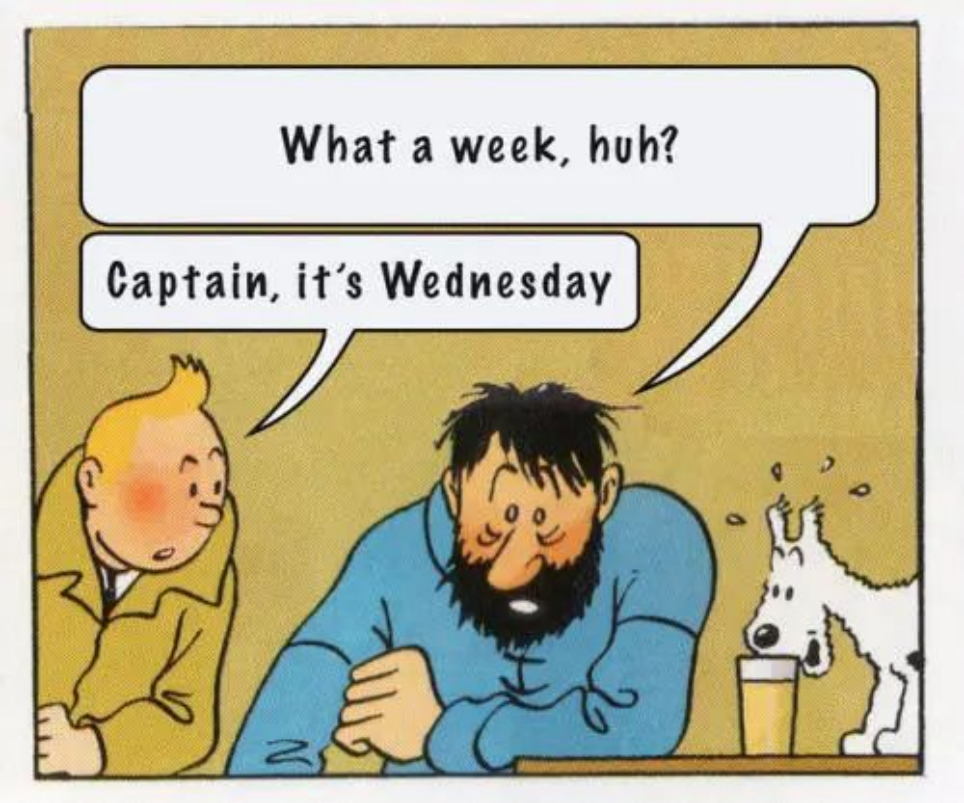
8/10: Final Review!



still true!

Divide and Conquer

Sorting &

Randomization

Data Structures

Graph Search

Dynamic

Programming

Greed & Flow

not on exam! → Special Topics

The final is not in our usual room!

- 200-002 Lane History Corner (southeast corner of the Main Quad, right by the end of the Oval)

- Terry's mnemonic:
  - 200-002 is a palindrome
  - Nvidia Auditorium is not a palindrome

Thanks, Terry.

Small announcements

- I said I'd send the exam to SCPD at 4 PM, so I need the time after class today to make any last changes. So I won't be available in Huang Basement as usual today, sorry :(
- I will have my usual office hours on Thursday (combined online + in person in Durand 319)
- Congrats to those who solved the Pre-HW6 puzzle (Yumou was first, also Quentin, Shi Yi, Will so far)
- If you are not sure about your exam arrangements please reach out to me + Ziang (OAE) or me + Rishu otherwise
  - note: Rishu is on a plane right now so please include me

Format

- 180 minutes
- Less time pressure than the midterm, but still rigorous / many opportunities to show depth of understanding
  - Many 2 pt. T/F, 3 pt. computations... don't get so hung up on one that you don't work on a 15 pt. algorithm!
- Covers all six units, but:
  - lighter on Unit 6
  - some more weight on Units 4 and 5 (the stuff since the midterm), but **fully cumulative** (don't just review the stuff since the midterm)
- Emphasis on:
  - stuff we saw a lot vs. minor corners
  - concepts, short computations, examples rather than formal proofs / details of proofs from class

Some question types

- Short computations (e.g., work through a small example of an algorithm)
- True/false, but no justification needed
- Design (and maybe write code/pseudocode for) an algorithm
  - I know these take a while, especially when you have to code. So there won't be too many.
- Fill in details of a proof, or find a flaw in a proof

Some stuff that's not on the final

- Karatsuba's and Strassen's algorithms
- Modular arithmetic
- Distributed stuff like HW2 Problem 5
- Amortization
- 0/1 knapsack
- Edit distance
- Top-down vs. bottom-up per se
  - still good to understand the paradigms, but no Qs like "is this top-down or bottom-up?"
- Job scheduling, Huffman encoding
- Bipartite matching
- HW6-only stuff (Stable Marriage, Kruskal's)
- Random details from HWs
  - only a couple exceptions, like Floyd-Warshall
- Coming up with a whole proof from scratch
  - You would be provided scaffolding, and it would be a familiar proof type.

