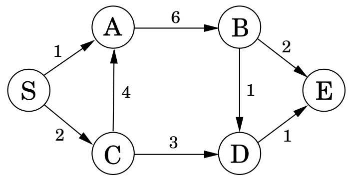
Lecture 11: Dynamic programming

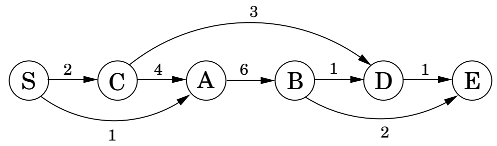
In this lecture we introduce a powerful algorithmic technique called dynamic programming using several example problems:

- Longest increasing subsequences
- Edit distance
- Knapsack
- Chain matrix multiplication
- Shortest paths
- Travelling salesman problem
- Independent sets in trees

First example of dynamic programming

- Shortest paths are particularly easy in DAGs (see Lecture 8 / Chapter 4 in the course book for general shortest paths algorithms)
- This is because nodes in a DAG can be linearized: there is an ordering of the nodes such that all edges go from left to right.





First example of dynamic programming

- So how to determine the distance from a given source node S to a node v?
- If we know the shortest distances from S of all the predecessors $u_1, \ldots, u_n$ of v, this is easy:
- The minimum distance is obtained through a predecessor u_i for which the distance from S to u_i + the distance from u_i to v is minimum, that is,

$$\mathrm{dist}(v) = \min\{\mathrm{dist}(u_1) + \ell(u_1, v), \ldots, \mathrm{dist}(u_n) + \ell(u_n, v)\}$$

- This holds for all nodes.

First example of dynamic programming

- If we compute the dist values in a linear order mentioned above, we are guaranteed to have all the required information for a node v when we get to that node.
- This is because the nodes are ordered so that all predecessors of a node v are before v in the ordering.
- Now we can compute all distances in a single pass:

```
initialize all dist(·) values to ∞
dist(s) = 0
for each v ∈ V \ {s} in linearized order do
    dist(v) = min{dist(u) + ℓ(u, v) : (u, v) ∈ E}
end
```

First example of dynamic programming

- This is dynamic programming, an algorithmic paradigm, where a problem is solved by identifying a collection of subproblems (here $\{dist(u) : u \in V\}$) and tackling them one by one, smallest first, using the answers to smaller problems to solve larger ones, until all subproblems are solved.
- However, in dynamic programming we are not given a DAG but it is implicit.
- Its nodes are the subproblems that we define and its edges are the dependencies between subproblems: if to solve subproblem B, we need the answer to subproblem A, then there is a (conceptual) edge from A to B.
- In this case A is seen as a smaller subproblem than B.

Longest increasing subsequence

- In this problem, the input is a sequence of numbers $a_1, a_2, \ldots, a_n$.
- A subsequence is any subset of these numbers taken in order $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots i_k \leq n$
- An increasing subsequence is one in which the numbers get strictly larger.
- The task is to find an increasing subsequence of greatest length.
- For example, the longest subsequence of

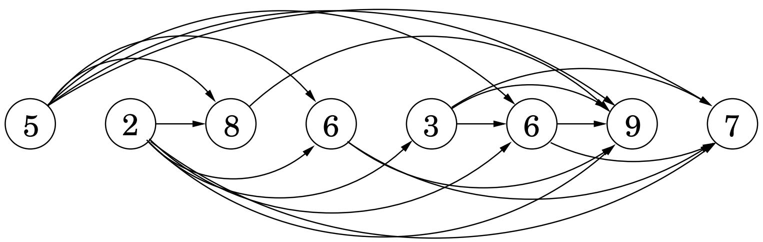
$$5, 2, 8, 6, 3, 6, 9, 7$$

is

$$2, 3, 6, 9$$

Longest increasing subsequence

- The solution space can be analysed using a graph $G = (V, E)$ where for each element a_i in the sequence there is a node $i \in V$ and there is an edge $(i, j) \in E$ whenever $i < j$ and $a_i < a_j$.



- Now there is a one-to-one correspondence between increasing subsequences and paths in this DAG.
- So the task is to find the longest path in the DAG.

Algorithm for longest increasing subsequence

```
for j = 1, 2, ..., n do
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
end
return max_j L(j)
```

where $\max\{\} = 0$

The basic idea:

- $L(j)$ is the length of the longest path ending at j (+1).
- $L(j)$ is then the length of the longest path to one of this predecessors + 1

Longest increasing subsequence

- This is dynamic programming.
- To solve the problem we defined a collection of subproblems $\{\mathrm{L}(j) : 1 \leq j \leq n\}$ such that

  There is an ordering on the subproblems and a relation that shows how to solve a subproblem given the answers to subproblems that appear earlier in the ordering.

