More Dynamic Programming

Today, we’ll discuss three applications of dynamic programming: to origami, to trees, and to tree-like graphs. The most important part of this lesson, and the one you should pay the most attention to understanding all the details of, is dynamic programming on trees.

1 2D Map Folding

Note: There’s a lot of background for the 2D map folding problem, but try to speed through it because it’ll take a while before you get to material directly relevant to today’s topic of dynamic programming.

The map folding problem is: You’re given a piece of paper with an \( n \times n \) grid of squares drawn on it. The goal is to fold along the lines until you’re left with just a single (very thick) square.

We’re going to add two additional constraints to this:

1. Each edge is going to have an M (for mountain) or V (for valley) drawn on it. This will indicate the type of fold you must make along that edge. *(If people are unfamiliar with this terminology, demonstrate mountain and valley folds on a piece of paper.)*

2. You can only make simple folds, meaning that you’re limited to folds where the crease goes all the way across the piece of paper. In other words, you can only make a fold if there’s a uniform line (row/column) of edges all having the same M/V designation.

A \( 4 \times 4 \) map with M/V assignments. There is one uniform line and therefore one simple fold, indicated by the dashed line.

Now the problem is: Fold the grid into a single square using the fewest possible simple folds, while respecting the M/V assignments. (Assume that it’s always possible, and we’re simply trying
Recitation 7: More Dynamic Programming

to minimize the number of folds. The problem of deciding whether a given 2D crease pattern is flat-foldable is still open.\(^1\)

We start by making a few observations that will help simplify the problem:

- **Folding causes edges to land on top of other edges. What must be true in order to make it possible to fold the resulting paper while still respecting all M/V assignments?**
  
  Assignments on opposite sides of the crease must be exact opposites, that is, every M needs to land on top of a V and vice versa.

- **What type of M/V assignments are possible at a vertex with four incident edges?**
  
  Let’s read off the edge assignments in clockwise order. WLOG, the possibilities are MMMM, MMMV, MMVV, MVMV, and all of the above with M and V swapped.

  MMMM is impossible because no fold causes Ms to land on Vs. MMVV is impossible because there’s no way to make a simple fold. MVMV is impossible because no fold causes Ms to land on Vs (and vice versa). Thus, the only possibility is **MMMV** (and its opposite, **VVVM**).

  ![diagram](diagram.png)

  There are two possibilities for how a vertex can look. (Of course, these can be rotated too.)

- **At each step of the process, can we always find a uniform line? If there are multiple, what can we say about them? (Are they horizontal or vertical? M or V?)**
  
  We’re guaranteed that the starting grid is foldable. Any subset of a foldable grid is also foldable (since making it larger only makes it harder to fold). After each step, we’re left with a subset of the original grid, so after each step, we’re left with a foldable grid. A foldable grid always has a uniform line, so yes, we can always find a uniform line.

  Due to the constraint on vertex types, if there are multiple uniform lines, they must be all horizontal or all vertical; uniform lines cannot cross. It is possible to have both M and V uniform lines, as long as they are parallel.

- **Assume that all uniform lines are vertical. What happens after folding all of them?**
  
  The resulting grid is still foldable (by the above) but doesn’t have any vertical uniform lines left, so therefore, it must have at least one uniform horizontal line.

- **Assume that all uniform lines are vertical. What order can we fold them in? Does order matter?**

\(^1\)http://courses.csail.mit.edu/6.849/fall12/lectures/L02.pdf
You need to fold all the vertical uniform lines before any horizontal ones appear. Not every folding order is possible, but among all the possible/valid folding orders, all of them have the exact same end result. However, some folding orders may require fewer folds than others (it may be possible for a fold to cause two uniform lines to land on top of one another), so we should consider all orders to see which one is the best.

