Lecture 18: Chapter 5 Part 3

Backtracking

CS3310

0-1 Knapsack Problem

The greedy solutions to the 0-1 Knapsack problem we previously discussed were only partially successful.

The Problem:

- A thief breaks into a jewelry store with a knapsack which will break if the total weight of the items stolen exceeds some maximum weight W.
- Each item has a value and a weight.
- How do we maximize the total value of the items taken while not breaking the bag?

0-1 Knapsack Problem

S = { $item_1$, $item_2$, …, $item_n$ }
w[i] = weight of $item_i$
p[i] = profit of $item_i$
W = maximum weight the knapsack can hold

A = bag of stolen items.

- The brute-force solution is to consider all subsets of the n items, discard the ones that exceed W, and take one of the remaining subsets with maximum total profit.
  - This solution is $\boldsymbol{\Theta(2^n)}$
- None of the greedy solutions we discussed worked. However, there is a backtracking algorithm that arrives at an optimal solution.

0-1 Knapsack Problem

- Unlike the sum-of-subsets problem, the 0-1 knapsack problem is an **optimization problem**.
- With sum-of-subsets, we select and reject items until their total weight = W
  - i.e. if weight = W when we reach a node, we *know* we have found a solution.
  - Other solutions from the instance are also possible.
- With the 0-1 knapsack problem, the goal is to find the combination of items that leads to the *greatest* profit without going over W.
  - When we reach a node, we do *not* know if it is a solution until the entire tree has been searched.
    - i.e. Reaching a different node might provide a higher profit!

0-1 Knapsack Problem

With optimization problems, we backtrack a little differently than before:

- For the 0-1 Knapsack Problem, we track the best profit reached so far and a list of the items we stole/rejected to get there.
- If we visit a node with a greater total profit than the best profit found so far, we update the best profit so far as well as the list of items.
- Finding a greater profit at a node doesn't mean we can backtrack yet. We may find a better solution at one of the node's descendants (by stealing more items).
- Therefore, we <u>always</u> visit a promising node's children with optimization problems.

0-1 Knapsack Problem

A general algorithm for backtracking in the case of optimization problems

```
void checknode (node v)
{
     node u;
     if (value(v) is better than best)
          best = value(v);
     if (promising(v))
          for (each child u of v)
               checknode(u)
}
```

- `best` is the value of the best solution found so far, and `value(v)` returns the value reached at node `v`

0-1 Knapsack Problem Components

While traversing the state space tree, we keep track of the following global variables:

- `maxProfit`: Initialize to 0. When we reach a node with a greater `profit` than `maxProfit` and `weight < W`, update `maxProfit` to `profit`.
- `bestSet`: An array indicating the items to steal and reject to achieve `maxProfit`. When we update `maxProfit`, update `bestSet` to show how we reached that profit.
- `p`: An array containing the price of each item.
  - i.e. `p[i]` contains the price of item *i*
- `w`: An array containing the weight of each item.
  - i.e. `w[i]` contains the weight of item *i*

0-1 Knapsack Problem

- Start by ordering the items in **nonincreasing** order by p_i / w_i
- In a state space tree, we begin at the root where $i = 0$ and perform a depth first search.
  - Moving left indicates stealing item $i + 1$
  - Moving right indicates rejecting item $i + 1$

- When we visit a node in our traversal:
  - `profit`: the sum of the profits of the items stolen up to and including that node.
  - `weight`: the sum of the weights of those items.

- When should we update `maxProfit` and `bestSet`?

0-1 Knapsack Problem

- Start by ordering the items in **nonincreasing** order by p_i / w_i
- In a state space tree, we begin at the root where $i = 0$ and perform a depth first search.
  - Moving left indicates stealing item $i + 1$
  - Moving right indicates rejecting item $i + 1$

- When we visit a node in our traversal:
  - `profit`: the sum of the profits of the items stolen up to and including that node.
  - `weight`: the sum of the weights of those items.

- When should we update `maxProfit` and `bestSet`?
  - If `profit > maxProfit` and `weight <= W`.
- Finally, see if the node we are visiting is promising. If it is, we visit its children.

