches. [4] [2] Secondly, a search technique can be added to the branching method that allows for the inconclusive branches to both be explored with relatively little penalty to the speed of the computation. As a result of this, the decision to fill the knapsack by best first seems to be a reasonable one.

Bound Method

Once a node has been branched on, it is necessary to compare the two differing branches to find the next best value to branch on. Furthermore, it is also necessary to be able to recognise when the fractional knapsack problem has generated a solution to the integer problem.

Presented here is an algorithm that uses two different approaches to bounding a result. In the Horowitz-Sahni case the bound is used in order to prevent excessive forwards steps. Indeed, if the LP-relaxation to the knapsack problem for the items yet to be filled will not allow for a solution that exceeds the current best solution to be found then a branch is discarded and the algorithm tracks back to a previous node. This algorithm therefore uses the bound to establish if it is necessary to search a node, and in doing so the Horowitz-Sahni approach ensures that is branches on as few items as it can. Again, this allows for the algorithm to move deeper into the tree faster, thus allowing for larger knapsack instances to be solved.

On the other hand the secondary algorithm stage uses the bound to explicitly decide which branch to move on. In order to do this, upper bounds on the solution are generated for each branch by computing the bound for the solution with either the node item in the knapsack or the item out of it. A rule is then followed which states that the branch with the greatest upper bound should be selected as the next node to move to. In the algorithm presented here a stopping rule is also used that if the bound is equal to the profit of the solution then the algorithm terminates. The drawbacks of this approach are discussed below.

Two different bounds are presented here: the Dantzig Bound and the Martello-Toth Bound. It will be shown that both bounds are tight and that the Martello-Toth Bound is always tighter than the Dantzig bound, thus allowing an optimal solution to be reached faster. The precise difference in performance will be presented in the results section.

Definition 1 (Dantzig's Bound)**.** Let U_0 denote Dantzig's Bound:

$$U_0 = \sum_{i=1}^{m-1} p_i + \hat{c}\lfloor r_m \rfloor \tag{1}$$

Where $m-1$ denotes the number of items in the knapsack, p_i denotes the profit values, r_m is the ratio of the critical item, and $\hat{c}$ denotes the residual weight capacity after the greedy algorithm has filled the knapsack.

It can be shown that this bound is derived from a solution to the fractional knapsack problem, and the tightness of this bound can also be shown.

Proposition 1.

a) U_0 is an optimal solution to the relaxed program.

b) U_0 is a two-approximation of the solution.

Proof. a) This proof is due to [5]. To see this assume the items have been ordered as per the greedy algorithm and first note that any optimal solution, x^* to the continuous knapsack problem must by definition fill the knapsack. This will see the solution contain $m-1$ whole items and then a fractional amount of the m^{th} value and thus $\sum_{i=1}^{m-1} w_j x_j + \frac{\hat{c}}{w_m} = M$. We show x^*_m takes the final term in this expression, and is not zero, by contradiction.

Suppose for some $j < m-1$ there exists a value in the optimal solution of $x^*_k < 1$. As a result of this there must also exist $x*_k > x_i$ for $k > m-1$. For some $\epsilon > 0$ it would then be possible to increase the value of x^*_k by ϵ, and decrease the value of x^*_q by ϵ, to give a value for the objective function that is greater than the maximal solution. Hence a contradiction.

On the other hand if $x^*_k > 0$ for some $k > m-1$ then it is clear that for some $j < m-1$ we have $x^*_j < 1$. Again this allows for these values to be adjusted to create a new solution that is returns a value greater than the optimal solution. This contradiction thus shows that the optimal solution is given as:

$$\sum_{i=1}^{m-1} p_j x^*_j + \hat{c} r_i$$

As a consequence of the fact that $\hat{c}r_i$ is not an integer it is therefore proven that an upper bound on the integer solution to the knapsack problem is given by Dantzig's Bound.
b) To see that the bound gives a two-approximation it is necessary to show that, if z is the optimal value of the solution, then $U_0 < 2z$. To see this note that both $\sum_{i=1}^{m-1} p_i$ and p_m are feasible solutions to the integer problem and so we see:

$$U_0 < \sum_{i=1}^{m-1} p_i + p_m < z + z = 2z$$

as required.

□

The Dantzig Bound is not the best possible bound for a branch and bound approach however. Indeed it will be shown below that the Dantzig Bound is itself an upper bound for the Martello-Toth bound, and so it is possible to implement this tighter bound to reduce the number of branches the algorithm needs to explore. This then allows for the optimal solution to be located faster than when the Dantzig Bound is applied.

