CS 6363.003.21S Lecture 10—March 2, 2021

Main topics are #dynamic_programming and #greedy_algorithms with #example/maximum_independent_set_in_trees and #example/class_scheduling.

Dynamic Programming on Trees

- Last time, we finished by looking at the maximum independent set problem on trees.
- An *independent set* of a graph is a subset of vertices with no edges between them. The *maximum independent set* problem asks for the largest independent set in a graph.
- Suppose we're given a rooted tree T with n vertices. Let's compute the size of the maximum independent set in T.
- Again, we'll start with a backtracking approach. Like before, we can decide to do something with the root (which in this case is given to us) and then recurse on the subtrees.
- We'll try to decide if the root belongs to the maximum independent set or not.
- Suppose we decide the root does not belong to the maximum independent set. We can treat each subtree of the root as its own version of the problem, because they do not share edges. We'll just ask for their maximum independent sets.
- And if we do decide to include the root, we can't include the children nodes, but we can include all the grandchildren. We can ask the Recursion Fairy to find maximum independent sets of the grandchildren's subtrees.
- So let MIS(v) denote the size of the maximum independent set in the subtree rooted at v. Let w (downarrow) v mean "w is a child of v". **[First recursive call should be on w]**

$$MIS(v) = \max\left\{\sum_{w\downarrow v} MIS(v),\ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\right\}$$

- We need to compute MIS(r) where r is the root of T.
  - Subproblems: Each recursive subproblem takes a node.
  - Memoization: The surprise here is that we don't make a new array. Instead, we'll use the tree itself by storing each MIS(v) in a new field v.MIS.
  - Dependencies: Each entry MIS(v) depends upon the children and grandchildren of v.
  - Evaluation order: And you've likely seen a way to process children and grandchildren before a node. We'll use a standard *post-order* traversal of the tree.
  - Space and time: We're storing one number per vertex, so we use **O(n)** space. Time is more subtle. The time taken to compute MIS(v) for each vertex varies depends on how many children and grandchildren it has. So let's turn the analysis around. The algorithm will spend time proportional to the total number of MIS lookups it performs. Each vertex counts as a child at most once and as a grandchild at most once.

Therefore, the algorithm will run in **O(n)** time total.

- And here's the algorithm. It's still recursive, but that's the easiest way to implement post-order tree traversal. **[Maybe do MIS(w) separately from using its value]**

```
MIS(v):
  withoutv ← 0
  for each child w of v
      withoutv ← withoutv + MIS(w)
  withv ← 1
  for each grandchild x of v
      withv ← withv + x.MIS
  v.MIS ← max{withv, withoutv}
  return v.MIS
```

- There's another way we could have solved this problem so that we don't have to worry about grandchildren.
- If we take the root into our independent set, we cannot include the roots of the children subtrees. So we could ask for the maximum independent subsets of the children subtrees that *don't* include their roots.
- Similarly, we could ask for maximum independent sets that *must* include roots.
- So let MISyes(v) denote the size of the maximum independent subset of the subtree rooted at v that *includes* v.
- And MISno(v) is defined the same except we exclude v.
- Now we just want to know max{MISyes(r), MISno(r)}, and we can use the following recurrence.

$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}$$

- Nearly all the details are the same as before, but we get a simpler dynamic programming algorithm.