0-1 Knapsack Problem

To determine if a node is promising:

1. Initialize two variables: `bound` to `profit` and `totweight` to `weight`.
2. Greedily choose the items we haven't considered yet, adding their profits to `bound` and their weights to `totweight`. We stop when we try to choose an item that would bring `totweight` above W. We call this item *k*.
3. Assume we can take <u>part</u> of item *k*. Calculate the fraction of *k* that would fit in the bag and add that fraction's profit to `bound`.
    - Expanding beyond the current node won't lead to a profit equal to `bound`. Rather, `bound` is an *upper bound* on the profit we can achieve by doing so.
        - i.e. it is impossible to reach a profit higher than `bound` by continuing.
        - How can we use this info to know when to backtrack?

0-1 Knapsack Problem

To determine if a node is promising:

1. Initialize two variables: `bound` to `profit` and `totweight` to `weight`.
2. Greedily choose the items we haven't considered yet, adding their profits to `bound` and their weights to `totweight`. We stop when we try to choose an item that would bring `totweight` above W. We call this item *k*.
3. Assume we can take <u>part</u> of item *k*. Calculate the fraction of *k* that would fit in the bag and add that fraction's profit to `bound`.
    - Expanding beyond the current node won't lead to a profit equal to `bound`. Rather, `bound` is an *upper bound* on the profit we can achieve by doing so.
        - i.e. it is impossible to reach a profit higher than `bound` by continuing.
        - How can we use this info to know when to backtrack?
            - If `bound <= maxprofit`, we backtrack.

0-1 Knapsack Problem

To determine if a node is promising, `totweight` and `bound` are computed with the following equations:

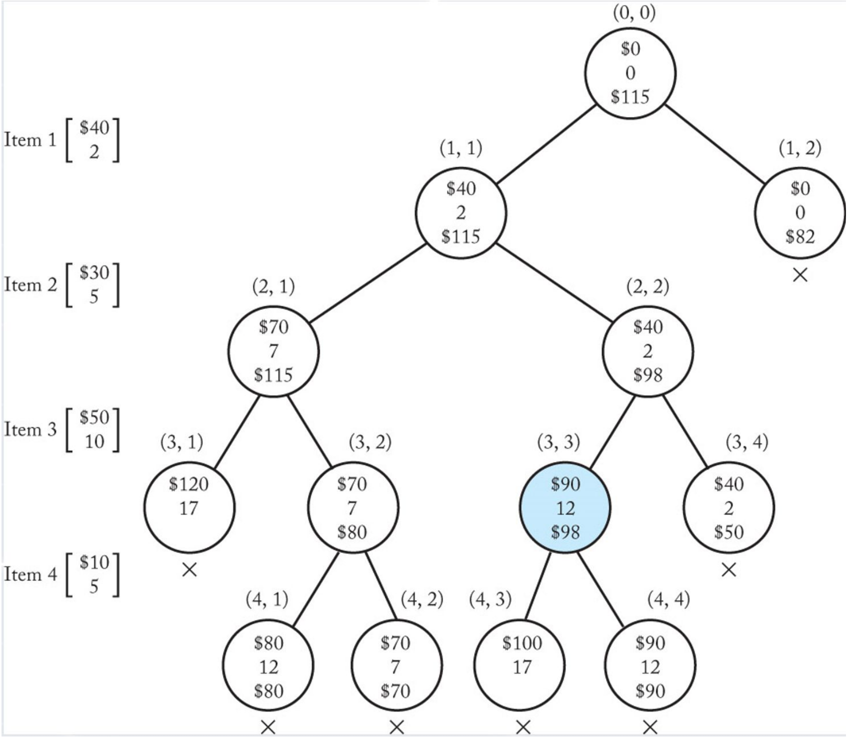
- `j`: The index of the first item we have not yet considered.
- `k`: The index of the item that, if taken, would bring `totweight` over W.

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for } k\text{th item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}.$$

0-1 Knapsack Problem

- At the dummy node, no items have been stolen or rejected yet.
- In each node we write:
  - Top: the overall profit gained for taking that item.
  - Middle: the overall weight.
  - Bottom: the node's bound.
- After determining the dummy node is promising, we move left to steal the first item.



W = 16 *maxprofit* = $90

0-1 Knapsack Problem Step 1

To initialize the problem:

- Initialize `maxprofit` to \$0, `include` and `bestSet` to empty
- Visit the dummy node: `profit` = \$0, `weight` = 0

What's next?