Definition 2 (Martello-Toth Bound)**.** Let U_a and U_b denote the upper and lower bounds for the Martello-Toth bound, U_1:

$$U_a = \sum_{i=1}^{m-1} p_i + \hat{c}\lfloor r_{m+1} \rfloor$$

$$U_b = \sum_{i=1}^{m-1} p_i + \lfloor p_m - (w_m - \hat{c})r_{m-1} \rfloor$$

$$U_1 = \max(U_1, U_2)$$

The reason that this bound is preferred is because of the following result:

Proposition 2. *For all knapsack problems $U_1 \leq U_0$.*

Proof. Again, the following proof is due to [5].
Due to the fact that, for any $i \in \{1, \ldots, n\}$ we have $r_i \geq r_{i+1}$ it is clear that $U_a \leq U_0$.
To prove that $U_b \leq U_0$ first observe that $r_m \leq r_m - 1$ and, as the m^{th} item does not fit into the knapsack, $w_m > \hat{c}$. Hence, as both brackets are less than zero, we have the bound:

$$(\hat{c} - w_m)(r_m - r_{m-1}) > 0$$

and this can be rearranged to show that:

$$\hat{c}\, r_m > p_m - (w_m - \hat{c})r_{m-1}$$

and so $U_b < U_0$.

Hence we are able to conclude that $U_1 \leq U_0$ for any knapsack problem.

□

A critic of this bound may argue that the need to compute not one, but two, bounds to calculate the overall result slows down the algorithm. This is true, however the reduction in speed is very slight, and it is argued in [5] and [6] that this loss is offset by the time saved through having to compute fewer branches. As a consequence of this it is concluded that the Martello-Toth Bound offers the most efficient branching method of the two.

In the algorithm presented, however, it is not always the case that the Martello-Toth bound is used. This is because the first component of the algorithm follows the Horowitz-Sahni method and so utilises the Dantzig Bound. When the second stage of the algorithm kicks in, however, the Martello-Toth Bound is used. In the results section the increase in speed this offers is discussed.

Search Method

One final point to be discussed when looking into the methodology behind the presented algorithm is the method by which the algorithm searches the nodes. Broadly speaking, two forms of search exist: the breadth first search, and the depth first search.

In a breadth first search, the bound would be calculated for every node on each branch. So, for example, even if the first item was found to be inside the optimal solution the search would still examine the bound given from the second item after removal of the first. The advantage of this approach is that it is thorough, and uses less memory: there is no need to keep track of the best values, as the algorithm searches through every branch that is created. A disadvantage to this, and it is a disadvantage that can prove to be significant [6], is the time taken for the algorithm to return the optimal solution. That said, it is argued in [6] that a breadth first search can be an acceptable approach for problems of size $n \leq 100$.

On the other hand a depth first search moves onto branches for items further away from the parent node far faster. It does this by selecting the branch with the best bound, accepting the conclusion, and then branching on the next item in the list. The advantage of this approach is that, in combination with a tight upper bound such as the Martello-Toth bound, the algorithm can quickly discard branches not worth exploring and hone in on an optimal solution. Cho et al note in [1] that the depth first search is far more effective than the breadth first search, and this is the conclusion also reached in [6]. It is for this reason that a depth first search is used in the branch and bound method presented here.

The Rationale Behind Combining Two Methods

The Horowitz-Sahni method is a practical tool for solving small to medium size datasets, however is not expected to efficiently solve harder instances. [5] Indeed, as Martello notes, the Horowitz-Sahni algorithm is instead a useful starting point for several of the more advanced algorithms. It is this reasoning that led to the decision, when faced with the intractable pisinger2 and pisinger4 datasets, to build on the existing algorithm.

In order to do this, the Martello-Toth algorithm was constructed. It should be noted, however, that this is not an exact construction as the backtracking steps are performed less frequently and, instead, the search method is biased towards moving deeper into the tree. The reason for this speed is down to the fact that the algorithm will move further into the tree before backtracking than the Horowitz-Sahni method. In practice, this means that it is possible for the algorithm to search much deeper than before in the same amount of time.

On the other hand, a criticism of this approach is that the stopping rule implemented; specifically the decision to terminate if the profit and bound match, implying an integer solution has been found; is that if an integer solution exists that is smaller than the true optimum, and that lies on the branches formed by accepting item 1, then the algorithm will return an incorrect result. This occurs here with the 4096 instance.