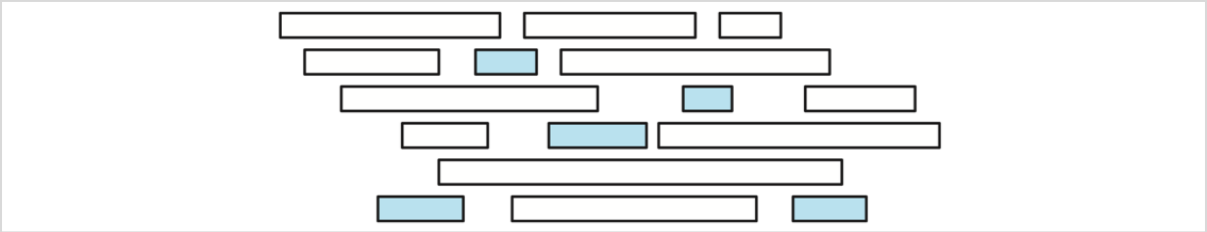
```
MIS(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + MIS(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```

Class Scheduling

- And that's all I want to say about dynamic programming for now. We'll come back to a few examples later in the semester as we talk about graph algorithms.
- But for now, I want to discuss another algorithm design paradigm that's based on

recursion, but has a much higher risk/reward tradeoff than backtracking and dynamic programming: greedy algorithms.

- As always, we'll look at an example before discussing the overall framework or how to argue your greedy algorithms actually work.
- Suppose you're picking classes for next semester, and your number one goal is to graduate as quickly as possible, so you want to take as many courses as possible. Don't worry, these classes you're considering don't require any actual work; you just need to ~~be present~~ have an opening in your schedule for the lecture. By some fluke all classes next semester are scheduled on Mondays only, but you're not allowed to take classes scheduled for conflicting times.
- Formally, let S[1 .. n] be the start time of all n courses offered and F[1 .. n] be their end times; so $0 \le S[i] < F[i]$ for all i.
- You want to find a *maximal conflict-free schedule* which is a maximum size subset X of {1, 2, …, n} such that for each i,j in X either S[i] > F[j] or S[j] > F[i].
- Another way to think about it is you have this set of overlapping intervals representing the time span for each course. Find the largest subset of intervals that don't overlap.

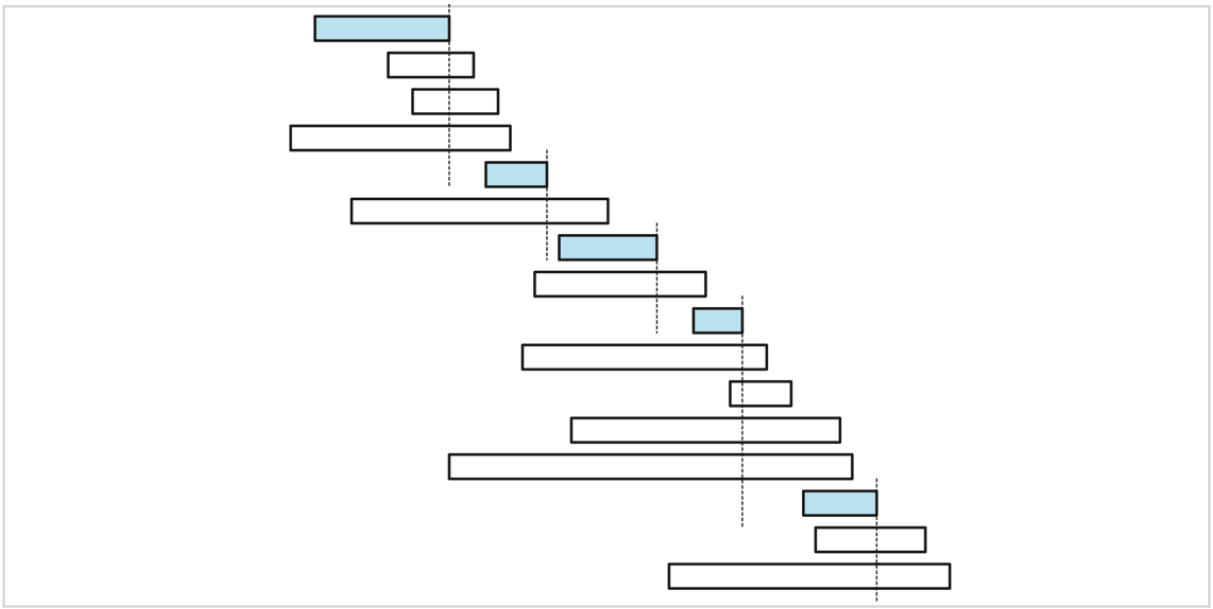


- If we were designing a backtracking or dynamic programming algorithm, we might loop over all the intervals, trying to guess one to include in our schedule. For each guess, we'd recursively build the best schedule for the subset of intervals that don't conflict.
- These subsets of classes do have some nice structure to them, so we don't need to consider *every* subset as the input to a recursive subproblem. But even still, we'd at best get an O(n^3) time dynamic programming algorithm using this strategy. There's another dynamic programming strategy that works better, but it still leads to an O(n^2) time algorithm. Can we do even better?
- For this specific problem, yes. It turns out we can *greedily* choose the class that finishes first and then recursively build a best schedule for the remaining classes.
- Let's prove this greedy strategy works, and then we'll look at an efficient implementation of the algorithm.
- Lemma: At least one maximal conflict-free schedule includes the class that finishes first.
  - Let f finish first, and consider any maximal conflict-free schedule X.
  - If X includes f, we're done.
  - Otherwise, let g be the first class to finish in X.
  - f finishes before g, so f does not conflict with any classes in X \ {g}.
  - So remove g and replace it with f. The new schedule is just as large and it agrees with

our first choice.

- And just to be complete (and emphasize how we're really doing recursion), let's argue that our overarching recursive strategy is correct.
  - We know we can start with the class f that finishes first.
  - Given that choice, we cannot take classes conflicting with f.
  - But we can use any subset of non-conflicting classes that don't conflict with f. In particular, we want a maximal set of such classes which the Recursion Fairy finds by induction.
- We can write our strategy as an iterative algorithm by scanning through the class list ordered by finishing time, and every time we see a new class that doesn't conflict with the last one we chose, taking it. This procedure returns the indices of the classes sorted by finishing time.