$W = 16$ `maxprofit` = \$0 `bestSet` = { }

$w = \{ 2, 5, 10, 5 \}$ $p = \{ 40, 30, 50, 10 \}$ `include` = { }

0-1 Knapsack Problem Step 1

To initialize the problem:

- Initialize `maxprofit` to \$0, `include` and `bestSet` to empty
- Visit the dummy node: `profit` = \$0, `weight` = 0

Is this node promising? Take remaining items until one would bring `totweight` over W

- Calculate k: $2 + 5 + 10 = 17$ and $17 > 16 \therefore k = 3$
- Calculate `totweight`: $weight + w_1 + w_2 = 0 + 2 + 5 = 7$
- Calculate `bound`: $profit + p_1 + p_2 + (W - totweight)\ (p_3 / w_3)$
  $= 0 + \$40 + \$30 + (16 - 7) \times (\$50/10) =$ **\$115**

`bound > maxprofit` and `weight` < 16. <u>Promising</u>.

$W = 16$ `maxprofit` = \$0 `bestSet` = { }

$w = \{ 2, 5, 10, 5 \}$ $p = \{ 40, 30, 50, 10 \}$ `include` = { }

0-1 Knapsack Problem Step 2

Step 2?

W = 16 $\quad$ `maxprofit` = \$0 $\quad$ `bestSet` = { }

w = { 2, 5, 10, 5 } $\quad$ p = { 40, 30, 50, 10 } $\quad$ `include` = { }

0-1 Knapsack Problem Step 2

Steal item 1 by setting `include[1]` to 1

- `profit` = \$0 + \$40 = \$40 `weight` = 0 + 2 = 2
- 2 < 16 and \$40 > \$0 so *we update* `maxprofit` and `bestSet`

Determine if the current node is promising:

- j: 2 k: 2 + 5 + 10 = 17 and 17 > 16 ∴ $k = 3$
- `totweight: weight +` w_2`: 2 + 5 = 7`
- `bound: profit +` p_2 `+ (W - totweight) (`p_3 `/` w_3`)`
  = \$40 + \$30 + (16 - 7) × (\$50 / 10) = **\$115**

\$115 > `maxprofit` and 2 < 16. Promising.

W = 16 `maxprofit` = **\$40** `bestSet` = { **1** }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { **1** }

0-1 Knapsack Problem Step 3

Steal item 2 by setting `include[2]` to 1

- `profit` = ?? `weight` = ??

$W = 16$ `maxprofit` = \$40 `bestSet` = { 1 }

$w = \{ 2, 5, 10, 5 \}$ $p = \{ 40, 30, 50, 10 \}$ `include` = { 1, **1** }

0-1 Knapsack Problem Step 3

Steal item 2 by setting `include[2]` to 1

- `profit` = \$40 + \$30 = \$70 `weight` = 2 + 5 = 7
- 7 < 16 and \$70 > \$40 so we update `maxprofit` and `bestSet`

Determine if the current node is promising:

- j: 3 k: 7 + 10 = 17 and 17 > 16 $\therefore k = 3$
- `totweight: weight = 7`
- `bound: profit + (W - totweight) (p_3 / w_3)`

  = \$70 + (16 - 7) × (\$50 / 10) = **\$115**

\$115 > `maxprofit` and 7 < 16. Promising.

W = 16 `maxprofit` = **\$70** `bestSet` = **{ 1, 1 }**

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 1 }

0-1 Knapsack Problem Step 4

Steal item 3 by setting `include[3]` to 1

- `profit` = \$40 + \$30 + \$50 = \$120 `weight` = 2 + 5 + 10 = 17

`weight > W`, so we backtrack. What do we do next?

W = 16 `maxprofit` = \$70 `bestSet` = { 1, 1 }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 1, 1 }

0-1 Knapsack Problem Step 5

Reject item 3 by setting `include[3]` to 0.

- `profit` = \$40 + \$30 + \$0 = \$70 `weight` = 2 + 5 + 0 = 7
- 7 < 16 but \$70 = \$70 so `maxprofit` and `bestSet` do not change

Determine if the current node is promising:

- j: 4 k: 7 + 5 = 12 and 12 < 16 $\therefore$ k = 5
- Since $k > n$, we simply add the profit of the remaining items to compute `bound`:

  `profit + p`$_4$ = \$70 + \$10 = **\$80**

\$80 > `maxprofit` and 7 < 16. Promising.