The reason for the inclusion of this method, however, is that the Horowitz-Sahni algorithm does not return solutions to the pisinger2 and pisinger4 datasets in a realistic timeframe. Indeed, the algorithm failed to return an optimal solution after two hours on each set. The fact that the secondary algorithm is able to return a solution, however, explains why this method has been included in the final result. A comparison of the two methods features in the results section, along with the final timings for each dataset.

How does the Algorithm Work?

The first stage of the algorithm is to order the items using the greedy technique. Once this has been performed, the Horowitz-Sahni method begins.

This part of the algorithm fills the knapsack item by item until the knapsack cannot be filled anymore. This is then either stored as the best possible solution so far, or ignored as it is worse than this current optimal value. The algorithm then backtracks to the most recently filled item and removes it, before continuing to move forward and fill the knapsack until it is full. Once the algorithm can no longer move backwards, it terminates. If at any point during the filling the upper bound indicates that the knapsack value will not exceed the current best solution, the algorithm will also backtrack to the most recently filled item and remove it. This speeds up computation by reducing the number of branches needed to find the optimal solution.

In practice, this method works well until the harder datasets are reached. Then, as already noted, the algorithm takes a long time to terminate. If, after 300 backtrack iterations, the algorithm has not terminated the current best value is stored and step two begins. It should be noted that the iteration limit has been set through experimentation, and that this could need to be increased after further testing. In practice the knapsacks were still solved successfully for as little as 150 iterations, though, so the limit offers some room for manoeuvre.

The second stage of the algorithm is based upon the Martello-Toth method and as such moves through the tree and, on each branch, computes a new upper bound based on the removal of an item. If the upper bound is greater than the existing upper bound, the algorithm chooses to remove the item and continues down the tree. Once a solution is found that matches the bound, the algorithm terminates. If the algorithm reaches the bottom of the tree, it backtracks to the node where the best profit offered so far was seen and then repeats the branching process. This then allows for the algorithm to move forward even if no better bound has been seen. In practice this has not needed to happen more than twice before an optimal solution was found, however it should be said that in some knapsack instances the algorithm could be stuck backtracking for far longer than the Horowitz-Sahni method without finding an optimal solution. Further testing will be needed to understand when this is the case.

This method, then, allows for a rapid descent down the branches and for the algorithm to return integer solutions in a far faster time frame than the Horowitz-Sahni algorithm. The drawback of this method, of course, is that if an integer solution exists that exceeds the upper bound computed so far, but that is not optimal, then the algorithm will return an incorrect value. This is why the algorithm, alone, is not a suitable alternative to the Horowitz-Sahni method as it cannot guarantee a solution that is correct.

Once both components of the algorithm have run, their two respective optimal solutions are compared. The best one is then returned as the solution to the knapsack instance.

Further Methods Implemented

Whilst the implementation of a stronger bound, and the addition of a breadth searching method to explore alternative branches, both enhanced the performance of the code there was still difficulty finding the optimal solution to the Pisinger0 dataset. The reason for the algorithm failing was because of the fact that the greedy algorithm was not able to order the items effectively. This is due to the fact they had different profits and weights and yet identical ratios. Further, the small size of the dataset ($n = 20$) is believed to have confounded the algorithm.

In order to overcome this issue the algorithm first groups the items into the different ratios possible, and then the algorithm sorts the data in these groups from highest to lowest profit. So, for example, the first item listed is the item with the best profit to weight ratio with the highest overall profit. This method of sorting the items then allows for the correct solution to be calculated. This is because the algorithm is filling the items with the best ratio, but also the items which offer the greatest benefit to the objective function, first.

Two further points are worth noting. Firstly, the algorithm only performs this sorting on datasets where there is at least two ratios that are equal. This is to avoid an unnecessary reduction in computation speed. The second is that there is no claim here that this two stage sorting is suitable for all problems. Indeed an intractable problem could be created that could defeat the algorithm. None are presented here, though.

Results

The performance of the algorithms presented here can be measured based on their ability to compute an optimal solution both accurately and quickly. Here it is presented that the Horowitz-Sahni method and the modified Martello-Toth method both fail to solve every instance of the knapsack provided. The combination of the two, however, is able to achieve this.

Algorithm Performance

Instance	Horowitz-Sahni	Modified Martello-Toth	Combined Algorithm
4	34	34	34
8	63	63	63
10	1590	1590	1590
20	2088	2088	2088
30	3736	3736	3736
100	22565	22565	22565
1536	993172	993172	993172
4096	2812405	2812196	2812405
Pisinger0	1428	1428	1428
Pisinger2	DNF	277046	277046
Pisinger3	138115	138116	138116
Pisinger4	DNF	464754	464754

Table 1: Comparison of Optimal Solutions By Algorithm

As seen from the table, no solution to the pisinger4 dataset was found. Indeed, the combined algorithm terminates at a value two less than the true optimum. Otherwise, it is clear that the combination of the two algorithms allows for the widest set of solutions to be found, thus indicating it to be the best method of the three.

