Dynamic Programming

Dynamic programming is another technique for solving optimization problems efficiently. It is more general than greedy algorithms but also often leads to slower algorithms.

Shortest paths

Recall that Dijkstra's algorithm correctly computes shortest paths only if all edge weights are non-negative. The Bellman-Ford algorithm can cope with negative edge weights. Here's the idea: Consider a shortest path from s to v and assume it has k edges. Then, if u is v's predecessor on this path, then the subpath from s to u must also be a shortest path, and it has $k-1$ edges. Thus, if we know the shortest paths with $k-1$ edges from s to all neighbours of v, we can compute the shortest path with k edges from s to v. More generally, if we know the shortest paths with *at most* $k-1$ edges from s to v and to all of v's neighbours, then we can compute the shortest path from s to v with at most k edges. Let $d(v,k)$ be the length of this path. This length is given by the following recurrence:

$$d(v,0) = \begin{cases} 0 & v = s \\ \infty & v \neq s \end{cases}$$

because only s can be reached from s without following any edges.

$$d(v,k) = \min\left(d(v,k-1),\ \min_{(w,v)\in E}\left(d(w,k-1) + \omega((w,v))\right)\right)$$

for $k > 0$. Thus, given $d(v,k-1)$ for all $v \in V$, computing $d(v,k)$ for all $v \in V$ takes $O(m)$ time. Since every shortest path from s to v has at most $n-1$ edges, $d(v,n-1)$ is in fact the distance from s to v. Thus, to compute $dist(s,v)$ for all $v \in V$, we need to compute $d(v,k)$ for all $v \in V$ and for all $0 \le k \le n-1$. This takes $O(n) + (n-1)\,O(m) = O(nm)$ time.

All-pairs shortest paths

What if we don't only want the distance from s to all vertices in G but the distances between all pairs of vertices in G. We could apply Bellman-Ford n times to solve this problem in $O(n^2 m)$ time. If the graph is dense ($m \in \Theta(n^2)$), this is the same as $O(n^4)$ time. Using Floyd-Warshall's algorithm, we can reduce the time to $O(n^3)$. Similar to Bellman-Ford's algorithm, it exploits that every subpath of a shortest path must be itself a shortest path.

We number the vertices from 1 to n. Now, for $0 \le i \le n$, let $d_i(v,w)$ be the length of the shortest path from v to w that has no vertex $j > i$ as an internal vertex. Then $dist(v,w) = d_n(v,w)$, so our goal is to compute $d_n(v,w)$ for all $v, w \in V$. Again, we develop a recurrence for $d_i(v,w)$. For $i = 0$, we have

$$d_0(v,w) = \begin{cases} w((v,w)) & \text{if } (v,w) \in E \\ \infty & \text{otherwise} \end{cases}$$

because a path containing no internal vertex $j > 0$ cannot have any internal vertices. Thus, we can compute $d_0(v,w)$ for all $v,w \in V$ by initializing $d_0(v,w) = \infty$ and then iterating over all edges in E and updating $d_0(v,w) = w((v,w))$ for every edge $(v,w) \in E$. This takes $O(n^2)$ time.

Now, for $i > 0$, consider the shortest path from v to w that has no internal vertex $j > i$. There are two possibilities:

1. If it does not contain vertex i, then it does in fact not contain any internal vertex $j > i-1$. Thus, its length is $d_{i-1}(v,w)$.

2. If it does contain i, then the subpath from v to i must be a shortest path without internal vertices $j > i-1$, as does the subpath from i to w. Thus, the path has length $d_{i-1}(v,i) + d_{i-1}(i,w)$.

We do not know which of these two cases applies, but one of them must apply. Thus,

$$d_i(v,w) = \min(d_{i-1}(v,w), d_{i-1}(v,i) + d_{i-1}(i,w))$$

Since this takes $O(1)$ time to compute for every pair (v,w), we can compute $d_i(v,w)$ for all $v,w \in V$ in

$O(n^2)$ time, provided we have already computed $d_{i-1}(v,w)$ for all $v,w \in V$. Since we need to compute $d_i(v,w)$ for all $0 \le i \le n$, this gives a running time of $O(n^3)$.

