Greedy Algorithms

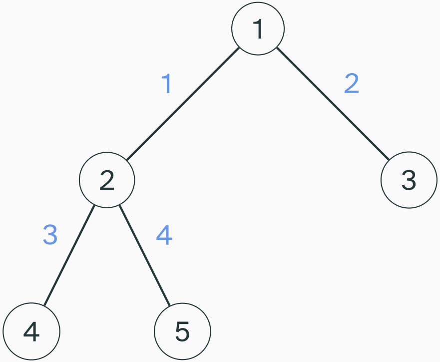
Sam McCauley

February 24, 2025

Welcome Back!

- No announcements today
- Any questions?

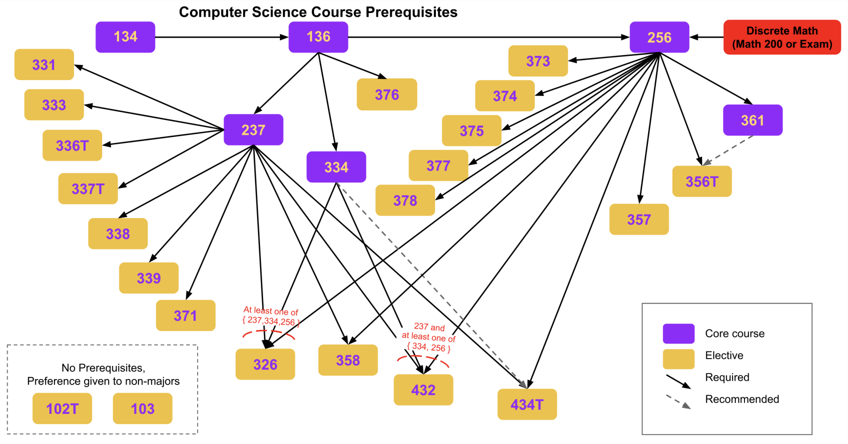
Quick Fact



- Any tree has n vertices and $n - 1$ edges.
- Any connected graph with $n - 1$ edges and n vertices is a tree.
- (Classic proof by induction to formalize.)

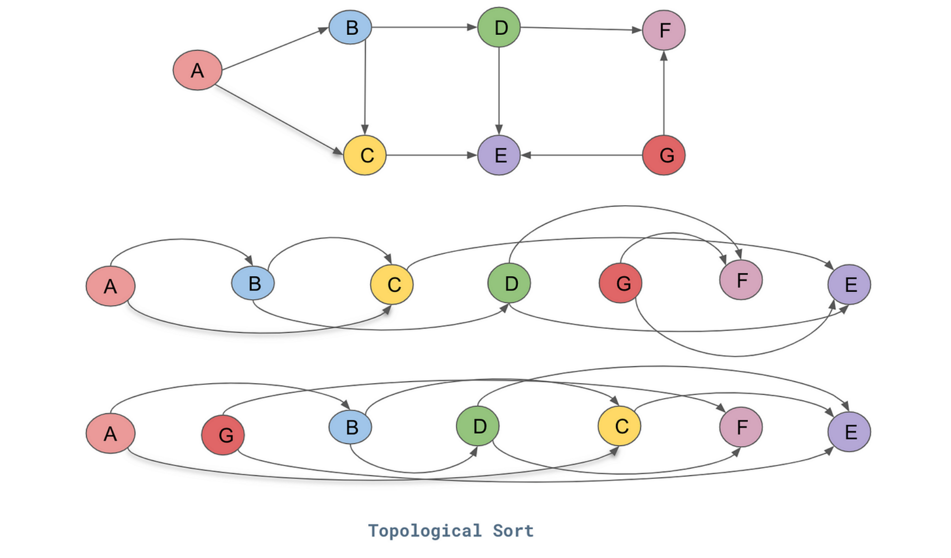
Topological Ordering

Topological Ordering



- Goal: Order the vertices of a graph so that for any edge (u, v), u comes before v in the final order
- *Example:* find a sequence of all courses satisfying prerequisites

Topological Ordering (a.k.a. Topological Sort)



Source: `https://medium.com/@konduruharish/topological-sort-in-typescript-and-c-6d5ecc4bad95`

DAGs and Toplogical Ordering

We want to show that:

Theorem

A graph G has a topological ordering if and only if G is acyclic.