- The running time is $\mathcal{O}(n^2)$:
  - Computing $\mathrm{L}(j)$ takes time proportional to the indegree of j, this is linear in $|E|$ and at most $\mathcal{O}(n^2)$ (given that predecessors of a node j are known).
- L values give the only the length of an optimal subsequence. To recover the subsequence itself some further bookkeeping is needed as for the shortest paths (see Lecture 8 / Chapter 4 in the course book).

Edit distance

- In this problem the task is to determine how close given two words are.
- A natural measure of distance is the extent to which the words can be aligned.
- For example, two possible alignments of SNOWY and SUNNY:

S	-	N	O	W	Y
S	U	N	N	-	Y

Cost: 3

-	S	N	O	W	-	Y
S	U	N	-	-	N	Y

Cost: 5

- The "-" symbol indicates a gap (any number of these can be placed in either string).
- The cost of an alignment is the number of columns in which the letters differ.

Edit distance

- The edit distance between two strings is the cost of their best possible alignment.
- Edit distance can be seen as the minimum number of edits (insertions, deletions, and substitutions of characters) needed to transform the first string to the second.

Edit distance using dynamic programming

- What are the subproblems?
- The input is two strings $x[1 \cdots m]$ and $y[1 \cdots n]$.
- We could consider the edit distance between some prefix $x[1 \cdots i]$ of x and some prefix $y[1 \cdots j]$ of y. Call this subproblem $E(i, j)$
- The goal is to compute $E(m, n)$.

For instance, the subproblem $E(7, 5)$:

$$\boxed{\mathrm{E\ X\ P\ O\ N\ E\ N}}\ \mathrm{T\ I\ A\ L}$$

$$\boxed{\mathrm{P\ O\ L\ Y\ N}}\ \mathrm{O\ M\ I\ A\ L}$$

Edit distance using dynamic programming

- How to express $E(i,j)$ in terms of "smaller" subproblems?
- Consider the alignment between $x[1\cdots i]$ and $y[1\cdots j]$ and the rightmost column: only three cases possible

$$\begin{array}{|c|}\hline x[i] \\ \hline - \\ \hline\end{array} \quad \text{or} \quad \begin{array}{|c|}\hline - \\ \hline y[j] \\ \hline\end{array} \quad \text{or} \quad \begin{array}{|c|}\hline x[i] \\ \hline y[j] \\ \hline\end{array}$$

- In the first case, the cost is 1 + that of the remaining alignment $x[1\cdots i-1]$ and $y[1\cdots j]$ (subproblem $E(i-1,j)$).
- In the second case, the cost is 1 + that of the remaining alignment $x[1\cdots i]$ and $y[1\cdots j-1]$ (subproblem $E(i,j-1)$).
- In the third case, the cost is that of the remaining alignment $x[1\cdots i-1]$ and $y[1\cdots j-1]$ (subproblem $E(i-1,j-1)$) + 1 if $x[i] \neq y[j]$ and + 0 if $x[i] = y[j]$.

Edit distance using dynamic programming

- Hence, we have expressed the subproblem $E(i,j)$ in terms of three smaller subproblems $E(i-1,j), E(i,j-1), E(i-1,j-1)$.
- We do not know which of them leads to the best solutions, so we need to try them all and pick the best:

$$E(i,j) = \min\{1+E(i-1,j), 1+E(i,j-1), \mathrm{diff}(i,j)+E(i-1,j-1)\}$$

where $\mathrm{diff}(i,j) = 0$ if $x[i] = y[j]$ and otherwise 1.

Edit distance using dynamic programming

- The answers to all the subproblems $E(i,j)$ form a two-dimensional table which should be solved in an order where $E(i-1,j), E(i,j-1), E(i-1,j-1)$ are solved before $E(i,j)$.
- But what are the base cases $E(i,0)$ and $E(0,j)$?
- $E(i,0)$ is the edit distance between the 0-length prefix of y (the empty string) and the first i letters of x and this distance is clearly i.
- Similarly, $E(0,j) = j$.