W = 16 `maxprofit` = \$70 `bestSet` = { 1, 1 }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 1, 0 }

0-1 Knapsack Problem Step 6

Steal item 4 by setting `include[4]` to 1

- `profit` = \$40 + \$30 + \$0 + \$10 = \$80 `weight` = 2 + 5 + 0 + 5 = 12
- 12 < 16 and \$80 > \$70 so we update `maxprofit` and `bestSet`

Determine if the current node is promising:

- No other items to add so `bound` = \$80
- Since \$80 = \$80, the current node is <u>nonpromising</u>.
  - Recall that with optimization problems, a node is *nonpromising* if its children don't need to be visited (or if it has no children)

W = 16 `maxprofit` = **\$80** `bestSet` = **{ 1, 1, 0, 1 }**

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 1, 0, 1 }

0-1 Knapsack Problem Step 7

Reject item 4 by setting `include[4]` to 0

- `profit` = \$40 + \$30 + \$0 + \$0 = \$70 `weight` = 2 + 5 + 0 + 0 = 7

Determine if the current node is promising:

- No other items to steal so `bound` = \$70
- `bound < maxprofit`, so we backtrack

W = 16 `maxprofit` = \$80 `bestSet` = { 1, 1, 0, 1 }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 1, 0, 0 }

0-1 Knapsack Problem Step 8

Reject item 2 by setting `include[2]` to 0

- `profit` = \$40 + \$0 = \$40 `weight` = 2 + 0 = 2

Determine if the current node is promising:

- j: 3 k: 2 + 10 + 5 = 17 and 17 > 16 ∴ $k = 4$
- `totweight: weight +` w_3 `= 2 + 10 = 12`
- `bound: profit +` p_3 `+ (W - totweight)` (p_4 / w_4)
  = \$40 + \$50 + (16 - 12) × (\$10 / 5) = **\$98**

- \$98 > `maxprofit` and 2 < 16. <u>Promising</u>

W = 16 `maxprofit` = \$80 `bestSet` = { 1, 1, 0, 1 }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 0 }

0-1 Knapsack Problem Step 9

Steal item 3 by setting `include[3]` to 1

- `profit` = \$40 + \$50 = \$90 `weight` = 2 + 10 = 12
- \$90 > \$70 and 12 < 16, so we update `maxprofit` and `bestSet`

Determine if the current node is promising:

- j: 4 k: 12 + 5 = 17 and 17 > 16 $\therefore k = 4$
- `totweight: weight = 12`
- `bound: profit + (W - totweight) (p` $_4$ `/ w` $_4$ `)`
  = \$90 + (16 - 12) × (\$10 / 5) = **\$98**
- \$98 > `maxprofit` and 2 < 16. <u>Promising</u>

W = 16 `maxprofit` = **\$90** `bestSet` = **{ 1, 0, 1 }**

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 0, 1 }

0-1 Knapsack Problem Step 10

Steal item 4 by setting `include[4]` to 1

- `profit` = \$90 + \$10 `weight` = 12 + 5 = 17
- 17 > 16, so we immediately backtrack

W = 16 `maxprofit` = \$90 `bestSet` = { 1, 0, 1 }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 0, 1, 1 }

0-1 Knapsack Problem Step 11

Reject item 4 by setting `include[4]` to 0

- `profit` = \$90 + \$0 = \$90 `weight` = 12 + 0 = 12

Determine if the current node is promising:

- No other items to add so `bound` = \$90
- `bound = maxprofit`, so we backtrack

W = 16 `maxprofit` = \$90 `bestSet` = { 1, 0, 1 }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 0, 1, 0 }

0-1 Knapsack Problem Step 12

Reject item 3 by setting `include[3]` to 0

- `profit` = \$40 + \$0 = \$0 `weight` = 2 + 0 = 2

Determine if the current node is promising:

- j: 4 k: 2 + 5 = 7 and 7 < 16 ∴ k = 5
- `totweight: weight +` w_4 `= 7`
- `bound: profit +` p_4 `=` \$40 + \$10 = **\$50**

`bound < maxprofit`, so we backtrack

W = 16 `maxprofit` = \$90 `bestSet` = { 1, 0, 1 }

w = { 2, 5, 10, 5 } p = { 40, 30, 50, 10 } `include` = { 1, 0, 0 }

0-1 Knapsack Problem Step 13

Reject item 1 by setting `include[1]` to 0

- `profit` = \$0 `weight` = 0

Determine if the current node is promising:

