9.1 Dynamic Programming on Graphs

9.1.1 Path Counting: Dynamic programming in DAG

Suppose that we have a DAG $G = (V, E)$ of $n$ nodes. Let $s$ and $t$ be two nodes of $G$. We want to count how many paths (not necessarily shortest) from $s$ to $t$.

A common mistake that many people make is to exhaustively search the paths and count them. Note that there might be an exponential number of paths, so this brute-force approach would be prohibitively slow. To understand how many paths there are, let’s consider the following extreme case. Suppose that there are $n$ nodes in the graph, labeled $0, 1, 2, \ldots, n - 1$. For every pairs $(i, j)$ such that $i < j$, there is a directed edge from node $i$ to node $j$. The source $s$ is node 0, and the target $t$ is node $n - 1$. This is the DAG of the most number of edges on $n$ nodes, and the number of paths from $s$ to $t$ here is the maximum among DAGs of $n$ nodes. See Figure 9.1 for an illustration of the case $n = 4$.

![Figure 9.1: A maximally connected DAG, on four nodes.](image)

Let $A_k$ denote the number of paths from node 0 to node $k$ in our DAG. Note that since edges only go forward, to count $A_k$, we only need to draw the graph up to node $k$. Below are the values of $A_k$ for small values of $k$.

<table>
<thead>
<tr>
<th>Index $k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path count $A_k$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

It looks like for $k \geq 1$, we have $A_k = 2^{k-1}$. So the number of paths from $s$ to $t$ can be exponential. We therefore should avoid an exhaustive search of those paths, and yet manage to count them. To handle this complex problem, let’s use the specialization technique get some clue on how to attack. Consider a DAG on $n$ nodes where nodes again are labeled from 0 to $n - 1$. For every pair $(i, j)$ such that $j \in \{i + 1, i + 2\}$, there is a directed edge from $i$ to $j$. See Figure 9.2 for an illustration of $n = 8$.

Let $B_k$ denote the number of paths from node 0 to node $k$ in our second DAG. Note that since edges only go forward, to count $B_k$, we only need to draw the graph up to node $k$. Below are the values of $B_k$ for small values of $k$. 

9-1
Well, so \( \{B_k\} \) is the Fibonacci sequence. In other words, \( B_k = B_{k-1} + B_{k-2} \) for every \( k \geq 2 \). But nodes \( k-1 \) and \( k-2 \) are the only nodes that have edges to node \( k \), so maybe there is a connection between this relationship and the recurrence? To see if our hunch is correct, let’s revisit the maximally connected DAG. Here node \( k \) has incoming edges from nodes \( 0, \ldots, k-1 \). We have \( A_k = 2^{k-1} \), and 

\[
A_0 + \cdots + A_{k-1} = 1 + (1 + 2 + \cdots + 2^{k-2}) = 1 + (2^{k-1} - 1) = 2^{k-1}.
\]

Voila! We have \( A_k = A_0 + \cdots + A_{k-1} \). So our hunch is again true for this particular case. To continue our victory, consider a DAG \( G = (V, E) \) of \( n \) nodes that are labeled \( 0, \ldots, n-1 \). Edges only go forward, meaning that for any pair \((i, j)\) such that \( i > j \), there is no edge from \( i \) to \( j \). Let \( F_k \) be the number of paths from node \( 0 \) to node \( k \) in \( G \). Let \( S_k \) be the set of nodes that have edges to node \( k \) in \( G \). Then a plausible recurrence is

\[
F_k = \sum_{i \in S_k} F_i.
\]

How would we justify this? What’s the relationship between \( S_k \) and the paths from \( 0 \) to \( k \)? Nodes in \( S_k \) have edges coming to node \( k \). So if we want to go from node \( 0 \) to node \( k \), in the last hop, we’ll move from a node in \( S_k \) to \( k \). Thus for paths that have to land to a node \( i \in S_k \) right before jumping to node \( k \), the number of those paths is exactly \( F_i \). So the recurrence above is justified.

