Two Ways of Thinking about Dynamic Programming
Jeff Edmonds

Here are two set of steps that you could follow to design a new dynamic programming algorithm. The first are the steps that I teach in 3101 using birds and friends. The second are new. Hopefully, it will help you understand dynamic programming better and not simply confuse you more. The homework will ask you to design an algorithm for a problem using the first method and then another (better) algorithm using the second method.

1 Bird and Friend Steps From 3101

When providing a dynamic programming algorithm and its proof of correctness, the following are all the paragraphs that should be included, their headings, and what they should contain. (See HTA pg 268 for the full description of the steps)

1) Specifications: This is likely part of the question and hence does not need to be written. However, before writing anything consider: What is the set of instances, for each instance what is its set of solutions, and for each solution what is its cost/value.

Algorithm using Trusted Bird and Friend: I have my instance I. The little bird knows a solution to it.

2) Question for Bird: I ask the little bird ... **Choose something like one of the following:**

- Is the last object from the instance in the solution?
- What is the last object in the solution?
- The solution forms a tree. What is the object at the root?

2') Possible Answers from Bird: The list of possible answers that she may give is .... (Enumerate them with $k \in [K]$). (For each such answer) Assume that she gives me answer k.

3) Constructing Subinstances: I give my friend the subinstance $subI$. **(Describe)**. He gives me an optimal solution $optSubSol$ for it.

4) Constructing a Solution for My Instance: I produce an optimal solution $optSol$ for my instance I from the bird's answer k and the friend's solution $optSubSol$. **(Describe)**

5) Costs of Solution: Similarly, I compute the cost $cost$ of our solution $optSol$. **(Describe)**

Recursive Back Tracing Algorithm:

6) Best of the Best: I can trust the friend because he is a recursive version of myself. Not actually having a little bird, I try all her answers and take best of best.

7) Base Cases: The base case instances and their solutions are ... **(Describe)**

Dynamic Programming Algorithm:

1) The Set of Subinstances: The set of subinstances $subI$ ever given to me, my friends, their friends is ... **(Describe)**. Note that this set is closed under this "sub"-operator and all (or at least most) of these subinstances are needed.

3) Construct a Table Indexed by Subinstances: I index these subinstances with i (and maybe j) so that $subI[i,j]$ denotes the subinstance **(Describe)**. I build a table indexed by these subinstances so that $optS[i,j]$ stores an optimal solution for $subI[i,j]$, $cost[i,j]$ the cost of this solution, and $birdAdvice[i,j]$ stores the birds advice given on this subinstance. (Actually we don't store the solution because it is too big.)

6) The Order in which to Fill the Table: The friends solve their subinstances (and the table is filled) in an order so that nobody has to wait. (from smaller to larger instances). **(Describe order)**

8) Code:

algorithm *DynamicProgrammingAlg* (I)

$\langle pre-cond \rangle$**:** I is my instance.

$\langle post-cond \rangle$**:** *optSol* is an optimal solution for I and *cost* is it's cost.

```
begin
    % Table: subI[i,j] denotes the subinstance (Describe).
             optSol[i,j] stores an optimal solution for it (Describe)
    table[i,j] cost, birdAdvice

    % Base Cases: Describe the base cases and their solutions.
    loop over base cases
        optSol[basecases] = its solution
        cost[basecases] = its cost
    end loop

    % General Cases: Loop over subinstances in the table.
    for i ∈ [range]
        for j ∈ [range]
            % Solve instance subI[i,j] and fill in table entry ⟨i,j⟩.
            % Try each possible bird answer.
            for k ∈ [K]
                % The bird and Friend Alg: see above
                optSol_k = Describe how to construct the solution to our instance from the
                           bird's advice k and the solution optSol[friend] to our friends in-
                           stance subI[friend].
                cost_k = Describe how to construct the cost of this solution from the bird's
                         advice k and the cost cost[friend] of our friends solution.
            end for
            % Having the best, optSol_k, for each bird's answer k, we keep the best of these best.
            k_min = "a k that minimizes cost_k"
            % optSol[i,j] = optSol_{k_min}
            cost[i,j] = cost_{k_min}
            birdAdvice[i,j] = k_min
        end for
    optSol = AlgWithAdvice(I, birdAdvice)
    return ⟨optSol, cost[initial instance]⟩
end algorithm
```

8') Constructing the Solution: We would run the recursive algorithm with the bird's advice to find the solution to our instance. We exclude this step from our answer.