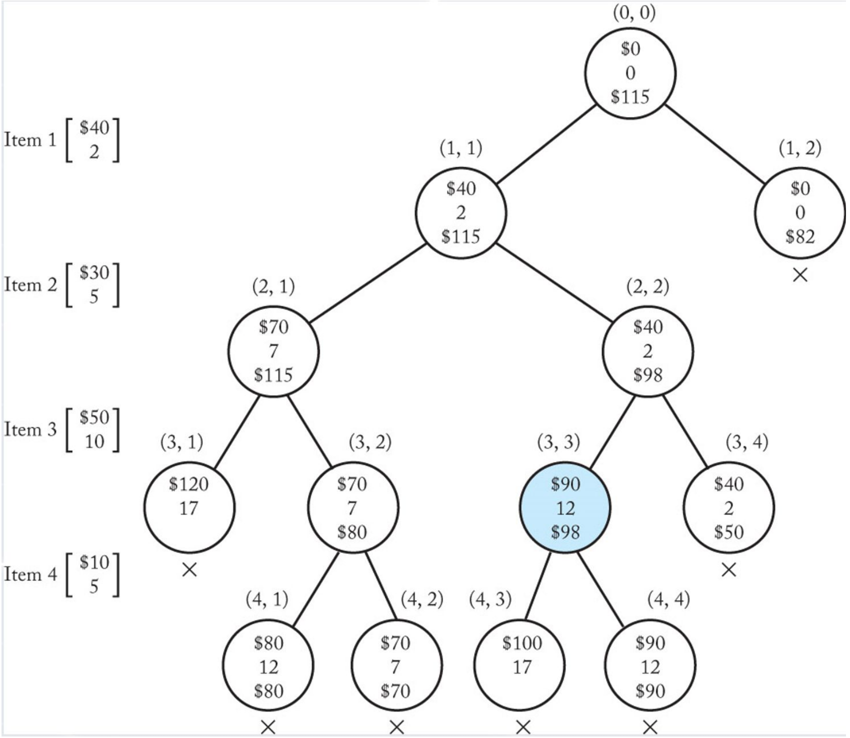
- j: 2 k: 5 + 10 + 5 = 20 and 20 > 16 $\therefore k = 4$
- `totweight`: `weight` + w_2 + w_3 = 0 + 5 + 10 = 15
- `bound`: `profit` + p_2 + p_3 + (W - *totweight*) × (p_4 / w_4)
  = \$0 + \$30 + \$50 + (16 - 15) × (10 / 5) = **\$82**

`bound < maxprofit`, so we backtrack. Nowhere to backtrack to, so we're done!

W = 16	`maxprofit` = \$90	`bestSet` = { 1, 0, 1 }
w = { 2, 5, 10, 5 }	p = { 40, 30, 50, 10 }	`include` = { 0 }

0-1 Knapsack Problem

➢ The pruned state space only has 13 nodes, whereas the entire state space tree has 31 nodes.



W = 16 *maxprofit* = \$90

0-1 Knapsack Pseudocode

```
void knapsack (index i, int profit, int weight)
{
      if (weight <= W && profit > maxprofit)
      {
            // We have found the best set reached so far.
            maxprofit = profit;
            numBest = i;
            bestset = include;
      }

      if (promising(i))
      {
            include[i + 1] = "yes";  // steal item i + 1
            knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
            include[i + 1] = "no";   // reject item i + 1
            knapsack(i + 1, profit, weight);
      }
}
```

0-1 Knapsack Pseudocode

```
bool promising (index i, int profit, int weight)
     index j, k, int totweight, double bound;

     // if weight is greater than W the bag can't hold item we just took, so return false
     if (weight >= W)
          return false;
     else
          j = i + 1;
          bound = profit;
          totweight = weight;
          while (j <= n && totweight + w[j] <= W) // grab as many items as possible
               totweight = totweight + w[j];
               bound = bound + p[j];
               j++;
          k = j;
          if (k <= n)
               bound = bound + (W - totweight) * p[k]/w[k]; // grab fraction of kth item
     return bound > maxprofit;
```

In-Class Exercise

Solve the following instance of the 0-1 Knapsack problem by drawing the pruned state space tree. Show `profit`, `weight`, `bound`, and `totweight` at each node visited.

$W = 5$

i	p_i	w_i
1	$102	3
2	$20	2
3	$60	4
4	$40	1