Greedy Algorithms

Sam McCauley

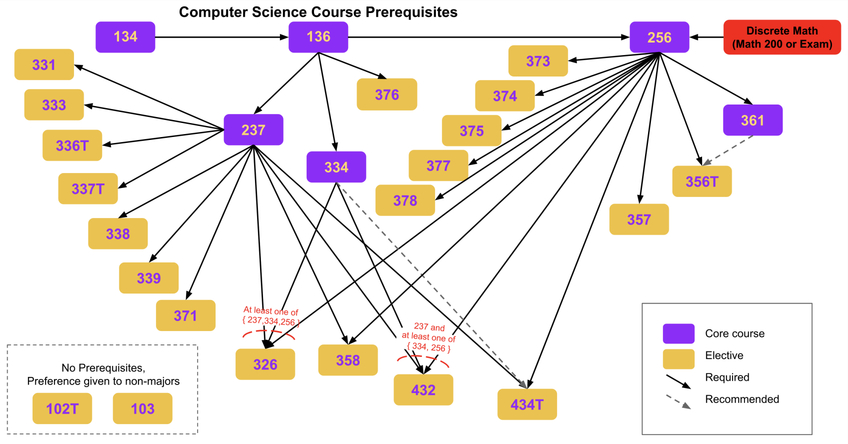
February 19, 2024

Welcome Back!

- Midterm discussion Thursday
- Topics this week (greedy and MST) will be on the midterm
- Will decide if Dijkstra’s is on midterm based on how far we get
- Fully optional, 1-question assignment released Thursday for practice with greedy algorithms
- Any other questions before we start?

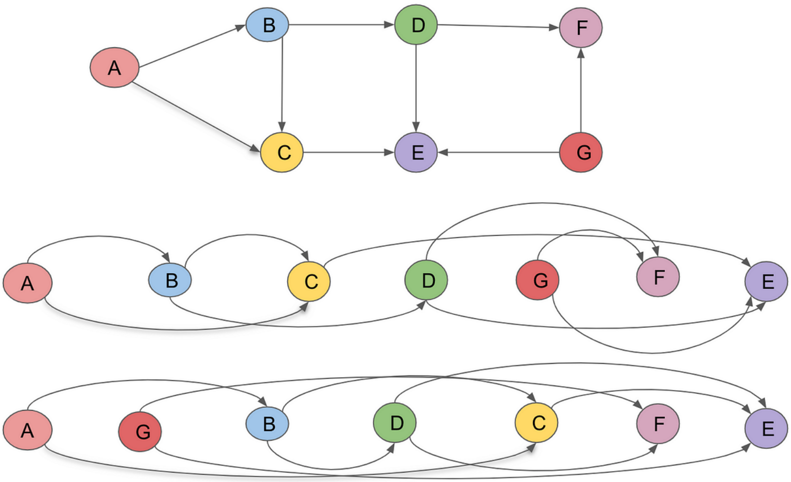
Topological Ordering

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)



Topological Sort

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

DAGs and Toplogical Ordering

Theorem

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

Proof: ($\Rightarrow$) if G has a topological ordering, G cannot have a cycle.

We'll prove ($\Leftarrow$) (if G is acyclic, then G has a topological ordering) in the next few slides.

DAGs and Toplogical Ordering

First, let's prove the following.

Lemma

Every DAG has a vertex with indegree 0.

Proof: Assume the contrary: there exists a DAG G where all vertices have indegree > 0.

Pick a vertex v_0. Find some v_1 such that $(v_1, v_0) \in G$. In general, for each v_i, find a v_{i+1} where $(v_{i+1}, v_i) \in G$.

After n steps, we have a sequence $v_0, v_1, v_2, \ldots, v_n$. One vertex must repeat (why?) (Answer: pidgeonhole principle).

Let $v_i = v_j$ for some $j > i$. Then stepping back through the sequence, $v_j, v_{j-1}, v_{j-2}, \ldots, v_i$ is a cycle. Contradiction.

Topological Ordering: Simple Algorithm

```
L = ∅
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
return L
```

- Can we prove that this algorithm works?

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

Finding Topological Ordering with DFS

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

- Running time?
- $O(n+m)$
- Why does this work?
  - Basic idea: similar to the cycle-finding proof. Every edge (u, v) has that v finishes before u. We *prepend* a vertex when it finishes, so u is before v in L.
- Example [On Board #1]

