Dynamic Programming

CMSC 27200: Honors Theory of Algorithms

January 30 to February 10, 2023

Corresponding section(s) of Kleinberg-Tardos: 6.1, 6.2, 6.4, 6.5, 6.8

1 Overview

In dynamic programming, we solve problems by solving smaller subproblems and then combining the solutions for smaller subproblems into solutions for larger subproblems and eventually the entire problem. To design a dynamic programming algorithm, we need to do the following:

1. Determine the subproblems which we need to solve.
2. Find a recurrence relation for how to obtain the answer to a subproblem from the answers to smaller subproblems.

To implement the dynamic programming algorithm, we do the following:

1. Store the answers to the subproblems which we've solved so far (memoization). This allows us to avoid repeating our work if we need the answer to a given subproblem multiple times.
2. Iteratively compute the answer to the next subproblem using the recurrence relation.

Example 1.1. *Consider the Fibonacci sequence $F(1) = 1$, $F(2) = 1$, $F(n) = F(n-1)+F(n-2)$. We can find $F(n)$ using dynamic programming as follows:*

1. *Subproblems: What is $F(k)$ where $k \in [n]$?*
2. *Recurrence relation: $F(k) = F(k-1) + F(k-2)$*

To implement the dynamic programming algorithm, we can store the values $\{F(k)\}$ in an array and iteratively compute the next value.

```
1. $F[1] = 1$
2. $F[2] = 1$
3. For $i = 3$ to $n$,
   $F[i] = F[i-1] + F[i-2]$
```

This algorithm takes $O(n)$ arithmetic operations (because the numbers involved get very large, these arithmetic operations actually take $O(n)$ time for a total running time of $O(n^2)$).

1.1 Difference between dynamic programming and divide and conquer

In both divide and conquer algorithms and dynamic programming algorithms, we split up the problem into subproblems and use the solutions to these subproblems to solve the entire problem. However, with divide and conquer algorithms, we split each problem/subproblem into disjoint pieces and the answer to each subproblem is only used once (though combining the answers to the subproblems may be intricate). On the other hand, with dynamic programming algorithms the subproblems may overlap and we may use the answer to a subproblem multiple times.

Visually, we can represent what is going on with the following directed graph.

1. We have a vertex v_p for each subproblem p.
2. We have an edge from v_p to v_q if and only if subproblem p appears in the recurrence relation for solving subproblem q.

For divide and conquer algorithms, this graph is a tree. For dynamic programming algorithms, this graph could be any acyclic graph.

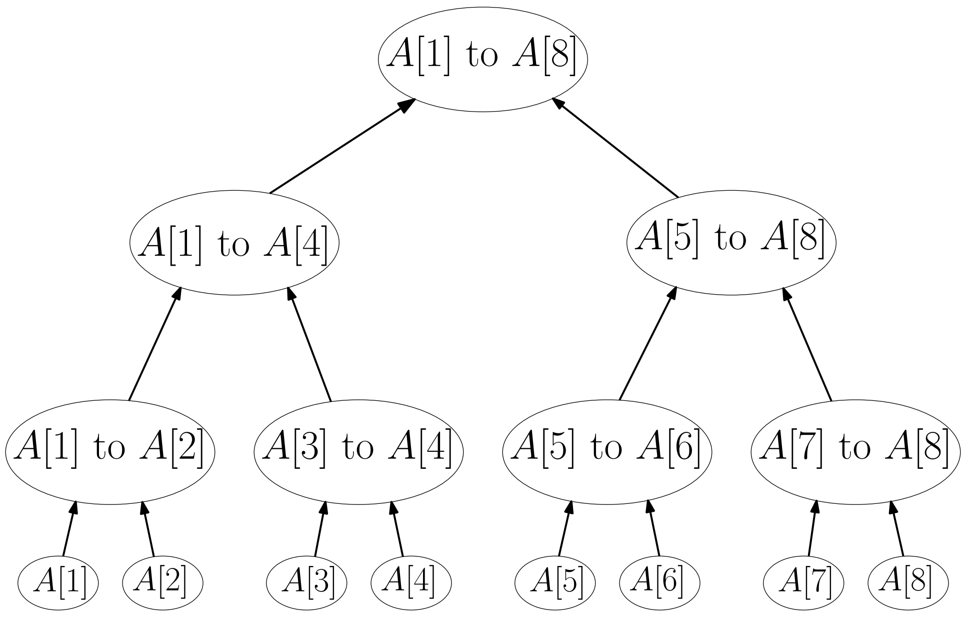

Figure 1: This figure shows the graph for mergesort on an array A with 8 elements (starting from index 1).

