CS320 The knapsack problem
Search and Discrete Optimization

- **Discrete Optimization Problem (S,f)**
  - S: space of feasible solutions (satisfying some constraints)
  - f : S → R
    - function that maps all feasible solutions on the real numbers
  - **Objective**: find an optimal solution \( x_{opt} \) such that
    \[
    f(x_{opt}) \leq f(x) \text{ for all } x \text{ in } S
    \]
    or
    \[
    f(x_{opt}) \geq f(x) \text{ for all } x \text{ in } S
    \]
- **Search application domains**
  - planning and scheduling
  - VLSI layout
  - pattern recognition
Integer Linear Problems

• Given a vector of variables $x$ and a set of linear constraints on these variables, expressed as a matrix vector inequality
  \[ Ax \leq b \]

• optimize (minimize or maximize) a linear cost function in these variables, expressed as a vector inner product $cx$. 
Simplex algorithm: Constraints form a convex space
Simplex algorithm: maximizing $cx = \text{moving it as far as possible North East}$
Example: Knapsack problem

• Is a 0/1 ILP problem
• Given
  – n objects - each with weight \( w_i \) and profit \( p_i \)
  – a knapsack with capacity \( M \)
• Determine a subset of the objects such that
  – total weight \( \leq M \)
  – total profit \textbf{maximal}
Knapsack is an ILP

• In the case of the knapsack problem the variables $x_i$ can only take values 0 and 1,
• there is only one constraint $Wx \leq M$
• the goal is to maximize $Px$
• where $W$ and $P$ are two size n vectors of positive numbers.
Example

- $W = 5 4 6 1$
- $P = 7 8 9 4$
- $M = 10$
Example

- \( W = 5 4 6 1 \)
- \( P = 7 8 9 4 \)
- \( M = 10 \)

- Choice vector?
Example

- $W = 5 \ 4 \ 6 \ 1$
- $P = 7 \ 8 \ 9 \ 4$
- $M = 10$

- Choice vector: $1 \ 1 \ 0 \ 1$
- Profit: 19
State space for knapsack

- State space is a tree
  - Information stored per state
    1. Capacity left. 2. Profit gathered. 3. Objects taken
- Root: level 0: no object has been considered
- At level i
  - Consider object i
  - Two possible states
    - take object
    - don’t take object
- Size of the state space
  - exponential
state space

• State stores **Capacity Left, Profit Gathered.**
  – Possibly: Objects taken so far.

• At level 0 (the root of the state space) no object has been considered.

• To go from level i to level i+1, consider object i.

• There are two possible next states:
  – take object i, or
  – do not take object i.

• Size state space is exponential in n.
Divide and Conquer (DivCo) solution

- Exhaustive recursive search technique
- Hopelessly inefficient, but simple

```c
knapdc (M, i)
// M: capacity left, i: level, return optimal profit from level i down
if (M < W[i])
    if (i < n) return knapdc(M, i+1)  // don’t take
    else return 0
else if (i < n)
    take = knapdc(M-W[i], i+1) + P[i]
    leave = knapdc(M, i+1)
    return max(take, leave)
else return P[i];
```
Dynamic Programming Solution

• Go through the state space bottom-up
  – select 1 object, then 2 objects, ......., all objects
  – very much like make change
  – use solutions of smaller sub-problem to efficiently compute solutions of larger one

<table>
<thead>
<tr>
<th>1</th>
<th>1..2</th>
<th>1..3</th>
<th>1..i-1</th>
<th>1..i</th>
<th>objects considered</th>
</tr>
</thead>
</table>

\[ S_0: \text{profit if object i not taken: } V_{i-1}[x] \]
\[ S_1: \text{profit if object i taken: } V_{i-1}[x-Wi] + P_i \]

\[ V_i[x] = \max(S_0,S_1) \]
\[ 0 \leq x \leq M \]
Dynamic programming main loop

PrevV = vector[0..M], all elements 0;
V = vector[0..M];
for i = 1 to n { // number of objects
    for j = 0 to M { // capacity
        if (j < W[i])  V[j] = PrevV[j]
        else
            S0 = PrevV[j];
            S1 = PrevV[j-W[i]] + P[i];
            V[j] = max[S0,S1];
    }
    PrevV = V;
}
return V[M]
Solution / choice Vector

• Notice that this solution does not return the choice vector, only the optimal profit value.