Algorithms you should be friends with

- What do I mean by this?
  - Be able to work through a small example
  - Understand the major details of the analysis of running time

Algorithms you should be friends with

- MergeSort
- k-Select
- RadixSort
- QuickSort
  - (know overall idea behind analysis, not so much the gross summations at the end)
- Karger's
  - (more the operational details than the analysis details)
- Heapsort
- Binary search

- BFS
- Dijkstra's
- DFS / Topological sort
- Kosaraju's
- Bellman-Ford
- Floyd-Warshall (HW5)
- Unbounded knapsack
- Prim's
- Ford-Fulkerson

Data structures you should be friends with

- Arrays and linked lists
  - Not a focus of this course, I know. But remember, e.g., why we needed heaps and couldn't just do the same with an array or a linked list.
- Hash tables
- Bloom filters
- Min- (or max-) heaps
  - and understand their array-based binary tree implementation
  - Know what a Fibonacci heap is used for
- Red-black trees
  - less so the rules than the key ideas / guarantees of binary search trees in general

Lecture 8 Key Points

- Understand how BFS and DFS explore a tree in different ways.
- The notion of **finishing time** in DFS is very useful. It
  - establishes a topological order
  - powers Kosaraju's Algorithm
  - can be useful in its own right (e.g., influencers)
- Be comfortable with adjacency lists and adjacency matrices as representations of a directed graph.
- Know what an SCC is and understand the idea of how Kosaraju's Algorithm finds them.

Lecture 8 Miniproblem

Given a directed unweighted graph *G*, write an algorithm to determine:

- the set of vertices in *G* that could potentially have the **largest** finishing time in a DFS of *G*.
- the set of vertices in *G* that could potentially have the **smallest** finishing time in a DFS of *G*.

Lecture 8 Miniproblem Solution

Consider the metagraph of SCCs. Because this graph is necessarily acyclic, any vertex that could have the largest finishing time in a DFS of *G* must be in a "source" SCC (i.e., one that has no incoming edges from other SCCs). That is, suppose there were some other SCC pointing into our vertex's SCC; then our vertex's SCC would have to be completely finished before any node in that other vertex could be finished.

If we run Kosaraju's Algorithm, we naturally recover the metagraph of SCCs, and then we can check each SCC to see if it is a source.

Similar reasoning holds for "sink" SCCs and smallest finishing times.

Lecture 9 Key Points

- This was the midterm review. It might not hurt to watch that again.

- Also consider reviewing the official solutions to the midterm.

Lecture 10 Key Points

- Remember that a negative cycle is a cycle with a *total* negative cost, not just with one or more negative edges.

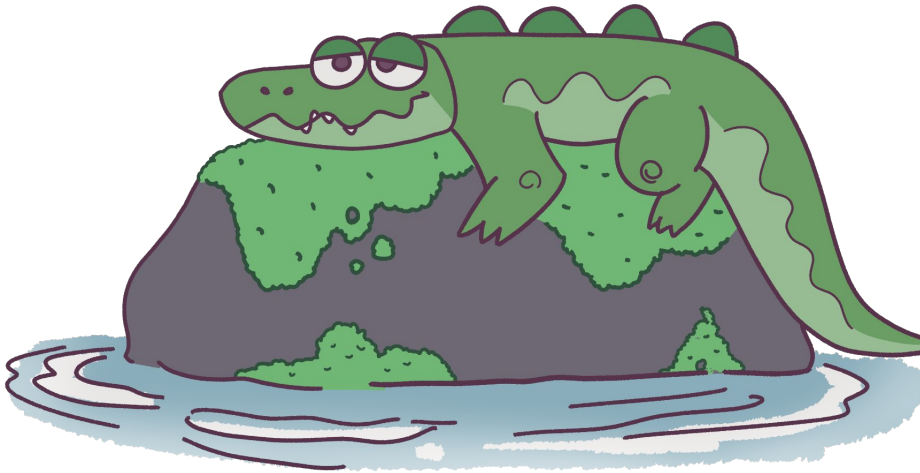
- Understand the idea behind Bellman-Ford – repeatedly "relax" all edges – and how it avoids the problem Dijkstra's can have with a negative edge.

- The Mario example is not testable per se but illustrates the idea of optimal substructure well.
  - What's the best score we can have when we're this far along and in the air / not in the air?

Lecture 10 Miniproblem

Waverly suggests modifying Bellman-Ford as follows: instead of choosing an edge order at the start and then relaxing edges in that order each round, choose a vertex order at the start, and in each round, relax all the edges of each vertex (in arbitrary order).

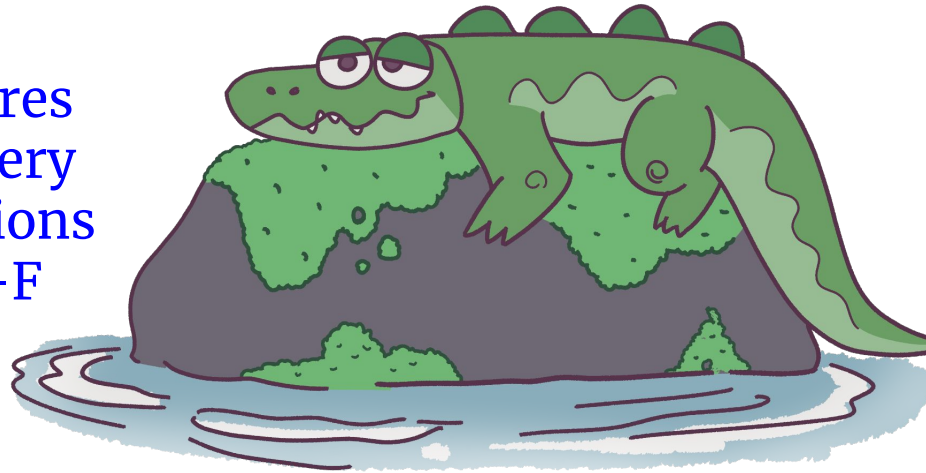
Does this work?