Let’s get back to our original problem. First, we somehow need to label our nodes \( 0, 1, \ldots, n-1 \) such that (i) \( s \) is labeled \( 0 \), and (ii) edges only go forward. This means that we need a topological ordering of the nodes, but \( s \) needs to be at the top. This won’t be possible if \( s \) has incoming edges. However, note that those edges won’t contribute to any path starting from \( s \), since there’s no cycle. Hence, we can delete all incoming edges to \( s \), and topologically sort the graph such that \( s \) is node \( 0 \). Let \( F_k \) denote the number of paths from node \( 0 \) to node \( k \). The recurrence suggests us to use dynamic programming to compute all \( F_k \). The code is given below.

\[
F[0] \leftarrow 1 \\
\text{for } k \leftarrow 1 \text{ to } n-1 \text{ do} \\
\quad F[k] \leftarrow 0 \\
\quad \text{for every every incoming edge } (i, k) \text{ to node } k \text{ do } F[k] \leftarrow F[k] + F[i]
\]

To analyze the running time, note that we have to consider every node, and for each node \( k \), we consider all of its incoming edges. Hence the total running time is \( O(m + n) \), where \( m \) is the number of edges.

**Exercise 9.1 (Shortest paths in DAG)** Suppose that we are given a weighted DAG of \( n \) nodes and \( m \) edges, where weights might be negative. Let \( s \) be a node in the DAG. Find the shortest paths from \( s \) to all nodes in the graph, using \( O(m + n) \) time.
9.1.2 Shortest paths up to \( k \) hops: Dynamic programming in general graphs

Suppose that we are given a weighted, directed graph of \( n \) nodes in which weights may be negative. Our goal is to find a shortest path from a source \( s \) to a target \( t \), subject to the restriction that this path must have at most \( k \) hops.

At the first glance, it is unclear how to do dynamic programming in a general graph. Dynamic programming requires some order so that we can find a recurrence. For a DAG, we can use the topological order of its nodes, but a general graph does not offer this kind of ordering. To resolve this issue, we transform our graph to a DAG as follows. We essentially make \( k + 1 \) copies of the original graph, organized in \( k + 1 \) layers:

for each node \( A \) in the original graph, the DAG will have nodes \( A_0, A_1, \ldots, A_k \). Edges only go from one layer to the next: for each directed edge \((A, B)\) of weight \( w \) in the original graph, we create directed edge \((A_0, B_1), (A_1, B_2), \ldots, (A_{k-1}, B_k)\) of the same weight \( w \). To enforce the constraint that each \( A_0, \ldots, A_k \) are just the same node, we create directed edges \((A_0, A_1), (A_1, A_2), \ldots, (A_{k-1}, A_k)\) of weight 0. See Figure 9.3 for an illustration.

![Diagram](image)

Figure 9.3: A weighted graph (left) and its transformed DAG (right), for \( k = 3 \). Dashed edges in the DAG have weight 0.

The transformed graph above is indeed a DAG, since edges only go from a lower layer to a higher layer. If the original graph has \( n \) nodes and \( m \) edges, the resulting DAG has \((k + 1)n\) nodes and \(k(m + n)\) edges. A shortest path from \( s \) to \( t \) in the original graph, of up to \( k \) hops, will become a shortest path from \( s_0 \) to \( t_k \) in the transformed DAG. Then from Exercise 9.1, we can find such a shortest path in the DAG using \( O(k(m + n)) \) time. We then can convert it back to the corresponding path in the original graph using \( O(k) \) time.\(^1\) Hence the total running time is \( O(k(m + n)) \) time.

**Exercise 9.2** The algorithm above uses \( O(k(m + n)) \) space. If the graph is big and so is \( k \) then we may not have enough memory to run it! Reduce the space usage to \( O(m + n) \) but keep the \( O(k(m + n)) \) running time.

\(^1\)For instance, in the example in Figure 9.3, if the shortest path in the DAG is \( A_0 \rightarrow B_1 \rightarrow D_2 \rightarrow D_3 \), by dropping the subscripts we obtain \( A \rightarrow B \rightarrow D \rightarrow D \). Then by removing duplicate nodes, we obtain \( A \rightarrow B \rightarrow D \).
9.1.3 Paint (ACM Regional Contest 2016, USA Southeast)

**Question:** You are painting a fence with $n$ slats, numbered from 1 to $n$. There are $k$ artists, each willing to paint their design on a specific portion of the fence. However, artists will never agree to have their slats painted over, so they will only paint their portion of the fence if no one else will paint any part of it. You want to select a set of painters that does not conflict to minimize the number of unpainted slats. For example, suppose there are 8 slats, and 3 painters. One painter wants to paint slats 1 → 3, one wants to paint 2 → 6, and one wants to paint 5 → 8. By choosing the first and last painters, you can paint most of the slats, leaving only a single slat (slat 4) unpainted, with no overlap between painters.

