Lecture 15: Dynamic Programming

We now discuss the coin change problem. As we mentioned before, the simple greedy algorithm works for some instances of this problem, but does not work for other situations. We now apply dynamic programming to this problem. Let us say we have \( k \) distinct coins, ordered by value in the decreasing order: \( d_1, d_2, \ldots, d_k = 1 \). Here \( d_i > d_{i+1} \). For example, we may have \( d_1 = 25, d_2 = 10, d_3 = 5, d_4 = 1 \). Our goal is to make changes for \( n \) cents using the smallest number of coins. We assume there are unlimited supply of each type of coins.

1. We define the subproblem: \( C[i] \) is equal to the minimum number of coins for \( i \) cents.
2. Initialization: \( C[j] = \infty \) if \( j < 0 \), and \( C[0] = 0 \).
3. Recurrence: \( C[i] = 1 + \min_{x=1}^{k} (C[j - d_x]) \).
4. Based on this recurrence, it is straightforward to write down the algorithm (omitted).
5. Time: \( O(nk) \). This is because there are \( n \) iterations, each taking \( O(k) \) time.
6. How to find the opt solution? Backtrack again. Omit: find out how you get there by using which coin at that step, and recursively search.

In this lecture, we continue our discussion of dynamic programming. We discussed two problems: 0-1 Knapsack, matrix chain multiplication and weighted activity selection.

Recall that in the 0-1 Knapsack problem, we have \( n \) items, each weight \( w_i \) (which is an integer) pound and worth \( v_i \) dollars. The goal is find a subset of items as valuable as possible but no more than \( W \) pounds (here \( W \) is a given integer). For each item, we must either take or not take and thus is called 0-1 Knapsack.

Suppose the optimal solution contains \( k \) items, \( b_1, b_2, \ldots, b_k \). Suppose we remove \( b_k \), then we have the following property: the remaining items, \( b_1, b_2, \ldots, b_{k-1} \) must the most valuable items under the constraints that total weight is no more than \( W - w_k \). Otherwise, we have a contradiction that \( b_1, b_2, \ldots, b_k \) are the most valuable items with no more than \( W \) total weight.

Now we define \( M[i, w] \) as the highest value (in dollar) when we choose items from 1 to \( i \), and total weight is no more than \( w \). Clearly, the solution to our Knapsack problem is simply \( M[n, W] \). It is easy to see that \( M[0, w] = 0 \), and \( M[i, 0] = 0 \) for all \( i/w \). When \( i \geq 1 \) and \( w > 0 \), we have: \( M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w]) \) if \( w_i < w \), and otherwise \( M[i, w] = M[i-1, w] \). This is because we either take item \( i \) or not. If we take item \( i \), then the remaining capacity of Knapsack is reduced by the weight of item \( i \). If not, the capacity remains unchanged. In either case, we should pack as valuable as we can using the remaining items.

1. \( \text{for } w = 0 \text{ to } W \text{ do} \)
2. \( \quad M[0, w] = 0 \)
3. \( \text{end for} \)
4. \( \text{for } i = 1 \text{ to } n \text{ do} \)
5. \( \quad M[i, 0] = 0 \)
6. \( \text{for } w = 1 \text{ to } W \text{ do} \)
7. \( \quad \text{if } w_i \leq w \text{ then} \)
8. \( \quad \quad M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w]) \)
9. \( \quad \text{else} \)
10. \( \quad \quad M[i, w] = M[i-1, w] \)
11. \( \quad \text{end if} \)
12. \( \text{end for} \)
13. \( \text{end for} \)
14. \( \text{return } M[n, W]. \)

Running time: we have \( O(nW) \) cells in \( M \) array, each taking \( O(1) \) to compute. So the total time is \( O(nW) \). Is this a polynomial-time algorithm? Not quite but it works well when \( W \) is relatively small.
Lecture 16: Dynamic programming and Graph Algorithms: BFS

We first discussed two more examples on dynamic programming. The first problem is matrix chain multiplication. Again, this is a problem that is well explained in the textbook. I want to emphasize several things. First, finding the subproblems is often one of the most important aspects of using dynamic programming. For this problem, we define $M[i,j]$ (where $i \leq j$) as the smallest amount of computation needed to multiple matrices $A_i \ldots A_j$. This is natural: maybe the optimal solution will put parenthesis right before $A_i$ and after $A_j$. Of course, we are not sure, but the key idea of DP is to compute the results for all the subproblems and then figure out the overall solution from these subproblems. I will skip the rest of details (since the textbook explains well).

I suggest you to experiment with small examples to understand why the proposed DP algorithm works.

The second problem is assembly line scheduling. Suppose we have two assembly lines of (say) cars. Each has $n$ stations $S_{i,j}$, for $i=1,2$ (referring to which of the two assembly lines), and $j=1 \ldots n$. We assume $S_{1,j}, S_{2,j}$ have the same functions. Note each station $S_{i,j}$ takes processing time $a_{i,j}$. A part enters=ing one assembly line can switch to the other line with a cost $t_{i,j}$ (when the switch happens from AL $i$ station $j$ to the other line station $j+1$). Now the problem is, minimize the total time for this car by picking which stations to work on this car. Again, we apply dynamic programming. We define: $f_i[j]$ = the fastest time to move the car to station $S_{i,j}$ (which includes the time spent at $S_{i,j}$). Then the minimum time for the car to go through the assembly line is $\min(f_1[n], f_2[n])$. We initialize $f_1[1] \leftarrow a_{1,1}$, $f_2[1] \leftarrow a_{2,1}$. Then, for $j=2 \ldots n$, and for $i=1 \ldots 2$, we let $i'$=the other assembly line; then $f_i[j] = \min(f_i[j-1] + a_{i,j}, f_{i'}[j-1] + t_{i,j-1} + a_{i',j})$.