Lecture 10 Miniproblem Solution

Waverly suggests modifying Bellman-Ford as follows: instead of choosing an edge order at the start and then relaxing edges in that order each round, choose a vertex order at the start, and in each round, relax all the edges of each vertex (in arbitrary order).

Does this work? Yes! B-F only cares that every edge gets visited in every round. The order of these visitations might affect how long it takes B-F to converge, but does not affect correctness.



Lecture 11 Key Points

- Reviewing edit distance is good practice for DP problems in general, but it's not on the final.

- DP solutions are often easier to code in a top-down way (start at the original problem, make recursive calls, memoize), but a bottom-up method avoids the large call stack (and possible stack overflow) and is a better long-term choice.

- Understand how unbounded knapsack works – the key idea is how we use previous estimates to get current estimates.

Lecture 11 Miniproblem 1

In the Mario example, we were able to save space by only retaining the previous and current columns of the DP array in memory, instead of the whole array.

Can we do something similar with the array for the unbounded knapsack problem?

Lecture 11 Miniproblem 1 Solution

In the Mario example, we were able to save space by only retaining the previous and current columns of the DP array in memory, instead of the whole array.

Can we do something similar with the array for the unbounded knapsack problem? Kinda. If the heaviest item's weight is w^*, we will always need to look w^* items backward in the DP array in each round. So we can throw out any part of the table that is more than w^* steps away. If w^* is small, this could be a big savings. If w^* is small, it probably doesn't help much.

Lecture 11 Miniproblem 2

In CS109, there is an infamous problem that boils down to determining the probability that the sum of 79 die rolls is 300 or less.

Let $C(t, d)$ be the number of ways to get a total of exactly t when rolling d dice. For instance, $C(5, 1) = 1$, and $C(5, 2) = 4$, since in order to sum to 5, two dice can come up (1, 4), (2, 3), (3, 2), or (4, 1).

Write a recurrence for $C(t, d)$. Specifically:

- Don't forget to include any base case(s) you need.
- The non-base-case part of the recurrence should involve a sum of multiple terms.

Lecture 11 Miniproblem 2 Solution

Here's one way to frame it:

- Base cases:
  - $C(0, 0) = 1$: there is trivially one way to make a total of zero with zero dice.
  - $C(\text{anything else}, 0) = 0$.
  - $C(\text{anything}, \text{anything negative}) = 0$.
  - $C(\text{anything negative}, \text{anything}) = 0$.
- Recurrence:
  - $C(t, d) = C(t-1, d-1) + C(t-2, d-1) + C(t-3, d-1) + C(t-4, d-1) + C(t-5, d-1) + C(t-6, d-1)$

Example:

$C(4, 2) = C(3, 1) + C(2, 1) + C(1, 1) + C(0, 1) + C(-1, 1) + C(-2, 1)$
$= C(3,1) + C(2, 1) + C(1, 1) + C(0, 1) + 0 + 0$

$C(3, 1) = C(2, 0) + C(1, 0) + C(0, 0) + C(-1, 0) + C(-2, 0) + C(-3, 0) = 0 + 0 + 1 + 0 + 0 + 0$

etc.

(There are versions of this that cut down on the recursive calls a bit, e.g., having C(1, 1) through C(6, 1) as base cases equaling 1.)

```
curr = [0] * 301
curr[0] = 1

for i in range(n):
    nxt = [0] * 301
    for j in range(301):
        for k in range(1, 7):
            if j + k <= 300:
                nxt[j + k] += curr[j]
    curr = nxt

print(sum(curr) / 6**79)
```

Lecture 11 Miniproblem 3

This is actually the one I didn't fully get through at the end of Lecture 10...

We have an undirected, unweighted graph. A lizard is at a starting vertex s. There are n eggs, each at a different vertex v_i. The lizard needs to collect at least k **different** eggs and return to the start, in as few moves as possible.

Lecture 11 Miniproblem 3 Solution

We want to do a "single source shortest paths" BFS on a new implicit graph in which each vertex is a tuple of the form:
(vertex ID in the original graph, binary string showing which eggs have been collected)

Then we find the minimum number of steps needed to reach each of the states (start vertex, 110), (start vertex, 101), (start vertex, 011), and take the minimum of those.

Lecture 12 Key Points

- Understand the common method for showing greedy algorithms are correct: we have never ruled out an optimal solution.
  - but be wary – it is easy to mess up the key part of the proof. e.g. HW6 Question 1 part 4 (walking on a graph)

- Be familiar with the activity selection problem.

Lecture 12 Miniproblem 1

Show that the following does **not** work for the activity selection problem:

- Sort the intervals by start time.
- Repeat the following:
    - Select the first activity.
    - Remove all overlapping activities.

Lecture 12 Miniproblem 1 Solution

We should select the orange and green activities, for a total score of 2. But the red one has the earliest start time, so we choose it and rule out the other two, getting only a score of 1.

Lecture 12 Miniproblem 2

Is the following greedy algorithm correct?

Top-Down Topological Sort:

Input: A directed acyclic graph G.

- Repeat the following:
    - For each vertex in G, see how many other vertices it can reach.
    - Remove the (or a) vertex in G that can reach the *most* other vertices. Make it the next element in the growing topological order.

Lecture 12 Miniproblem 2 Solution

This is very slow, but correct. Informally: Suppose that the graph still two vertices *x* and *y* for which a correct topological ordering must have *x* before *y*. Then this means there must be a path from *x* to *y*. So whatever vertices *y* can reach, *x* can also reach those vertices, plus (at least) *y* itself. So *y* cannot be chosen.

Lecture 13 Key Points

- Be able to work through a small example of Ford-Fulkerson.
  - Understand what the residual graph is for, and how to use it.
  - How do we get a final answer? And how do we find a minimum s-t cut if we want one?