For example, consider the following above, with three horizontal uniform lines. Depending on the folding order, it will take either 2 or 3 folds to fold along all three lines, though the end result of either order is the same. *(If confused, try this with an actual piece of paper!)*

```
  M  M  
M V M M
  V  V  V
V M M V
  V  V  
```

*Our original map example, but one fold further in than last time. At this step, you have a choice about the order in which to proceed.*

The above observations yield the following algorithm for 2D map folding: Find all (horizontal/vertical) uniform lines and consider the 1D problem of folding just along these lines. Determine how to fold these lines with the fewest total folds. Now find all (vertical/horizontal) uniform lines in the resulting 2D pattern and repeat.

```
  M  M  
M M M M
  V  V  V
V M M V
  V  V  
```

*The 1D problem. Each M or V represents a uniform line (in the 2D map); creases with neither an M nor a V are non-uniform lines (and will be dealt with in a future 1D problem).*

Can we solve the 1D problem with dynamic programming?

- **Subproblems:** After each fold, the resulting M/V pattern is a substring of the original pattern, so let’s make these substrings our subproblems.
- **Guess:** We can guess where to make the first fold. (However, not all guesses are valid! We need to check if opposite sides of the fold match each other.)
- **Recurrence:** 0 if this substring has no creases, otherwise 1 + the minimum over all guesses.

Running time: There are $O(n^2)$ subproblems, $O(n)$ guesses, and it takes $O(n)$ time to check that a guess is valid. This takes $O(n^4)$ total for a single instance of the 1D dynamic program.
Getting back to the original problem: We need to spend $O(n^2)$ time finding uniform lines, and then we run an $O(n^4)$ dynamic program. We may need to repeat this process up to $n$ times, so the overall running time is $O(n^5)$.

It’s possible both to tighten the analysis of this algorithm, and to improve the algorithm itself. For instance, there exists an $O(n^2)$ dynamic program.²

## Maximum-Weight Independent Set

2.1 On General Graphs

On general graphs, the maximum-weight independent set problem is NP-hard. As such, we don’t know a polynomial time algorithm to find maximum weight independent sets! One thing we can do is see if the problem becomes any easier if we consider only certain classes/types of graphs.

2.2 On Trees

Pick an arbitrary vertex $r$ to be the root of the tree. What happens if you remove $r$ from the graph? (The tree is split into multiple smaller trees.) Note that, if $G$ weren’t a tree, this wouldn’t work.

If we compute the maximum weight independent set on each of these subtrees, does that help us solve the problem on $G$?

Let’s try to write a dynamic program:

- **Subproblems**: For each $v \in V$, consider the graph $G_v$ which is the subtree of $G$ rooted at $v$. Let $\text{MWIS}(v)$ be the maximum weight of any independent set of $G_v$.

- **Guess**: Is $v$ in the maximum-weight independent set?

- **Recurrence**: Let $v_1, \ldots, v_k$ be the children of $v$. If we guess NO, then any union of independent sets of $G_{v_1}, \ldots, G_{v_k}$ gives an independent set of $G_v$ of weight $\sum_k \text{MWIS}(v_k)$.

  However, if we guess YES, then we may not be able to union the independent sets of each

²http://courses.csail.mit.edu/6.849/fall12/lectures/L02.pdf
child subtree—we may accidentally include both \( v \) and one of its children, which is not independent.

How can we fix this? What we really need to know is the maximum-weight independent set for a subtree where we definitely don’t include the root in any independent sets. For each subtree, let’s keep track of the following values:

- \( \text{MWIS}_\text{in}(v) \) is the maximum weight among independent sets of \( G_v \) that include \( v \) in the set.
- \( \text{MWIS}_\text{out}(v) \) is the maximum weight among independent sets of \( G_v \) that do not include \( v \) in the set.
- We can now compute \( \text{MWIS}(v) = \max \left( \text{MWIS}_\text{in}(v), \text{MWIS}_\text{out}(v) \right) \).