Dynamic programming algorithm for edit distance

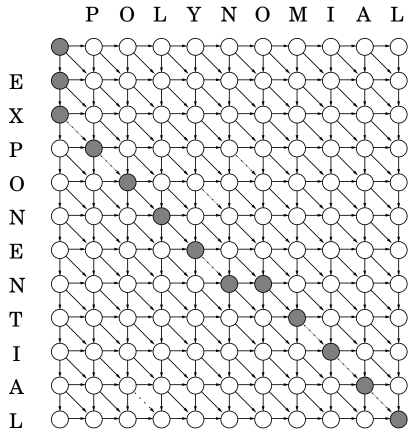
for $i = 0, 1, 2, \ldots, m$ **do**
    $E(i,0) = i$
end
for $j = 0, 1, 2, \ldots, n$ **do**
    $E(0,j) = j$
end
for $i = 0, 1, 2, \ldots, m$ **do**
    for $j = 0, 1, 2, \ldots, n$ **do**
        $E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), \mathrm{diff}(i,j) + E(i-1,j-1)\}$
    end
end
return $E(m,n)$

The underlying DAG

- Every dynamic program has an underlying DAG structure.
- Nodes are subproblems and edges capture the precedence constraints.
- Edges from nodes $u_1, \ldots, u_k$ to node v mean that subproblem v can only be solved once the answers to $u_1, \ldots, u_k$ are known.

The underlying DAG

- In the edit distance problem, the subproblems are of the form (i, j) and there are edges from $(i-1, j), (i, j-1), (i-1, j-1)$ to (i, j).
- In this graph edge lengths can be set so that edit distance corresponds to shortest path!
- Set lengths to 1 for all edges except for those edges from $(i-1, j-1)$ to (i, j) with $x[i] = y[j]$ for which the length is set to 0 (dotted lines in the figure).
- Now edit distance is the length of the shortest path from (0,0) to (m, n).



Knapsack

- In this problem the input is a set of n items of weight $w_1, \ldots, w_n$ and value $v_1, \ldots, v_n$ and a weight limit W
- The task is to select a set items of at most total weight W but having the most value.
- We consider two versions of the problems.
- In knapsack with repetition there are unlimited quantities of each item available.
- In knapsack without repetition only one instance of each item is available.

Knapsack

- For example, take $W = 10$ and

Item	Weight	Value
1	6	30
2	3	14
3	4	16
4	2	9

- For knapsack with repetition the optimal solution is to pick one instance of item 1 and two of item 4 (total value 48).
- For knapsack without repetition the optimal solution is to pick items 1 and 3 (total value 46).

Knapsack with repetition

- What are the subproblems?
- We can shrink the original problem in two ways:
  - Look at fewer items ($1, 2, \ldots, j$ for some $j \leq n$) or
  - consider a smaller weight limit $w \leq W$
- Consider the latter case first.
- $K(w)$ = maximum value achievable with a knapsack of capacity w.
- If the optimal solution to $K(w)$ includes an instance of item i, then removing this instance leaves an optimal solution to $K(w - w_i)$.
- This means that $K(w) = K(w - w_i) + v_i$ for some i
- ... but we do not know which so we try them all:

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\}$$

Dynamic programming algorithm for knapsack with repetitions

```
K(0) = 0
for w = 1 to W do
    K(w) = max{K(w − w_i) + v_i : w_i ≤ w}
end
return K(W)
```

The algorithm fills in a one-dimensional table of length $W + 1$ in left-to-right order. Each entry can take up $\mathcal{O}(n)$ time to compute. Thus, the overall running time is $\mathcal{O}(nW)$.

Knapsack without repetition

- For the case without repetition, a better way of forming subproblems is to limit the items (one of each only allowed).
- $K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, \ldots, j$.
- Now for each item j, either j is needed or not needed to achieve the optimal value:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Dynamic programming algorithm for knapsack without repetitions

```
Initialize all K(0, j) = 0 and all K(w, 0) = 0
for j = 1 to n do
    for w = 1 to W do
        if w_j > w then
            K(w, j) = K(w, j − 1)
        else
            K(w, j) = max{K(w, j − 1), K(w − w_j, j − 1) + v_j}
        end
    end
end
return K(W, n)
```

The algorithm fills in a two-dimensional table with $W + 1$ rows and $n + 1$ columns. Each entry can be computed in constant time. Thus, the overall running time is $\mathcal{O}(nW)$.