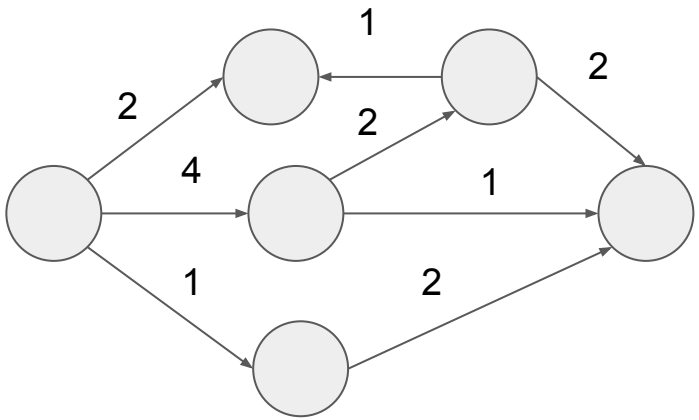
- Bipartite *matching* is not covered, but you should still know what a bipartite graph is (from earlier in the course)

Lecture 13 Miniproblem

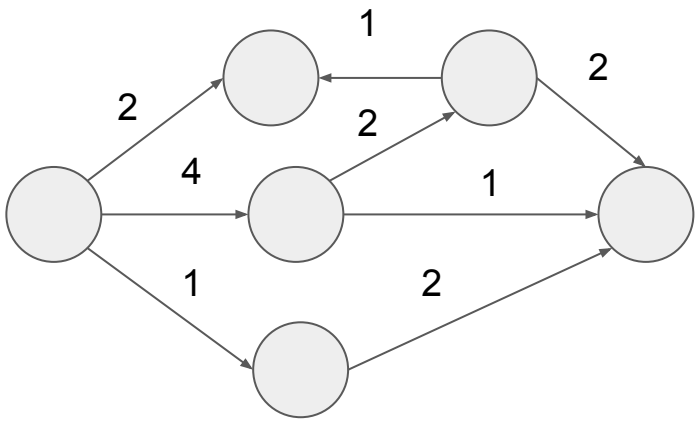
Let's just make up some random graph on the board and find its max flow and a min cut!

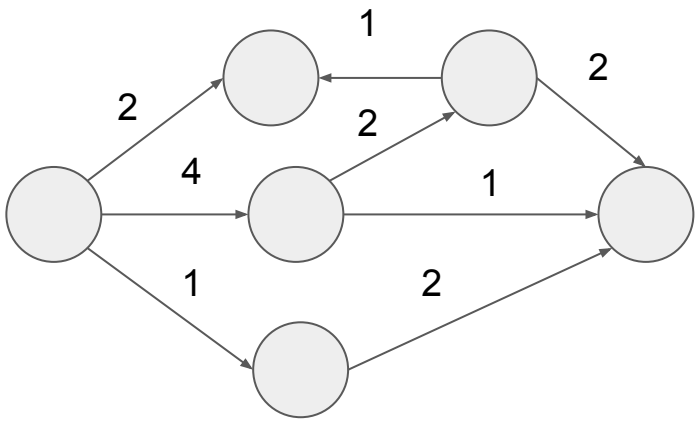
Lecture 13 Miniproblem Solution

The specific example that I threw together on the board is not really worth repeating, but something unexpected did happen, so I'll detail that.