It is also possible to compare the run time taken by the three different algorithms.

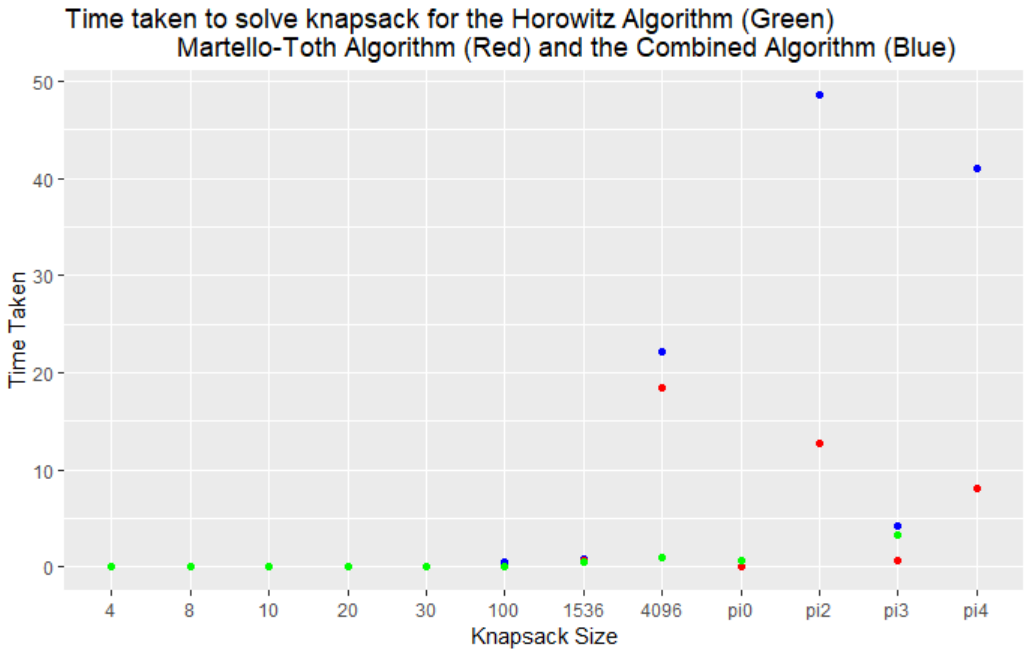

Figure 1: Comparison of Algorithm Performance by Time

As can be seen, the combined algorithm is considerably slower in the harder instances than both the individual algorithms. This is used to conclude that, whilst the algorithm is more accurate than the components it contains, it is not as efficient as it could be. Indeed the Martello-Toth algorithm is faster, but not more accurate than the combined algorithm when implemented here.

It is also possible to demonstrate a comparison in the efficiency of the bounds available to the Martello-Toth aspect of the algorithm. Specifically, how much faster is the Martello-Toth bound than the Dantzig bound when

