**Lecture Notes CS:5350** Dynamic Programming for Knapsack Problem
**Lecture 23**: Nov 19, 2019
**Scribe**: Andrew West

## 1 Dynamic Programming Recurrence for Knapsack

As discussed in the previous lecture, we can interpret the recurrence of the dynamic programming method for the knapsack problem. The recurrence is defined as $OPT(j, C)$, which is interpreted at looking over the items indexed from 1 to $j$ with available capacity $C$. There are 3 cases of this recurrence:

$$OPT(j, C) = \begin{cases} \max\{OPT(j-1, C-s_j) + v_j, OPT(j-1, C)\} & \text{for } 1 \leq j \leq n \text{ and } s_j \leq C \leq B \\ OPT(j-1, C) & \text{for} 1 \leq j \leq n \text{ and } 1 \leq C < s_j \\ 0 & \text{for } j = 0 \text{ or } C = 0 \end{cases}$$

Where $B$ is the original capacity of the knapsack given in the input to the problem. With this recurrence defined above, we can compute an optimal solution to Knapsack in $O(n * B)$.

## 2 Approach to Finding Solution to Knapsack

One approach to finding the optimal solution to the knapsack problem is to first find the optimal values of the problem, then from there determining the actual optimal solution. We can see this informally by looking at the grid created by the function of $OPT(j, C)$ described above with $j$ values from 1 to $n$ and $C$ values from 0 to $B$.

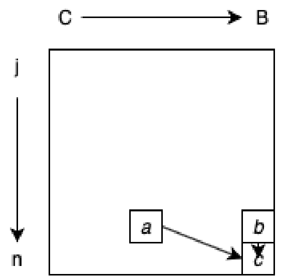

Figure 1: Grid created by $OPT(j, C)$ with grid locations specified by letters $a$, $b$, and $c$

The process for determining the values begins by looking at location $c$ on the grid. From there, we decide if item $n$ is in the optimal solution the problem. Once this is determined, we go to the corresponding location, either $a$ or $b$, based on our previous decision. We repeat this procedure until we have run out of space, $C = 0$, or when we run out of items to take into consideration, $j = 0$. The process of finding values takes linear time according to $n$. From this, we can gather the values and return a proper optimal solution to the problem. However, the one caveat is that this solution takes $O(n * B)$ time as described in the previous section which is linear in terms of $n$, but is also relative to the size of the input $B$. Now we will show an improvement that can be made to this process which will reduce overall running time of the algorithm.

## 3 Improvement to Dynamic Programming Approach for Knapsack

### 3.1 Goal

The intent of this improvement is to deliver an algorithm that has a running time of $O(n * \min\{B, V\})$, where $V = \sum_{i=1}^{n} v_i$. Although this doesn't look to be drastic reduction in the overall running time of the algorithm. One can reason that if we consider an example where the values given in the input are drastically lower, even when summed, than that of the overall capacity. The running time of the algorithm will be substantially less in terms of $V$ than in terms of $B$.

### 3.2 Process

If we consider row $j-1$ from the above example, we can see that this row can be interpreted as a function of all possible values of OPT that we can get from the inputs of items from 1 to $j-1$ and the input $C$ shown above each box of the row as shown in figure 2. In the figure, the values $V_i$ represent the appearance (and repetition) of the same OPT value for the given column in the row $j-1$; it should be reinforced that these values $V_1$ to $V_t$ are not equivalent to the values $v_i$ for $1 \leq i \leq n$ which are inputs to the knapsack problem.. The values of $V_1$ through $V_t$ are established such that $0 < V_1 < V_2 < ... < V_t \leq V$, where $t$ is also bounded by $V$. This means that one can have as many as $V$ distinct integers in the row for $j-1$.

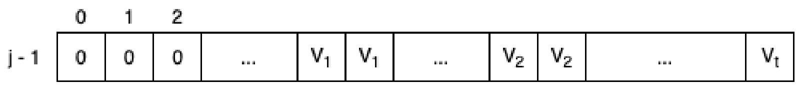

Figure 2: Row $j-1$ as interpreted using variables $V_i$ to represent the distinct OPT values

Once we have reconsidered the approach to viewing each row we continue this improvement by encoding these new ways of interpreting the rows into linked lists. As shown in figure 3, this can be done by setting each node of the linked list to hold the information of $V_i$ as well as the column index, represented by $c_i$, which states the column where value $V_i$ initially appeared in the row. This process will be done for all $1 \leq i \leq t$. At this point, we have a linked list representation for row $j-1$ that has size of at most $V$.

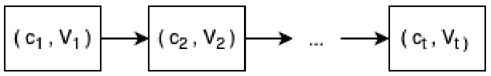

Figure 3: Linked list representation of the new j-1 row interpretation

In order to proceed with the improvement, we must answer the question of how do we get from the linked list established previously for row $j-1$, to a linked list for row $j$ in $O(V)$ time. As touched on informally in class, once this question is answered we will do this for all $n$ rows of the grid and get a complete running time of $O(n * V)$. Answering this question also leads us into the $O(n * \min\{B, V\}$ running time algorithm for this problem.