Lecture 13 Miniproblem Solution

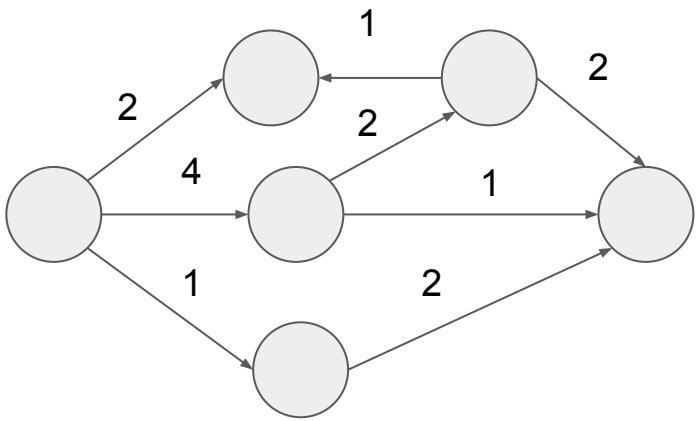


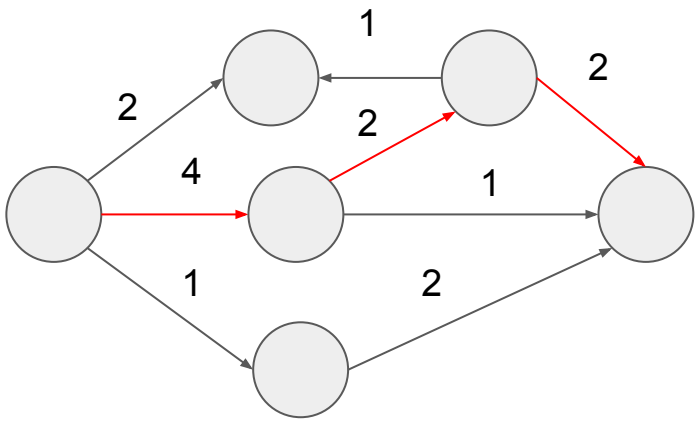


residual graph

Lecture 13 Miniproblem Solution

(arbitrary) augmenting path, bottleneck is size 2

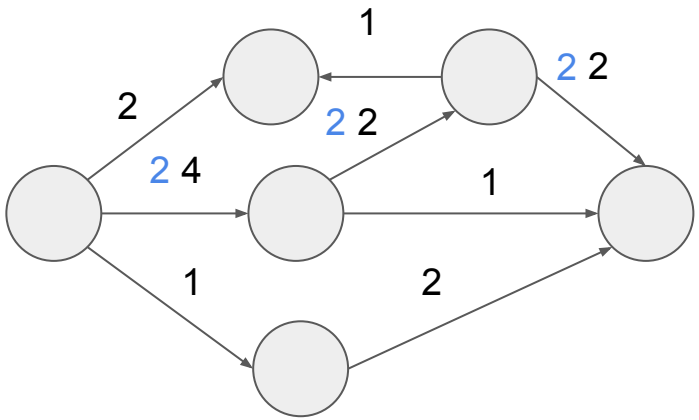




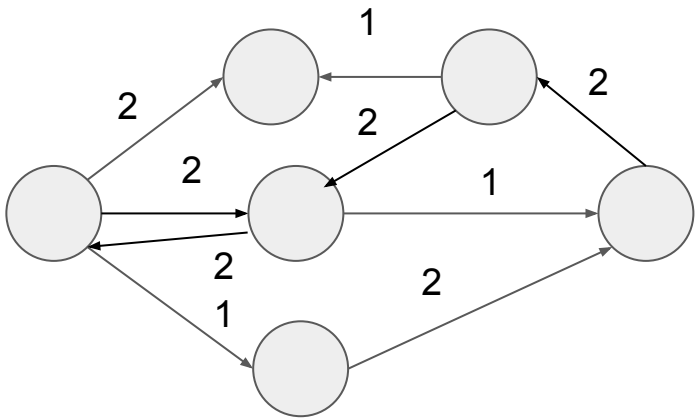
residual graph

Lecture 13 Miniproblem Solution

add 2 units of flow along each of those edges

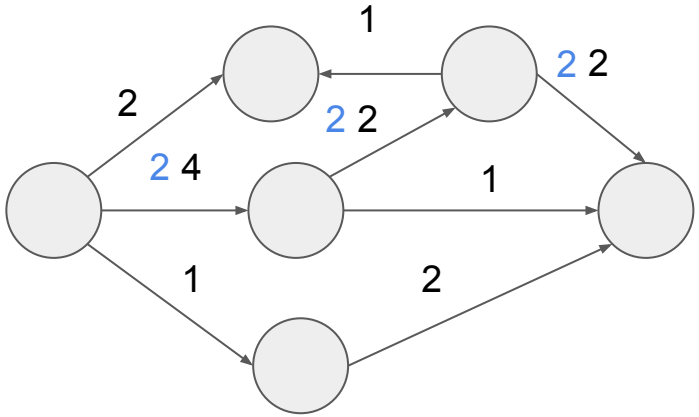


reverse 2 units of capacity along each of those edges

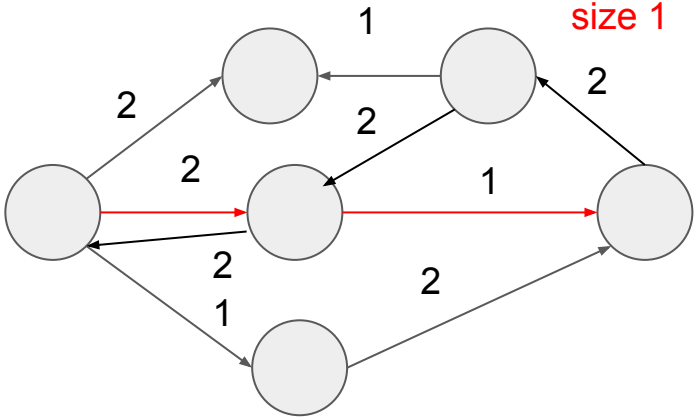


residual graph

Lecture 13 Miniproblem Solution

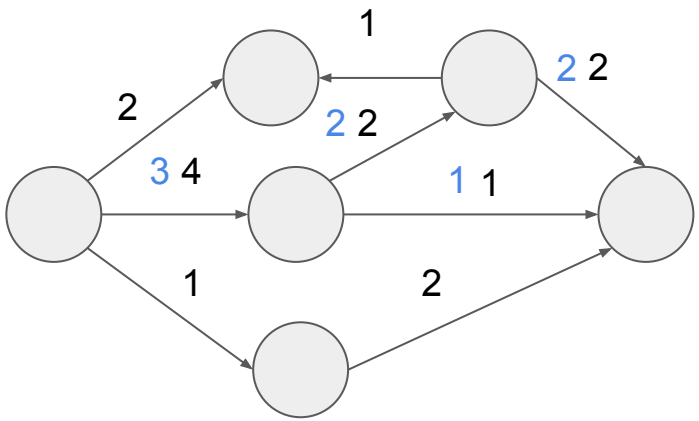


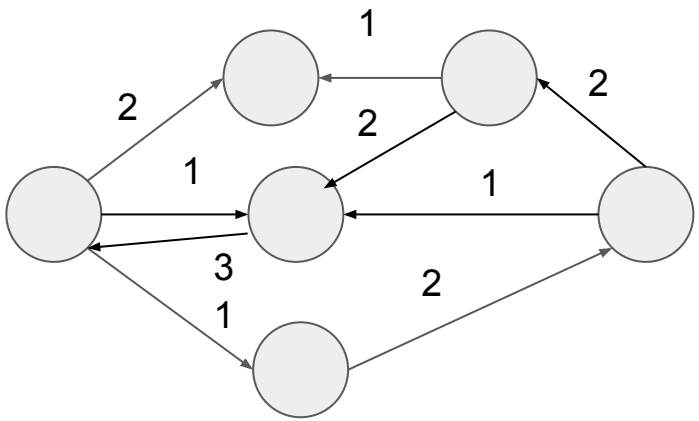
(another arbitrary) augmenting path, bottleneck is size 1