9) Running Time: The number of subinstances in the table is ....
The number of bird answers is ....
The running time is the product of these ....

2 Designing a Dynamic Programming Algorithm via the "Where are you" Abstraction and/or "a Reduction to Shortest Paths" Abstraction

Understanding dynamic programming is hard. To help with this, we have already presented the bird/friend abstraction. We will now present a second abstraction. The difference is not huge – just a different story used to describe the same underlying math. It shifts the algorithm designer's initial attention from determining

the question for the little bird to determining the set of subinstances to be solved. It also shifts how one thinks about these subinstances. Hopefully this new abstraction will help your intuition and not confuse you further.

A Reduction to Shortest Paths: This new abstraction of dynamic programming can be considered to be a reduction to the shortest paths in the leveled graph problem. Read HTA 19.8. Given a new optimization problem and an instance I for it, our goal is to design a dynamic programming algorithm which will find an optimal solution for I. The reduction involves constructing a leveled graph G_I such that there is one-to-one correspondence between the solutions for I and the st-paths in G_I and such that the cost/value of each solution is the same as the cost of its corresponding path. Then the dynamic programming algorithm that finds a shortest path in G_I can be used to find an optimal solution for I.

Over View of the new Abstraction: We will refer to this second abstract as the "Where are you?" abstraction. The key difference between the bird/friend abstraction and it has to do with how one thinks about the subinstances. In the first abstraction, after the bird answers a question, you must formulate a subinstance to give your friend. This subinstance needs to meet the precondition of your same computational problem and needs to be smaller than your instance. These restrictions limit the scope of your thinking. In the new abstraction, you go on a journey (without friends) in search of a solution to your instance. The sequence of questions to the bird, "What is the first edge in the shortest path?", "What is the second?", are viewed as the road you are traveling along forking and you needing to choose which direction to continue following. The path you take specifies the sequence of choices you make and as such specifies the solution that you choose. You can add lengths to the road segments that you travel along so that the total length of the path you travel is equal to the cost of the solution corresponding to this path. Then your task of finding the optimal solution is reduced to the task of finding the shortest from the source to the destination through this leveled graph. If the possible paths you could take fork and refork into a tree of exponential size, then searching for the best path amounts to the recursive backtracking algorithm and to the brute force algorithm of trying each of the possible solutions. However, if we can collapse this tree into a DAG (directed acyclic graph) then the running time will be much less. The key to collapsing the tree is to forget the path you took to get where you are and to focus on *where* is is that you are. More specifically, what about our current state will influence your future choices.

Where You Currently Are Splitting The Task: Suppose you are traveling from home to school and you are currently standing at Spadina station. Being at Spadina splits your problem into two completely separate independent problems. The first subtask is to determine an optimal path is from home to Spadina. The second subtask is determining a optimal path is from Spadina to school. These two subtasks are independent in that if you concatenate an optimal solution for the first and an optimal for the second, you will obtain a path from home to school that is optimal from amongst those that go through Spadina Station. What remains in obtaining an overall optimal solution is to repeat this processes for all possible locations that you might stand. This set of possible locations corresponds to the set subinstances that need to be solved by the dynamic programming algorithm.

The detailed steps for designing a dynamic programming algorithm using this abstraction are as follows.

Goal: Be clear about the specification for your new optimization problem. Consider an instance I to the problem. Our goal is to find an optimal solution for it.

***st*-Path:** Recall that an instance to the shortest path in a leveled graph problem consists of a leveled graph G for which there is a known linear ordering of the nodes, the edges have known weights, and the source node s and the sink node t are given. The goal is to find a shortest weighted path from s to t.