Chain matrix multiplication

- Suppose we want to multiply a number of matrices with different dimensions.
- Because matrix multiplication is associative, i.e,

$$A \times (B \times C) = (A \times B) \times C$$

different orders are possible.
- Multiplying an $m \times n$ matrix with a $n \times p$ matrix gives a $m \times p$ matrix and takes roughly mnp multiplications.
- Moreover, different evaluation orders (parenthesizations) of multiplication lead to different costs.

Chain matrix multiplication

- Consider matrices:
  $A(50 \times 20)$, $B(20 \times 1)$ $C(1 \times 10)$, $D(10 \times 100)$
- Now evaluation $A \times B \times C \times D$ has different costs in different evaluation orders:

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120 200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60 200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7 000

- Observe that the natural greedy approach to always perform the cheapest matrix multiplication available (second option) does not lead to an optimal solution.

Chain matrix multiplication

- So the input is a sequence of matrices $A_1, \dots, A_n$ with dimensions $m_0 \times m_1, m_1 \times m_2, \dots, m_{n-1} \times m_n$ and the task is to give an ordering (parenthesization) so that the cost of the matrix multiplication $A_1 \times \cdots \times A_n$ is the lowest.
- What are the subproblems?
- For $1 \leq i \leq j \leq n$ define:

$$C(i,j) = \text{minimum cost of multiplying } A_i \times \cdots \times A_j$$

- Now for any k ($i \leq k < j$), this problem can be divided to two subproblems: multiplying $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$. So the cost $C(i,j)$ is the cost of the two subproblems plus the cost of combining the results.
- As we do not know which is the best way of splitting, we have to try them all

$$C(i,j) = \min_{1 \leq k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

Dynamic programming algorithm for chain matrix multiplication

```
for j = 1 to n do
    C(i, i) = 0
end
for s = 1 to n − 1 do
    for i = 1 to n − s do
        j = i + s
        C(i, j) = min{C(i, k) + C(k + 1, j) + m_{i−1} · m_k · m_j : i ≤
        k < j}
    end
end
return C(1, n)
```

The algorithm fills in a two-dimensional table whose entries take $\mathcal{O}(n)$ time to compute. Thus, the overall running time is $\mathcal{O}(n^3)$.

Shortest paths continued

- In the shortest reliable path problem the input is a graph G with lengths on the edges, two nodes s and t and an integer k.
- The task is to find a shortest path from s to t that uses at most k edges.
- Let us define for each node v and each integer $i \le k$, $\mathrm{dist}(v, i)$ to be the length of the shortest path from s to v that uses i edges.
- We can set $\mathrm{dist}(s, 0) = 0$ and $\mathrm{dist}(v, 0) = \infty$ for other nodes v.
- Then

$$\mathrm{dist}(v, i) = \min_{(u,v)\in E}\{\mathrm{dist}(u, i-1) + \ell(u, v)\}$$

- This leads to a dynamic programming algorithm in a natural way.

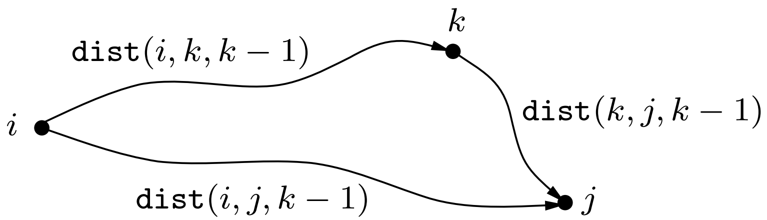
All-pairs shortest paths

- If we want to find the shortest path between all pairs of vertices, the straightforward approach to running the general shortest-path algorithm $|V|$ times, once for each starting node, is not the most economical.
- A better alternative is the $\mathcal{O}(|V|^3)$ Floyd-Warshall algorithm.
- The idea: subproblems are obtained by restricting the permissible intermediate nodes on the path.
- Number the nodes in V as $\{1, 2, \ldots, n\}$ and let $\mathrm{dist}(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates.
- Initially, $\mathrm{dist}(i, j, 0) = \ell(i, j)$ if $(i, j) \in E$ and otherwise $\mathrm{dist}(i, j, 0) = \infty$.