residual graph

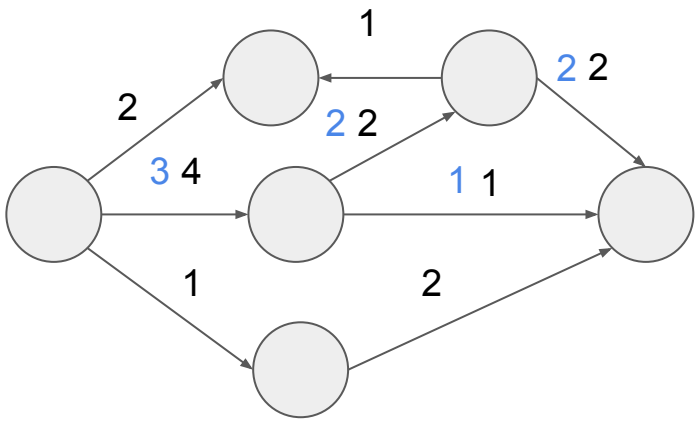
Lecture 13 Miniproblem Solution

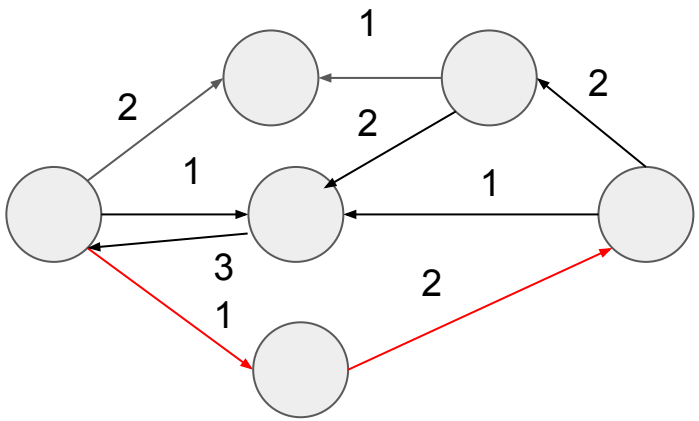




residual graph

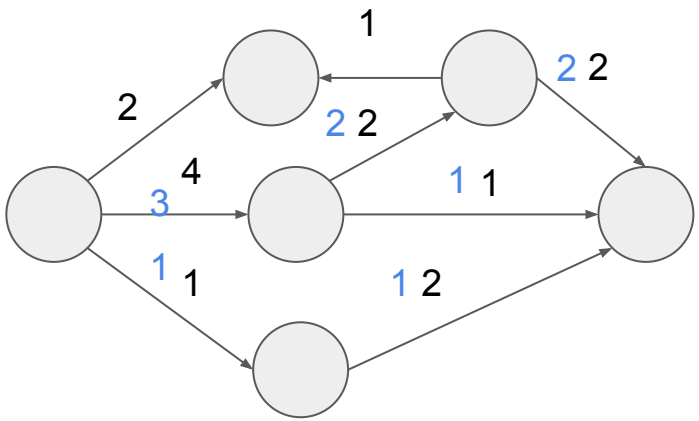
Lecture 13 Miniproblem Solution

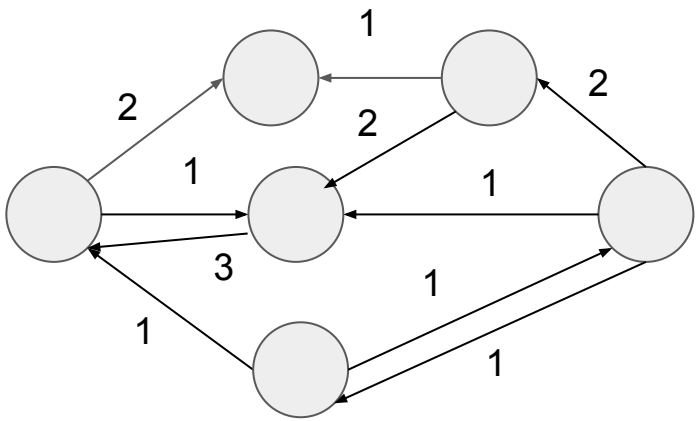




residual graph

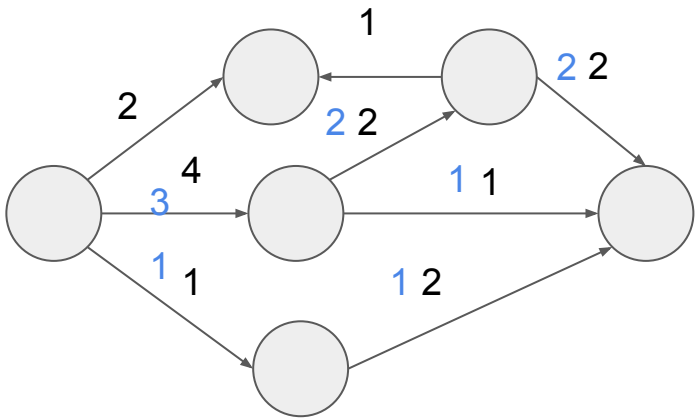
Lecture 13 Miniproblem Solution



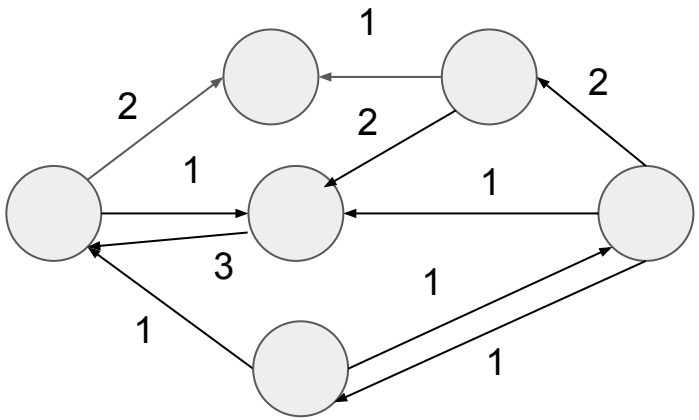


residual graph

Lecture 13 Miniproblem Solution

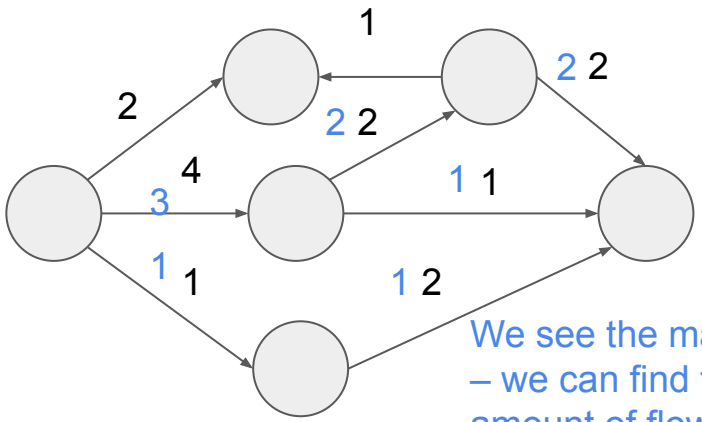


no more augmenting paths!

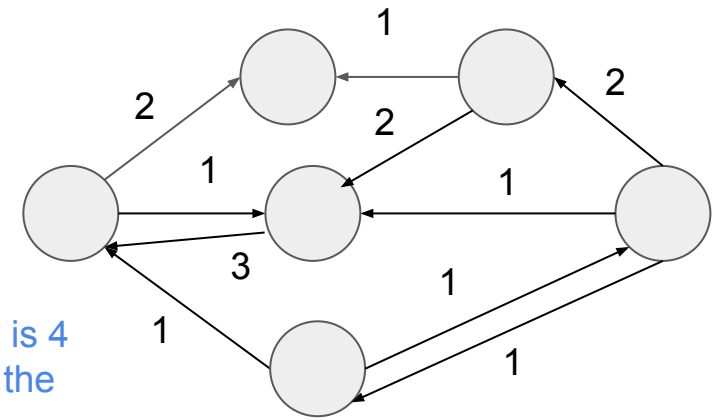


residual graph

Lecture 13 Miniproblem Solution



We see the max flow is 4 – we can find this as the amount of flow (not capacity!) coming out of the source or going into the sink.

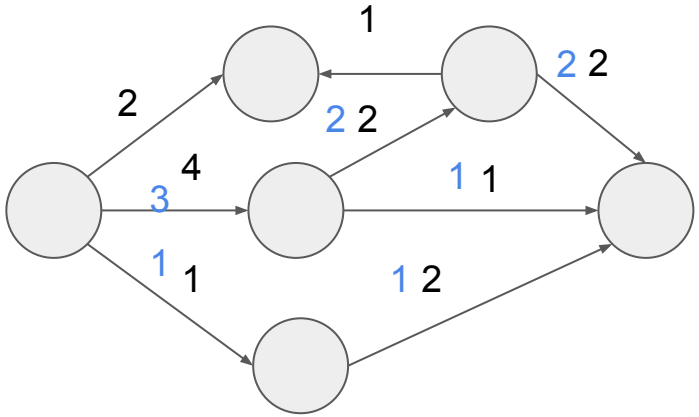


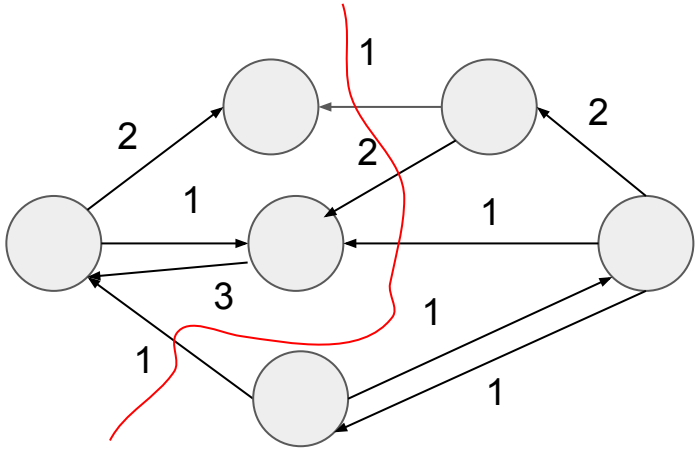
residual graph

Lecture 13 Miniproblem Solution

In lecture I think I said something like "see how far the flow can go before getting stuck, and look at the backwards edges and there's your min cut", but this isn't actually true here.