Sequence alignment

Sequence alignment tells us something about how different gene sequences may have evolved. For more than two sequences, the problem is NP-hard. For two sequences, we can solve the problem efficiently in $O(nm)$ time, where n and m are the lengths of the two input sequences.

The input consists of the two sequences over an alphabet Σ. (For gene sequences, $\Sigma = \{A, C, G, T\}$.) We are also given a $|\Sigma| \times |\Sigma|$ **similarity matrix**. This matrix represents the likelihood of replacing one amino acid with another one for every possible pair of amino acids (characters). Finally, we are given a vector of Σ **gap penalties**, which represent the likelihood of dropping or introducing an amino acid.

An **alignment** of two strings A and B are two strings $A^* = \langle a_1^*, a_2^*, \ldots, a_t^* \rangle$ and $B^* = \langle b_1^*, b_2^*, \ldots, b_t^* \rangle$ of the same length over the alphabet $\Sigma^* = \Sigma \cup \{\sqcup\}$ and such that A^* and B^* can be obtained from A and B by inserting spaces. We also assume that $a_i^* \neq \sqcup$ or $b_i^* \neq \sqcup$ for every $1 \le i \le m$ because otherwise we can obtain

a shorter alignment by dropping position i from A^* and B^*. The score of the alignment is

$$S(A^*, B^*) = \sum_{i=1}^{t} s(a_i^*, b_i^*), \text{ where}$$

$$s(a_i^*, b_i^*) = \begin{cases} \delta(a_i^*) & \text{if } b_i = \sqcup \\ \delta(b_i^*) & \text{if } a_i = \sqcup \\ M(a_i^*, b_i^*) & \text{otherwise.} \end{cases}$$

M is the similarity matrix and δ is the vector of gap penalties.

Our goal is to find an alignment with the smallest possible score. To do this using dynamic programming, we need a recurrence characterizing an optimal solution. We have three possibilities for the last position in (A^*, B^*). Let $m = |A|$ and $n = |B|$.

Case 1: $a_t^* = a_m$ and $b_t^* = b_n$. Then $(A^*[1, t-1], B^*[1, t-1])$ is an optimal alignment for $A[1, m-1]$ and $B[1, n-1]$, where $X[i, j] = \langle x_i, x_{i+1}, \ldots, x_j \rangle$ for any sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$.

Case 2: $a_t^* = \sqcup$ and $b_t^* = b_n$. Then $(A^*[1, t-1], B^*[1, t-1])$ is an optimal alignment for A and $B[1, n-1]$.

Case 3: $a_t^* = a_m$ and $b_t^* = \sqcup$. Then $(A^*[1, t-1], B^*[1, t-1])$ is an optimal alignment for $A[1, m-1]$ and B.

This gives us the following recurrence for the score $S(i,j)$ of an optimal alignment for $A[1,i]$ and $B[1,j]$.

$$S(i,j) = \begin{cases} \sum_{k=1}^{j} \delta(b_k) & i = 0 \\ \sum_{k=1}^{i} \delta(a_k) & j = 0 \\ \min\left(\delta(a_i) + S(i-1,j),\ \delta(b_j) + S(i,j-1),\ M(a_i,b_j) + S(i-1,j-1)\right) & i>0,\ j>0 \end{cases}$$

We can compute the score $S(m,n)$ of an optimal alignment for A and B using this recurrence in $O(mn)$ time:

```
Sequence Alignment (A, B, M, δ)
m = |A|
n = |B|
S[0,0] = 0
for i = 1 to m do
    S[i,0] = S[i-1,0] + δ(A[i])
for j = 1 to n do
    S[0,j] = S[0,j-1] + δ(B[j])
for i = 1 to n do
    for j = 1 to m do
        S[i,j] = S[i-1,j-1] + M[A[i], B[j]]
        S1 = S[i,j-1] + δ[B[j]]
        if S1 < S[i,j] then S[i,j] = S1
        S2 = S[i-1,j] + δ[A[i]]
        if S2 < S[i,j] then S[i,j] = S2
```

```
return S[m,n]
```

Note that all values needed by iteration (i,j) are computed by earlier iterations and thus are already stored in S. Thus, each iteration takes constant time.

