Introduction to Dynamic Programming[*]

Michael P. Kim[†]

CS 4820 — Spring 2024

1 Case Study: Max Weight Independent Set on a Path

Given a graph $G = (V, E)$, a subset of the vertices $S \subseteq V$ is called an *independent set* if no two $u, v \in S$ form an edge in the graph; i.e., $(u, v) \notin E$. The *maximum weight independent set* problem asks the following question:

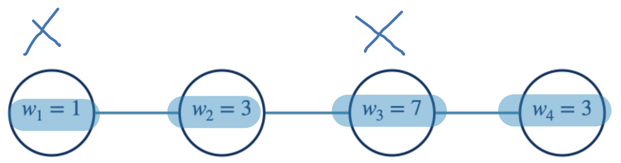
> **Given:** a graph $G = (V, E)$ with non-negative vertex weights $W = \{w_u\}_{u \in V}$
> **Find:** the independent set $S \subseteq V$ that maximizes the total weight $W_S = \sum_{u \in S} w_u$

NP

As we'll see later in the course, the maximum independent set problem on general graphs (even without weights) cannot be solved efficiently.[1] For today's lecture, we focus on the setting where the input is a path graph—namely, a connected graph with degree at most 2.

> **Given:** a *path graph* $P = (V, E)$ with non-negative vertex weights $W = \{w_u\}_{u \in V}$
> **Find:** the independent set $S \subseteq V$ that maximizes the total weight $W_S = \sum_{u \in S} w_u$

For instance, consider the path graph below. For notational convenience (here and throughout the lecture) assume that the vertices are labeled with $\{v_1, \ldots, v_n\}$, increasing from left to right.



MWIS

In this instance, the maximum independent set is $S = \{v_1, v_3\}$ with a weight of $w_1 + w_3 = 8$. Another candidate independent set is $S' = \{v_2, v_4\}$ and has weight $w_2 + w_4 = 6$. Of course, in this instance, we can convince ourselves that S is the max independent set by trying all subsets of the vertices. But, for larger instances of path graphs, can we do better?

[*]Notes and lecture heavily influenced by Tim Roughgarden's offering of CS 161 at Stanford University.
[†]Cornell University, `mpk@cs.cornell.edu`
[1]Unless P = NP.

2 Greedy Approach

So far in 4820, the key algorithm design paradigm we've seen is greedy. Can we design a greedy algorithm for the problem at hand? Let's consider a few candidate "priority" functions.

Perhaps the first greedy algorithm would process vertices in order of weight. That is, sort the vertices by w_u, then for each vertex $u \in V$, add u to S if u does not have a neighbor in S already. While natural, it is not hard to construct an instance where this greedy heuristic fails.[2] 1→4→7→5

A second greedy algorithm might process the vertices simply in order specified by the path. Starting from the left (or perhaps the right), add in vertices as long as they don't contradict the independence condition. This strategy can be seen to be quite limited, always returning an independent set with alternating vertices. Again, you should think about how to construct an instance on which this strategy fails.[3]

1→4→7→5→2 1→7→4→10

While these greedy heuristics can be made to work on the unweighted path graphs, the weights seem to introduce a challenge that thwarts greedy approaches. In particular, it is not clear that we can make our decisions myopically and irrevocably. It seems that we need a more global view of the solution space.

3 Dynamic Programming

In the remainder of the lecture, we'll explore a new paradigm for algorithm design, called *dynamic programming*, that implicitly explores all possible solutions. Key to this paradigm is that we search the complete solution space *implicitly*: enumerating all possible solutions, of course, would take exponential time. We will show that, for problems like the weighted independent set problem on path graphs, the value of a given solution can be expressed in terms of the value of *partial solutions*. This property—that full solutions can be decomposed into partial solutions—is the quintessential aspect of a problem that often suggests a dynamic programming algorithm.[4]

3.1 Optimal Solutions on the Path Graph

To begin, let's consider a hypothetical optimal solution to the problem—an independent set $S \subseteq V$ of the path's vertices with maximum weight w_S—and see if we can understand properties that the solution must satisfy. Here is one such property.