• In order to recreate the choice vector, the whole table $T$ of vectors needs to be kept. (or does it?)

• After it is filled the choice vector can be retrieved backwards.
  – Starting at $T[n,M]$ the choice (take or don't take) for object $n$ can be determined, and also which element of the previous vector needs to be examined.
  – Very much like LCS
**Example (table transposed)**

- \( W = 5 \ 4 \ 6 \ 1 \quad P = 7 \ 8 \ 9 \ 4 \quad M = 10 \)

<table>
<thead>
<tr>
<th>cap</th>
<th>obj</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 10</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0 7 7 7 7 7 7 7 7 7 7</td>
</tr>
</tbody>
</table>
Example

- \( W = 5 \ 4 \ 6 \ 1 \quad P = 7 \ 8 \ 9 \ 4 \quad M = 10 \)

<table>
<thead>
<tr>
<th>cap</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>7</th>
<th>7</th>
<th>7</th>
<th>7</th>
<th>7</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1:2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
Example

- \( W = 5 \ 4 \ 6 \ 1 \quad P = 7 \ 8 \ 9 \ 4 \quad M = 10 \)

<table>
<thead>
<tr>
<th></th>
<th>cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>0 1 2 3 4 5 6 7 8 9 10</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0 7 7 7 7 7 7</td>
</tr>
<tr>
<td>1:2</td>
<td>0 0 0 0 8 8 8 8 8 15 15</td>
</tr>
<tr>
<td>1:3</td>
<td>0 0 0 0 8 8 9 9 9 15 17</td>
</tr>
</tbody>
</table>
Example

- \( W = 5 \ 4 \ 6 \ 1 \quad P = 7 \ 8 \ 9 \ 4 \quad M = 10 \)

<table>
<thead>
<tr>
<th>cap</th>
<th>( 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>.obj</td>
<td>1</td>
</tr>
<tr>
<td>1:2</td>
<td>0 0 0 0 8 8 8 8 8 15 15</td>
</tr>
<tr>
<td>1:3</td>
<td>0 0 0 0 8 8 9 9 9 15 17</td>
</tr>
<tr>
<td>1:4</td>
<td>0 4 4 4 8 12 12 13 13 15 19</td>
</tr>
</tbody>
</table>
Example

- \( W = 5 4 6 1 \quad P = 7 8 9 4 \quad M = 10 \)

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 \\
1:2 & 0 & 0 & 0 & 0 & 8 & 8 & 8 & 8 & 8 & 15 & 15 & \\
1:3 & 0 & 0 & 0 & 0 & 8 & 8 & 9 & 9 & 9 & 15 & 17 & \\
1:4 & 0 & 4 & 4 & 4 & 8 & 12 & 12 & 13 & 13 & 15 & 19 & \text{take 4}
\end{array}
\]
Example

- $W = 5 \ 4 \ 6 \ 1 \quad P = 7 \ 8 \ 9 \ 4 \quad M = 10$

1 0 0 0 0 0 7 7 7 7 7 7
1:2 0 0 0 0 8 8 8 8 8 8 15 15
1:3 0 0 0 0 8 8 9 9 9 15 17 not 3
1:4 0 4 4 4 8 12 12 13 13 15 19 take 4
Example

- $W = 5 \ 4 \ 6 \ 1$  
- $P = 7 \ 8 \ 9 \ 4$  
- $M = 10$

\[
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
1:2 & 0 & 0 & 0 & 0 & 8 & 8 & 8 & 8 & 8 & 15 & 15 \\
1:3 & 0 & 0 & 0 & 0 & 8 & 8 & 9 & 9 & 9 & 15 & 17 \\
1:4 & 0 & 4 & 4 & 4 & 8 & 12 & 12 & 13 & 13 & 15 & 19 \\
\end{array}
\]

- take 1
- take 2
- not 3
- take 4
DP and bands

• Band: area in a column vector in DP array where value does not change
  – How many bands in column 1?
  – How many in col₂, at most in col₂?
  – How would you represent a band?

