0/1 Knapsack Problem: A Branch and Bound Approach

Sumanth

June 18, 2025

Abstract

The 0/1 Knapsack Problem seeks to maximize the total value of items included in a knapsack without exceeding its weight capacity, where each item is either included or excluded. The **branch and bound** approach optimizes this by exploring promising solutions while pruning paths with bounds worse than the current best. This document details the problem formulation, pseudocode, complexity analysis, practical applications, and implementation considerations, including bound calculation and the use of a queue for BFS, as noted by the contributor, with suggestions for priority queue optimization.

1 Introduction

The 0/1 Knapsack Problem is a classic combinatorial optimization problem where, given a set of items with weights and values, the goal is to select a subset that maximizes total value while keeping the total weight within a knapsack's capacity. The **branch and bound** approach efficiently explores the solution space by maintaining an upper bound on possible solutions and pruning unpromising paths. This method is widely used in resource scheduling and optimization problems, offering significant performance improvements over brute-force methods.

2 Problem Formulation

Given:

- n items, each with weight w_i and value v_i.
- A knapsack with capacity W.

Find a subset of items to include such that the total weight does not exceed W, and the total value is maximized. Formally, maximize:

$$\sum_{i=1}^{n} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W, \quad x_i \in \{0,1\}$$

where $x_i = 1$ if item i is included, and $x_i = 0$ otherwise. The output is the maximum value and the selected items.

2.1 Key Idea

The algorithm uses **branch and bound** to explore a decision tree, where each node represents a partial solution (items selected up to a level, current weight, and value). At each node:

- Compute an upper bound on the best possible value achievable.
- Prune the node if its bound is less than the current best solution.
- Branch by including or excluding the next item.

Nodes are processed using a queue (BFS), with potential optimization via a priority queue for best-bound-first search.

3 0/1 Knapsack Algorithm

The algorithm uses a queue to process nodes in BFS order, calculating bounds to prune unpromising paths. Below is the pseudocode:

3.1 Bound Calculation

The `computeBound` function estimates the maximum possible profit by:

- Adding the current profit.
- Greedily including remaining items (sorted by value-to-weight ratio) until the capacity is exceeded.
- Using fractional inclusion for the last item to tighten the bound.

3.2 Priority Queue Optimization

As noted by the contributor, the implementation uses a queue (BFS). Using a **priority queue** to process nodes with the highest bound first can improve performance by exploring promising paths earlier.

4 Complexity Analysis

- **Time Complexity**: $O(2^n)$ in the worst case, as all subsets may be explored. In practice, pruning reduces the number of nodes significantly.
- **Space Complexity**: $O(2^n)$ in the worst case for the queue storing all nodes, though pruning limits queue size in practice.

5 Practical Applications

The 0/1 Knapsack Problem is used in:

- **Resource Scheduling**: Allocating limited resources to tasks for maximum benefit.
- **Finance**: Optimizing investment portfolios under budget constraints.

Algorithm 1 0/1 Knapsack Algorithm (Branch and Bound)

Input: Arrays $w[1 \dots n]$, $v[1 \dots n]$, capacity W
Output: Maximum value, selected items
Initialize queue Q with root node $(\text{level} = 0, \text{profit} = 0, \text{weight} = 0, \text{bound} = \text{computeBound}(0,0,0))$
Initialize maxProfit $\leftarrow 0$, bestItems $\leftarrow \emptyset$
while Q is not empty **do**
  $u \leftarrow Q$.dequeue()
  if u.bound $\leq$ maxProfit **then**
    continue ▷ Prune node
  end if
  if u.level $= n$ **then**
    if u.profit $>$ maxProfit **then**
      maxProfit $\leftarrow u$.profit
      Update bestItems with u.items
    end if
    continue
  end if
  $v \leftarrow$ new node ▷ Include item u.level $+ 1$
  v.level $\leftarrow u$.level $+ 1$
  v.weight $\leftarrow u$.weight $+ w[v$.level$]$
  if v.weight $\leq W$ **then**
    v.profit $\leftarrow u$.profit $+ v[v$.level$]$
    v.bound $\leftarrow$ computeBound$(v$.level$, v$.profit$, v$.weight$)$
    if v.bound $>$ maxProfit **then**
      Q.enqueue(v)
    end if
  end if
  $w \leftarrow$ new node ▷ Exclude item u.level $+ 1$
  w.level $\leftarrow u$.level $+ 1$
  w.weight $\leftarrow u$.weight
  w.profit $\leftarrow u$.profit
  w.bound $\leftarrow$ computeBound$(w$.level$, w$.profit$, w$.weight$)$
  if w.bound $>$ maxProfit **then**
    Q.enqueue(w)
  end if
end while
Return maxProfit, bestItems
Procedure computeBound$(level, \text{profit}, \text{weight})$:
Initialize bound $\leftarrow$ profit, remWeight $\leftarrow W -$ weight
for $i =$ level $+ 1$ to n **do**
  if remWeight $\geq w[i]$ **then**
    bound $\leftarrow$ bound $+ v[i]$
    remWeight $\leftarrow$ remWeight $- w[i]$
  else
    bound $\leftarrow$ bound $+ \left(\frac{\text{remWeight}}{w[i]}\right) \cdot v[i]$
    break
  end if
end for
Return bound

- **Operations Research**: Solving combinatorial optimization problems in logistics and production.
- **Cryptographic Analysis**: Certain knapsack-based cryptographic systems.

6 Example

Given $n = 4$, weights $w = \{2, 1, 3, 2\}$, values $v = \{12, 10, 20, 15\}$, and capacity $W = 5$, the algorithm (with items sorted by v_i/w_i) yields:

Item	1	2	3	4
Weight	1	2	2	3
Value	10	12	15	20

Selected items: {1, 3} (weight = $1+2=3$, value = $10+15=25$). The branch and bound tree prunes nodes where the bound is less than 25.

7 Limitations and Extensions

- **Limitations**:
  - Exponential worst-case complexity for large n.
  - Memory-intensive for large queues without effective pruning.
- **Extensions**:
  - **Priority Queue**: Use a max-heap for best-bound-first search to reduce explored nodes.
  - **Dynamic Programming**: Solve in $O(n \cdot W)$ time for pseudo-polynomial cases.
  - **Approximation Algorithms**: Use greedy or FPTAS for near-optimal solutions in large instances.

8 Implementation Considerations

- **Bound Tightness**: Sort items by value-to-weight ratio to compute tighter bounds, improving pruning.
- **Priority Queue**: Replace the BFS queue with a priority queue (e.g., `std::priority_queue` in C++) for best-bound-first search.
- **Numeric Stability**: Use appropriate data types (e.g., `long long` for large values).
- **Node Representation**: Store nodes efficiently with level, profit, weight, and bound to minimize memory usage.
- **Output Format**: Track selected items using a bit vector or list for clear solution reporting.

9 References

- GeeksForGeeks: 0/1 Knapsack using Branch and Bound
- Wikipedia: Knapsack Problem