To prove this we showed (last class):

Lemma

Every DAG has a vertex with indegree 0.

Let's review the algorithm we saw last class based on this lemma.

Topological Ordering: Simple Algorithm

```
while L has length less than n:
    find a vertex v with indegree 0
    if no such vertex exists:
        return that the graph has a cycle
    add v to the end of L
    remove v and its outgoing edges from G
```

- Running time?
- How can we store vertices with indegree 0?
  - Use a stack of vertices with indegree 0, and an array storing indegree of all vertices
  - Initialize array by examining edges one by one
- Time to remove vertex and edges with adjacency list?
- Overall: $O(n + m)$ time

DAGs and Toplogical Ordering

We're ready to prove our theorem (and show that the algorithm is correct).

Theorem

A graph G has a topological ordering, and the algorithm finds it, if and only if G is acyclic.

Proof.

If G is acyclic, then by the lemma our algorithm always finds a vertex of degree 0, so it returns a list L containing all n vertices. We are left to show that L is a topological ordering. Consider a vertex v in L. Since v has indegree 0 in G when it is added to L, for any edge (v', v) in the original graph, v' must have already been placed in L.

Now consider the case where G has a cycle C. Assume by contradiction that the algorithm does not return that G has a cycle. In this case, the algorithm must add all vertices to L. Let v be the first vertex in C added to L by the algorithm. But v must have an incoming edge in C, which is placed later in L. □

Finding Topological Ordering with DFS

```
DFS-Cycle(s):
    mark s as active
    for each neighbor v of s:
        if v is active:
            report that there is a cycle
        if v is not finished:
            DFS-Cycle(v)
    mark s as finished
    add s to the front of L
```

- Running time?
- $O(n + m)$
- Why does this work?
- What does it mean for a vertex to be active? Let's do an example on the board

Finding Topological Ordering with DFS

```
DFS-Cycle(s):
    mark s as active
    for each neighbor v of s:
        if v is active:
            report that there is a cycle
        if v is not finished:
            DFS-Cycle(v)
    mark s as finished
    add s to the front of L
```

Claim: Vertex v is active if and only if DFS−Cycle(v) was called, but has not yet finished.

Short proof: We mark v as active only when DFS−Cycle(v) is called; we mark v as finished when DFS−Cycle(v) finishes

Finding Topological Ordering with DFS

```
DFS-Cycle(s):
    mark s as active
    for each neighbor v of s:
        if v is active:
            report that there is a cycle
        else if v is not finished:
            DFS-Cycle(v)
    mark s as finished
    add s to the front of L
```

In pairs: Let’s say the algorithm returns that there *is* a cycle. Can you write a short proof for why it is correct?

Proof: Since v is active, $\mathrm{DFS}(v)$ has called but has not yet finished. Then there is a path from s to $\mathrm{DFS}(v)$ in the DFS tree; combining with the edge (v, s) gives a cycle.

Finding Topological Ordering with DFS

```
DFS-Cycle(s):
    mark s as active
    for each neighbor v of s:
    mark s as finished
        if v is active:
            report that there is a cycle
        else if v is not finished:
            DFS-Cycle(v)
    add s to the front of L
```

Other direction: Let's prove that if there is a cycle, the algorithm finds it.

Let v be the first vertex in the graph explored by DFS that is in a cycle; let C be that cycle and let v' be the vertex before v in C.

By our observation from last class, $\texttt{DFS-Cycle}(v)$ explores all unmarked vertices reachable from v before completing. So $\texttt{DFS-Cycle}(v')$ will be called while v is active; and the algorithm will return that there is a cycle.

Greedy Algorithms

Algorithmic Design Paradigms

We will look at the following algorithmic paradigms in this class.

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
- Network Flow

Algorithmic Design Paradigms

- Greedy Algorithms $\Leftarrow$ we are here!
- Divide and Conquer
- Dynamic Programming
- Network Flow

Making Change Optimally

- What are the fewest number of coins and bills to make $x?
- Anyone have an algorithm?
- Does this *always* work? Yes. But it’s not obvious!