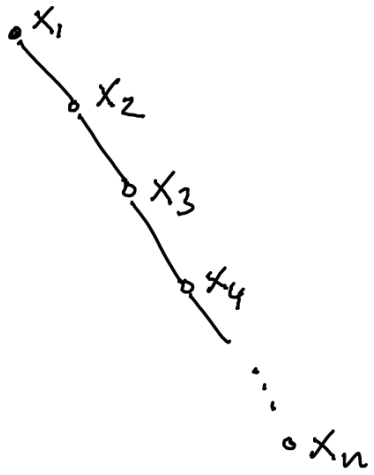
Exercise: Given S, can you compute an optimal alignment (not just its score) in $O(n+m)$ time?

Optimal Binary Search Trees

Balanced binary search trees are worst-case optimal but may not be optimal if different keys have different probabilities to be searched for. For example, given n elements $x_1, x_2, \ldots, x_n$ with probabilities $p_1, p_2, \ldots, p_n$ to search for them, the expected cost of a search using a tree T is

$$C(T) = \sum_{i=1}^{n} d_T(x_i) p_i$$

where $d_T(x)$ is x's depth in T. Now assume $p_i = 2^{-i}$ for $1 \le i < n$ and $p_n = 2^{-n+1}$, so $\sum_{i=1}^{n} p_i = 1$. Then $C(T) = \Theta(\lg n)$ if T is perfectly balanced because x_1 has logarithmic depth in T. The highly skewed tree



has cost $C(T) = \sum_{i=1}^{n-1} \frac{i}{2^i} + \frac{n}{2^n} \leq \sum_{i=1}^{\infty} \frac{i}{2^i} + O(1) \in O(1)$.

Thus, it achieves much better expected query cost in this case.

Side note: $\sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=1}^{\infty} i x^i$, for $x = \frac{1}{2}$

But $\sum_{i=1}^{\infty} i x^i = x \sum_{i=1}^{\infty} i x^{i-1} = x \cdot \frac{d \sum_{i=1}^{\infty} x^i}{dx}$

$= x \cdot \frac{d\left(\sum_{i=0}^{\infty} x^i - 1\right)}{dx}$

$= x \cdot \frac{d\left(\frac{1}{1-x} - 1\right)}{dx}$

$= x \cdot \frac{1}{(1-x)^2}$

Plugging in $x = \frac{1}{2}$, we obtain $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$.

So, given the probabilities $p_1, p_2, \ldots, p_n$, our goal is to build an binary search tree T over $x_1, x_2, \ldots, x_n$ that minimizes $C(T)$.

Observe the similarity to Huffman codes. Here we want to ensure that elements with high probability are close to the root while less probable elements may be deeper in the tree. In Huffman trees, frequent characters are close to the root while infrequent ones may have greater depth. More formally, we have essentially the same cost measure:

Huffman	Optimal search tree
$\sum_{i=1}^{n} f_i d_T(x_i)$	$\sum_{i=1}^{n} p_i d_T(x_i)$

So why can't we just use Huffman's algorithm to find an optimal search tree? The problem is that Huffman's algorithm does not guarantee an ordering of the leaves, while an optimal search tree still needs to be a search tree. This small difference complicates matters substantially. So let us try to understand the structure of an optimal search tree again:

If $n=1$, it has a single node: $\circ x_1$

If $n>1$, all we know is that the root stores some element x_i, the left subtree must be an optimal search tree for $x_1, \dots, x_{i-1}$ and the right subtree must be an optimal search tree for $x_{i+1}, \dots, x_n$.