• Observation: you don’t need to store value for all capacities, just for bands
  – Faster and less storage
  – Diminishing returns
  – Still worth it
Example

- \( W = 5 \ 4 \ 6 \ 1 \quad P = 7 \ 8 \ 9 \ 4 \quad M = 10 \)

  1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 7 \ 7 \ 7 \ 7 \ 7 \ 7

  1:2 \ 0 \ 0 \ 0 \ 0 \ 8 \ 8 \ 8 \ 8 \ 8 \ 15 \ 15

  1:3 \ 0 \ 0 \ 0 \ 0 \ 8 \ 8 \ 9 \ 9 \ 9 \ 15 \ 17

  1:4 \ 0 \ 4 \ 4 \ 4 \ 8 \ 12 \ 12 \ 13 \ 13 \ 15 \ 19
Example

- \( W = 5 \quad 4 \quad 6 \quad 1 \quad P = 7 \quad 8 \quad 9 \quad 4 \quad M = 10 \)

1:2

\[
\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 7 \\
0 & 0 & 0 & 0 & 8 & 8 & 8 \\
15 & 15 & 15 & 15 & 15 & 15 & 15
\end{array}
\]

1:3

\[
\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 0 & 8 & 8 \\
0 & 0 & 0 & 0 & 8 & 9 & 9 \\
15 & 15 & 15 & 15 & 15 & 17 & 17
\end{array}
\]

1:4

\[
\begin{array}{ccccccc}
1 & 0 & 0 & 0 & 0 & 8 & 12 \\
4 & 4 & 4 & 4 & 8 & 12 & 13 \\
15 & 15 & 15 & 15 & 15 & 15 & 19
\end{array}
\]

5,7
Example

- \( W = 5 \quad 4 \quad 6 \quad 1 \quad P = 7 \quad 8 \quad 9 \quad 4 \quad M = 10 \)

\[
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
1:2 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

- critical points: 4,8 6,9 9,15 10,17

\[
\begin{array}{cccccc}
1:3 & 0 & 0 & 0 & 0 & 8 \\
1:4 & 0 & 4 & 4 & 4 & 8
\end{array}
\]

- critical points: 1,4 4,8 5,12 7,13 9,15 10,19
Critical points from \(i-1\) to \(i\):

\[ V_i(j) = \max(V_{i-1}(j), V_{i-1}(j-w_i) + P_i) \]

\[ V_{i-1}(j) \] previous point set
Critical points

• Critical points from $i-1$ to $i$:

\[ V_i(j) = \max(V_{i-1}(j), V_{i-1}(j-w_i) + P_i) \]

- $V_{i-1}(j)$ previous point set
- $V_{i-1}(j-w_i) + P_i$ previous point set
  shifted up $P_i$
- shifted right $w_i$
Critical points

• Critical points from i-1 to i:

\[ V_i(j) = \max(V_{i-1}(j), V_{i-1}(j-w_i) + P_i) \]

- \( V_{i-1}(j) \): previous point set
- \( V_{i-1}(j-w_i) + P_i \): previous point set shifted up \( P_i \)
- \( w_i \): shifted right
- \( \max \): take top points

dominated point
Further space optimization

• Keeping the whole table is space consuming.
• Optimization: only keep the middle column and determine the point on the backwards path in that middle column.
• Then recursively recompute two sub problems with in total half the size of the original, so the time complexity is at most twice the time complexity of the original algorithm, but space complexity is now \(O(M)\) instead od \(O(M^*n)\).