Knapsack: As a working example, consider the knapsack problem. An instance consists of $\langle V, \langle v_1, p_1\rangle, \ldots, \langle v_n, p_n\rangle\rangle$. Here, V is the total volume of the knapsack. There are n objects in a store. The volume of the i^{th} object is v_i, and its price is p_i. A solution is a subset $S \subseteq [1..n]$ of the objects that fit into the knapsack, i.e., $\sum_{i \in S} v_i \leq V$. The cost (or success) of a solution S

is the total value of what is put in the knapsack, i.e., $\sum_{i \in S} p_i$. Given a set of objects and the size of the knapsack, the goal is fill the knapsack with the greatest possible total price.

Stocks: Read the stock market question in the homework. The input instance to your problem consists of $I = \langle T, S, Price \rangle$, where T is an integer indicating your last day to be in the market, S is the set of $|S|$ stocks that you consider, and $Price$ is a table such that $Price(t, s)$ gives the price of buying one share of stock s on day t. Buying stocks costs an overhead of 3%. Hence, if you buy p dollars worth of stock s on day t, then you can sell them on day t' for $p \cdot (1 - 0.03) \cdot \frac{Price(t',s)}{Price(t,s)}$. You have one dollar on day 1, can buy the same stock many times, and must sell all your stock on day T. You will need to determine how you should buy and sell to maximize your profits.

A Sequence Choices; Each Path ≈ A Solution: As said, this "Where are you?" abstraction shifts the algorithm designer's initial attention from determining the question for the little bird to determining the set of subinstances to be solved. However, before we can consider "Where you are," you do have to understand that your journey is amounts to a sequence of decisions in the process of choosing a solution to your instance I and that possible solution corresponds to a different possible path. To be consistent with the bird/friend technique, let us consider the end of the solution first.

***st*-Path:** To construct an st-path in G backwards, we start at node t, we choose the last edge $\langle u_r, t \rangle$ in the path, then the second last $\langle u_{r-1}, u_r \rangle$, then the third, and so on. We are done when we reach node s.

Knapsack: A solution to the knapsack instance would be constructed simply by considering the objects $\langle v_n, p_n \rangle, \langle v_{n-1}, p_{n-2} \rangle, \ldots$ backwards and deciding at each fork in the road whether to put the next object in or not.

Stocks: There are a lot of different question we might ask the bird. Keep in mind that the running time of the algorithm depends on the number of answers she might give. If we ask "Do should I sell the stock that I own today?", then there are two answers she can give. If we ask "When should I sell the stock that I own?", then there are T answers she can give. If we ask "What shock should I buy today?", then there are $|S|$ answers she can give. If we ask "What shock should I buy today and when should I sell it?", then there are $|S| \cdot T$ answers she can give. We are not going to worry at the moment about the details of question to ask. All we know is that our journey learns the what stocks to buy and when to buy and sell them. Because we are advised to journey backwards, we move backwards in time.

Remember Only What is Necessary to Continue: The path you followed to your current location/state determines what you have chosen so far about the solution. Your current location/state, however, determines what of that information you have remembered. While the number of possible paths is exponential, we want number of possible locations to be polynomial. Hence, we only remember what is necessary to continue our journey. You must remember some of the following.

1. Independent of how you got here, *where* are you currently?
2. When considering our current location, there is a complete symmetry between whether we are traveling forward or backwards along a path. But express the state in term of future path towards the beginning of the solution.
3. How much of the input instance I have you considered so far and how much remains?
4. How of much of the instance's resources you have used up and how much remain?
5. The key question is: What about the choices you have made so far will influence what choices you are allowed to make in the future?

Each possible answer to these questions determines your current location. Each will translate into a subinstance.

***st*-Path:** When constructing an st-path, a state in which we might be in is that we have constructed the path backwards from t to some node v and now we are standing at node v. However, according to (a), how we got here does not matter, only our current location v. Hence, we will have a *state* for each node v in the graph G. Note that the choices we are allowed to make in the future in our path to s depends only on this current state v.

Knapsack: At some point during this process of deciding which objects to put in the knapsack, we have considered the objects $n, n{-}1, \ldots, i{+}1$ and we still must consider the objects $i, i{-}1, \ldots, 1$. The input's resource is the volume V of the knapsack. Let $V - v$ denote the volume of knapsack that we have filled so far so that v denotes the volume yet to be filled. Which objects have been put in so far does not influence what can be put in in the future except for this i and this remaining volume v. Hence, we will have a *state* for each $\langle i, v\rangle$. Being in this state means that we still must choose which of the first i objects will go in and we have v volume left.