All-pairs shortest paths

- If we now consider a new intermediate node k, then the shortest path from i to j uses it once or not at all:

$$\mathrm{dist}(i, j, k) = \min\{\mathrm{dist}(i, k, k-1) + \mathrm{dist}(k, j, k-1), \mathrm{dist}(i, j, k-1)\}$$



All-pairs shortest paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ∞
    end
end
for all (i, j) ∈ E do
    dist(i, j, 0) = ℓ(i, j)
end
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) =
              min{dist(i, k, k − 1) + dist(k, j, k − 1), dist(i, j, k − 1) }
        end
    end
end
```

The travelling salesman problem (TSP)

- In this problem the input is a set of n cities and a matrix $D = (d_{ij})$ of intercity distances.
- The task is to find a tour that
  1. starts and ends at city 1,
  2. visits all other cities exactly once, and
  3. has the minimum total length.
- This is a very difficult problem with no guaranteed polynomial time algorithm known.
- The brute force technique would examine all possible tours but would have $\mathcal{O}(n!)$ time complexity.
- Using dynamic programming we can do a bit better (but not polynomial time).

Subproblems for the TSP

- A suitable subproblem:
  for a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1 and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.
- If $|S| \geq 2$, we can set $C(S, 1) = \infty$.
- Now $C(S, j)$ can be determined from subproblems by considering the second-to-last city, which has to be some $i \in S$, with the overall path length equal to distance from 1 to i plus the distance from i to j
- We choose the best such i:

$$C(S, j) = \min_{i \in S : i \neq j} \{C(S - \{j\}, i) + d_{i,j}\}$$

- The subproblems can be ordered by $|S|$.

Dynamic programming algorithm for TSP

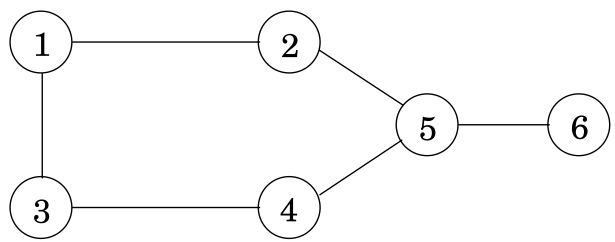
$C(\{1\}, 1) = 0$
for $s = 2$ *to* n **do**
    for *all subsets* $S \subseteq \{1, 2, \ldots, n\}$ *of size* s *and containing* 1 **do**
        $C(S, 1) = \infty$
        for *all* $j \in S, j \neq 1$ **do**
            $C(S, j) = \min\{C(S - \{j\}, i) + d_{i,j} : i \in S, i \neq j\}$
        end
    end
end
return $\min\{C(\{1, 2, \ldots, n\}, j) + d_{j,1} : 2 \leq j \leq n\}$

There are at most $2^n \cdot n$ subproblems and each take linear time to compute. Hence, the total run time is $\mathcal{O}(n^2 2^n)$.

Independent sets in trees

- A subset of nodes $S \subseteq V$ is an independent set of graph $G = (V, E)$ if there are no edges between the vertices in S.
- Finding a largest independent set in a graph is a very difficult problem with no guaranteed polynomial time algorithm known.

For this graph the largest independent set is $\{2, 3, 6\}$:



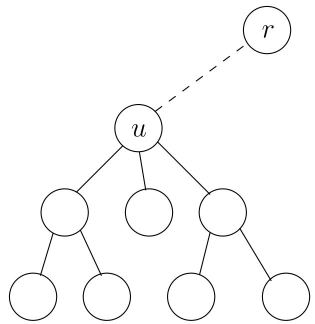
Independent sets in trees

- But if the graph is a tree, a linear time dynamic programming algorithm is available.
- Select one of the nodes in the tree as the root r.
- In a rooted tree, each node u defines a subtree, that is, the subtree induced by the nodes v for which the path from v to r contains u. (Or, the subtree "hanging from" u, when we draw the tree so that the root r is the topmost node.)
- Hence, a suitable subproblem:
  $I(u)$ = size of the largest independent set in the subtree hanging from u.
- Now dynamic programming can proceed bottom-up in the rooted tree and the final goal is $I(r)$ where r is the root of the entire tree.

Independent sets in trees

- Suppose we know the largest independent sets of all subtrees below a certain node u.
- Now computing $I(u)$ has two cases: any independent set either includes u or it does not. Hence,

$$I(u) = \max\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w)\}$$



Summary

- Dynamic programming is a general algorithmic technique that is applicable to a wide range of problems.
- In dynamic programming, a problem is divided into a set of subproblems such that there is
  1. an ordering on the subproblems, and
  2. a rule that shows how to solve a subproblem given the answers to subproblems that appear earlier in the ordering.
- Every dynamic programming algorithm has an underlying DAG structure where the nodes are subproblems and edges capture the precedence constraints; that is, a subproblem can only be solved once the answers to its predecessors in the DAG are known.