Contents Lecture 6

- Weighted interval scheduling
- Recursion or Iteration
- Subset sums and knapsacks
- RNA: dynamic programming over intervals
- DNA: sequence alignment (lab 5)
Weighted interval scheduling

- Recall interval scheduling:
  - Input is a set of requests $r_i$ with start $s_i$ and finish $f_i$ times
  - Two requests conflict if their intervals overlap
  - We want to select the maximum number of nonconflicting requests
  - We have seen this can be done with a greedy algorithm which selects as next request the request with earliest finish time and which does not conflict with any already selected request

- In the weighted interval scheduling problem each request has a value $v_i$
- Now we want to maximize the sum of values $v_i$ of the selected requests
- If anybody can figure out a greedy algorithm for that, congratulations!
- We will take an approach which at first may seem to be extremely slow
Assume the requests are named such that $f(1) \leq f(2) \leq \ldots \leq f(n)$

Each request has a value $v(i)$

In interval scheduling we started with $r_1$

Now we will instead consider each request starting with the last, $r_n$

Let $T$ be an optimal schedule and $OPT(n)$ be the sum of the selected $v(i)$ from requests $r_1, r_2, \ldots, r_n$.

We will make our own optimal schedule $S$, also with value $OPT(n)$
Our algorithm

- \( p(n) \) is the request with maximum \( f(i) \) which does not conflict with \( r_n \)
- We need to decide if \( r_n \) should be selected or not, so we have two cases:
  1. \( r_n \) is selected: in this case \( OPT(n) = v(n) + OPT(p(n)) \)
  2. \( r_n \) is not selected: in this case \( OPT(n) = OPT(n - 1) \)
- To decide which case to use, we evaluate both and see which is best!

function \( OPT(n) \)

\[
\begin{align*}
\text{if} \ n &= 0 \ \text{then} \\
\text{return} \ 0 \\
\text{a} &\leftarrow v(n) + OPT(p(n)) \\
\text{b} &\leftarrow OPT(n - 1) \\
\text{return} \ \max(a, b)
\end{align*}
\]

- This algorithm will recompute \( OPT(i) \) a huge number of times
- If for all \( i \) \( p(i) = i - 1 \) then \( OPT \) will be called \( 2^n \) times
- This seem to be useless but we can fix the problem. How?
Memoization

- Note that the value of $OPT(i)$ obviously never changes
- So when we have computed $OPT(i)$ we can remember the value
- We save it in an array and use it next time $OPT(i)$ is needed
- Let $m[1], m[2], \ldots, m[n] = -\infty$ initially

```plaintext
function OPT(n)
    if $n = 0$ then
        return 0
    else if $m[n] = -\infty$ then
        $a \leftarrow v(n) + OPT(p(n))$
        $b \leftarrow OPT(n - 1)$
        $m[n] \leftarrow \max(a, b)$
    return $m[n]$
```

- Remembering values like this is called **memoization**
Avoiding recursion

- Recursion is beautiful and simplifies life but the cost, as we know, is that it takes time to make function calls
- We can just produce the array $m$ directly

```plaintext
procedure make_table(n)
i ← 1
while $i \leq n$
    $a ← v(i) + m[p(i)]$
    $b ← m[i - 1]$
    $m[i] ← \max(a, b)$
    $i ← i + 1$

function OPT(n)
    return $m[n]$
```
Dynamic programming

- What we just saw is an example of **dynamic programming**
- We express a solution in terms of solutions to **overlapping** smaller problems
- Note the difference between divide and conquer where we solve **independent** small problems
- The power of dynamic programming comes from avoiding recomputing already solved subproblems
- We will see several examples how we should think when using dynamic programming
- This technique was invented in the 1950’s by Richard Bellman
- In this context programming is not ”computer programming” but instead finding an optimal solution, or, program, to achieve typically a military scheduling problem (as linear programming in mathematics)
- Bellman wanted a fancy name so he could continue working on this with funding from the US defence
The subset sum problem: a variant to the knapsack problem

- How to bring as much hand luggage as possible on a flight
- You are allowed to bring at most $W$ kilograms of hand luggage
- You have $n$ items, and each item $i$ has weight $w_i$
- Select a subset $S$ of these items so that
  - $T = \sum_{i \in S} w_i \leq W$
  - $T$ is as large as possible
- No greedy algorithm is known for this problem
- How can we use dynamic programming here???
- We need to consider both weights $w_i$ and $W$
- If we select item $i$ with weight $w_i$ we have $W - w_i$ left...
- In the normal knapsack problem, each item has both a value and a weight but in principle these problems are similar (we have weight = value)
Dynamic programming approach