Change Cannot Always be Made Greedily

The old British system had (among others) the following coins:

Coin:	penny	threepence	sixpence	shilling	florin	half-crown
Value:	1	3	6	12	24	30

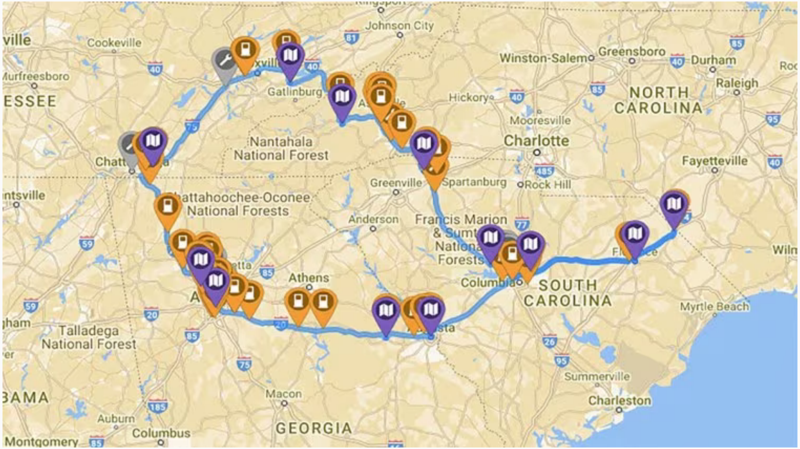
- Can you come up with an amount for which the greedy algorithm does not use the correct number of coins?

- One example: 48. The greedy algorithm gives three coins: 30 + 12 + 6. But we can do it with two florins (24 + 24)

Greedy Algorithms

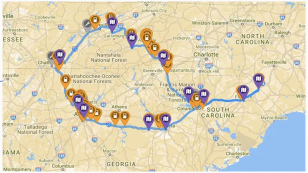
- Greedy algorithms make simple local decisions to obtain an optimal solution
- Are almost always fast!
- Question: can you show that your greedy algorithm is *always correct* for the given problem?

Filling Up on ~~Gas~~ Electricity



- You are driving an EV with a range of 200 miles
- Charging stations along route at distance $d_1, d_2, \ldots, d_n$ from start
- Goal: find the minimum number of charging stops to complete the trip

Filling Up on ~~Gas~~ Electricity



- Given sorted list of stops $d_0 = 0, d_1, d_2, \ldots, d_n, d_{n+1}$
  - d_0 is the start and d_{n+1} is the destination
- Find the smallest set of stops, including d_0 and d_{n+1}, that differ by at most 200 miles
- Greedy algorithm: Start with d_0. Repeatedly do the following: take the farthest-away stop that is less than 200 miles away
- Running time? $O(n)$
- The hard part is showing that this algorithm is correct!

Proof of Correctness

- We'll prove the following invariant: let's say greedy arrives at stop d_i after exactly k stops. Then for *any* other route that arrives at d_j in exactly k stops, we have $j \leq i$.

Proof of Correctness

- Maintain the following invariant: let's say greedy arrives at stop d_i after exactly k stops. Then for *any* other route that arrives at d_j in exactly k stops, we have $j \leq i$.
- If this invariant is satisfied, we are optimal. (Why?)
  - Let greedy have cost C. No algorithm is "past" greedy after $C - 1$ stops, so no algorithm reaches the end in $\leq C - 1$ stops.
- Greedy stays ahead proof strategy

Proof of Correctness

Lemma

If greedy arrives at stop d_i after exactly k stops, then for any other route that arrives at d_j in exactly k stops, we have $j \leq i$.

Proof: By induction. (I.H. is the lemma). Base case: greedy reaches d_0 after 0 stops; all other algorithms must also be at d_0 after 0 stops.

Inductive step: assume the I.H. for some k. Assume the contrary for $k+1$: greedy reaches some stop d_I, whereas some other algorithm A reaches stop d_J with $J > I$.

Let d_j be the previous stop reached by A, and d_i be the previous stop reached by greedy. (Diagram [on blackboard]) We have $d_J - d_j < 200$. And by the I.H., $j \leq i$.

But then $d_J - d_i < 200$, so greedy could also have reached d_J! This contradicts the definition of greedy: it would have chosen d_J rather than d_I.