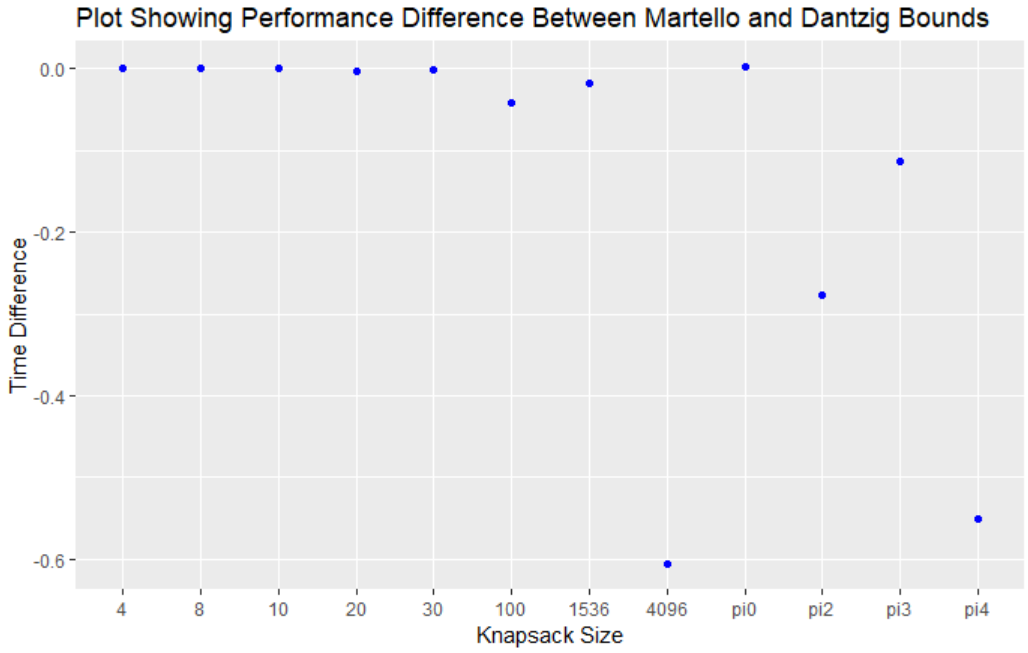

Figure 2: Comparison of Algorithm Performance by Bound

implemented in an algorithm? As can be seen in the plot below, the answer is that the Martello-Toth bound is faster in all cases, however this is a negligible difference in all but the largest cases presented here. As this benefit comes with minimal extra work, however, it is concluded that this benefit is worthwhile. As such, any future algorithms would be designed to use a bound at least as tight as the Martello-Toth bound.

Further Work

Perhaps the main source of further work is to rectify the Modified algorithm so that it is able to solve all instances presented. In particular it would be excellent to see a solution to the pisinger4 dataset. Beyond this, several other thoughts could be considered.

The first could be to try implementing a branching strategy that removes not one but two items. Initial attempts to implement this suggest that this does not add any value to the Horowitz-Sahni algorithm, but may add speed substantially to the modified Martello-Toth algorithm. As a matter of fact the 100 and 1536 item instances both ran around 40% faster with this addition. It does not, however, increase the accuracy of the algorithm and the 4096 instance was still solved incorrectly.

One reason for this increase in speed is that the algorithm is able to eliminate items that do not add value more quickly. These items could be useful for the final solution, though, so it is likely that this will be too coarse a comb for a realistic branch and bound approach as an instance could easily be formulated that would defeat it by pairing the items in the optimal solution together.

Another source of further work could be to implement some of the other methods proposed in [5]. These could then be used to either further enhance the combined algorithm presented, or to provide a more efficient solution to the instances given here. Finally, it should be said that these algorithms presented are by no means perfect. Indeed many harder knapsacks exist in the literature and further work could be to find and solve these to allow further improvements to be made.