Further time optimization

• The dynamic programming solution creates more elements of the table than it uses. Eg it uses only 1 element of the last column, and only 2 of the one but last.

• We can partially build the dynamic programming table ``on demand'' in a top down fashion.

• We initialize the table with eg -1, and only evaluate a table element if we need it and it has not been computed yet.

• This technique is called memoization.
Bounds for knapsack

• Strengthening / simplifying assumption
  – objects sorted according to profit/weight ratio
  – it's smarter to take the diamond than the silverwork

• In a certain state, with a certain capacity left
  – Lowerbound (lwb) = sum of profit of objects with highest profit/weight ratio, s.t sum of corresponding weights does not exceed capacity. **lwb is achievable.**
  – Upperbound (upb) = lwb + fraction of profit of next object s.t. capacity left = zero. **upb is not achievable** if an actual fraction is taken. But it can be used to prune the search space.
Lowerbound Upperbound

• lwb is the **GREEDY** Solution
  – Computing the greedy solution is fast: $O(n)$

• Question: is the greedy solution optimal?
Lwb  Upb

• **lwb is the GREEDY Solution**
  – Computing the greedy solution is fast: $O(n)$
• Question: is the greedy solution optimal?
  – No
  – counter example shows it:
  – but LWB and UPB together can be used to prune the search space
Branch and Bound: B&B

• Branching rule
  – How to move to new states from a current state

• Bounds
  – How to compute an upper bound $U$, lower bound $L$ in state $S$

• Selection rule
  – Which state to branch to next
    • DEPTH first, BREADTH first, BEST first
    • Parallel approach: take a number of branches at the same time

• Elimination rule
  – Infeasible - cannot lead to a solution
  – Bounded - eliminate $S$ if upper bound of $S <$ current best lower bound
    (best solution so far for 0/1 ILP problem)
  – Dominated - eliminate $S$ if it is DOMINATED by another state $Q$
Depth First Branch & Bound (DFBB)

• Searching the state space
  – Traverse the state space depth first (left to right)
  – Maintain the best solution BS = best LWB found so far
  – Before going into a new sub-tree
    • Compute upper bound Upb of that sub-tree
    • If Upb < BS, prune that sub-tree (do not evaluate the tree) Why does it work?
  – DFBB is much faster than Div&Co
B&B for knapsack

• Branch
  – Two new states: take next object or don’t take

• Bound
  – Obtained by Greedy solution

• Select
  – Breadth first allows other options to be discovered soon

• Eliminate ...
B&B eliminate

• Eliminate
  – Infeasible: Weight > Capacity
  – Bounded: Upb < BS (current global lower bound)
  – **BFS allows for elimination of dominated nodes:**
    Two states S and Q on the same level in the search tree, Q dominates S if
    it has the same or more capacity left and more profit gathered
Knapsack Example

- Capacity M = 7, Number of objects n = 3
- $W = [5, 4, 3]$  $P = [10, 7, 5]$  (ordered in $P/W$ ratio)

CL : Capacity Left
PG : Profit Generated
L,U : Bounds of the state
Knapsack B&B optimizations

• When taking an object you don’t need to recompute upb and lwb
• If we don’t take we can subtract Pi from upb and add Wi to capLeft and restart the upb computation where we left off last time
• If capLeft < min(weight of the remaining objects): stop computing
Smart Domination

• Domination
  – If (capLeft1 > capLeft2 and profit1 > profit2) kill task2
  – Naïve implementation compares all pairs: O(#tasks²)

1: if no tasks at level i dominate each other, what about all takes? What about all don’t takes?

2: if the tasks are sorted by decreasing capLeft, last task in queue Cl,Pl, next task considered Cn,Pn
  • We know Cn<=Cl so if Pn<Pl kill Cn,Pn
  • Keeping queue sorted: merge while doing the dominance test, now is O(#tasks)