With this in place, we can fix the recurrence for our dynamic program:

- For non-leaves:
  - \( \text{MWIS}_\text{in}(v) = w(v) + \sum_k \text{MWIS}_\text{out}(v_k) \), because \( v \)'s children definitely can’t be included in the set
  - \( \text{MWIS}_\text{out}(v) = \sum_k \text{MWIS}(v_k) \), because we don’t care if \( v \)'s children are included in the set
- For leaves:
  - \( \text{MWIS}_\text{in}(v) = w(v) \)
  - \( \text{MWIS}_\text{out}(v) = 0 \)
- \( \text{MWIS}(v) = \max \left( \text{MWIS}_\text{in}(v), \text{MWIS}_\text{out}(v) \right) \)

We can now compute subproblems from the leaves up, and each subproblem will only depend on the values of previously solved subproblems. Suppose \( G \) has \( n \) vertices and \( m \) edges. There are \( O(n) \) subproblems, 2 guesses for each subproblem, and \( O(k) \) time per guess. We can bound \( k = O(n) \) for an overall \( O(n^2) \) running time.

However, it’s possible to make our analysis even tighter:

\[
\sum_{v \in V} 2 \times \# \text{children}(v) = 2m = 2n - 2.
\]

The first part comes directly from the above analysis. Each child uniquely corresponds to an edge in the graph, so we can rewrite the sum in terms of \( m \). Finally, for a tree, we know that \( m = n - 1 \). Therefore, the running time is \( O(n) \).

### 2.3 On Tree-Like Graphs

Consider the following graph \( H \):
Let’s call this particular graph $H$. Note that $H$ is not a tree.

This is not a tree, so we can’t use the algorithm we just developed. If you tried to, you would find subproblems that are dependent on one another, so you can’t solve either one!

However, $H$ has enough tree-like properties that the maximum-weight independent set problem is not NP-hard on $H$. In fact, we can adapt our dynamic program to work on $H$.

To start, we’ll create a tree $H'$ based on $H$. For each triangle of three vertices in $H$, add a vertex to $H'$. Two vertices of $H'$ are connected by an edge if their corresponding triangles in $H$ share an edge. Here’s the $H'$ we construct:

\[
\begin{array}{c}
\text{H is on the left, and the H' produced by the described process is on the right. H' is a tree.}
\end{array}
\]

Now that we have a tree, we can adapt our dynamic program to work!

Note: Trying to make generalized statements would require an entire arsenal of new notation, so from here on out, the notes will often stick to specific examples rather than the general case.

If $v$ is a vertex of $H'$, how many children can $v$ have? Neighbors of $v$ correspond to triangles in $H$ adjacent to $v$’s triangle, so there can be at most three neighbors. If you pick the root of $H'$ to be a vertex with fewer than three neighbors (such a vertex must always exist), then after rooting $H'$, each vertex has at most two children.

Our subproblems are now subtrees rooted at vertices of $H'$. Even though the subproblems are subtrees of $H'$, we still want independent sets of $H$, not $H'$, so we need to examine the corresponding
vertices in $H$.

For an arbitrary subproblem (given by a vertex $v$ and a subtree $H'_v$), let the vertices of $H$ corresponding to $v$ be $A, B, C$. Our guess is that either none of $A, B, C$ are in the maximum-weight independent set, or exactly one of them is. (We can’t have two of them, because that wouldn’t be independent! Remember, every pair is connected by an edge.)

So, we need to compute five values: $\text{MWIS}_A(v)$ is the maximum weight among independent sets that include $A$ in the set, and likewise for $\text{MWIS}_B(v)$ and $\text{MWIS}_C(v)$. For the “none” guess, we have $\text{MWIS}_\emptyset(v)$, which is among all independent sets that include none of $A, B, C$. Finally, as in the tree problem, $\text{MWIS}(v)$ is the max taken over the previous four values.

We need to be very careful about what $\text{MWIS}$ represents because we don’t want to ever double-count a weight. Let’s match up vertices of $H$ with vertices of $H'_v$. If $H$ has $n$ vertices, then $H$ has $n - 2$ triangles, so some vertex of $H'$ will need to be matched to three vertices of $H$. Let’s create the following matching: the root of $H'_v$ is matched to all three vertices in its corresponding triangle, and then we depth-first search $H'$, at each step matching the current vertex of $H'$ to the unique unmatched vertex in its corresponding triangle.