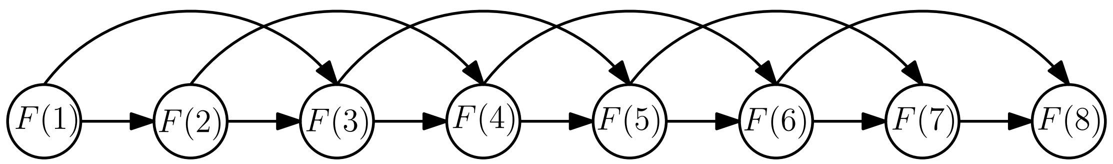

Figure 2: This figure shows the graph for the dynamic programming algorithm for evaluating the Fibonacci numbers.

2 Weighted Interval Scheduling

Problem 2.1 (Weighted Interval Scheduling: Maximizing the Total Weight of the Jobs Accepted)**.** *In this interval scheduling problem, we have one processor and we are given n jobs. Each job has a set time interval $[a_i, b_i]$ when it must use the processor if it is accepted and a weight w_i. We are then asked to find a schedule S which maximizes the total weight of the accepted jobs.*

This problem can be stated more precisely using the following definition:

Definition 2.2. *Given n intervals $[a_1, b_1], \ldots, [a_n, b_n]$, we say that a sequence $S = (i_1, \ldots, i_m)$ of elements of $[n]$ is a valid schedule if $\forall j \in [1, m-1](b_{i_j} \leq a_{i_{j+1}})$ (the jobs $i_1, \ldots, i_m$ are in sequential order and job i_{j+1} does not start before job i_j finishes). We define $|S| = m$ to be the number of elements in S (i.e. the number of jobs accepted).*

Precise problem statement: Given n intervals $[a_1, b_1], \ldots, [a_n, b_n]$, find a valid schedule $S = (i_1, \ldots, i_m)$ which maximizes $\sum_{j=1}^{|S|} w_{i_j}$

Preprocessing: We first sort the times $\{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$ and let $(t_1, \ldots, t_{2n})$ be the sorted times. For simplicity, we assume that all of the elements of $\{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$ are distinct so there are no ties.

To solve this problem, we consider the following subproblems: For all $t \in \{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$, what is the maximum total weight which can be completed by time t?

Definition 2.3. *We define $\mathcal{S}_t = \{S = (i_1, \ldots, i_m) : S \text{ is a valid schedule}, b_{i_m} \leq t\}$. In other words, $\mathcal{S}_t$ is the set of all valid schedules which are completed by time t.*

Definition 2.4. *Define $w(t)$ to be the maximum total weight which can be completed by time t. More precisely,*

$$w(t) = \max_{S=(i_1,\ldots,i_m)\in\mathcal{S}_t} \{\sum_{j=1}^{|S|} w_{i_j}\}$$

We now have the following recurrence relation:

Lemma 2.5. *Assuming that all the elements of $\{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$ are distinct, for all $k \in [2n-1]$,*

1. *If $t_{k+1} = a_i$ for some $i \in [n]$ then $w(t_{k+1}) = w(t_k)$.*
2. *If $t_{k+1} = b_j$ for some $j \in [n]$ then $w(t_{k+1}) = \max\{w(t_k), w(a_j) + w_j\}$.*

Proof. This equality can be proved by either manipulating the sets of possible solutions to the subproblems or by proving both an upper bound and a lower bound on $w(t_{k+1})$.

Proof which manipulates the sets of possible solutions:

If $t_{k+1} = a_i$ for some $i \in [n]$ then any job which finishes by time t_{k+1} also finishes by time t_k so $\mathcal{S}_{t_{k+1}} = \mathcal{S}_{t_k}$ and thus $w(t_{k+1}) = w(t_k)$. If $t_{k+1} = b_j$ for some $j \in [n]$ then

$$\begin{aligned}\mathcal{S}_{t_{k+1}} &= \mathcal{S}_{t_k} \cup \{S' = (i_1, \ldots, i_m) \in \mathcal{S}_{t_{k+1}: i_m = j}\} \\ &= \mathcal{S}_{t_k} \cup \{S' = (i_1, \ldots, i_m) : i_m = j, S = (i_1, \ldots, i_{m-1}) \in \mathcal{S}_{a_j}\}\end{aligned}$$

This implies that

$$max_{S' \in \mathcal{S}_{t_{k+1}}}\{w(S')\} = \max\{\max_{S' \in \mathcal{S}_{t_k}}\{w(S')\}, \max_{S \in \mathcal{S}_{a_j}}\{w(S)\} + w_j\}$$

and thus $w(t_{k+1}) = \max\{w(t_k), w(a_j) + w_j\}$

Proof by showing an upper an lower bound:

Again, we can observe that if $t_{k+1} = a_i$ for some $i \in [n]$ then any schedule which finishes by time t_{k+1} also finishes by time t_k so $w(t_{k+1}) = w(t_k)$.

If $t_{k+1} = b_j$ then we can show that $w(t_{k+1}) = \max\{w(t_k), w(a_j) + w_j\}$ by showing that $w(t_{k+1}) \geq \max\{w(t_k), w(a_j) + w_j\}$ and by showing that $w(t_{k+1}) \leq \max\{w(t_k), w(a_j) + w_j\}$.

To show that $w(t_{k+1}) \geq \max\{w(t_k), w(a_j) + w_j\}$, observe that by the definition of $w(t_k)$ there is a schedule S that finishes by time t_k with total weight $w(t_k)$. This schedule S finishes by time t_{k+1} so we have that $w(t_{k+1}) \geq w(t_k)$. Similarly, by the definition of $w(a_j)$, there is a schedule S which finishes by time a_j with total weight $w(a_j)$. Adding job j to this schedule gives a schedule S' which finishes by time $t_{k+1} = b_j$ and has total weight $w(a_j) + w_j$ so we have that $w(t_{k+1}) \geq w(a_j) + w_j$. Putting these pieces together, we have that $w(t_{k+1}) \geq \max\{w(t_k), w(a_j) + w_j\}$, as needed.

To show that $w(t_{k+1}) \leq \max\{w(t_k), w(a_j) + w_j\}$, observe that by the definition of $w(t_{k+1})$ there is a schedule S' which has weight $w(t_{k+1})$ and finishes by time t_{k+1}. There are two cases to consider. If S' does not contain job j then S' finishes by time t_k and thus has weight at most $w(t_k)$ which implies that $w(t_{k+1}) = w(S') \leq w(t_k)$. If S' contains job j then S' consists of a schedule S which finishes by time a_j plus the job j. $w(S) \leq w(a_j)$ so we have that $w(t_{k+1}) = w(S') = w(S) + w_j \leq w(a_j) + w_j$. In both cases, $w(t_{k+1}) \leq \max\{w(t_k), w(a_j) + w_j\}$, as needed. □

Remark 2.6. *If the assumption that there are no ties is not true then we can preprocess the problem by subtracting ϵj from each b_{i_j} and adding ϵj to each a_{i_j} for a sufficiently small $\epsilon > 0$. This will make this assumption true while not affecting which schedules are valid. Equivalently, we can break ties as follows:*

1. *For all $i, j \in [n]$, if $a_i = b_j$ then we put a_i before b_j.*
2. *For all $i < j \in [n]$, if $a_i = a_j$ then we put a_i before a_j.*
3. *For all $i < j \in [n]$, if $b_i = b_j$ then we put b_i before b_j.*

3 The Bellman-Ford Algorithm for Shortest Paths

Problem 3.1 (Shortest Path Problem)**.** *In the shortest path problem with negative edge weights, we are given the edge lengths of a directed graph G on n vertices. These edge lengths may be negative, but G may not contain a negative cycle. We are then asked to find a path of minimum length from some vertex s to another vertex t.*

More precisely, we are given the following data:

1. *A directed graph $G = (V, E)$ togther with a starting vertex $s \in V$ and a destination vertex $t \in V$.*

2. *For each edge $e = (u, v) \in E$, we are given its length $l_e = l_{uv}$.*

We are then asked to output a path P from s to t such that $\sum_{e \in E(P)} l_e$ is minimized.

To solve this problem, we consider the following subproblems: For all $k \in [n]$ (where $n = |V(G)|$) and all $v \in V(G)$, what is the minimum length of a walk from s to v which uses at most k edges?

Definition 3.2. *We define $d_k(v)$ to be minimum length of a walk from s to k which uses at most k edges.*

Proposition 3.3. *If there are no negative cycles in G then for all $k \in \mathbb{N}$ and all $v \in V(G)$, $d_k(v)$ is also equal to the minimum length of a path from s to v which uses at most k edges.*

Proof. Since all paths are also walks, we just need to show that for any walk W from s to v which uses at most k edges, there is a path P from s to v which uses at most k edges which has equal or smaller length than W. To show this, let W be a walk from s to v which uses at most k edges. We can turn W into a path P by deleting all cycles from W. Since there are no negative length cycles in G, the length of P is at most the length of W, as needed. □

We now have the following recurrence relation:

Lemma 3.4. $d_{k+1}(v) = \min\{d_k(v), \min_{u \in V(G):(u,v) \in E(G)}\{d_k(u) + l_{uv}\}\}$

Proof. To see that $d_{k+1}(v) \le \min\{d_k(v), \min_{u \in V(G):(u,v) \in E(G)}\{d_k(u) + l_{uv}\}\}$, we make the following observations:

1. By definition, there is a walk W of length $d_k(v)$ from s to v which uses at most k edges and thus uses at most $k+1$ edges, so $d_{k+1}(v) \le d_k(v)$
2. By definition, for all $u \in V(G)$ there is a walk W of length $d_k(u)$ from s to u which uses at most k edges. Adding the edge (u, v) to this walk we obtain a walk from from s to v of length $d_k(u) + l_{uv}$ which uses at most $k+1$ edges.

Conversely, to see that $d_{k+1}(v) \ge \min\{d_k(v), \min_{u \in V(G):(u,v) \in E(G)}\{d_k(u) + l_{uv}\}\}$, let W be a walk from s to v of minimum length which uses at most $k+1$ edges. We now have the following cases:

1. If W uses at most k edges then W has length at least $d_k(v)$ and thus $d_{k+1}(v) \ge d_k(v) \ge \min\{d_k(v), \min_{u \in V(G):(u,v) \in E(G)}\{d_k(u) + l_{uv}\}\}$.
2. If W uses exactly $k+1$ edge and the final edge of W is (u, v) then W has length at least $d_k(u)+l_{uv}$ and thus $d_{k+1}(v) \ge d_k(u)+l_{uv} \ge \min\{d_k(v), \min_{u \in V(G):(u,v) \in E(G)}\{d_k(u) + l_{uv}\}\}$

□

3.1 Running Time Analysis

Let $n = |V(G)|$ and let $m = |E(G)|$. Naively, there are n^2 subproblems $d_k(v)$ to compute and each $d_k(v)$ takes $O(n)$ time to compute so we obtain a bound of $O(n^3)$. We can improve this bound somewhat by observing that each $d_k(v)$ takes $O(indegree(v))$ time to compute. Thus, for each $k \in [n]$, computing all of the $d_k(v)$ takes time $O(\sum_{v \in V} indegree(v)) = O(m)$ as $\sum_{v \in V} indegree(v) = |E| = m$. Thus, the Bellman-Ford algorithm takes $O(mn)$ time.

3.2 Finding Negative Cycles

If there is a negative cycle in G which is reachable from s then we can find infinitely negative walks to any vertex v which is reachable from the negative cycle. Using the Bellman-Ford algorithm, we can check whether this happens by checking if there are any updates when we look at walks of length n. If there are no negative cycles reachable from s then since all paths in G have length at most $n-1$, there will be no updates. Conversely, if there are no updates for a given k then we have already found the shortest walks from s to every vertex v and we can stop. If there is a negative cycle reachable from s then there is no shortest walk from s to any vertex in this cycle (as we can make these walks have arbitrarily negative length) so there must be updates for all k.

4 Knapsack

Problem 4.1 (Knapsack)**.** *In the knapsack problem, we are given n objects with weights w_i and values v_i as well as a knapsack with capacity C. We are then asked for the maximum total value of objects which can be carried in the knapsack. More precisely, what is $\max_{I \subseteq [n]: \sum_{i \in I} w_i \leq C} \{\sum_{i \in I} v_i\}$?*

To solve this problem when the weights w_i and the capacity C are integers, we consider the following subproblems: what is the maximum total value which can be taken from the first k objects without exceeding weight w?

Definition 4.2. *Define $v(k, w) = \max_{I \subseteq [k]: \sum_{i \in I} w_i \leq w} \{\sum_{i \in I} v_i\}$*

We have the following recurrence relation

Lemma 4.3. $v(k+1, w) = \max\{v(k, w), v(k, w - w_{k+1}) + v_{k+1}\}$

Proof. To prove that $v(k+1, w) \geq \max\{v(k, w), v(k, w - w_{k+1}) + v_{k+1}\}$, we make the following observations:

1. By definition, there is an $I \subseteq [k]$ such that $\sum_{i \in I} w_i \leq w$ and $\sum_{i \in I} v_i = v(k, w)$. Since $I \subseteq [k+1]$, $v(k+1, w) \geq v(k, w)$.
2. By definition, there is an $I \subseteq [k]$ such that $\sum_{i \in I} w_i \leq w - w_{k+1}$ and $\sum_{i \in I} v_i = v(k, w - w_{k+1})$. Taking $I' = I \cup \{k+1\}$, $\sum_{i \in I'} w_i \leq w$ and $\{\sum_{i \in I'} v_i = v(k, w - w_{k+1}) + v_{k+1}$. Thus, $v(k+1, w) \geq v(k, w - w_{k+1}) + v_{k+1}$.

Conversely, by definition there is an $I \subseteq [k+1]$ such that $\sum_{i \in I} w_i \leq w$ and $\sum_{i \in I} v_i = v(k+1, w)$. We now have the following two cases:

1. If $k+1 \notin I$ then $I \subseteq [k]$ and thus $v(k+1, w) \leq v(k, w) \leq \max_{\{v(k,w), v(k, w - w_{k+1}) + v_{k+1}\}}$.
2. If $k+1 \in I$ then taking $I' = I \setminus \{k+1\}$, $\sum_{i \in I'} w_i \leq w - w_{k+1}$ and $\sum_{i \in I'} v_i = v(k+1, w) - v_{k+1} \leq v(k, w - w_{k+1})$ and thus

$$v(k+1, w) \leq v(k, w - w_{k+1}) + v(k+1, w) \leq \max\{v(k, w), v(k, w - w_{k+1}) + v_{k+1}\}$$

□

Remark 4.4. *While the size of the input is $O(nlog_2(C))$, this algorithm runs in time $poly(n, C)$ which is not polynomial in the input size if C is very large. Thus, we say that this algorithm is a pseudo-polynomial time algorithm rather than a polynomial time algorithm. As we will see later in the course, knapsack is NP-hard so unless $P = NP$, there is no polynomial time algorithm for knapsack.*

5 Longest Common Subsequence

In the longest common subequence problem, we are asked to find the longest common subsequence contained in two strings $A = a_1a_2 \ldots a_n$ and $B = b_1b_2 \ldots b_m$. More precisely, we want to find sequences of indices $I = (i_1, i_2, \ldots, i_l)$ and $(j_1, j_2, \ldots, j_l)$ such that:

1. $\forall k \in [2, l], 1 \leq i_{k-1} < i_k \leq n$ and $1 \leq j_{k-1} < j_k \leq m$ (the sequences of indices I and J are in increasing order)
2. $\forall k \in [l], a_{i_k} = b_{j_k}$ (the subsequences of A and B with indices I and J are the same)

and the length l is maximized.