```
GREEDYSCHEDULE(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```



- This figure shows the sorted set of intervals and what we're skipping past.
- For running time, we have the time it takes to sort which is O(n log n). The rest of the algorithm is a simple O(n) time for loop, so the overall running time is O(n log n).

Greedy Algorithms and Exchange Arguments

- We just saw an example of a greedy algorithm.
- Not everybody describes it this way, but I like to think of it as "backtracking without the backtracking". As with standard backtracking algorithms, we need to make a single

decision that is as simple and self-contained as possible. Each possible choice leads to an appropriate recursive call that handles the remaining decisions.

- The big difference, though, is that there is one "obviously" best choice that we can commit to before doing any recursive calls or learning any consequences. So, we make that choice, perform its recursive call, and return the overall result without checking if any of the other choices would have been better.
- As shown earlier, you may not necessarily write describe your algorithm as a recursive procedure, but I still think about it recursively during the design phase.
- Coming up with plausible sounding greedy strategies is easy. Finding one that is correct and arguing for its correctness is much, much harder.
- Almost always, proving our first choice correct requires an exchange argument:
    1. Start with some optimal solution (not one necessarily found by ours or any other algorithm. Just the best choice over the set of solutions). If the optimal solution agrees with our first choice, then great! Otherwise…
    2. Do some kind of "exchange" in the optimal solution so it does use our first choice.
    3. Argue that the exchange didn't increase the cost of the solution, so our new solution must also be optimal.
- Sometimes, the new solution ends up being even better than the original one, implying that the hypothetical solution not using our first choice was actually a contradiction. We'll see an example of that when we do minimum spanning tree algorithms sometime next month.
- Of course, this strategy is still based on backtracking, so you should argue that your recursive call is useful as well.
- And like a lot of things, this is not the *only* way to think about greedy algorithms. Erickson suggests that exchange arguments should try to fix the "first" difference between greedy and optimal solutions, and he offers an alternative proof for the class scheduling problem to reflect this viewpoint.
- But I like to teach the backtracking way of thinking about it, because it keeps you all thinking recursively and it solidifies the goal with the exchange argument.
- All that said, you almost never want to use a greedy algorithm if your goal is a provably correct algorithm for a problem. It's extremely rare that the problem is structured so nicely that you can do an exchange argument, so you're better off just using dynamic programming. I've had some freedom this semester in what examples I choose for techniques like divide-and-conquer and dynamic programming. Correct greedy algorithms are so rare that I've already used one of the two examples that appear in pretty much every introductory lecture on the topic. Thursday, we'll go over the other introductory example. By the end of the semester, we'll have seen pretty much every example that most computer scientists ever learn.