Graph Algorithms. Graph is one of the most important concepts in algorithms. We will spend several weeks on algorithms for graphs. In this lecture, we study two basic algorithms for graph search: breadth-first search (BFS) and depth-first search (DFS). Many of you have known these two algorithms before. But I suggest you to refresh your memory and also think a little deeper about these two algorithms.

First, BFS can find the distances of nodes to a source node. Here, distance just means the smallest number of edges connecting the two nodes. See the textbook for the BFS algorithm. In class, I presented a simpler version of the algorithm: no coloring is used since we just use the distance to check whether a node is visited before. The key question on BFS is why the distances set by the BFS algorithm are correct. The correctness is based on three properties that holds for all $d = 0,1,2,\ldots$ at some point of time (note it is some time, not always): (1) each node with distance (from source $s$) $\leq d$ is set properly, (2) all other nodes are not discovered (i.e. distance is set to $\infty$), (3) the queue contains precisely the nodes with distance $d$. We can see why these hold by using induction. At $d=0$, this is almost trivial: we only have $s$ in stack (whose distance is 0, and $s$ is the only such node). Then assume at some point of time, these three properties hold. Then, we can show that there exists some moment where the three properties hold for $d+1$. Let us focus on one part: the distance set by the first node $u$ of the queue is correct. Note, in this case, a newly discovered node $v$ (distance currently set to $\infty$) will be set to distance $d+1$. Why is this action correct: can $dist(v) \geq d+2$ or $dist(u) \leq d$? First, $dist(v)$ must be at most $d+1$ because $v$ reachable in one hop from a distance $d$ node. Second, $dist(v)$ can not be $d$ or smaller because by the induction assumption, all such nodes (with distance $d$ or smaller) have already been set (and thus $v$’s distance can not be $\infty$. We are not quite done yet, but I hope you have seen where I am leading to. You can finish the details yourself.

Lecture 17: DFS and its applications

The main topic of this lecture is DFS, which is like exploring the maze. Just like exploring the maze, we must try to avoid going in cycles. The tools for this is (a) chalk for marking visited places, and (b) a roll of strings to backtrack. These are easily done by having a flag for each graph node (which is marked when it is visited), and by recursion. In class, I described a slightly simpler algorithm than the one in textbook. There, we just use a flag visited(u) to indicate whether u is visited or not (instead of colors in the textbook). The rest of algorithm is the same as the textbook (read the section if you did not write down all the steps in class). A key utility of DFS is timestamp, which tells when a node is initially discovered and finished. This time interval [u.d, u.f] is often quite useful for some problems. We will continue on this next week.

The timestamp can be very useful. The time interval (discovery time, finish time) has a useful property: time interval for any two nodes traversed by DFS must be either disjoint or one contain another. See the textbook for proof and more explanations.

DFS also helps to classify the graph edges into four types: (a) Tree edge: along which a node is first visited. Why these edges form trees? Any node at most one incoming edge. A node can be discovered just once. (b) Forward edge: lead to an already visited descendant. (c) Back edge: lead to an ancestor (and form a cycle). (d) Cross edge: lead to a node has been explored before but not descendants or ancestors. Forward edge must
go from an ancestor (in the order of node discovery) to a descendant. For example, we have nodes a, b, c and
d, and DFS starts from a and visit b from a, c from b, and then return to c and visit d. In this case, if there is
an edge from d to c, it is not a forward edge because d is not the ancestor of c in DFS tree (that is, we did not
visit c during the traversal of d). Thus, edge from d to c is a cross edge. This means that just like tree edge, the
time interval of forward edge is just like tree edge: starting node’s time interval contains that of ending node.
We also discussed how to determine the types of edges based on the timestamp (time interval). That is, how
can we determine the edge type by just looking at the time intervals? Refer to your notes or textbook if you
forgot.

First, a directed graph G is acyclic iff G has no back edges in DFS. We proved it in class (review it and it
is in textbook). Thus, we can use DFS on the graph, determine the type of each edge and report the graph is
acyclic if there is no back edge.

Now the next application of topological sort. If G is a DAG, we can order its nodes along a line (called
linearization), so that edges go one direction. A useful property is that each DAG allows a topological order.
Here an observation helps: for each edge \((u, v)\) in a DAG, we have \(u.f > v.f\). Why? Suppose \(u.f < v.f\). Then,
can \((u, v)\) be cross edge? No: cross-edge means u’s finish time is later. Can be forward/tree? No: \(u.f > v.f\) in
those cases. So must be back edge. But we know NO BACK edges, since G has no cycles. So what should be
the starting? The one with largest \(u.f\) (called source). Review your notes to see how we did this and how to
analyze the running time.

Now we move to the topic of finding the strongly connected components. Strongly connected component:subset of nodes in the graph, such that each pair of nodes in the subset can go to each other in both directions.
Suppose we shrink each component into a node, this created a condensed graph (called component graph). This
graph must be DAG, otherwise we can merge the components along the cycle form a larger strongly connected
component.

Suppose you start from some node the sink component (no outgoing edges to any other components in
component graph) and perform DFS, you traverse inside this component only. But this only works for sink
component. Second, the graph with highest finish time must be in the source component (i.e. no other
component connects to it). Why? We will continue next time.