Topological Sort and Dynamic Programming on Directed Acyclic Graphs

Faraz Mirza

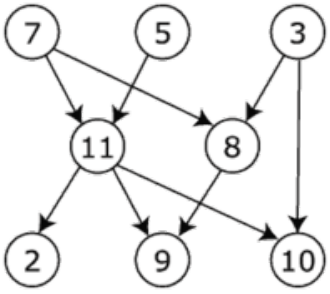
May 28, 2021

Dynamic programming can be applied as a general technique to solve a variety of problems that can be defined by states and relationships between these states. The nature of dynamic programming allows us to easily apply and translate it to a special type of graph. Although dynamic programming on graphs can be approached using standard methods, it is helpful to understand the inherent connections between directed acyclic graphs, and the concept of dynamic programming.

1 Introduction and Motivation

There are two main elements present in dynamic programming algorithms, namely states and transitions. If we represent states as nodes in a graph, we can represent transitions as directed edges between two nodes in the graph, implying that one state can be determined as a function of the nodes which feed into it.

Particularly, this graph should be acyclic, meaning that there should be no path that starts at a node and leads back to the same node, otherwise when calculating the value of a state we would be stuck in an infinite loop. Pictured below is an example of a directed acyclic graph. Notice that there is a hierarchy of nodes, which will be discussed later. As a side note, this hierarchy is also present in trees, which can often be considered as a special type of directed acyclic graph.



To make this idea more concrete, we will look at a well known example. The longest increasing subsequence problem asks us to find the size and identity of the largest set of elements in an array that has elements with increasing value when they are ordered by their indices. If we represent each element of the array as a node in the directed graph, and add directed edges from each node to all nodes representing elements that have a higher index, we can rephrase our problem as the longest path in the directed graph such that the value of the element that each node represents along the path is increasing. In the example above, one such longest increasing path includes the nodes $3, 8$, and 9.

The algorithm to solve this problem would stay the same, but the reasoning behind why we can store the transition which gives us the longest increasing subsequence ending at each element to eventually provide us with the entire longest increasing subsequence becomes more clear. We can backtrack from the final element of the longest increasing subsequence, iteratively resetting the current element represented by the state to the one which provided the optimal transition.

In this case, the value of a node equals 1+the maximum value of all nodes that feed into it which represent a lesser element in the array, initializing the value of each node at 0. However, our transition could be anything taking into account the data stored in each state. For example, if we were instead trying to figure out the shortest path from a source node to a given node, we would make our transition the minimum of all lengths to nodes that feed into the current node plus the corresponding edge weights.

2 Topological Sort

One such algorithm that operates on directed acyclic graphs is known as topological sort. Topological sort answers the question, "If I have a list of directions saying that one task must be done before another, what is a valid ordering of all tasks?" To formalize this problem, we will represent the constraint of task a being before task b as a directed edge from a to b in a graph where tasks are nodes. This graph can not feasibly have cycles, because that would mean that in order to begin a task, we must have already completed that task.

When thinking of an algorithm to solve this ordering problem, we can first think about what we would do in a real life situation. We know that the first task that we do can not have any task that must be before it. So at the beginning of our algorithm, we can take any node that has an in-degree of 0, remove it and its edges from the graph, and add it to our ordering. Now we have a new graph with possibly more nodes that have an in-degree of 0. At each step we only take into account the in-degree of nodes, so reconstructing the graph is not necessary.

Instead we can keep a queue storing each node that has an in-degree of 0. At each step, we remove the front element of the queue and add it to our ordering. We then simulate deleting the node's edges by decreasing the in-degree of each of its children by 1. If any of these children then have an in-degree of 0, we can add them to the queue. In the case that there are multiple valid orderings of tasks, this algorithm will prioritize nodes added to the queue first, thus making relevant the usually arbitrary ordering of nodes in an entry of an adjacency list. Since each edge and node is processed in constant time, the complexity of this algorithm is $O(V + E)$.

If instead we wanted one specific valid ordering, for example the lexicographically smallest valid ordering, we could use a priority queue instead of a queue, allowing us to remove the node with the lexicographically smallest value at each step. This changes our complexity to $O(E + V \log V)$.

Alternatively, we could use a recursive topological sort algorithm. When processing a node, we know that we must include the subgraph reachable by each of its children in the ordering only after including the current node. Because of this, we can recursively call the topological sort on each of its children, and then add the current node to the beginning of the resulting sorted list of nodes. Essentially, we are computing this ordering by the preforming a depth-first search and adding each node to the beginning of a list after it completes all of its recursive calls, so this runs in $O(V + E)$ time.

3 Pseudocode

The pseudocode for the iterative and recursive algorithms are included below in order. These specific versions also identify when the topological sort can not be completed because the graph has a cycle.

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

The depth-first search implementation tests for cycles by using two separate flags representing whether a node has been visited, added to the ordering, or neither.

```
L ← Empty list that will contain the sorted nodes
while exists nodes without a permanent mark do
    select an unmarked node n
    visit(n)

function visit(node n)
    if n has a permanent mark then
        return
    if n has a temporary mark then
        stop   (not a DAG)

    mark n with a temporary mark

    for each node m with an edge from n to m do
        visit(m)

    remove temporary mark from n
    mark n with a permanent mark
    add n to head of L
```

4 Sample Problems

All of these problems can be solved using topological sort or dynamic programming on directed graphs. These concepts do come up often in USACO competitions, but usually are more complex than simply implementing a dynamic programming algorithm and outputting the value for a state. As an implementation detail, it is

usually better to use recursion with memoization for such problems, since there is not always a clear node hierarchy to iterate over given in the problem statement.

http://www.usaco.org/index.php?page=viewproblem2&cpid=1017
http://www.usaco.org/index.php?page=viewproblem2&cpid=838
http://www.usaco.org/index.php?page=viewproblem2&cpid=212
http://www.usaco.org/index.php?page=viewproblem2&cpid=551