- \( T = \sum_{i \in S} w_i \leq W \), maximize \( T \)
- Consider an optimal solution which can choose from \( n \) items for an allowable weight \( W \)
- Either item \( n \) is included or it is not. Excluding \( n \) may be due to \( w_n > W \) or because it is simply better to skip it
- For instance if the items have weights \( \{3, 7, 8\} \) and \( W = 10 \) it is better to skip the 8 kg item
- If we then select the 7 kg item, we clearly have \( W - 7 \) kg left

\[
OPT(n, W) = \begin{cases} 
0, & n = 0 \\
OPT(n - 1, W), & w_n > W \\
\max(OPT(n - 1, W), \ w_n + OPT(n - 1, W - w_i)), & \text{otherwise}
\end{cases}
\]
Running time

- This is not polynomial time!
- The running time is dependent on the value of $W$!
RNA is a string $B = b_1 b_2 \ldots b_n$ over the alphabet $\{C, G, A, U\}$.

Compared with DNA it is single stranded and due to this there are secondary structures when it connects to itself according to certain rules.
Secondary structures

A secondary structure is a matching $S = \{(b_i, b_j)\}$

A pair is shown as two molecules connected with a dashed edge

A molecule can only pair with at most one other molecule

$A$ and $U$ can pair, and $C$ and $G$ can pair

Pairing molecules cannot be too close: $(b_i, b_j) \in S \Rightarrow i < j - 4$

No crossing pairs: if $i < j < k < l$ then $(b_i, b_k)$ and $(b_j, b_l)$ cannot both be in $S$

The problem is to find an $S$ with a maximal number of pairs
An \( OPT(i, j) \) function

\[
\cdots \quad U \quad C \quad G \quad A \quad G \quad U \quad G \quad A \quad G \quad C \quad \cdots
\]

\( b_t \) \hspace{2cm} \( b_j \)

- Initially called with \( OPT(1, n) \)
- Then for some arbitrary call we have \( OPT(i, j) \)
- \( b_j \) is our right most symbol, or molecule.
- When \( b_j \) pairs with some \( b_t \) the noncrossing condition splits up our remaining interval in two halves:
  - \( b_i \ldots b_{t-1} \)
  - \( b_{t+1} \ldots b_{j-1} \)
An $OPT(i, j)$ function

- Case 1: $i \geq j - 4$: $OPT(i, j) = 0$
- Case 2: There is no available molecule to create a pair for $b_j$: $OPT(i, j) = OPT(i, j - 1)$
- Case 3: Taking rules used in Cases 1 and 2 into account, a $t$ is selected which maximizes:
  
  $OPT(i, j) = 1 + \max_t\{OPT(i, t - 1) + OPT(t + 1, j - 1)\}$
String alignment: how similar are two strings?

- Comparing "abcd" and "abд" we can say that there is a ’c’ missing
- Comparing "abcd" and "abed" we may say:
  - the ’c’ and ’e’ should have been the same but where not, or
  - the right string has a missing ’c’ and the left a missing ’e’
- We can put a value on these differences:
  - For a mismatch: there is a cost of $\alpha_{pq}$ with $p$ and $q$ being Unicode characters or members of some other alphabet such as symbols in DNA strings
  - If there is a missing character: $\delta$
- For instance, $\alpha_{qw}$ may be 1 since ’q’ and ’w’ are close on a keyboard and $\alpha_{qk} = 3$ since they are more distant
- For a missing character, we may give it a cost $\delta = 2$ for instance
- To say how similar two strings are, we want to find the smallest cost of ”fixing” the strings so they become identical.
An example of using $\alpha_{pq}$

- Assume $\alpha_{cd} = 3$
- Of course $\alpha_{pp} = 0$ for every character $p$
- We can compare "abc" and "abd"
- Starting from the end we simply note the cost $\alpha_{cd} = 3$ and move on to the next pair of characters
- "ab" and "ab" remain with no cost
An example of using $\delta$

- We again compare "abc" and "abd"

- Starting from the end we either see this as
  - the left string misses a ’d’, or
  - the right string misses a ’c’

- Let us use the first case. It means we ”insert” the ’-’ and get: "abc-" i.e. there is a gap in the left string

- We don’t actually insert any ’-’ in the algorithms we will see soon, but the dashes are used when printing the output

- The gap in the left string is removed together with the ’d’ in the right string

- We then have "abc" and "ab"
X, Y, and OPT\((i, j)\)