## 4 Polynomial-time Approximation Scheme (PTAS) for Knapsack

Before we get into the specifics of this approximation scheme for Knapsack, let us the definition of a polynomial-time approximation scheme (PTAS). A PTAS is defined in a previous lecture as a schematic for an algorithm that takes an input $\epsilon$ and generates a family of algorithms described as $\{A_\epsilon | \epsilon > 0\}$ where each algorithm has an approximation value $\alpha$ of $1-\epsilon$ for minimization problems and $1+\epsilon$ for maximization problems. The algorithms created by the schemes run in polynomial time relative to the input, hence the name polynomial-time approximation scheme; however, the problem lies that the running time is arbitrary on the size of $\epsilon$. Therefore, if the running time relative to epsilon is extreme, the closer we are approximation wise to the optimal solution the slower the algorithm will run.

### 4.1 Goal

For any $epsilon > 0$, we want an algorithm $A_\epsilon$ that yields a $(1-\epsilon)$-approximation for the knapsack problem where $\epsilon$ is an extra parameter to the algorithm and is not considered as part of the input. This discrepancy allows us to classify the algorithms we generate as approximation algorithms. If we didn't, we would create algorithms that were polynomial relative to the input and thus contradict the definition of an approximation algorithm.

### 4.2 Notes

1. $A_\epsilon$ runs in polynomial time in size of input to Knapsack, but its dependence on $\epsilon$ may be arbitrary. Recall things that describe the input of knapsack: $n$, $\log B$ where $B$ is the capacity of the knapsack, or $\log V$ where $V = \sum_{i=1}^{n} v_i$. Therefore as a running time, we could see anything related to $n^2$, $(\log B)^4$, $(logV)^{1/2}$, etc, and $\epsilon$ could be non-polynomial.

2. If $A_\epsilon$ returns a solution $S_\epsilon$ and $OPT$ is the optimal value of a solution, then $\sum_{i \in S_\epsilon} v_i \geq (1-\epsilon) * OPT$. (Notice: Since this is a maximization problem, we are considering the result relative to the optimal value as $\geq (1-\epsilon)$, rather than $\leq (1+\epsilon)$ as done for minimization problems.

### 4.3 Algorithm

```
1. $M \leftarrow \max_{1 \leq i \leq n} v_i$
2. $\mu \leftarrow \frac{\epsilon * M}{n}$ // scaling down factor $(\frac{v_i}{M} \leq 1) * \frac{n}{\epsilon}$, so we scale between $[0, \frac{n}{\epsilon}]$
3. $v'_i \leftarrow \frac{v_i}{\mu}$ for $i = 1$ to $n$
4. Solve KNAPSACK using the method discussed previously in lecture with $B$, $\{s_1, s_2, ..., s_n\}$,
   $\{v'_1, v'_2, ..., v'_n\}$ and return the solution.
```

## 4.4 Running Time

The running time of this improved dynamic programming algorithm is $O(n * V)$, where $V = \sum_{j=1}^{n} v'_j \leq \sum_{j=1}^{n} \frac{n}{\epsilon} = \frac{n^2}{\epsilon}$. $\therefore$ running time of the algorithm is $O(\frac{n^3}{\epsilon})$. Note: the dependency on $\epsilon$ for this algorithms in this PTAS is $\frac{1}{\epsilon}$ which is linear. This actually classifies the PTAS in a specific subset called Fully Polynomial-time Approximation Scheme (FPTAS). FPTAS is classified as PTAS that has a linear dependence on $\epsilon$.

**Theorem.** *Let $S_\epsilon$ be the solution returned by $A_\epsilon$ and let $OPT$ be the value of an optimal solution. Then, $\sum_{i \in S_\epsilon} v_i \geq (1-\epsilon) * OPT$.*

*Proof.* If we consider some arbitrary original value given to the problem $v_i$ and the corresponding $v'_i$ value that we get from the algorithm. We also know that since $v'_i$ is set as the floor of the value $\frac{v_i}{\mu}$. This implies that:

$$v'_i \leq \frac{v_i}{\mu} < v'_i + 1 \text{ which leads to } v_i - \mu < \mu * v'_i$$

The below step of $\mu \sum_{i \in S_\epsilon} v'_i \geq \mu \sum_{i \in O} v'_i$ is allowed because $S_\epsilon$ is the $OPT$ of of reduced problem; therefore, every other solution is greater than or equal to this solution. With this information we can complete our proof:

$$\begin{aligned}
\sum_{i \in S_\epsilon} v_i &\geq \sum_{i \in S_\epsilon} \mu v'_i \\
&= \mu \sum_{i \in S_\epsilon} v'_i \\
&\geq \mu \sum_{i \in O} v'_i, \text{ where O is the optimal solution of original problem} \\
&> \sum_{i \in O} (v_i - \mu) \\
&= \sum_{i \in O} v_i - \sum_{i \in O} \mu \\
&= OPT - \sum_{i \in O} \frac{\epsilon * M}{n}, \text{ recall that } \mu = \frac{\epsilon * M}{n}, \text{ where } M = \max_{1 \leq i \leq n} v_i \text{ and } n \text{ is number of items.} \\
&\geq OPT - n * \frac{\epsilon * M}{n}, \text{ because } O \text{ can have at-most } n \text{ items} \\
&= OPT - \epsilon * M \\
&\geq OPT - \epsilon * OPT, \text{ since } OPT \geq M \\
&= (1-\epsilon) OPT.
\end{aligned}$$

□