Lecture outline: greedy algorithms II

- Knapsack
- Scheduling
  - minimizing time
  - with deadlines
Greedy algorithms

- Problem:
  - Optimize cost (w.r.t. objective function)
  - Solutions are composed from components (candidates)
  - A validity check function
  - A feasibility check function (checks whether the partial solution can be completed to a valid solution)

- Greedy algorithm
  - goes step by step
  - maintains partial solution and possible extensions
  - selects the best possible extensions (how?)
Greedy verbatim

Greedy() {
    initialize_partial_solution();
    while(there are extensions) {
        do {
            choose the best extension;
            check its validity;
            while(it’s not valid);
            add that extension to get a new partial solution
            see if the problem is solved, if so return;
        }
    }
}
Knapsack

- $n$ objects, weights $w_i$, values $v_i$
- a knapsack can hold maximal weight of $W$
- maximize the $\$ value of what we fit

(2lb, $5)  (3lb, $2)  (5lb, $100)  (12lb, $6)

max of 9lb
Knapsack: much simpler case?

• or is it?
• $n$ objects, weights $w_i$, values $v_i$, $w_i = v_i$
• a knapsack can hold maximal weight of $W = \frac{1}{2} \sum_{i=1}^{n} w_i$
• can we fill it up full?
• No polynomial time algorithm is known!

\[(2\text{lb}) \quad (3\text{lb}) \quad (5\text{lb}) \quad (12\text{lb})\]

max of $11\text{lb}$
Knapsack: breakable objects

- $n$ objects, weights $w_i$, values $v_i$
- a knapsack can hold maximal weight of $W$
- we can break objects, fractions $x_i \in (0, 1)$
- maximize $\sum_{i=1}^{n} x_i v_i$ subject to $\sum_{i=1}^{n} x_i w_i \leq W$

\[
\begin{align*}
  x_1 &= 1 & (2\text{lb}, \$5) \\
  x_2 &= 2/3 & (3\text{lb}, \$2) \\
  x_3 &= 1 & (5\text{lb}, \$100) \\
  x_4 &= 0 & (12\text{lb}, \$6)
\end{align*}
\]

$\text{max of 9lb}$
**Knapsack verbatim: greedy algorithm**

```cpp
float knapsack(const vector<float>& w, const vector<float>& v, vector<float>& x, float wmax) {
    for(int i=0; i<x.size(); ++i) x[i] = 0;
    float weight = 0, value = 0;
    while(weight<wmax) {
        i = best_remaining_object();
        if(weight+w[i]<=wmax) {
            x[i] = 1; weight+=w[i]; value+=v[i];
        } else {
            x[i] = (wmax-weight)/w[i]; weight+=w[i]; value+=v[i];
        }
    }
    return value;
}
return value;
```
Knapsack: strategies

- Choose the most valuable
- Sequence: 3, 4.

\[ x_1 = 0 \quad x_2 = 0 \quad x_3 = 1 \quad x_4 = 1/3 \]

(2lb, $5) \quad (3lb, $2) \quad (5lb, $100) \quad (12lb, $6)

max of 9lb, value $102
Knapsack: strategies

- Choose the lightest
- Sequence: 1, 2, 3.

\[ x_1 = 1 \quad x_2 = 1 \quad x_3 = 4/5 \quad x_4 = 0 \]

(2lb, $5) \quad (3lb, $2) \quad (5lb, $100) \quad (12lb, $6)

max of 9lb, value $87
Knapsack: strategies

- Choose the best value per unit weight (that is $v_i/w_i$)
- Sequence: 3, 1, 2.
- Provably the best strategy

\[
\begin{align*}
x_1 &= 1 \\
&\text{(2lb, $5) } \\
x_2 &= 2/3 \\
&\text{(3lb, $2) } \\
x_3 &= 1 \\
&\text{(5lb, $100) } \\
x_4 &= 0 \\
&\text{(12lb, $6) }
\end{align*}
\]

max of 9lb, value $106.33$
Knapsack: greedy best theorem

- Choose the best value per unit weight (that is $v_i/w_i$)
- **Theorem:** If objects are selected in order of decreasing $v_i/w_i$, then the algorithm knapsack finds an optimal solution.

