Zach Ho
Kearns
CPE 349

Knapsack Algorithms – Final Report

Enumeration Brute Force Approach:

This is the most basic approach of the four. This approach finds the optimal solution by checking the values of every possible solution and picking the best.

Specifically, a set of items is represented by a binary string of "0's" and "1's" with length equal to the number of items. A "1" represents taking that item, while a "0" indicates not taking that item. This algorithm generates every binary string with length equal to the number of items, calculates the value of each possible set, and returns the maximum value set that satisfies the capacity limit.

For smaller input sizes, this approach will work, however, as the input size increases, the amount of time this approach would take to return an answer would quickly become infeasible.

Greedy Approach:

This approach turned out to be the fastest of the four, however it does not always return an optimal solution, but rather an estimation that's "close to" optimal. This approach finds its solution by continually taking the item with the best value-to-weight ratio until the capacity limit is reached.

This algorithm has the initial overhead of sorting the items by decreasing value-to-weight ratio. By first sorting the items, this approach can always choose to take the item that has the current best value per unit weight. The drawback of the greedy approach is that the optimal set of items may not necessarily be returned.

This approach was viable for all the tested input sets/sizes. In fact, it was the fastest approach of the four. If an estimation that's close to the optimal solution is acceptable and speed is more important, this approach is recommended.

Dynamic Approach:

This approach was the most reliable in both returning the optimal solution and well as finishing in a "reasonable" amount of time. This approach finds its solution by dynamically filling in a table that holds the optimal solution at a given point.

This algorithm has no initial time overhead as there is no sorting to be done, however, memory will be needed to create both the solution table, and well as the traceback table which holds the decisions for taking or not taking a given item.

This approach was viable for all the tested input sets/sizes. While slightly slower than the Greedy approach, this algorithm guarantees that the optimal solution shall be returned. If returned value is more important than running time than this algorithm is recommended.

Branch and Bound Approach:

This approach returns the optimal solution, however uses a lot of memory. Branch and Bound finds its solution by searching a binary tree which represents the search space of the problem and eventually finding the optimal solution.

The search space of the problem is represented by a binary tree made up of nodes. A node's left child represents the option of taking a given item, while the right child represents not taking the child. Each node has a running tally of its total value, weight, and items contained in the set, plus has a bound value which represents the potential max value that can be obtained by following that branch of the tree. By calculating bounds and comparing their values to the value of the current solution, the algorithm is able to skip over certain branches, essentially shrinking the search space.

This approach was able to return the optimal solution for smaller input sizes, however, once the input size started getting larger, the amount of memory used by this algorithm grew quickly. For the final test case, this algorithm was unable to complete without throwing an "Out of Memory" error and terminating prematurely.

Solutions Found by Various Algorithmic Approaches for Different Input Sets/Sizes:

	easy20	easy50	hard50	easy200	hard200
Enum	726(519)	N/A	N/A	N/A	N/A
Greedy	692(476)	1096(240)	16538(10038)	4075(2634)	136724(111924)
Dyn Prog	726(519)	1123(250)	16610(10110)	4092(2658)	137448(112648)
B'n'B	726(519)	1123(250)	16610(10110)	4092(2656)	137077(112277)

Note: Branch and Bound was only run for ~hour for the hard200 test case.

Runtimes of Various Algorithmic Approaches for Different Input Sizes/Sets:

	easy20	easy50	hard50	easy200	hard200
Enum	1011537068 ns	N/A	N/A	N/A	N/A
Greedy	4079 ns	6053 ns	6185 ns	34738 ns	36712 ns
Dyn Prog	1001370 ns	1347836 ns	14736451 ns	20929284 ns	131532342 ns
B'n'B	403180 ns	671746 ns	1279148 ns	8660598 ns	N/A

Greedy Algorithm:

```
public void KS_Greedy(int numItems, ArrayList<Item> items, int
capacity) {
        // Copy the list of items so that the original isn't altered
by sort
        ArrayList<Item> itemsCopy = new ArrayList<>();
        for (int i = 0; i < numItems; i++)
            itemsCopy.add(new Item(items.get(i)));

        // Sort the copied list of items by decreasing value of
value/weight
        // ratio to ensure you get the most value per unit weight
        Collections.sort(itemsCopy, Item.byRatio());

        int setItemCount = 0;
        int setVal = 0;
        int setWeight = 0;
        boolean maxWeight = false;

        // Take items until capacity is reached
        while (!maxWeight && setItemCount < numItems) {
            Item itemN = itemsCopy.get(setItemCount);
            if (setWeight + itemN.wt < capacity) {
                setVal += itemN.val;
                setWeight += itemN.wt;
                setItemCount++;
            }
            else {
                maxWeight = true;
            }
        }

        // Determine the set of items taken
        ArrayList<Item> maxSet = new ArrayList<>();
        for (int j = 0; j < setItemCount; j++)
            maxSet.add(itemsCopy.get(j));
        // Sort the set by increasing item index
        Collections.sort(maxSet, Item.byIndexAscend());

        // Print results
        System.out.println("Greedy solution (not necessarily optimal):
"
         +setVal+" "+setWeight);

        for (Item item : maxSet)
            System.out.print(item.id + " ");
        System.out.print("\n");
    }
```

Dynamic Algorithm: This approach was gathered from the code at
http://introcs.cs.princeton.edu/java/96optimization/Knapsack.java.html

```
public void KS_Dynamic(int numItems, ArrayList<Item> items, int
capacity) {
        // table[n][w] = max profit for items 1..n with weight limit w
        // taken[n][w] = does solution for items 1..n take item N?
        int[][] table = new int[numItems+1][capacity+1];
        boolean[][] taken = new boolean[numItems+1][capacity+1];

        long startTime = System.nanoTime();
        // Fill in both tables with appropriate values
        for (int n = 1; n <= numItems; n++) {
            Item itemN = items.get(n-1);
            for (int w = 1; w <= capacity; w++) {
                // Determine values for taking or not taking item N
                int dontTake = table[n-1][w];
                int take = itemN.wt <= w ?
                 itemN.val + table[n-1][w-itemN.wt] : 0;
                // Take the max of taking or not taking item N and
fill in
                // corresponding cell in taken table
                table[n][w] = Math.max(dontTake, take);
                taken[n][w] = (take > dontTake);
            }
        }
        long endTime = System.nanoTime();
        long duration = endTime-startTime;
        double seconds = (double)duration / 1000000000.0;
        System.out.println("duration for dynamic = "+seconds);

        // Determine item set
        ArrayList<Item> itemSet = new ArrayList<>();
        int setVal = 0;
        int setWeight = 0;

        for (int n = numItems, w = capacity; n > 0; n--) {
            Item itemN = items.get(n-1);
            if (taken[n][w]) {
                itemSet.add(itemN);
                w -= itemN.wt;
                setVal += itemN.val;
                setWeight += itemN.wt;
            }
        }
        Collections.sort(itemSet, Item.byIndexAscend());

        // Print results
        System.out.println("Dynamic Programming solution: "+setVal+"
"+setWeight);
        for (Item item : itemSet)
            System.out.print(item.id + " ");
```

```
        System.out.print("\n");
    }
```

Branch and Bound Algorithm:

```
public void KS_BnB(int numItems, ArrayList<Item> items, int capacity)
{
        Collections.sort(items, Item.byRatio());
        Node root = new Node();
        Node solution = new Node();
        PriorityQueue<Node> queue = new PriorityQueue<>();

        root.getBound(numItems, items, capacity);
        queue.offer(root);

        long startTime = System.nanoTime();
        // The queue shall consist of nodes which are the root of a
promising
        // branch of the search space tree
        while(!queue.isEmpty()) {
            Node popped = queue.poll();
            Item itemN = items.get(popped.lvl);

            if (popped.bnd > solution.val) {
                Node take = new Node(popped);
                take.val += itemN.val;
                take.wt += itemN.wt;
                take.itemSet.add(itemN.id);

                if (take.wt <= capacity && take.val > solution.val) {
                    solution = take;
                    //printSolution(solution);
                }
                take.getBound(numItems, items, capacity);
                if (take.bnd > solution.val)
                    queue.offer(take);

                Node dontTake = new Node(popped);
                dontTake.getBound(numItems, items, capacity);
                if (dontTake.bnd > solution.val)
                    queue.offer(dontTake);
            }
        }
        long endTime = System.nanoTime();
        long duration = endTime-startTime;
        double seconds = (double)duration / 1000000000.0;
        System.out.println("duration for enum = "+seconds);

        //Print results
        Collections.sort(solution.itemSet);
        System.out.println("Using Branch and Bound the best feasible "
         +"solution found: "+solution.val+" "+solution.wt);
```

```
        for (Integer i : solution.itemSet)
            System.out.print(i + " ");
        System.out.print("\n");
    }
```

Method to calculate bound used by Branch and Bound:

```
public void getBound(int numItems, ArrayList<Item> items, int
capacity) {
        int i = lvl;
        int setWeight = wt;
        bnd = val;

        if (wt >= capacity) {
            bnd = 0;
        }
        else {
            while (i < numItems && setWeight + items.get(i).wt <=
capacity) {
                bnd += items.get(i).val;
                setWeight += items.get(i).wt;
                i++;
            }
            if (i < numItems)
                bnd += ((capacity - setWeight) * items.get(i).rat);
        }
    }
```