Appendix 1: The Algorithm

```
knapsack.solver <- function(file, B){

  #Also input for the bound type
  start <- Sys.time()                    #Start the timer

  if(is.character(file) == FALSE){
    return("File needs to be a character")
  }
  knp <- read.table(file)
  row <- nrow(knp)
  n <- knp[1,1]
  M <- knp[1,2]
  p <- knp[(2:row),2]
  w <- knp[(2:row),1]
  if(nrow(knp) - 1 == length(p) & nrow(knp) -1 == length(w)){  #Check data has been read in correctly
    sol <- list(profits = p, weights = w, bound = M, number_of_items = n)
    print("Data has been read in correctly")
  }else{
    return("Data has not been read in correctly")
  }

 #First write the horowitz algorithm in:

  horowitz <- function(p, w, M){

    x <- p/w  #Ratio Vector

    #First order the variables#

    sort.x <-  sort(x,decreasing = TRUE)   #Order ratios
    index.x <- order(x,decreasing = TRUE)  #Create ordered index of variables

    sort.p <- c()
    sort.w <- c()

    for(i in 1:length(p)){    #Recreate the profit and weight variables in the right order
      sort.p[i] <- p[index.x[i]]
    }

    for(i in 1:length(w)){
      sort.w[i] <- w[index.x[i]]
    }

    if(length(unique(sort.x) != length(sort.x))){ #Deals with the cases where ratios are equal
      sort.matrix <- cbind(sort.p,sort.w,sort.x,index.x)
      sort.matrix <- as.data.frame(sort.matrix)     #Allow for sorting on multiple cols
      sort.matrix <- with(sort.matrix, sort.matrix[order(sort.x, sort.p, decreasing = TRUE),])
      sort.p <- sort.matrix$sort.p      #Recall the vectors form the data frame
      sort.w <- sort.matrix$sort.w
      sort.x <- sort.matrix$sort.x
      index.x <- sort.matrix$index.x
```

```
}

profit <- 0
weight <- 0
items <- c()

#Initialise

z <- 0
z.hat <- 0
c.hat <- M    #residual
j <- 1        #index
n <- length(p)
y <- rep(0,n)       #store optimal items
y.hat <- y
s <- FALSE     #Used for stopping rule
revisit <- 1   #Used for stopping rule
iterations <- 0

#functions required

greedy <- function(p,w, c.hat, z.hat, M, j){    #greedy algorithm 1
  greedy.profit <- z.hat
  greedy.weight <- M - c.hat
  i <- j
  repeat{                          #Needed to get the if loop to repeat
    if(greedy.weight + w[i] <= M & i <= length(w)){  #If next item can fit in the knapsack
      greedy.profit <- greedy.profit + p[i]   #Add profit value, weight value and item index
      greedy.weight <- greedy.weight + w[i]
      i <- i+1
    }else{
      i <- i-1    #Go back to the right node
      break
    }
  }
  u <- sum(sort.p[j:i]) + floor((c.hat - sum(sort.w[j:i]))*(p[i]/w[i]))

  return(u)
}

forward <- function(sort.p, sort.w, c.hat, y.hat, z.hat, j){  #Forward step
  while(sort.w[j] <= c.hat & j <= length(sort.w)){
    c.hat <- c.hat - sort.w[j]
    z.hat <- z.hat + sort.p[j]
    y.hat[j] <- 1
    j <- j+1
  }
  if(j <= n){
    y.hat[j] <- 0
    j <- j+1
  }
  return(list(profit = z.hat, residual = c.hat, solution = y.hat, j = j))
}
```

```
backward <- function(p, w, c.hat, y.hat, z.hat, j){  #Backward step
  k <- max(which(y.hat[1:(j-1)] == 1))
  if(k == 1){
    s == TRUE
  }else{
    c.hat <- c.hat + w[k]
    z.hat <- z.hat - p[k]
    y.hat[k] <- 0
    j <- k+1
  }
  return(list(profit = z.hat, residual = c.hat, solution = y.hat, j = j))
}

#begin the algorithm#

while( s == FALSE & revisit <= 2){    #Stopping rule
  iterations <- iterations + 1
  repeat{
    if(j < n){
      u <- greedy(sort.p, sort.w, c.hat, z.hat, M, j)    #Perform greedy algorithm
      if(z.hat + u <= z){
        break
      }
      fwd <- forward(sort.p, sort.w, c.hat, y.hat, z.hat, j)
      z.hat <- fwd$profit    #Update values after forward step
      c.hat <- fwd$residual
      y.hat <- fwd$solution
      j <- fwd$j
    }else if(j == n){
      fwd <- forward(sort.p, sort.w, c.hat, y.hat, z.hat, j)
      z.hat <- fwd$profit     #Update values after forward step
      c.hat <- fwd$residual
      y.hat <- fwd$solution
      j <- fwd$j
    }else if(j > n){
      break
    }
  }

  #update the solution with new solution value
  if(z.hat > z){
    z <- z.hat
    for(l in 1:n){
      y[l] <- y.hat[l]
    }
  }


  j <- n
  if(y.hat[n] == 1){
    c.hat <- c.hat + sort.w[n]
    z.hat <- z.hat - sort.p[n]
    y.hat[n] <- 0
  }
```

```
    #backtrack step
    bwd <- backward(sort.p, sort.w, c.hat, y.hat, z.hat, j)
    z.hat <- bwd$profit     #Update values after forward step
    c.hat <- bwd$residual
    y.hat <- bwd$solution
    j <- bwd$j

    if(sum(y.hat) == 1){     #If we revisit zero more than once then we are done
      revisit <- revisit + 1
    }

    if(iterations > 300){
      break
    }
  }

  for(i in 1:n){
    if(y[i] == 1){