- **Proof outline:** if all the weights are 1 then it is optimal. Otherwise, order them so that $v_i/w_i$ decrease, and consider $V(x) = \sum_{i=1}^{n} x_i v_i$.

- For some other choice $y$ prove that (choose special $j$)

\[
V(x) - V(y) = \sum_{i=1}^{n} (x_i - y_i) v_i = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i} \geq 0.
\]

\[
V(x) - V(y) \geq \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_j}{w_j} = \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0.
\]
Scheduling: min time in system

- A single server
- $n$ customers
- service times $t_i$
- would like to minimize average time that a customer spends in the system
- since $n$ is fixed we can just minimize $T(p) = \sum_{i=1}^{n} \sum_{k=1}^{i} t_{p_i}$. 
Scheduling: min time in system

• Claim: serving in order of increasing service time is optimal
• First of all, $T(p) = \sum_{i=1}^{n} \sum_{k=1}^{i} t_{pk} = \sum_{k=1}^{n} (n - k + 1) t_{pk}$.
• $p$ is a permutation
Scheduling: min time in system

- If not in order, then we find \( i < j \) such that \( t_{p_i} > t_{p_j} \)
- Exchange them!
- Total cost will be better
- Implementation trivial: running time \( O(n \log n) \)
Scheduling with deadlines

- A single server
- \( n \) jobs, service time is the same (unit).
- At any time just one job
- Deadlines \( d_i \): profit \( g_i \) is earned only if \( t_i \leq d_i \).
- \( t_i = \infty \) if it’s not executed

Example: \( n = 5 \)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( g_i )</td>
<td>40</td>
<td>15</td>
<td>20</td>
<td>60</td>
<td>30</td>
</tr>
<tr>
<td>( d_i )</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Scheduling with deadlines

- Greedy strategy: take the fattest job still available as long as the set stays feasible
- Feasible set allows at least one feasible sequence of execution
- **Lemma**: set of $k$ jobs if feasible if and only if indexed in order of non-decreasing deadlines the sequence $1, \ldots, k$ is feasible.
  - **Only If Proof**: if sequence is not feasible then $r > d_r$ for some $r$, but then there are at least $r$ jobs whose deadlines are before or on $t = r - 1$, so that the set is not feasible.
  - **Theorem**: greedy algorithm is optimal. Proof in the book (p.208)
Scheduling with deadlines: slow algorithm

void sequence(const vector<int>& d, vector<int>& j) {
    assert(j.empty());
j.push_back(0);
    for(i=1; i<d.size(); ++i) {
        r = j.size()-1;
        while( r>=0 && d[j[r]]>max(d[i], r) )
            --r; // while usefully shiftable
        if(d[i]>r) { // r contains first non-shiftable
            j.push_back(-1);
            for(int m=j.size()-1; m>r; --m)
                j[m+1] = j[m];
            j[r+1] = i;
        }
    }
} // so, how slow is this?
Scheduling with deadlines faster

• The same algorithm
• Different feasibility check:
• Start with an empty schedule of length $n$
• Schedule a job $i$ at time

$$t_i = \max \{ k : k \leq \min(d_i, n - 1) \text{ and } k \text{ is free} \}$$
• To implement define $n_t = \max \{ k : k \leq t \text{ and } k \text{ is free} \}$
• Two slots in the same set if their $n_t$ are the same
• Use disjoint sets to maintain “the first available slot before or on $t$”
Scheduling with deadlines: faster algorithm

- Algorithm is in the book p. 214
- Need to maintain array of earliest available times since labels of the disjoint set do not guarantee the minimality when merging them
  - find the desired time (deadline or $n$)
  - get its label $l$ and the earliest available
  - if there is a slot insert yourself in there and merge set with label $l$ and the one immediately to the left
  - finally, compress the solution
- at most $2n$ finds and $n$ merges, so that without sorting we have $O(n\alpha(2n, n))$ where $\alpha$ is that slow growing function that is $< 4$. 