Lemma 3.1. *Consider a maximum weight independent set $S \subseteq V$. The final vertex v_n in the path graph is either in the independent set, or not. That is, $v_n \in S$ or $v_n \notin S$.*

This lemma needs no proof: we are simply observing the trivial fact that every vertex is either in or not in the subset S and applying it to the final vertex. Despite the triviality of this lemma, it

[2]Consider a path with three vertices, where the middle vertex has maximum weight, but the two ends sum to more than the middle.

[3]Construct a four vertex path, where the maximum independent set consists of the outer two vertices.

[4]You may wonder what aspects of the "programs" we write here are "dynamic." In truth, the terminology "dynamic programming" has nothing to do with our algorithm design approach, and everything to do with the politics of getting research funded. `https://en.wikipedia.org/wiki/Dynamic_programming#History_of_the_name`

will get us quite far in designing an algorithm for the problem.[5] The point of stating this simple lemma is that it gets us started thinking about a case analysis.

Consider the two cases. First, suppose that the final vertex is in the optimal independent set $v_n \in S$. What else can we conclude about S? By the definition of an independent set, we know that the second to last vertex must not be in the solution $v_{n-1} \notin S$, because it is neighbors with v_n. On the other hand, suppose that $v_n \notin S$. In this case, we can conclude that the optimal value W_S in the path graph P is the same as the value in the path graph with v_n removed.[6]

By Lemma 3.1, these cases are exhaustive, so we can hope to *characterize* the solutions to the entire problem in terms of two different subproblems. To better reason about subproblems, let P_i for $i \in [n]$ denote the path graph including the first i vertices (so, $P = P_n$). Further let $\textbf{WIS}(P_i)$ denote the weight of the maximum independent set of the ith path graph. With the case analysis and notation in place, we can show the following lemma.

Lemma 3.2. *Consider a maximum weight independent set $S \subseteq V$ on the path graph $P = (V, E)$ with $n \geq 2$ vertices. The maximum weight W_S is equal to $w_n + \textbf{WIS}(P_{n-2})$ or $\textbf{WIS}(P_{n-1})$. Specifically, the following equality holds.*

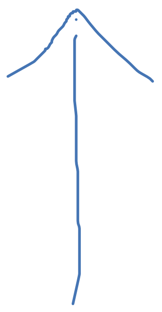
$$W_S = \max\{w_n + \textbf{WIS}(P_{n-2}),\ \textbf{WIS}(P_{n-1})\}$$

Proof Idea. Assume that $n \geq 2$. Suppose $v_n \in S$. Then, we know that $v_{n-1} \notin S$, as there is an edge (v_{n-1}, v_n). With v_{n-1} not in S, we know that the remaining vertices $V \setminus \{v_n, v_{n-1}\}$ remain feasible. Removing these two end vertices and considering an optimal solution to P_{n-2}, we conclude that $W_S = w_n + \textbf{WIS}(P_{n-2})$ (by the feasibility of all the nodes in P_{n-2} and the assumption that W_S was the optimal value for P).

Suppose $v_n \notin S$. In this case, then the optimal solution is simply the optimal solution on P_{n-1}, so $W_S = \textbf{WIS}(P_{n-1})$.

The maximum weight independent set on P is the maximum over these two exhaustive cases. $\square$

Immediately, this recursive formulation of the optimal value suggests an algorithm.



[5] Let this be a lesson that, often in algorithm design, it's wise to start by writing down what you know about the solution to your problem, no matter how "obvious" it is to you.

[6] In thinking about this problem, we can even conclude that v_{n-1} must be in the maximum weight independent set: otherwise, v_n should have been included in S. Note that this is *not typical* of dynamic programming problems. As we'll see, we do not need this property to design a correct algorithm.

```
ComputeWIS($P_k$):
    • if $k = 0$: return 0
    • if $k = 1$: return $w_1$
    • Let $W_{k-1} \leftarrow$ ComputeWIS($P_{k-1}$)
    • Let $W_{k-2} \leftarrow$ ComputeWIS($P_{k-2}$)
    • if $w_k + W_{k-2} > W_{k-1}$:
        – return $w_k + W_{k-2}$
    • return $W_{k-1}$
```

Correctness. Correctness of the algorithm requires proof, but follows by induction, following the proof of Lemma 3.2.

