CS320
More on the knapsack problem
Search and Discrete Optimization

• Discrete Optimization Problem \((S,f)\)
  – \(S\): space of feasible solutions (satisfying some constraints)
  – \(f: S \rightarrow \mathbb{R}\)
    • function that maps all feasible solutions on the real numbers
  – Objective: find an optimal solution \(x_{\text{opt}}\) such that
    \[ f(x_{\text{opt}}) \leq f(x) \text{ for all } x \text{ in } S \]
    or \[ f(x_{\text{opt}}) \geq f(x) \text{ for all } x \text{ in } S \]
  – 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 $cx$ is 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 $\leq M$
  – maximum total profit
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.
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 next states
    • take object
    • don’t take object

• Size of the state space
  – exponential
Divide and Conquer 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
  – use solutions of smaller sub-problem to efficiently compute solutions of larger one

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

\[
V_i[x] = \max(S_0, S_1)
\]

\[
0 \leq x \leq M
\]

objects considered
Sparse representation: 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 (consider hybrid with memoization)
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 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

- \( W = 5461 \quad P = 7894 \quad M = 10 \)

1:2
- 0 0 0 0 8 8 8 8 15 15

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

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

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

1:2

$$
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8 & 8 & 8 \\
0 & 0 & 0 & 8 & 9 & 9 \\
0 & 4 & 4 & 8 & 12 & 12 \\
\end{array}
$$

Critical points: 4,8 6,9 9,15 10,17

1:3

$$
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8 & 8 & 8 \\
0 & 0 & 0 & 8 & 9 & 9 \\
0 & 4 & 4 & 8 & 12 & 12 \\
\end{array}
$$

Critical points: 4,8 5,12 7,13 9,15 10,19

1:4

$$
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8 & 8 & 8 \\
0 & 0 & 0 & 8 & 9 & 9 \\
0 & 4 & 4 & 8 & 12 & 12 \\
\end{array}
$$

Critical points: 1,4 4,8 5,12 7,13 9,15 10,19
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
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 right \( w_i \)
- Lifted up \( P_i \)
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 right \( w_i \)
- lifted up \( P_i \)

max: take top points
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. E.g., 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 profits 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: $O(n)$
  But not optimum solution!

(But usually close.)
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**
  - How to prune nodes out of the stat space
    - **Infeasible** - cannot lead to a solution
    - **Bounded** - eliminate $S$ if upper bound of $S <$ current best lower bound solution so far
    - **Dominated** - eliminate $S$ if it is DOMINATED by another state $Q$
      S dominated by $Q$ if we can prove that $Q$ will produce same or better solution
Knapsack Example

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

\[\begin{align*}
\text{CL} &: 7, \quad \text{PG}: 0 \\
\text{L} &: 10, \quad \text{U}: 13.5 \\
\text{CL} &: 7, \quad \text{PG}: 0 \\
\text{L} &: 12, \quad \text{U}: 12 \\
\text{CL} &: 2, \quad \text{PG}: 10 \\
\text{L} &: 10, \quad \text{U}: 13.5 \\
\text{CL} &: 7, \quad \text{PG}: 0 \\
\text{L} &: 5, \quad \text{U}: 5 \\
\text{CL} &: 3, \quad \text{PG}: 7 \\
\text{L} &: 12, \quad \text{U}: 12 \\
\text{CL} &: 2, \quad \text{PG}: 10 \\
\text{L} &: 10, \quad \text{U}: 13.3 \\
\text{CL} &: 3, \quad \text{PG}: 7 \\
\text{L} &: 7, \quad \text{U}: 7 \\
\end{align*}\]

- CL : Capacity Left
- PG : Profit Generated
- L,U : Bounds of the state

**Leaf**
- CL: 7, PG: 0
  L: 10, U: 13.5

**Optimum**
- CL: 3, PG: 7
  L: 12, U: 12

**Bounded**
- CL: 7, PG: 0
  L: 5, U: 5

**Infeasible**
B&B for knapsack

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

• Bounds
  – Obtained by Greedy solution

• Select
  – Breadth first allows good lower bounds to be discovered soon, and thus large parts of the state space to be pruned
B&B eliminate

• Eliminate
  – Infeasible: Weight > Capacity left
  – Bounded: Upb < 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
Smart Domination

Domination

– If (capLeft1 > capLeft2 and profit1 > profit2) kill task2
– Naïve implementation compares all pairs: $O(#\text{tasks}^2)$

1: if no tasks at level $i$ dominate each other, what about all takes? What about all don’t takes? So keep all takes separate from all don’t takes. Then merge them and delete dominated tasks.

2: if the tasks are sorted by decreasing capLeft, last task enqueued $C_l,P_l$, next task considered $C_n,P_n$
   • We know $C_n \leq C_l$ so if $P_n < P_l$ kill $C_n,P_n$
   • Keeping queue sorted: merge while doing the dominance test, now is $O(#\text{tasks})$