Stocks: Suppose that if we are to buy and sell on a giving day, then we sell the stock that we have in the morning and we buy the next stock in the afternoon. The key about this is that at noon we may or may not own stock and at midnight we definitely own some stock. Hence, during our journey, there two types of states that we may be in. Let being in state $\langle t\rangle$ mean that it is currently noon on day t and I do not own any stock. Let being in state $\langle t, s\rangle$ meaning that it is currently midnight on day t and I own any stock s. Note that this is the only information that determines your future journey. What stocks you have bought and sold in the past does not influence what and when you buy in the future. Neither does the amount of money/stocks that you currently have.

Construct G_I: Construct a graph G_I to represent this process.

Nodes ≈ States: G_I will have a node u for each state that the process might be in. Going backwards t and s denote the initial and final states. (If there is more than one such states, then have extra nodes t and s and put an edge from all possible initial and final states to these.)

***st*-Path:** The graph G_I is identical to the graph G. This should not surprise us too much if we recall what our initial goal is. We want the task of finding the optimal st-path in G_I to be identical to the task of finding the optimal st-path in G.

Knapsack: G_I will have a state $\langle i, v\rangle$ for each $i \in [0, n]$ and each $v \in [0, V]$. Note that the states $\langle 0, v\rangle$ and $\langle i, 0\rangle$ are all final states because nothing else could be put in the knapsack. Hence, we will have a zero weight edge from the source node s to each of these nodes.

Stocks: The algorithm designer must decide whether to include nodes of type $\langle t\rangle$ and/or nodes of type $\langle t, s\rangle$. It is your choice. The running time of the algorithm depends on the number of subinstances/nodes. Note that there are T of the first type of states and $T \cdot |S|$ of the second type

Edges ≈ Choices: There is an edge $\langle u, v\rangle$ between two states/nodes if making one decision about the solution transitions you from state u to state v. If there are more than one way to transition between these two states, then it is fine to have more than one edge between u and v. In our dynamic program, each node u corresponds to a subinstance and each outgoing edge correspond to the different answers the bird might give for this subinstance.

Edge Weights ≈ Cost Choice: The weight/cost $w_{\langle u,v\rangle}$ of the edge $\langle u, v\rangle$ will be the cost/benefit added to the solution because of this choice.

Knapsack: Being in state $\langle i, v\rangle$ means that we still must choose which of the first i objects will go into the knapsack and we have v volume left. The next decision will be whether or not to put the i^{th} object in the knapsack. If we do, then we follow the edge from this state $\langle i, v\rangle$ to state $\langle i{-}1, v{-}v_i\rangle$. The weight of this edge will be p_i because this is the benefit of transitioning between these states. (If the object does not fit because $v{-}v_i < 0$, then the weight will be $-\infty$.) If we do not put the i^{th} object in the knapsack, then we follow the edge from this state $\langle i, v\rangle$ to the state $\langle i{-}1, v\rangle$. The weight of this edge will be zero.

Stocks: What edges are in the graph depend on which nodes $\langle t \rangle$ and/or $\langle t, s \rangle$ you include. Each step of your journey does not increase the amount you have by an additive amount but by a multiplicative amount. Let the weight of the edge be this multiplicative factor.

***st*-Path ≈ Solution:** A st-path through the graph G_I then corresponds to a sequence of choices that are made along the process of constructing a solution for the instance I. Hence, there will be a one-to-one correspondence between the st-paths in G_I and the solutions for I. More over, the edge weights are designed so that the cost of this path is the same as the cost/value of its corresponding solution.

Knapsack: A path from $t \approx \langle n, V \rangle$ to $\langle 0, v \rangle$ or $\langle i, 0 \rangle$ to s in this graph G_I corresponds to a solution to the knapsack problem, i.e. it specifies which objects to take and which not to take. The cost of the path is the sum of the edges, which will amount to the sum $\sum_{i \in S} p_i$ of the prices p_i of the objects put in the knapsack, which in turn is the value of the constructed solution.

