1 Non greedy algorithms (which we should have covered earlier)

1.1 Floyd Warshall algorithm

This algorithm solves the all-pairs shortest paths problem, which is a problem where we want to find the shortest distance between each pair of vertices in a graph, all at the same time. Floyd-Warshall runs in $O(n^3)$ time.

1.1.1 Idea

Label the nodes $1, \ldots, n$. Consider the intermediate vertices of shortest paths (i.e. the vertices on a path from u to v that are not equal to u or v).

Define a subproblem (u, v, k), where we consider the shortest path from u to v that only uses nodes 1 through k as intermediate vertices. (Then for all pairs of vertices (u, v), we are ultimately interested in the subproblems (u, v, n), which tell us the shortest distance from u to v that may include any of the vertices as intermediate vertices.)

Either this shortest path p uses node k, or it does not.

If it doesn't use node k, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Therefore, the shortest path for subproblem (u, v, k) is the same as the shortest path for subproblem $(u, v, k-1)$.

If it does use node k, then the shortest path (that uses the first k vertices as intermediate vertices) can be decomposed into the shortest path from u to k (that uses the first $k-1$ vertices as intermediate vertices) plus the shortest path from k to v (that uses the first $k-1$ vertices as intermediate vertices). (If this wasn't true, then we could replace either of the subpaths with a shorter path, deriving a contradiction.)

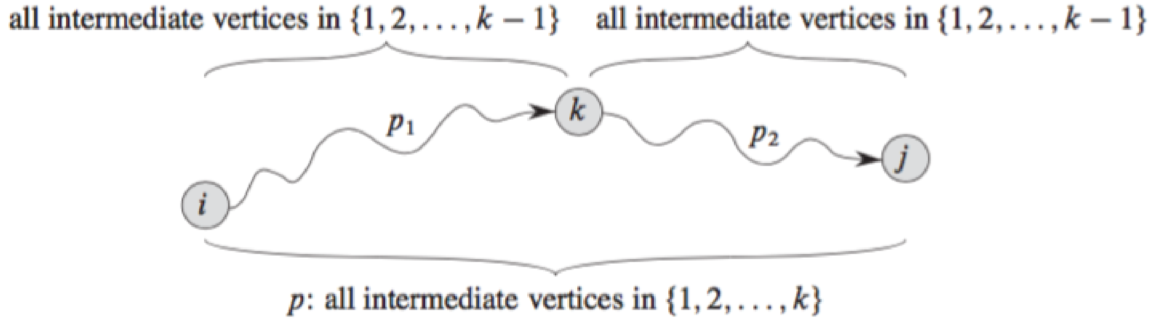
all intermediate vertices in $\{1, 2, \ldots, k-1\}$
all intermediate vertices in $\{1, 2, \ldots, k-1\}$
p_1
k
p_2
j
i
p: all intermediate vertices in $\{1, 2, \ldots, k\}$

Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

1.1.2 Recursive formulation

Let $d_{ij}^{(k)}$ be the solution to the subproblem (i, j, k), i.e. the weight of a shortest path from vertex i to vertex j where all the intermediate vertices are in the set $\{1, 2, \ldots, k\}$. Then

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

The base case, $k = 0$, is just the paths with no intermediate vertices (recall that the vertices are numbered starting from 1, and not starting from 0). So $d_{ij}^{(0)}$ is just w_{ij}, the weight of the edge from i to j. (If there is no edge from i to j, we say the weight of the edge is infinity.)

The output of the algorithm is just the matrix of distances $\delta(i, j) = d_{ij}^{(n)}$.

```
FLOYD-WARSHALL($W$)
$n = W.rows$
$D^{(0)} = W$
for $k = 1$ to $n$
    let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
    for $i = 1$ to $n$
        for $j = 1$ to $n$
            $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
return $D^{(n)}$
```

1.1.3 Shortest paths

To find the actual shortest paths in the graph (as opposed to just the shortest distances), we keep track of the predecessor matrices $\Pi^{(k)}$, where $\pi_{ij}^{(k)}$ is the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

To compute $\pi_{ij}^{(k)}$, observe that there are once again two cases: the case where k is on the shortest path from i to j that only includes intermediate vertices in the set $\{1, \ldots, k\}$, and the case where k is not on the shortest path.

If k is not on the shortest path, then $\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$, since the predecessor of j should be the same as the predecessor for the case where we only use the first $k-1$ vertices as intermediate vertices.