Now, what $\text{MWIS}(v)$ actually represents is: Take the subtree $H'(v)$ of $H'$ rooted at $v$, and consider the subgraph of $H$ induced by the vertices corresponding to vertices in $H'(v)$. Find the maximum weight independent set of this subgraph and sum up the weights of the vertices that are both in the independent set and matched to vertices of $H'(v)$.

Let’s work on an example. Take $H$ and $H'$ as drawn above, root $H'$ at $v_1$, and let’s consider computing the values at the root based on the values at its children, $v_2$ and $v_3$.

Let’s zoom into $H$ and $H'$, considering just the vertex $v_1$ of $H'$ and its two children, $v_2$ and $v_3$. We also label the vertices of $H$ with letters.

- $\text{MWIS}_A(v_1) = w(A) + \text{MWIS}_A(v_2) + \max\left(\text{MWIS}_\emptyset(v_3), \text{MWIS}_E(v_3)\right)$
- $\text{MWIS}_B(v_1) = w(B) + \max\left(\text{MWIS}_\emptyset(v_2), \text{MWIS}_D(v_2)\right) + \text{MWIS}_B(v_3)$
- $\text{MWIS}_C(v_1) = w(C) + \text{MWIS}_C(v_2) + \text{MWIS}_C(v_3)$
- $\text{MWIS}_\emptyset(v_1) = \max\left(\text{MWIS}_\emptyset(v_2), \text{MWIS}_D(v_2)\right) + \max\left(\text{MWIS}_\emptyset(v_3), \text{MWIS}_E(v_3)\right)$
- $\text{MWIS}(v_1) = \max\left(\text{MWIS}_A(v_1), \text{MWIS}_B(v_1), \text{MWIS}_C(v_1), \text{MWIS}_\emptyset(v_1)\right)$
If you understand the intuition behind each of these formulas, then it should hopefully be clear how to generate the formulas for other vertices.

The one big difference when writing the formulas for other vertices is that we only add $w(x)$ if $x$ is matched to the current vertex; the root is matched to all three of $A, B, C$ so we added the weight in all three cases, but for non-root vertices, we’ll only add the weight in one of those three cases. (Again, this is to ensure that each vertex in the maximum weight independent set has its weight counted exactly once.)

We can now proceed from the leaves of $H'$ up to the root, and each subproblem only depends on already-computed subproblems. Suppose $H$ has $n$ vertices. Then $H$ has $n - 2$ triangles, so $H'$ has $n - 2$ vertices and there are $n - 2$ subproblems. Each subproblem has 5 guesses and takes $O(1)$ per guess. Therefore, the overall running time is $O(n)$.

(Note: We apologize if the section on tree-like graphs has errors; this topic has a lot of subtleties, and it’s extremely difficult to get all of them right! We definitely don’t expect you to get all the subtleties right either, as long as you understand the overall idea.)

### 2.4 Treewidth

We only showed that this technique works on a single graph $H$, but it turns out that it generalizes to most graphs made out of triangles. (Not all, though: only the ones that are “tree-like.” It’s possible that the process described above will not yield a tree.)

However, we can generalize even more, to any graph! Given a graph, we can compute a quantity $k$ known as the treewidth. To find the treewidth, start with $k = 2$ and try to group vertices into overlapping groups of size $k + 1$ such that: If $(u, v)$ is an edge then there’s a group containing both $u$ and $v$, and all groups containing both are connected by edges. Additionally, for any vertex $v$, the groups containing $v$ need to form a tree. If this doesn’t work, keep on incrementing $k$ until it does. The minimum $k$ that works is the treewidth.

$H$ in our example above had treewidth $k = 2$ because the triangular structure of $H$ meant that groups of three vertices worked. Given a generic graph, we can show that $k \leq n - 1$ because putting all $n$ vertices into a single group always satisfies the desired properties.

The graph formed by grouping vertices (eg, $H'$) is called the *tree decomposition*. By running our dynamic program on the tree decomposition, we can solve the maximum-weight independent set problem in $O(2^k n)$ time on any graph. This is linear for graphs of constant treewidth, but in general, $k = O(n)$ and this algorithm is exponential. That makes sense because, as you may remember, the maximum-weight independent set problem is NP-hard on general graphs.