**The discovery dialog.** Below is a dialog between a teacher and a student for solving the problem above.

   **Student:** I think it’s a graph problem since it involves relationship (conflicts) among painters. Thus it seems intuitive that each painter corresponds to a node. Not sure about the edges. A plausible approach is to connect $u$ and $v$ only if they don’t have conflict.

2. **Teacher:** There’s something missing here. Consider the extreme case that everybody has conflict with everybody else. (For example, each painter wants to paint more than $n/2$ slats.) In this particular case, the graph is $n$ isolated node, and you can only pick a single node. How would you know which one you should pick?
   **Student:** So each node needs to have a weight, so that in the example above, I can pick the heaviest node.

3. **Teacher:** We actually prefer *edges* to have weight so that we can run classic graph algorithms. But let’s start with a node-weight graph first, if that’s more intuitive to you. We can always later transform it to an edge-weight graph later. How would you pick a weight for each node?
   **Student:** From the example above, it’s clear that the weight should be the number of slats the painter wants to cover.

4. **Teacher:** OK, so you have a node-weight graph. How would you formulate the output?
   **Student:** We want to find a group of conflict-free painters, which corresponds to a subgraph such that everybody is connected to everybody else. This kind of graph is known as a *clique*. So we have to find a clique of maximum weight.

5. **Teacher:** Do you have a brute-force algorithm?
   **Student:** I can look at every subset of painters—there are $2^k$ such subsets. For each subset $S$, I check if the corresponding subgraph is a clique in $O(N^2)$ time, where $N = |S|$, and compute their weights in $O(N)$ time. Totally, I’ll need $O(2^k \cdot k^2)$ time.

6. **Teacher:** Now, given a subset $S$ of $N$ painters with intervals $a_i \rightarrow b_i$, we want to check if they are conflict-free using less time than $\Theta(N^2)$. Is there a set of items that can be sorted by size or some key?
   **Student:** Well, I can keep the intervals in two copies, one sorted by $a_i$, and another sorted by $b_i$. For each interval $(a, b)$, it has conflict with an interval $(A, B)$ if either (i) $a \leq A \leq b$, or (ii) $A < a \leq B$. I now can use binary search to check if each interval has conflict with some other interval using $O(\log(N))$ time. Totally I only need $O(N \log(N))$ time.

7. **Teacher:** So for the clique checking problem, you need $O(N^2)$ time if you just use the graph, but you can improve that to $O(N \log(N))$ if you go back to the list formulation. What’s the reason?
**Student:** So in the list formulation, there is a way to reorder the painters so that we can perform binary search. In contrast, this ordering is lost if we simply look at the graph.

8. **Teacher:** So how would you enforce this ordering in the graph representation?

**Student:** I think now an edge should have direction. We have \( u \to v \) if these painters have no conflict, and the interval of \( u \) is before that of \( v \) in the fence. The graph has the following property: if \( u \to v \) and \( v \to w \), then of course \( u \to w \) as well. Constructing the graph takes \( O(k^2) \) time.

9. **Teacher:** If you have a list of \( q \) nodes, what’s the criteria that the corresponding painters are conflict-free?

**Teacher:** These \( q \) nodes must form a path \( v_1 \to v_2 \to \cdots \to v_q \).

10. **Teacher:** What’s the formulation of the output?

**Student:** We’ll find a path of maximum weight in the graph.

11. **Teacher:** But the longest path problem has very different kinds of solutions, depending on whether the graph is a DAG or not. Is your graph a DAG?

**Student:** A directed edge \( u \to v \) implies that the interval of \( u \) is before that of \( v \) in the fence. This order means that it’s impossible to have cycles. The graph is a DAG of \( k \) nodes and \( O(k^2) \) edges.

12. **Teacher:** The standard longest path algorithm needs a source node and a target node. If you try every pair of (source, target) then that will be too expensive.

**Student:** I can create a source and a target of weight 0. The source has directed edges to all original \( k \) nodes, and these \( k \) nodes in turn have directed edges to the target node. Note that adding these \( k \) edges do not create cycles, and the graph remains a DAG. The new graph has at most \( k + 2 \) nodes and at most \( k^2 + 2k \) edges.

13. **Teacher:** Your graph is node-weight, but standard graph algorithms are for edge-weight graphs. How would you deal with this?