Greedy Algorithms

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

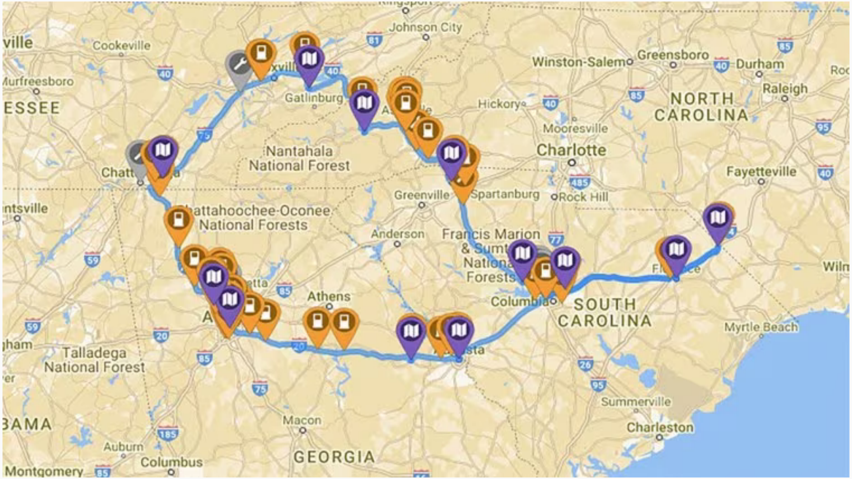
- Can you come up with an amount for which the greedy algorithm does not use the correct number of coins?

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

Greedy Algorithms

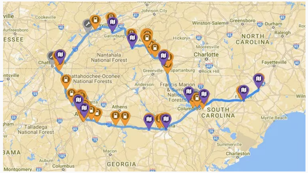
- Greedy algorithms make simple local decisions to obtain an optimal solution
- Are almost always fast!
- Question: can you show that your greedy algorithm is *always correct* for the given problem?

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 $d_0 = 0, d_1, d_2, \ldots, d_n, d_{n+1}$
- 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 we get to stop d_i after k stops. Then if *any* other route gets to d_j in k stops, we have $j \leq i$.

Proof of Correctness

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

Proof of Correctness

Lemma

If greedy reaches stop d_i after k stops, then if any other route gets to d_j in k stops, we have $j \leq i$.

Proof: By induction. (I.H. is the lemma). Base case: greedy reaches d_0 after 1 stop; all other algorithms must also be at d_0 after 1 stop.

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 Board #2]) 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

- Let's say we get to stop d_i after k stops. Then if *any* other route 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)

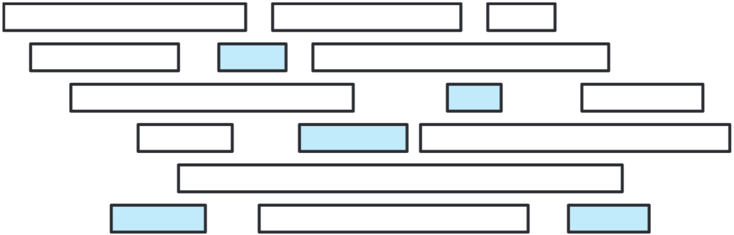


Figure 4.1. A maximum conflict-free schedule for a set of classes.

From Erikson Algorithms textbook

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

Class Scheduling (Interval Scheduling)

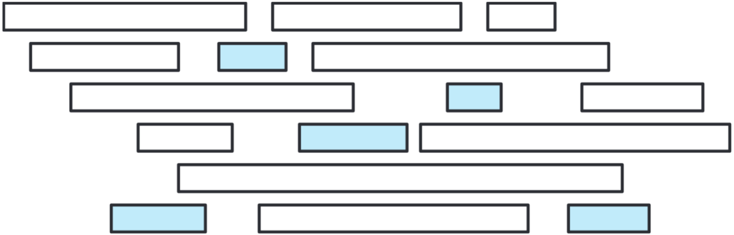


Figure 4.1. A maximum conflict-free schedule for a set of classes.

From Erikson Algorithms textbook

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

Class Scheduling (Interval Scheduling)

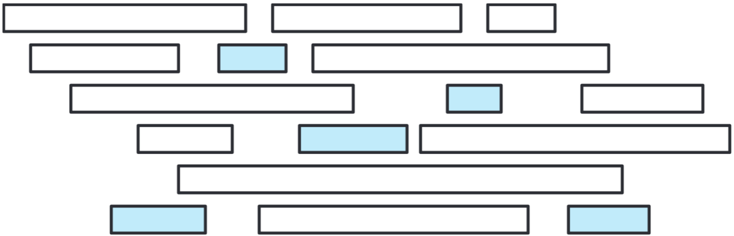


Figure 4.1. A maximum conflict-free schedule for a set of classes.

From Erikson Algorithms textbook

- [On Board #3] 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 Board #4]

Idea 2: Shortest Jobs First

- Repeatedly pick shortest remaining conflict-free job
- Counterexample: a very short job overlaps two jobs
- [On Board #5]

Idea 3: Fewest Conflicts First

- Repeatedly pick the conflict-free job that overlaps the fewest jobs

- Counterexample: [On Board #6]

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 Board #7]

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: 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!

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

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 Board #8]

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 Board #9] 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

Minimum Spanning Trees

Minimum Spanning Tree (MST)

- A “greedy” graph algorithm
- How many of you have seen a minimum spanning tree algorithm before?
- We’ll see two, and talk about MST structure

MST Problem Definition

Given a connected undirected graph G with positive *edge weights* w_e, a *spanning tree* is a set of edges $T \subseteq E$ such that:

- T is a *spanning tree*: T is a tree that connects all vertices, and
- T has *minimum weight*: for any spanning tree T',

$$\sum_{e \in T} w_e \leq \sum_{e \in T'} w_e.$$

In this class we will assume that *all edge weights are distinct*. It just makes the proofs simpler; Prim's and Kruskal's algorithm work without this assumption.