Running Time. To analyze the running time of ComputeWIS, we need to be more careful. As written, we have a recursive algorithm that makes 2 recursive calls at each iteration. Further, the size of the subproblems decreases very slowly. As a recurrence, the running time can be expressed as follows.

$$T(n) \geq T(n-1) + T(n-2) \geq 2 \cdot T(n-2)$$

We can lower bound this recurrence as $T(n) \geq 2^{n/2}$. *Exponential Time!* A careful accounting of the algorithm shows that, in fact, ComputeWIS computes the value for all feasible solutions *explicitly*, so it is no wonder that it would scale exponentially.[7]

3.2 Memoization and the Dynamic Programming Table

A further accounting of the recursive calls of ComputeWIS shows that many of the calls are identical. In the current formulation, we pay for ComputeWIS(P_k) for each invocation, even though the value is the same every time. If we record these values as we go, we can avoid redoing identical work. This trick is generally referred to as *memoization*.

In memoization, we keep a global array—called the *dynamic programming table*—to maintain a record of our previous computations. We memoize our algorithm to give MemoizedComputeWIS. The key difference is that we maintain a 1D-dynamic programming table W, initialized to all -1 values (as a bookkeeping measure). If we ever see an entry $W[k]$ with value -1, we recurse to compute the optimal independent set value, then replace $W[k]$ with the computed value. Then, in future recursive calls, we know that if we encounter a value $W[k] \neq -1$, we need not recurse.

[7]Can you come up with an exact characterization of the number of independent sets on a path? Consider where else you've seen a recurrence of the form $F_n = F_{n-1} + F_{n-2}$. fibonacci

```
MemoizedComputeWIS($P_k$):
  • if $k = 0$: return $0$
  • if $k = 1$: return $w_1$
  • if $W[k-1] = -1$:
      – $W[k-1] \leftarrow$ MemoizedComputeWIS($P_{k-1}$)
  • if $W[k-2] = -1$:
      – $W[k-2] \leftarrow$ MemoizedComputeWIS($P_{k-2}$)
  • if $w_k + W[k-2] > W[k-1]$:
      – return $w_k + W[k-2]$
  • return $W[k-1]$
$W \leftarrow [-1, \ldots, -1]$
return MemoizedComputeWIS($P$)
```

Correctness. You should convince yourself that nothing about the semantics of the algorithm have changed, so correctness stays the same.

Running Time. The memoized version of the algorithm is a tiny change from the original version, but its running time differs immensely.

First, let's note how many recursive calls can ever happen. MemoizedComputeWIS(P_k) is only called when $W[k] = -1$, and then $W[k]$ gets reassigned to some nonnegative value. By the fact that we only update $W[k]$ when it starts as -1, we conclude that for each $k \in [n]$, MemoizedComputeWIS(P_k) is called at most once. Further, how much work do we perform per update of $W[k]$? In each such call, $O(1)$ additional work is performed to perform look-ups in W and perform comparisons. Thus, in total—despite the recursive structure of the algorithm—we get away with performing $O(n)$ work to return the maximum weight independent set in P.

3.3 Iterative Dynamic Programming

We've shown a recursive formulation for solving our problem using dynamic programming. As you can see in KT §6.2, for any dynamic program, there is an equivalent formulation as a bottom-up iterative algorithm that operates on the dynamic programming table directly.

```
IterativeComputeWIS($P$):
  • $W \leftarrow [-1, \ldots, -1]$
  • $W[0] \leftarrow 0$; $W[1] \leftarrow w_1$
  • for $k = 2, \ldots, n$:
      – $W[k] \leftarrow \max\{w_k + W[k-2], W[k-1]\}$
  • return $W[n]$
```

Again, you should convince yourself that these programs are, in fact, equivalent. At times, recursion will be the easiest way to formulate the ideas behind the problem solving, but almost universally, the iterative solution will be easier (and faster) to implement.

↳ asymptotically & time-wise