This is enough to develop a recurrence for the cost $C(i,j)$ of an optimal search tree for $x_i, x_{i+1}, \dots, x_j$. $C(1,n)$ is then the cost of the optimal tree for the

whole set of elements $x_1, x_2, \ldots, x_n$.

$$C[i,j] = \begin{cases} 0 & i > j \\ P[i,j] + \min\limits_{i \le k \le j} \left(C[i,k-1] + C[k+1,j] \right) & i \le j \end{cases}$$

$$P[i,j] = \sum_{k=i}^{j} p_k$$

To use this recurrence in an algorithm, we need to ensure that all values $C[i',j']$ needed to compute $C[i,j]$ are computed before $C[i,j]$. Any such value $C[i',j']$ satisfies $j'-i' < j-i$. This gives the following algorithm.

```
OptimalSearchTree(p)
for i=1 to n do
    P[i,i] = p[i]
    for j=i+1 to n do
        P[i,j] = P[i,j-1] + p[j]
for i=1 to n+1 do
    C[i,i-1] = 0
for k=0 to n-1 do
    for i=1 to n-k do
        C[i,i+k] = ∞
        for j=i to i+k do
            C' = C[i,j-1] + C[j+1,i+k]
            if C' < C[i,i+k] then
                C[i,i+k] = C'
        C[i,i+k] = C[i,i+k] + P[i,i+k]
return C[1,n]
```

This algorithm clearly takes $O(n^3)$ time and correctly computes $C[1,n]$ because it implements our recurrence for $C[i,j]$. It only computes the cost of the optimal tree. To compute the tree itself, we need to determine for all $i \le j$ which element is stored at the root of an optimal tree for $x_i, \ldots, x_j$. Let us call this table R. We compute R by augmenting the above algorithm as follows:

```
OptimalSearchTree(p)
for i=1 to n do
    P[i,i] = p[i]
    for j=i+1 to n do
        P[i,j] = P[i,j-1] + p[j]
for i=1 to n+1 do
    C[i,i-1] = 0
for k=0 to n-1 do
    for i=1 to n-k do
        C[i,i+k] = ∞
        for j=i to i+k do
            C' = C[i,j-1] + C[j+1,i+k]
            if C' < C[i,i+k] then
                C[i,i+k] = C'
                R[i,i+k] = j
        C[i,i+k] = C[i,i+k] + P[i,i+k]
return R
```

Clearly the running time remains $O(n^3)$. Now, given R, we can compute the optimal tree using a simple recursive procedure, which we call as BuildTree $(x, R, 1, n)$.

```
BuildTree(x, R, i, j)
  if i > j then return null
  k = R[i,j]
  r = a new node
  r.key = x[k]
  r.left = BuildTree(x, R, i, k-1)
  r.right = BuildTree(x, R, k+1, j)
  return r
```

Given that $R[i,j]$ stores the index k of the root element of an optimal search tree for $x_i, \ldots, x_j$ and that the left and right subtrees must be optimal search trees for $x_i, \ldots, x_{k-1}$ and $x_{k+1}, \ldots, x_j$, respectively, BuildTree correctly builds an optimal search tree for $x_1, \ldots, x_n$. Its running time is given by the recurrence:

$$T(i,j) = \begin{cases} \Theta(1) & i > j \\ \Theta(1) + T(i,k-1) + T(k+1,j) & i \le j \end{cases}$$

It is easy to verify by induction that $T(i,j) \in \Theta(j-i+2)$ for all (i,j) such that $j \ge i-1$. Thus, $T(1,n) \in \Theta(n)$, that is, BuildTree takes linear time and the cost of finding the optimal tree is dominated by the dynamic programming portion, as is usually the case.