Building to an MST Algorithm

- Can we create an optimal MST on one vertex?
    - How about on two vertices?
- Idea: add minimum weight edge to tree
- Intuition as to why this is optimal?

Prim's Algorithm (Jarník's Algorithm)

First, choose a starting vertex u. Create a set of vertices, starting with $S \leftarrow \{u\}$ and a tree starting with $T \leftarrow \emptyset$.

While $|T| \leq n - 1$, find the min-cost edge $e = (u, v)$ such that one end $u \in S$ and $v \in V \setminus S$. Set $T \leftarrow T \cup \{e\}$ and $S \leftarrow S \cup \{v\}$.

Let's do an example [On Board #10]

First, how can we prove correctness? (Then we'll discuss how to find e efficiently, and the running time.)

Cut Property of MST

A *cut* is a partition of the vertices V into two subsets: S, and $V \setminus S$. A *cut edge* is an edge with one endpoint in S and the other in $V \setminus S$.

Lemma

For any cut S, let $e = (u, v)$ be the minimum weight cut edge. Then e is in every minimum spanning tree of G.

(Recall we are assuming that all weights are distinct)

Cut Property of MST: Proof

Lemma

For any cut S, let $e = (u, v)$ be the minimum weight cut edge. Then e is in every minimum spanning tree of G.

Proof: Assume the contrary: there is an MST T such that $e \notin T$.

There must be some path p from u to v in T. Let $e' = (u', v')$ be the first cut edge in p. Let's draw a diagram [On Board #11]

Consider the set T' created by removing e' from T and adding e. Therefore, T' has smaller weight than T. We claim that T' is a spanning tree: for any two vertices x, y there is a path from x to y in T'.

If $x, y \in S$, then the path from x to y in T is also a path in T'; same if $x, y \in V \setminus S$.

Say $x \in S$ and $y \in V$. Then let p_1 be the path from x to u, and p_2 be the path from v to y. Then p_1, e, p_2 is a path from x to y. $\square$

Proving Prim's Correct

- How can we use the cut property to prove Prim's algorithm correct?

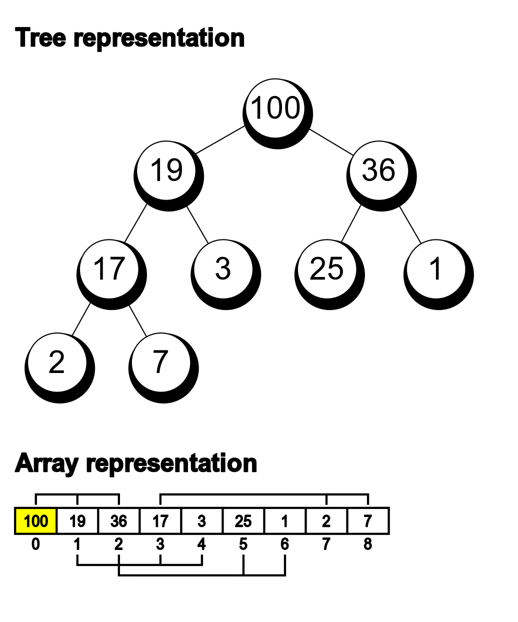
- Every edge we add is the smallest cut edge between S and $V \setminus S$; by the cut property we are done.

Implementing Prim's Algorithm

What do we need to be able to do?

- Maintain all cut edges!
- Must be able to insert new edges when adding a vertex to T
- Must be able to find minimum-weight cut edge (i.e. minimum-weight edge in the data structure) and remove it
- Note that: we will keep some edges from S to S in the data structure. If we remove such an edge we'll just skip it.
- What data structure can insert, and remove minimum weight?
- Answer: priority queue

Priority Queue



- Insert a new item (Insert)
- Remove minimum weight item (ExtractMin)
- Done using a heap

Heaps (Quick Review)

- Heap property: each item in the tree is smaller than either of its children
- Tree has minimum height; filled in left to right (“full” tree)
- Maintain implicitly in an array (do not need pointers!)
- Extract min, or insert a new item, in $O(n \log n)$ time
- Can build a heap in $O(n)$ time (!)

Prim's Algorithm (Jarník's Algorithm)

First, choose a starting vertex u. Create a set of vertices, starting with $S \leftarrow \{u\}$ and a tree starting with $T \leftarrow \emptyset$.

While $|T| \leq n-1$, find the min-cost edge $e = (u, v)$ such that one end $u \in S$ and $v \in V \setminus S$. Set $T \leftarrow T \cup \{e\}$ and $S \leftarrow S \cup \{v\}$.

To implement: each time we add a vertex to S, add its incident edges to T. To find the minimum cut edge, remove edges from T until we find a cut edge. Cost?

Need to do $\leq 2m$ inserts, and $\leq 2m$ extract mins (why?).

Running time: $O(m \log m)$.