      items[i] <- index.x[i]
    }
  }
  items <- items[!is.na(items)]
  for(i in 1:length(items)){
    weight <- weight + w[items[i]]
  }

  solution <- list(Profit = z, Weight = weight, Items = items, Iterations = iterations )


}

#If the Horowitz fails then we utilise this additional algorithm

#Function needed for branch and bound to run#

branch.bound <- function(p,w,M, B){   #Algorithm for Branch and Bound takes same inputs

  x <- p/w  #Ratio Vector

  #First order the variables#

  sort.x <-  sort(x,decreasing = TRUE)   #Order ratios
  index.x <- order(x,decreasing = TRUE)  #Create ordered index of variables

  sort.p <- c()
  sort.w <- c()

  for(i in 1:length(p)){    #Recreate the profit and weight variables in the right order
    sort.p[i] <- p[index.x[i]]
  }

  for(i in 1:length(w)){
    sort.w[i] <- w[index.x[i]]
```

```
}

if(length(unique(sort.x) != length(sort.x))){ #Deals with the cases where ratios are equal
  sort.matrix <- cbind(sort.p,sort.w,sort.x,index.x)
  sort.matrix <- as.data.frame(sort.matrix)     #Allow for sorting on multiple cols
  sort.matrix <- with(sort.matrix, sort.matrix[order(sort.x, sort.p, decreasing = TRUE),])
  sort.p <- sort.matrix$sort.p      #Recall the vectors form the data frame
  sort.w <- sort.matrix$sort.w
  sort.x <- sort.matrix$sort.x
  index.x <- sort.matrix$index.x
}


#Now call the greedy algorithm

#Function for the greedy search#

type <- B
greedy.v2 <- function(p, w, x, index, M){  #x arg is the list of items, index is their no
  profit <- 0    #Store the total profit, weight and items in solution
  weight <- 0
  items <- c()   #This will be a list of items, with item 1 being sort.x[1]
  i <- 1

  repeat{                          #Needed to get the if loop to repeat
    if(weight + w[i] <= M & i <= length(w)){  #If next item can fit in the knapsack
      profit <- profit + p[i]   #Add profit value, weight value and item index
      weight <- weight + w[i]
      items <- c(items, index[i])
      i <- i+1
    }else if(weight + w[i] > M & i <= length(w)){
      break    #Once the sack cannot be filled anymore, stop the loop
    }else if(i > length(w)){
      i <- i-1    #Return to the end of the items
      break       #No items left to fill the knapsack
    }
  }

  #Now calculate the bounds#
  if(type == "martello"){
    residual <- M - weight    #Remaining capacity in knapsack
    U.1 <- profit + floor(residual*x[i+1])  #First Martello Bound
    U.2 <- profit + floor(p[i] - (w[i] - residual)*x[i-1])  #Second Martello Bound
    if(is.na(U.1) == TRUE){   #Deals with when i+1 is out of range
      U.1 <- profit + floor(residual*x[i])   #Uses Dantzig bound for a fall back
    }
    bound <- max(U.1, U.2)    #Select best upper bound
  }else if(type == "dantzig"){
    residual <- M - weight    #Dantzig Bound
    bound <- profit + floor(residual*x[i])
  }else{
    return("Bound must be either danzig or martello")
  }
```

```
  #Now return the result#
  result <- list(Profit = profit, Weight = weight, Items = items, Bound = bound)
  #return(result)
}

#Initial Branch#
limit <- M
initial <- greedy.v2(p = sort.p, w = sort.w,x = sort.x,
                     index = index.x,  M = limit)  #Create the first solution

profit <- initial$Profit   #Now set the variables to the values of the initial search
weight <- initial$Weight
items <- initial$Items
index <- index.x
bound <- initial$Bound
alt.profit <- 0           #Storage for if the algorithm does not terminate

#Start the search using the filling the knapsack method#

node <- 1           #Index for the jth item to be removed, starting from 1
branches <- 0       #Counter for number of branches
backtrack.count <- 0  #If we backtrack to a new solution we add one
iterations <- 0

#Branch and Bound Begins#

while(profit < bound & length(items) < length(w) & backtrack.count <= length(w)) {
  #When profit and bound match result is optimal; also need to know to break if no items left
  #Also need to break if no nodes left to check

  iterations <- iterations + 1
  branches <- branches + 1   #Add a new branching point


  if(branches > length(w) & alt.profit != 0){ #If we found a better profit but the bound was never met
    #but an alternate profit was also found
    sort.p <- alt.sort.p   #Update values
    sort.w <- alt.sort.w
    sort.x <- alt.sort.x
    index <- alt.index
    node <- node.alt        #Return to previous nodes and branches
    branches <- branches.alt + 1
    backtrack.count <- backtrack.count + 1  #Avoids being permanently stuck backtracking
  }else if(branches > length(w) & alt.profit == 0){  #Does not terminate but no alternate solution
    break
  }