**Student:** I can apply the transform in Scribe 9 to turn a node-weight graph to an edge-weight one. The transformation cost is \( O(k^2) \).

14. **Teacher:** What’s the running time to find the longest path in your graph?

**Student:** The graph is a DAG of \( k + 2 \) nodes and \( O(k^2) \) edges, so I can solve this using \( O(k^2) \) time.

**Reflection.** In this problem, again you can see the usefulness of studying the brute-force algorithm, despite its horribly slow time \( O(2^k \cdot k^2) \). This algorithm presents another problem (checking if \( N \) painters are conflict-free) that lets us realize that the graph representation misses some ordering among the painters, and thus we should consider a directed graph instead.

It also useful to start with a node-weight graph, since in this problem constructing this graph is more intuitive. In contrast, if we try an edge-weight graph from the beginning, we might get lost because we couldn’t realize that we need a source and target nodes.

### 9.2 The Graph View of Dynamic-Programming Problems

In this section, we revisit some old dynamic-programming problems and show how to model them as graph problems. While efficiency remains the same, the graph approach often gives simpler and more intuitive solutions.
COIN CHANGING. Suppose that our currency has 3 types of coins: dimes (10 cents), nickles (5 cents), and pennies (1 cent). We want to give change to \( n \) cents to minimize the number of coins.

To see why this problem can be viewed as a graph problem, consider the following directed graph of \( n + 1 \) nodes: 0, 1, \ldots, \( n \). For each node \( i \geq 1 \), we create a directed edge \((i, i - 1)\). If \( i \geq 5 \), create another edge \((i, i - 5)\). Finally, if \( i \geq 10 \), create another edge \((i, i - 10)\). See Figure 9.4 for an illustration of \( n = 6 \).

In the graph above, each node has at most 3 outgoing edges, and thus there are at most \( 3(n+1) = O(n) \) edges in total. Each way to give change corresponds to a path from source \( n \) to destination 0. For example, for \( n = 6 \), if we first give a nickle and then a quarter, then this corresponds to the path \( 6 \rightarrow 1 \rightarrow 0 \). Therefore, a way to minimize the number of coins corresponds to a shortest path from \( n \) to 0. Since this is an unweighted graph, we can use BFS to find shortest paths from node \( n \) to every other node in \( O(n) \) time.

ROD CUTTING. Suppose that you have a rod of length \( n \). You’d like to cut it into pieces and sell them. The length of each piece must be an integer. There’s a table \( P[1 : n] \) that indicates the price \( P[i] \) of a piece of length \( i \). You’ll sell the pieces according to the listed price in \( P \). For example, suppose that your rod has length \( n = 4 \), and the table \( P \) is shown in Table 9.1. If you sell the rod as a whole, you’ll get \( P[4] = 14 \) dollars. If you instead cut it into two pieces, both of length 2, you’ll get \( P[2] + P[2] = 16 \) dollars.

To see why this problem can be viewed as a graph problem, consider the following directed graph of \( n + 1 \) nodes: 0, 1, \ldots, \( n \). For every pair \((i, j)\) with \( i > j \geq 0 \), create a directed edge \((i, j)\) with weight \(-P[i - j]\). See Figure 9.5 for the graph of Table 9.1.
The graph above is a DAG because edges only go backward, from a high node \( i \) to a low node \( j \). Each node has at most \( n \) outgoing edges, and thus there are at most \( n(n + 1) = O(n^2) \) edges. Each way to cut the rod corresponds to a path from source \( n \) to destination \( 0 \), and the profit is the negated value of the total weight of the path. For example, for \( n = 4 \) and the price table given in Table 9.1, if we cut the rod in two pieces of length 2 then this corresponds to the path \( 4 \rightarrow 2 \rightarrow 0 \), which has weight \(-16\). Thus our goal is to find the shortest path from source \( n \) to destination \( 0 \). From Exercise 9.1, this problem can be solved using \( O(n^2) \) time.

A variant of Rod Cutting. Consider the following variant of Rod Cutting. After each cut, the rod becomes weaker, so you can have at most \( k < n \) cuts. Again, we have to find a shortest path in a graph of \( n + 1 \) nodes and \( O(n^2) \) edges, but here we are restricted to paths of at most \( k + 1 \) hops. From Section 9.1.2, this can be solved within \( O(kn^2) \) time.