Proof of Correctness

- Shown: If greedy reaches stop d_i after k stops, then for *any* other route that gets to d_j in k stops, we have $j \leq i$.
- Questions about this problem, or the greedy stays ahead proof strategy?

Class Scheduling (Interval Scheduling)

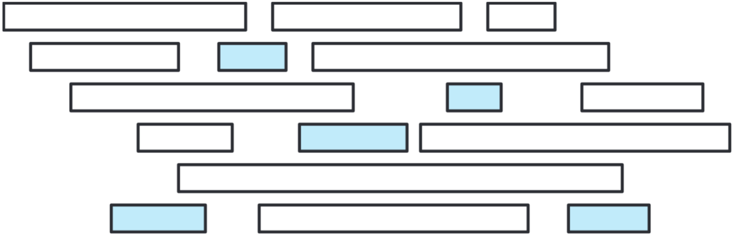


Figure 4.1. A maximum conflict-free schedule for a set of classes.

From Erikson Algorithms textbook

- Set of classes with start times $s_1 \ldots s_n$ and finish times $f_1 \ldots f_n$
  - I'll also call them jobs
- What is the maximum number of non-conflicting classes that can be scheduled?

Class Scheduling (Interval Scheduling)

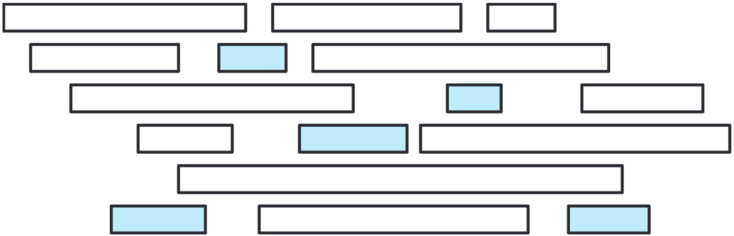


Figure 4.1. A maximum conflict-free schedule for a set of classes.

From Erikson Algorithms textbook

- Can be solved recursively (see Erikson textbook)—correct but slow
- Today: faster algorithm using greedy!

Class Scheduling (Interval Scheduling)

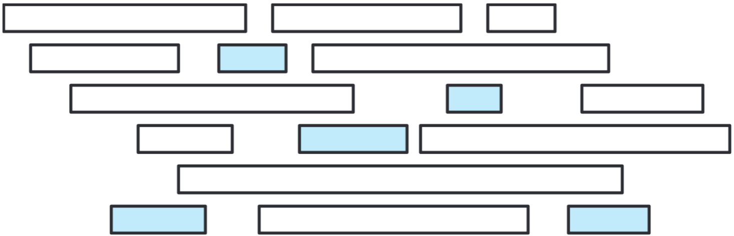


Figure 4.1. A maximum conflict-free schedule for a set of classes.

From Erikson Algorithms textbook

- [on blackboard] Ideas for greedy algorithms for this problem?
  - Not all of these will work! But I want to brainstorm different ways to be greedy.
  - Then we'll talk about counterexamples to some of these ideas

Idea 1: Greedily Choose by Start Time

- Repeatedly pick conflict-free job with earliest start time
- Counterexample: a very long job starts first
- [on blackboard]

Idea 2: Shortest Jobs First

- Repeatedly pick shortest remaining conflict-free job
- Counterexample: a very short job overlaps two jobs
- [on blackboard]

Idea 3: Fewest Conflicts First

- Repeatedly pick the conflict-free job that overlaps the fewest jobs
- Counterexample: [on blackboard]

Idea 4: Earliest Finish Time First

- Repeatedly pick the conflict-free job that ends first
- Counterexample?
- Believe it or not, this actually works
- Brief intuition: if we pick the course that ends earliest, that “frees us up” the soonest
    - Never make a *bad* decision: if another algorithm picked a later-ending job first, we can still take the rest of its schedule! [on blackboard]