If k is on the shortest path, then $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$, since the predecessor of j from i is the same as the predecessor of j that we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

So we have

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

And

$$\pi_{ij}^{(0)} = \begin{cases} \texttt{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

2 Greedy algorithms

When solving an optimization problem using dynamic programming, you make a “choice” at each step, and you find the optimal solution given the choice you’ve made. The idea is to try all possible choices and choose the one that leads to the best optimal solution.

For example, in the rod cutting problem, you are trying to find the maximum amount of money r_i you can get by cutting a rod of length i. So first you want to cut off a piece of length j (selling it for p_j dollars), and then you want to find the maximum amount of money you can get by cutting the remaining rod of length $i - j$. In the dynamic programming solution, we try this for all possible values of j, and find the max over all of the optimal solutions to find the overall optimal solution, i.e. $r_i = \max_{1 \le j \le i}(p_j + r_{i-j})$.

To find r_n, we just compute r_0, r_1, r_2, etc in sequence until we get to r_n.

With greedy algorithms, instead of looking at all the choices and deciding between them, we focus on one choice: the greedy choice. The greedy choice is the choice that looks best at any given moment. Once we have made it, we are free to solve a smaller subproblem, which we can also solve with a greedy choice, etc.

Importantly, there are many cases where greedy algorithms don’t work, even when you feel like they should. So even when a particular greedy algorithm seems intuitively obvious, it is important to rigorously prove correctness (more so than in other parts of this class). This often means arguing that (on each step) if the optimal solution doesn’t use the greedy choice, we can replace it with a better solution (or an equally good solution) that does use the greedy choice.

2.1 The activity selection problem

In this problem, you have a list of proposed activities $1, \ldots, n$. Activity i starts at time s_i and finishes at time f_i. You want to do as many activities as possible while avoiding time conflicts.

Formally, we want to find a maximum-size subset of mutually compatible activities. Activities i and j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

We assume that the activities are sorted in monotonically increasing order of finish time, i.e. $f_1 \leq f_2 \leq \cdots \leq f_n$.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Example: In the figure, the set $\{3, 9, 11\}$ consists of mutually compatible activities. It is not a maximum subset, since the subset $\{1, 4, 8, 11\}$ is larger. In fact, this subset is a largest subset of mutually compatible activities.

Note that there may be multiple optimal solutions – another one is $\{2, 4, 9, 11\}$. However, our algorithm only needs to find one of them.

2.1.1 Idea

Claim: We can find an optimal solution by always selecting the activity with the earliest finishing time f_1, then recursing on the activities whose start times are $\geq f_1$.

(The idea is to always select the activity that leaves the most resources available for other activities.)

Example: In the table, we start by choosing activity 1, which finishes at time 4. Now we recurse on activities that start after time 4 (i.e. activities 4, 6, 7, 8, 9, and 11). Then we choose activity 4, since that is the activity with the earliest finishing time in our remaining set. Now we recurse on activities 8, 9, and 11. We choose activity 8, which finishes at time 11, and then the only activity left over is activity 11. This produces the set $\{1, 4, 8, 11\}$.

2.1.2 Algorithm

We consider the intermediate subproblem S_k, which is the set of activities that start after activity k finishes. Our goal is to prove that we can find some optimal solution A_k to this subproblem that includes the greedy choice x as one of the activities. Then at each step of the algorithm, it will be safe to make the greedy choice, and then recursively consider

1. The activities that finish before x starts (note: this is the empty set, since x is guaranteed to be the activity in S_k with the smallest finishing time)
2. The activities that start after x finishes, which means we have to solve the subproblem S_x. (When we are solving S_x we can once again use the greedy choice, which in this context, means the activity in S_x with the earliest finishing time.)

Then, we may end up writing the pseudocode as follows:

```
RECURSIVE-ACTIVITY-SELECTOR($s, f, k, n$)
$m = k + 1$
while $m \leq n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
    $m = m + 1$
if $m \leq n$
    return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR($s, f, m, n$)
else return $\emptyset$
```

2.1.3 Correctness

Theorem: Consider any nonempty subproblem S_k, and let x be an activity in S_k with the earliest finish time. Then x is included in some maximum-size subset of mutually compatible activities in S_k.

Proof: Suppose we have an "optimal subset" of activities A_k (note that there may be more than one optimal subset). Let j be the activity in A_k with the earliest finishing time. If $j = x$, then we are done, since we have shown that x is some maximum size subset of mutually compatible activities of S_k.