Example 5.1. *Given the strings A = "excellent" and B = "applesauce", the longest common subsequence is "ece" which is letters 1, 3, and 4 (or 7) in "excellent" and letters 5, 9, and 10 in "applesauce".*

This problem can be solved using dynamic programming as follows:

Definition 5.2. *Let $S(i, j)$ be the longest common subsequence in $A_i = a_1a_2 \ldots a_i$ and $B_j = b_1b_2 \ldots b_j$ i.e. the longest common subsequence up to index i in A and index j in B.*

1. Subproblems: What is $S(i, j)$ for $i \in [n], j \in [m]$
2. Recurrence relation:
   (a) If $a_i = b_j$ then $S(i, j) = S(i - 1, j - 1) + 1$
   (b) If $a_i \neq b_j$ then $S(i, j) = \max\{S(i - 1, j), S(i, j - 1)\}$

   where we take $S(0, j) = S(i, 0) = 0$

To see why this recurrence relation holds, note that if $a_i = b_j$ then it is always optimal to match up a_i with b_j and we just need to find the longest common subsequence in $A_{i-1} = a_1a_2 \ldots a_{i-1}$ and $B_{j-1} = b_1b_2 \ldots b_{j-1}$. On the other hand, if $a_i \neq b_j$ then we cannot use both a_i and b_j. If we don't use a_i then our subsequence will have length at most $S(i - 1, j)$ and if we don't use b_j then our subsequence will have length at most $S(i, j - 1)$. Thus, the longest common subsequence in $A_i = a_1a_2 \ldots a_i$ and $B_j = b_1b_2 \ldots b_j$ has length $\max\{S(i - 1, j), S(i, j - 1)\}$.

Running time analysis: There are $O(n^2)$ subproblems $S(i, j)$ and it takes $O(1)$ time to compute each $S(i, j)$ from the answers to the previous subproblems so the total running time is $O(n^2)$.

6 Non-crossing matchings

Given an undirected graph G with vertices $V(G) = \{v_1, \dots, v_n\}$ arranged in a circle, what is the largest matching M with no crossings?

We can state this problem more precisely with the following definition:

Definition 6.1. *We say that a set of edges M is a non-crossing matching of G if $M \subseteq E(G)$ and for all pairs of edges $(v_i, v_j), (v_{i'}, v_{j'}) \in M$ where $i < j$ and $i' < j'$, neither of the following cases hold:*

1. $i < i' < j < j'$
2. $i' < i < j' < j$

or equivalently, one of the following cases holds:

1. $i < j \leq i' < j'$ *i.e.* (v_i, v_j) *comes before* $(v_{i'}, v_{j'})$
2. $i' < j' \leq i < j$ *i.e.* $(v_{i'}, v_{j'})$ *comes before* (v_i, v_j)
3. $i \leq i' < j' \leq j$ *i.e.* $(v_{i'}, v_{j'})$ *is inside of* (v_i, v_j)
4. $i' \leq i < j \leq j'$ *i.e.* (v_i, v_j) *is inside of* $(v_{i'}, v_{j'})$

Definition 6.2. *For all $i < j \in [n]$, we define $M(i, j)$ to be the size of the largest non-crossing matching M such that M only contains edges between the vertices $v_i, \dots, v_j$.*

1. Subproblems: What is $M(i, j)$ for all $1 \leq i < j \leq n$?
2. Recurrence relation:

$$M(i,j) = \max\left\{M(i, j-1), \max_{\{i' : i \leq i' < j, (v_{i'}, v_j) \in E(G)\}} M(i, i'-1) + M(i'+1, j-1) + 1\right\}$$

To see why this recurrence relation holds, note that either v_j is matched with a vertex $v_{i'}$ where $i' \in [i, j-1]$ or v_j is not matched with any of these vertices. If v_j is not matched with any of these vertices then the largest non-crossing matching we can obtain has $S(i, j-1)$ edges. If v_j is matched with a vertex $v_{i'}$ where $i' \in [i, j-1]$ then the largest non-crossing matching we can obtain within the vertices $v_i, \dots, v_j$ has $M(i, i'-1) + M(i'+1, j-1) + 1$ edges. Thus, the largest matching we can obtain if v_j is matched up has $\max_{i' : i \leq i' < j, (v_{i'}, v_j) \in E(G)} M(i, i'-1) + M(i'+1, j-1) + 1\}$ edges.

Running time analysis: There are $O(n^2)$ subproblems $M(i, j)$ and it takes $O(n)$ time to compute each $M(i, j)$ from the answers to the previous subproblems so the total running time is $O(n^2 * n) = O(n^3)$.