- \(X = "abc"\) and \(Y = "abd"\)
- \(X = x_1x_2x_3\) and \(Y = y_1y_2y_3\)
- The cost of an optimal alignment of \(X = x_1x_2 \ldots x_i\) and \(Y = y_1y_2y \ldots y_j\) is denoted \(OPT(i, j)\)
- \(OPT(i, 0) = i\delta\) since it ignores \(Y\) and aligns a string of \(i\) symbols with an empty string — which must be done with \(i\delta\)
- \(OPT(0, j) = j\delta\) for similar reason
- \(OPT(1, 1)\) is the minimum of \(\alpha_{x_1y_1}\) and \(2\delta\)
- It is clear what happens when we use \(\alpha_{x_1y_1}\) — we are ”charged” with the mismatch cost of \(\alpha_{x_1y_1}\), which in our example is \(\alpha_{aa} = 0\)
- In the other case, we use one \(\delta\) to skip either \(x_1\) or \(y_1\) and then another \(\delta\) to skip the other of \(x_1\) and \(y_1\)
We can view the alignment as a graph.

Another example: $X = "abdc$" and $Y = "bbcd$"

- A vertical $\delta$ eats one symbol from $X$ and leaves $Y$ unchanged
- A horizontal $\delta$ eats one symbol from $Y$ and leaves $X$ unchanged
- $OPT(i, j)$ is equivalent to finding a shortest path from $(0, 0)$ to $(i, j)"
An $OPT(i, j)$ function

- It is probably clear now how we can write an optimal function to find the set of $\alpha$ and $\delta$ operations with minimum cost
- We have two strings $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$
- Using dynamic programming we write:
  - Case 1: $OPT(i, j) = \alpha_{x_i, y_j} + OPT(i - 1, j - 1)$
  - Case 2: $OPT(i, j) = \delta + OPT(i, j - 1)$
  - Case 3: $OPT(i, j) = \delta + OPT(i - 1, j)$
- As usual, we evaluate all cases and select the minimum
- We compute a table $A[0..m][0..n]$ using the recurrence for $OPT$
- $A$ is initialized with $A[i][0] \leftarrow i\delta$ for each $i$, and
- $A$ is initialized with $A[0][j] \leftarrow j\delta$ for each $j$.
- In this algorithm, we need the full matrix for print the alignment.
- If we only need the value computed by $OPT$ then we only need space for the two most recently used columns.
Bellman-Ford shortest path algorithm

- Consider a directed graph \((V, E)\) with \(n\) nodes and \(m\) edges.
- Edge costs \(c_{vw}\) are allowed to be negative in this algorithm.
- No cycle may have negative cost though (obviously).
- The problem is to find the minimum cost path from \(s\) to \(t\).
- Let \(OPT(i, v)\) be the minimum cost of a path from \(v\) to \(t\) which uses at most \(i\) edges.
- The initial problem is \(OPT(n - 1, s)\) which can be solved by:

\[
OPT(i, v) = \begin{cases} 
0, & v = t \\
\infty, & i = 0 \\
\min\{OPT(i - 1, v), OPT(i - 1, w) + c_{vw}\}, & i \geq 1 
\end{cases}
\]

- We can create the table \(M\) with \(O(n^2)\) space from \(OPT(i, v)\).
- \(M\) can be created in time \(O(n^3)\) for a dense graph.
Creating M

```plaintext
int M[n][n]

procedure make_table(G, s, t)
    n ← |V|
    M[0][t] ← 0
    M[0][v] ← ∞ for v ∈ V − {t}
    i ← 1
    while i ≤ n − 1 do
        for v ∈ V do
            M[i][v] ← min{M[i − 1][v], M[i − 1, w] + c_vw}
```

- This is a direct translation from the OPT(i, v) recurrence
- M[i][v] is the shortest path from v to t with at most i edges
- The M table can be used to compute a shortest path from s to t
- The Bellman-Ford algorithm is better than this, as we will see next
The Bellman-Ford algorithm

- Consider the for-loop again:

\[
\text{for } v \in V \text{ do} \\
M[i][v] \leftarrow \min\{M[i-1][v], M[i-1, w] + c_{vw}\}
\]

- It checks each edge \((v, w)\) to discover a shorter path from \(v\)
- We do not need a two-dimensional matrix
- Each vertex can have two attributes: distance and succ

\[
\text{for } e = (v, w) \in E \text{ do} \\
\quad \text{if } \text{distance}(v) > c_{vw} + \text{distance}(w) \text{ then} \\
\quad \quad \begin{align*}
\text{distance}(v) & \leftarrow c_{vw} + \text{distance}(w) \\
\text{succ}(v) & \leftarrow w
\end{align*}
\]

- This gives a running time \(O(mn)\) — still \(O(n^3)\) in a dense graph
Fewer than $n - 1$ iterations

- If no change to any node occurs in the while-loop, it is safe to stop before all $n - 1$ iterations
- This is so since no change can happen in the future