If $j \neq x$, then consider the set of activities $A'_k = A_k - \{j\} \cup \{x\}$, where we swap out the first activity in the "optimal subset" with the first activity in the entire subproblem. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, j is the first activity in A_k to finish, and $f_x \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum size subset of mutually compatible activities of S_k, and it includes activity x.

2.1.4 Iterative pseudocode

In practice, it is easier to write the code iteratively, as follows:

```
GREEDY-ACTIVITY-SELECTOR($s, f$)
$n = s.length$
$A = \{a_1\}$
$k = 1$
for $m = 2$ to $n$
    if $s[m] \geq f[k]$
        $A = A \cup \{a_m\}$
        $k = m$
return $A$
```

This algorithm runs in time $O(n)$.

2.2 Knapsack problem

Now we will consider another application of dynamic programming, the knapsack problem. This problem is interesting in part because the greedy strategy doesn't work on one variant of the problem, but if we change the problem slightly, the greedy strategy does work.

2.2.1 0-1 knapsack problem

Suppose we have n items $\{1, \ldots, n\}$ and we want to decide which ones to take to the pawn shop. Each item i has a value v_i (which represents how much we could sell it for at the pawn shop), and a weight $w_i > 0$. Our knapsack can only hold a total weight of W, which constrains which items we can bring.

Question: How can we choose the best set of items to bring?

Formally: Choose a set of items $S \subseteq \{1, \ldots, n\}$ that maximizes $\sum_{i \in S} v_i$, subject to the constraint that $\sum_{i \in S} w_i \leq W$.

Greedy solution: Loop through all the items; on each iteration, choose the item with the greatest "bang for your buck", i.e. the greatest v_i/w_i.

Example where this doesn't work: Suppose your knapsack has size 50. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars.

The greedy strategy would choose item 1 and then item 2, leaving no room for item 3. This strategy is worth 160 dollars total. However, the optimal solution chooses items 2 and 3, which is worth 220 dollars.

Dynamic programming solution: On the nth step of the algorithm, we decide whether to include the nth item in the knapsack. If we include the nth item, then the problem reduces to stuffing a subset of the first $n-1$ items into a knapsack of size $W - w_n$. If we don't include the nth item, the problem reduces to stuffing a subset of the first $n-1$ items into a knapsack of size W.

Let $A[i, j]$ be the maximum amount of money you can get from stuffing the first i items into a knapsack of size j. (After populating the table, the answer will be stored in $A[n, W]$.) Then

$$A[i, j] = \max \begin{cases} A[i-1, j] \\ A[i-1, j-w_i] + v_i \end{cases}$$

And we can write the code as follows:

```
for j = 0 to W:
    A[0, j] = 0
```

```
for i = 1 to n:
    for j = 0 to W:
        if w[i] > j:
            A[i, j] = A[i - 1, j]
        else:
            A[i, j] = max(A[i - 1, j], A[i - 1, j - w[i]] + v[i])
```

Runtime: $O(nW)$

2.2.2 Fractional knapsack problem

In the fractional knapsack problem, you can take fractions of items, instead of either deciding to bring the item or not bring the item. In this case, the greedy strategy does apply. First we compute the value per pound v_i/w_i for each item. Then we take as much as possible of the item with the greatest value per pound. Once the supply of that item is exhausted, we move on to the item with the second-greatest value per pound, etc.

Correctness: Assume the items are sorted in order of decreasing v_i/w_i. Denote a solution to this problem by $s = (s_1, s_2, \dots, s_n)$, where s_i is the total amount of the ith item that we choose to carry. The greedy algorithm works by assigning $s_1 = \min(w_1, W)$, and then recursively solving the subproblem with the last $n - 1$ items, and a knapsack of weight $W - w_1$.

To show this strategy gives an optimal solution, we proceed by contradiction. Suppose the optimal solution is $s_1, s_2, \dots, s_n$, where $s_1 < \min(w_1, W)$. (Note that s_1 cannot be greater than $\min(w_1, W)$ because the total amount of item 1 is only w_1, so we are really just comparing to the greedy case, where $s_1 = \min(w_1, W)$.) Let i be the largest number such that $s_i > 0$. By decreasing s_i to $\max(0, s_i - (\min(w_1, W) - s_1))$ and increasing s_1 by the same amount, we get a better solution, which contradicts the optimality of the existing solution. Therefore, the greedy solution is the optimal solution.