  #Create item lists for the branches#

  in.p <- sort.p           #Vectors with item kept in
  in.w <- sort.w
  in.items <- sort.x
```

```
in.index <- index

out.p <- sort.p[-node]   #Vectors with the node item taken out
out.w <- sort.w[-node]
out.items <- sort.x[-node]
out.index <- index[-node]

#Perform greedy algorithm on the two different branches#

in.greedy <- greedy.v2(in.p, in.w, in.items, in.index, M=limit)
out.greedy <- greedy.v2(out.p, out.w, out.items, out.index, M=limit)

#Choose the branch with the greatest upper bound#

if(in.greedy$Bound <= out.greedy$Bound & out.greedy$Bound
   != out.greedy$Profit){   #Choose largest of bounds, so remove the node

  sort.p <- out.p    #Remove the node from the vectors
  sort.w <- out.w
  sort.x <- out.items
  index <- out.index

  profit <- out.greedy$Profit   #Overwrite with the new solution
  weight <- out.greedy$Weight
  items <- out.greedy$Items
  bound <- out.greedy$Bound

  node <- node     #Deletion of item will move onto the next node
}else if(in.greedy$Bound > out.greedy$Bound &
         out.greedy$Bound != out.greedy$Profit){     #Choose the best bound, so no items removed

  sort.p <- in.p     #No change to the vectors, as jth item will stay
  sort.w <- in.w
  sort.x <- in.items
  index <- in.index

  profit <- in.greedy$Profit     #Update the solution with the new value
  weight <- in.greedy$Weight
  items <- in.greedy$Items
  bound <- in.greedy$Bound

  node <- node+1    #Move to the next node
}else if(out.greedy$Bound == out.greedy$Profit){ #This is the optimal solution so terminate
  profit <- out.greedy$Profit   #Overwrite with the new solution
  weight <- out.greedy$Weight
  items <- out.greedy$Items
  bound <- out.greedy$Bound
}

if(out.greedy$Bound <  in.greedy$Bound &
   out.greedy$Profit > in.greedy$Profit){ #May need to search on a new branch
  alt.sort.p <- out.p     #Save the branch for a later date
  alt.sort.w <- out.w
  alt.sort.x <- out.items
```

```
      alt.index <- out.index

      if(alt.profit < out.greedy$Profit){    #If this gives a greater profit keep this aside

        alt.profit <- out.greedy$Profit     #Store alternate values
        alt.weight <- out.greedy$Weight
        alt.items <- out.greedy$Items
        alt.bound <- out.greedy$Bound

        node.alt <- node - 1    #Delete the same item and store for later
        branches.alt <- branches
        alt.solution <- list(Alternate_Profit = alt.profit, Alternate_Weight = alt.weight,
                             Alternate_Items = alt.items, Alternate_Bound = alt.bound)
      }
    }

  }

  #Solution Step#

  if(profit > bound){

    print(list(Profit = profit, Bound = bound))    #Print instance so we can debug
    return("Something has gone very wrong here...")


  }else if(length(items) == length(w)){   #For instances when all items are taken

    solution <- list(Profit = profit, Weight = weight, Items = items, Bound = bound,
                     Branches = "Greedy Solution took the lot")
    return(solution)

  }else if(branches >= length(w) | backtrack.count >= length(w)){

    solution <- list(Profit = profit, Weight = weight, Items = items, Bound = bound,
                     Branches = iterations)
    return(solution)

  }else if (profit == bound){


    solution <- list(Profit = profit, Weight = weight, Items = items, Bound = bound,
                     Branches = branches)
    return(solution)
  }
}

solution1 <- horowitz(p,w,M)          #Initialises the first solution
best.solution <- solution1
if(solution1$Iterations >= 300){      #Only use if Horowitz stalls
  solution2 <- branch.bound(p,w,M,B)     #Initialises the first solution
  if(solution1$Profit < solution2$Profit){   #Update best solution
    best.solution <- solution2
  }  else{
```

```
      best.solution <- solution1
    }
  }

  end <- Sys.time()
  time.taken <- end - start
  print(list(Time_taken = time.taken))
  return(best.solution)
}
```

References

[1] Cho, G. and Shaw, D. X. (1997). A depth-first dynamic programming algorithm for the tree knapsack problem. *INFORMS Journal on Computing*, 9(4):431–438.

[2] Greenberg, H. and Hegerich, R. L. (1970). A branch search algorithm for the knapsack problem. *Management Science*, 16(5):327–332.

[3] Lagoudakis, M. G. (1996). The 0-1 knapsack problem – an introductory survey. Technical report.

[4] Lauriere, M. (1978). An algorithm for the 0/1 knapsack problem. *Mathematical Programming*, 14(1):1–10.

[5] Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA.

[6] Zpfel, G., Braune, R., and Bgl, M. (2010). *The Knapsack Problem and Straightforward Optimization Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg.