0-1 Knapsack Optimization with Branch-and-Bound Algorithm

Salem Hildebrandt & Christopher Hanson
Computer Science
Simpson College
701 North C Street; Indianola, Iowa 50125
salem.hildebrandt@my.simpson.edu; christopher.hanson@my.simpson.edu

Abstract

The 0-1 Knapsack problem is a combinatorial optimization problem in which the subject must maximize the value of potential items for placement in a knapsack without exceeding its size constraint. This problem has been used in a variety of applications such as aiding financial investment problems as well as by Non-Governmental Organizations in supplying relief effort.

This paper focuses primarily on the best-first implementation of the branch-and-bound algorithm with thorough experimentation. Our experiments showed that the value which most directly affects the runtime of the algorithm was related to the size of the knapsack instead of the number of items. While the total number of items to choose to place in the knapsack did affect the runtime (as it has previously been found to do so), it was found that the size of the knapsack had an even greater effect on the runtime.

1 Problem Description

There exists a knapsack of predetermined size *K*, and several potential items each with a size *s* and value *v*. Each item can either be included in the knapsack or not included in the knapsack but cannot be partially included in the knapsack.

The subject is charged with placing items into a knapsack with the intent of maximizing obtained value without exceeding *K*, the size restriction, i.e., the goal is to maximize the value within the size constraint of the knapsack. [3] This can be expressed by

$$\max \sum_{k=1}^{i} v_k \qquad\qquad K \geq \sum_{k=1}^{i} s_k$$

where *i* is the total number of items placed in the knapsack.

A prime example of the 0-1 Knapsack Problem involves a hypothetical burglary. The subject of this scenario is the burglar, and he or she is attempting to steal potential items from a random individual’s house. These items may be of great, little, or no significance in terms of value. Due to the size constraint on the knapsack he or she is carrying, the burglar would likely have forgo acquiring a valuable grandfather clock and settle for a less valuable but highly portable necklace.

2 Approaches to Solving the 0-1 Knapsack Problem

We researched three major approaches, aside from branch-and-bound, to the 0-1 Knapsack Problem: brute force, the greedy approach, and dynamic programming.

2.1 Brute Force Algorithm

The brute force approach to the 0-1 Knapsack Problem lists every possible combination of potential items taking into consideration the size constraint on the knapsack. Then, the algorithm finds the combination of highest value. This particular approach is not efficient; its runtime is $O(2^n)$. Thus, it is rarely used.

2.2 Greedy Algorithm

There are three distinct strategies for implementing the greedy approach. The first strategy is to locate the item with the greatest value and then fill the knapsack with the particular item without violating its size restriction [2]. As soon as the initial item is appropriately utilized, the next most valuable item is placed into the knapsack. This process is repeated until no more items can be placed in the knapsack without violating its size constraint [2]. This strategy is rarely optimal because there is a great risk of meeting the size constraint with relatively few high-valued items whereas several smaller and slightly lower-valued items would make a better solution.

The second strategy for implementing the greedy approach is to fill the knapsack with the item of smallest size and continually move to the next smallest item until no more items can be placed in the knapsack [2]. This “smallest-first” strategy is not optimal because the knapsack could potentially contain a large number of small non-valuable items, neglecting to incorporate valuable, albeit large, items.

The third strategy considers the ratio of the object value to its size, hereafter considered the unit cost of the item. This strategy starts by placing the item with greatest unit cost into the knapsack and proceeds to the next greatest unit cost.

This third strategy will reach optimization when partial inclusiveness is allowed (an item can be partially included in the knapsack); however, it is not guaranteed to reach an optimal solution for the 0-1 Knapsack Problem without trial and error. Thus, the greedy approach to the 0-1 Knapsack Problem is considered heuristic.

2.3 Dynamic Programming

The dynamic programming approach to the 0-1 Knapsack Problem divides the problem into “sub-problems”. First, the algorithm determines whether or not to place the item of greatest value in the knapsack [3]. A that point, there are fewer items remaining for consideration. This process is repeated continually with recursion until the knapsack is filled without exceeding the size constraint [3]. The average runtime for the dynamic programming approach to the 0-1 Knapsack Problem is $O(nK)$, where K is the size constraint on the knapsack and n is the number of items [1].

3 Branch and Bound

There are two predominant implementations of branch-and-bound: breadth-first, and best-first. We implemented each of these approaches to the 0-1 Knapsack Problem.

3.1 Breadth-First

The breadth-first approach uses a queue to hold potential nodes of items according to their potential increase to the overall value of the knapsack (bound). This approach considers whether or not a node is promising according to whether or not the value of the knapsack would be greater if this item were to be placed in without exceeding the size constraint. However, once the algorithm has traversed a path, it backtracks to the last parent where the bound was still considered promising and evaluates the other children [3]. Because of this, sections of the state-space-tree on which the algorithm is executed are considered when they could have been eliminated prior to traversal.

3.2 Best-First

The best first approach utilizes a state-space-tree which consists of an initial state, a goal state, intermediate states, and transforming operators with pre-conditions and post-conditions [4]. For the 0-1 Knapsack Problem, the initial state is the empty knapsack, the goal state is the optimized knapsack within the size constraint, and the intermediate states are the various states of the knapsack with different item combinations but have not reached optimization [3].

The best-first algorithm eliminates unnecessary path traversals in the state-space-tree using the bounds on the nodes. These bounds indicate whether or not the traversal is promising; if the bound is greater than the total value of the root, it is considered promising. After this, the algorithm traverses the child that is most promising and continues this process until the solution cannot be further optimized. Unlike breadth-first, best-first does not traverse each child before eliminating the parent node. However, with best-first, the most promising traversal is not always the optimal solution; in many cases the algorithm has to backtrack and find a different path within the state-space-tree [3].

In summary, the key idea behind this algorithm is to calculate whether or not a combination of items is promising, then traverse the path and backtrack according to whether or not the size constraint is violated or more promising nodes appear.

4 Algorithm Description

We considered a variety of sources and implementations when exploring the algorithm; we decided that the work of Richard Neapolitan best suited our understanding of the branch-and-bound algorithm [3]. We adopted the general ideas behind the best-first implementation and the bound function implementation as described below.

The algorithm works with a linked-list of specified items which can be randomly generated or manually created, and an integer representing the size constraint on the knapsack. Each item is defined with its size and value. The state-space-tree consists of nodes whereby each node represents the current status of the knapsack with an integer representing the value of the items and a linked-list of the items in the knapsack.

The core of the algorithm is a 'while' loop that runs until the linked-list of nodes, considered a priority-queue, is empty. During the first run of the loop, the first node in the linked-list is retrieved and its values are held in a node variable, *v*. Then, a second node, *u*, is initialized to be empty and its values are calculated according to the values of *v*. The level, within the state-space-tree, of *u* is calculated according to the level of *v*; the level of *u* is always one greater than the level of *v*. The size of node *u* is equivalent to the sum of the size of *v* and the level position of *u*. The final component of node *u* is computed according to whether or not observing another node along the current traversal is promising; that is, whether or not the total size of the considered items is within the size constraint, and the value is greater than the value at the root of the current subtree.

Once these calculations are performed, the size of *u* is compared with the size constraint on the knapsack, and the value of *u* is compared with the previously obtained maximum value. Then, the new maximum value is set accordingly. The algorithm then adds node *u* to the linked-list if its potential value is greater than the previously calculated maximum value. Before the while loop starts its next run, *u* is reset, and the component calculations are performed again.

After the while loop reaches its terminating condition, the maximum value is printed.

4.1 Pseudocode

```
ALGORITHM   BestFirstBranchAndBound   (LinkedList   <Items>   Items,   int
MaxSize)
//Input:         A   linked-list   (priority-queue)   of   items   for
consideration
                 The size constraint on the knapsack (MaxSize)
//Output:        The  maximum  value  of  the  knapsack  found  by  the
algorithm
//Precondition:  The items are organized according to input

//n = number of items

PriorityQueue<Node> PQ //initialized empty
Size[n], Value[n]
Node u, v (parameters: level, value, weight, bound)

MaxValue = 0
Initialize the root, v(level ← -1, all else 0)
```

```
PQ.insert(v) //insert if possible

while (PQ is not empty)
      v ← PQ.deleteMax("pop" the first node)

      reset_u

      if(u.size does not exceed MaxSize & u.value exceeds MaxValue)
            Max Value ← u.value
      if(u.bound exceeds Max Value)
            PQ.insert(u)

      reset_u

      if(u.bound exceeds Max Value)
            PQ.add(u)

return Max Value

Method reset_u
      u = empty
      if(v.level = -1)
            u.level ← 0
      else if (v.level != (n-1))
            u.level ← v.level + 1
      u.size ← v.size + Size[u.level]
      u.value ← v.value + Value[u.level]
      u.bound ← Bound(MaxSize, n, Size, Value)
```

The branch-and-bound algorithm has the potential to generate all possible combinations of items depending on bound calculations of items [3].

4.2 Memory and Time Requirements

The algorithm requires a linked-list to store the potential items, a linked list or priority-queue to store the item combination nodes, and two arrays to hold the size and values of each item. Therefore, the memory requirements are O(N), where N is the number of items in the knapsack. While the runtime of the algorithm is exponential in the worst case, the real time of the algorithm is considerably quick due to the pruning of the search tree [3].

5 Results

Experimentation was performed utilizing three approaches: randomized generation, manual input, and verification of previous work. For the randomized generation experiments, we developed a program to generate potential items in large quantities and output the resulting optimized value. The manual input and verification testing required little in the way of additional programming. Most of the data sets were unique in that they did not come from other sources; however, for the purpose of ensuring accuracy with the algorithm, a single data set from Richard Neapolitan's work [3] was used. This seemed appropriate given that many components of our algorithm were inspired by his pseudocode. The randomized tests served to test the time requirements given various values for each significant aspect of the randomized generator. The input and verification tests were performed to exploit weaknesses of the branch-and-bound algorithm and test its accuracy. The following tables and figures give some insight into the obtained results.

5.1 Randomized Sets

Number of Items (n)	**Size of Knapsack (maxSize)**	**Size Upper Bound**	**Value Upper Bound**	**Time**	**Output (maxValue)**
100	100	10	10	12749	149
100	70	10	10	47	118
100	50	10	10	15	92
100	20	10	10	0	29

Table 1: Changes in *maxSize* (size of the knapsack)

Number of Items (n)	**Size of Knapsack (maxSize)**	**Size Upper Bound**	**Value Upper Bound**	**Time**	**Output (maxValue)**
50000	50	10	10	3000	101
10000	50	10	10	78	99
5000	50	10	10	16	87
500	50	10	10	0	104
120	50	10	10	0	78

Table 2: Changes in *n* (number of potential items)

Number of Items (n)	**Size of Knapsack (maxSize)**	**Size Upper Bound**	**Value Upper Bound**	**Time**	**Output (maxValue)**
1000	500	100	200	0	2075
1000	500	100	80	0	1006
1000	500	100	70	16	781
1000	500	100	60	0	539
1000	500	100	50	16	472
1000	500	100	40	0	326

Table 3: Changes in the upper bound for item value

Number of Items (n)	Size of Knapsack (maxSize)	Size Upper Bound	Value Upper Bound	Time	Output (maxValue)
1000	500	150	100	0	620
1000	500	80	100	47	1010
1000	500	70	100	16	1566
1000	500	60	100	1110	1411
1000	500	50	100	15937	1996

Table 4: Changes in the upper bound for item size

Based on these results, it appears as though the time required to compute the optimal value increases in several situations: as the size upper bound decreases, as the number of items increases, and as the size of the knapsack increases. While the results from Table 3 are not insignificant, they do not give much insight into the effect of value on runtime.

5.2 Manual Input / Verification Testing

```
Watch - Value: 40; Size: 2
Painting - Value: 30; Size: 5
Television - Value: 50; Size: 10
Doll - Value: 10; Size: 5
The optimal solution for the items given, and the size constraint 16 is: 90
```

Figure 1: Verification data set – from Neapolitan

```
Doll - Value: 8; Size: 2
Painting2 - Value: 18; Size: 6
Painting - Value: 20; Size: 7
Brick - Value: 2; Size: 18
Television - Value: 30; Size: 18
The optimal solution for the items given, and the size constraint 25 is: 46
```

Figure 2: Exploiting the "cropping" tendency of the branch-and-bound algorithm (should be 56)

The results from Figure 1 were expected given that the dataset was from Neapolitan's work [3]. Figure 2 concerns a dataset designed for the purpose of testing the algorithm, its results were verified manually.

6 Conclusion

To summarize our research, bar the dynamic programming approach, the branch-and-bound approach to the 0-1 Knapsack Problem is comparable in terms of runtime to various other algorithms. The runtime of branch-and-bound is greatly affected by the number of potential items, the size constraint on the knapsack, and the size of each item. And, finally, our implementation was successful in providing a solution to the 0-1 Knapsack Problem, given that branch-and-bound is renowned for sometimes selecting a solution that is not the optimal solution. Regarding further work, it seems as though an organizational function would be useful for inserting the potential items into the linked-list in such a way that the optimal is achieved more often if not every time the algorithm is run.

Acknowledgments

Many thanks to Dr. Lydia Sinapova for assistance in research, and to Dr. Mark Brodie, who evaluated the research paper and provided feedback with regards to its initial draft.

References

[1] Goddard, S. (n.d.). *Dynamic programming 0-1 knapsack problem* [PowerPoint slides]. Retrieved from http://cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf

[2] *Greedy algorithms* [PowerPoint slides]. Retrieved from http://www.radford.edu~nodie/classes/360/greedy.html

[3] Neapolitan, R. (2014). *Foundations of Algorithms* (5th ed.). Jones & Bartlett Learning.

[4] Sinapova, L. (2013). *Algorithm design paradigms* [PDF document]. Retrieved from faculty lecture notes website: faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250-2013/DesignParadigms.pdf