Stocks: A path backwards in time from state $t \approx \langle T \rangle$ to $s \approx \langle 0 \rangle$ in this graph G_I corresponds to a solution to the stock problem, i.e. it specifies which stocks to buy and when. Define the cost of the path to be not the sum but the product of the edges. Then this cost will amount to the total multiplicative factor that the your money increases from midnight on day zero to midnight on day T. This in turn is the value of the constructed solution.

Shortest *st*-Path ≈ Optimal Solution: It follows that an optimal solution for the instance I corresponds to a shortest st-path through the graph G_I.

Knapsack: Actually, an optimal path is one with maximal value $\sum_{i \in S} p_i$. However, in a leveled graph, it is just as easy to find a longest st-path.

Stocks: It is just as easy to find an st-path whose product of edge weights is maximized. (Alternatively take the logarithm of all the weights and then products translate into sums.)

Leveled Graph: The graph G_I produced is *leveled* because the process of constructing the solution orders the states into levels so that all edges point forwards.

Dynamic Programming Algorithm: The dynamic programming algorithm that finds the shortest path in G_I can be used to find the optimal solution for I. Or even better, you can construct a dynamic programming algorithm that directly solves your new problem.

States ≈ Nodes ≈ Subinstances: Given the instance I, the states in the process to construct the solution for I corresponds to nodes in G_I, which in turn correspond to subinstances that the dynamic programming needs to solve. The subinstance corresponding to a state is to find the optimal way continuing from here.

***st*-Path:** The state v indicates that we are standing at node v. The subinstance corresponding to this state is to find the shortest path from v to s – or forward from s to v.

Knapsack: The state $\langle i, v \rangle$ indicates that we still must choose which of the first i objects will go in and we have v volume left. The subinstance corresponding to this state is to find an optimal way of doing this.

The dynamic programming algorithm sets up a table in which to store the cost of an optimal solution for each of these subinstances (and the bird's advice). The algorithm will loop through these subinstances v starting with the sink node t and ending with the source node s.

Bird-Friend Algorithm: When considering the subinstance corresponding to the node v, the algorithm will ask the little bird for the last choice $\langle u_k, v \rangle$ in an optimal solution for the subinstance v. The set of answers to the birds questions correspond to the edges into v in the graph G_I. Not having such a bird, the algorithm will try each possibility. When trying the answer $\langle u_k, v \rangle$, the algorithm asks the friend for the optimal solution for the subinstance u_k. An optimal solution $optSol_k$ for v from amongst those solutions consistent with the bird's answer k is easily obtained from an optimal solution for u_k and the bird's advice k. The cost of this solution for v is the cost of this solution for u_k plus the weight $w_{\langle u_k, v \rangle}$ of the bird's choice. An optimal solution $optSol$ for v is obtained to by taking the best of $optSol_k$ optimized over all of the bird's answers k.

***st*-Path:** When considering the subinstance v, we are looking for an optimal path from s to node v. We ask the bird for the last edge in this path. If she answers $\langle u_k, v\rangle$, then we ask the friend to solve the subinstance u_k, namely for an optimal path from s to u_k. The shortest path from s to v from amongst those traveling through node u_k is the shortest path from s to u_k plus the last edge $\langle u_k, v\rangle$. The shortest over all path from s to v is the best of these optimized over v's incoming neighbors u_k.

Knapsack: When considering the subinstance $\langle i, v\rangle$, we are looking for an optimal subset of the first i objects to go into a knapsack of volume v. We ask the bird whether or not to put the last object in. If yes, we ask the friend $\langle i-1, v-v_i\rangle$ and add p_i to his solution. If no we ask the friend $\langle i-1, v\rangle$ and give the same solution.

Stocks: Your homework is to develop three different dynamic algorithms for this problem. The differ in $Time = \#$ of subinstances $\times$ # of bird answers.

The Table: This cost of this optimal solution for v and the bird's advice k are stored in the table index at v. When the friend is asked about u_k, his solution had already been stored in this table. The original instance I corresponds to the node t. Hence, the solution (or at least its cost) can be read from the table.

This completes the dynamic programming algorithm. This characterization of Dynamic Programming is, as far as I can tell, general enough to capture most every Dynamic Programming Algorithm that I can think of.