Earliest Finish Time First Proof Idea

- Let's say greedy gets some set of jobs G
- The optimal algorithm has some set of jobs O; assume by contradiction that O has a *strictly better* cost than G
- Proof idea: transform O into G one step at a time while keeping the same cost
- More formally: let's say O has C jobs, and O schedules k jobs that G does not (so $|O \setminus G| = k$), then there exists a schedule O' of C jobs that schedules $k-1$ jobs that G does not
- Applying the above repeatedly means that G is optimal! (Contradiction)

$$O \xrightarrow{\text{same cost}} O' \xrightarrow{\text{same cost}} O'' \xrightarrow{\text{same cost}} O'' \xrightarrow{\text{same cost}} O''' \ldots \xrightarrow{\text{same cost}} G$$

Earliest Finish Time First Proof Idea

- Let's say greedy gets some set of jobs G
- The optimal algorithm has some set of jobs O
- Proof idea: *transform* O into G one step at a time while keeping the same cost
- More formally: if O schedules k jobs that G does not, then there exists a schedule O' with the same cost as O that schedules $k - 1$ jobs that G does not
- Applying the above repeatedly means that G is optimal!

$$O \underbrace{\xrightarrow{\text{same cost}} O' \xrightarrow{\text{same cost}} O'' \xrightarrow{\text{same cost}} O'' \xrightarrow{\text{same cost}} O''' \ldots \xrightarrow{\text{same cost}}}_{k \text{ iterations}} G$$

Earliest Finish Time Proof

Lemma

If some schedule O schedules $k \geq 1$ jobs that G does not, then there exists a schedule O' with the same cost as O that schedules $k-1$ jobs that G does not

Proof: Let's write each schedule out in order of finish time:

- $O = o_1, o_2, \ldots, o_m$
- $G = g_1, g_2, \ldots, g_\ell$

Let j be the first index where O schedules a job that G does not. That means we can rewrite $O = g_1, g_2, \ldots, g_{j-1}, o_j, o_{j+1}, \ldots, o_m$.

Then we define O' by replacing o_j with g_j (why must g_j exist?), as follows: $O' = g_1, g_2, \ldots, g_{j-1}, g_j, o_{j+1}, \ldots, o_m$.

Clearly, we have that O' only schedules $k-1$ jobs that G does not.

TODO: We need to show that O' is a legal schedule.

Earliest Finish Time Proof

Lemma

If some schedule O schedules $k \geq 1$ jobs that G does not, then there exists a schedule O' with the same cost as O that schedules $k - 1$ jobs that G does not

Proof: We define O' by replacing o_j with g_j, as follows:
$O' = g_1, g_2, \ldots, g_{j-1}, g_j, o_{j+1}, \ldots, o_m$. We need to show that O' is a legal schedule.

We only need to show that g_j does not conflict with any other job in O' (why?)
(Answer: because O had no conflicts)

By definition of greedy, g_j cannot conflict with $g_1, \ldots, g_{j-1}$.

Since O is a legal schedule, o_j finishes before any job in $o_{j+1}, \ldots, o_m$ starts. By definition of greedy, g_j finishes before o_j. So g_j does not conflict with $o_{j+1} \ldots, o_m$.

Earliest Finish Time Algorithm

```
greedySchedule(J):
    sort J by finish time
    create empty list G
    for each job j in J:
        if j starts after last entry in G ends:
            add j to G
    return G
```

- We showed that this gives an optimal schedule!
- Running time?
- $O(n \log n)$ on n jobs

Earliest Finish Time Proof

- This is called an *Exchange Argument*: we repeatedly alter (exchange) an optimal solution, without increasing cost, until we get the greedy solution

- Proves that greedy is one of the optimal solutions!

- Let's do an example of how this proof works [on blackboard]

Greedy Proof Techniques

1. Greedy stays ahead

2. Exchange argument

Both are good ways to analyze a greedy algorithm! Oftentimes, both actually work—but sometimes one is easier than the other.

- If one is proving very difficult, try the other

- Can look quite similar

What if jobs are weighted?

Challenge question

- Suppose each job has a positive weight
- Goal: schedule the jobs with maximum weight that have no conflict
- [on blackboard] Can you come up with a counterexample where earliest deadline first does not work?

Greedy Algorithms Takeaway

- Greedy algorithms are a *sometimes* thing
- Usually fast; *Correctness* is the main question!
- Only use a greedy algorithm when you can show that it is correct
  - Starting in March we'll look at more sophisticated problem-solving techniques