This cut is proof that there are no more augmenting paths, but it's not a min cut.





residual graph

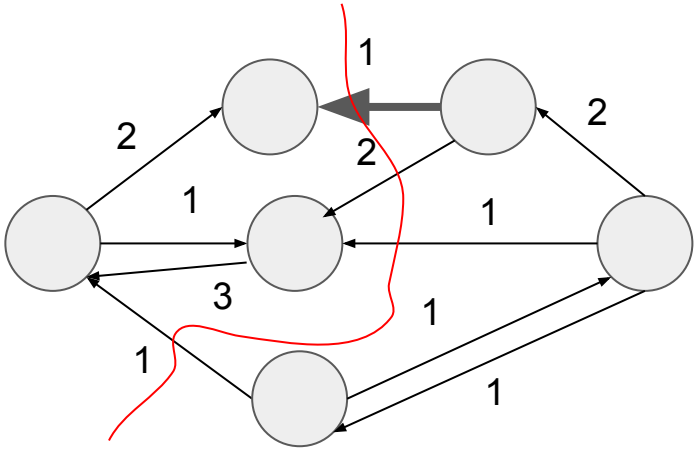
Lecture 13 Miniproblem Solution

In lecture I think I said something like "see how far the flow can go before getting stuck, and look at the backwards edges and there's your min cut", but this isn't actually true here.

This cut is proof that there are no more augmenting paths, but it's not a min cut.

This seems to be a common mistake.

The issue is with the edge I've bolded. It was always "backwards", but here we're treating it as if it got reversed.



residual graph

https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/

atrix Strings Hashing Linked List Stack Queue Binary Tree Binary Search Tree Heap

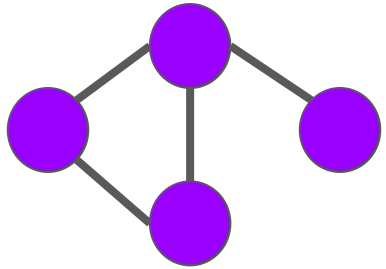
1. Run Ford-Fulkerson algorithm and consider the final residual graph.
2. Find the set of vertices that are reachable from the source in the residual graph.
3. All edges which are from a reachable vertex to non-reachable vertex are minimum cut edges. Print all such edges.

Lecture 13 Miniproblem Solution

A bit more detail about this kind of situation:
https://stackoverflow.com/questions/4482986/how-can-i-find-the-minimum-cut-on-a-graph-using-a-maximum-flow-algorithm

You will not be asked to identify an s-t min cut (in the context of Ford-Fulkerson) on the final.

Misc. review problems from earlier in the course if time



- Show that $n+161$ is $O(n^2)$.
- Find the probability that Karger's succeeds on the above graph.
- Design a data structure (for integers) that meets the usual running time guarantees for self-balancing BSTs, but can also find its smallest odd element and smallest even element in O(1) time.

Show that n+161 is O(n^2).

We will let $c = 100, n_0 = 2$.

Claim (Inductive Hypothesis): For all $n \geq 2, n + 161 \leq 100 \cdot n^2$.

Base case: Let $n = 2$. Then $2 + 161 = 163 < 100 \cdot 2 = 200$.

Inductive step: Suppose that the claim holds for all $2 \leq n < k - 1$. We will show that it also holds for $n = k$.

We have $k + 161 = ((k - 1) + 161 + 1)$.

Using the Claim inductively, $(k-1)+161 \leq 100 \cdot (k-1)^2 = 100k^2 - 200k + 100$.

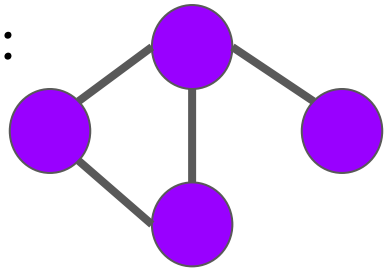
So $k+161 = ((k-1)+161)+1 \leq (100k^2 - 200k + 100)+1 = 100k^2 - 200k + 101$.

Because $2 \leq k$, $101 \leq 102 \leq 51k$, so we can continue the chain of inequalities:

$\leq 100k^2 - 200k + 51k = 100k^2 - 149k \leq 100k^2$.

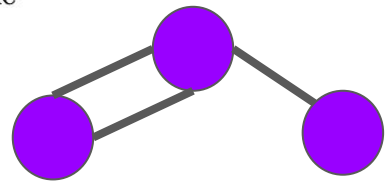
So we have shown $k + 161 \leq 100k^2$, showing that the claim also holds for k, and completing the proof.

Find the probability that Karger's succeeds on:



This graph has only one min cut: the edge to the lone vertex.

With probability $\frac{1}{4}$, we contract this edge in the first step of Karger's, and so we lose the min cut. Otherwise, with probability $\frac{3}{4}$, we contract one of the other edges; no matter which one we choose, we end up with:



Then from here we have a $\frac{1}{3}$ chance of picking the min-cut edge and failing, but we have a $\frac{2}{3}$ chance of picking one of the two edges on the left and thereby leaving behind only the min cut, and succeeding.

So the overall success probability is $\frac{3}{4} \cdot \frac{2}{3} = \boxed{\frac{1}{2}}$.

Design a data structure (for integers) that meets the usual running time guarantees for self-balancing BSTs, but can also find its smallest odd element and smallest even element in O(1) time.

We can use a red-black tree accompanied by two additional red-black trees, one holding only the even elements, and one holding only the odd elements. We also maintain variables holding the smallest elements in the even tree and odd tree.

When we insert into or delete from the structure, we insert into / delete from the "main" red-black tree, but we also insert into / delete from either the "even" tree or the "odd" tree according to whether the new value is even or odd. Then we update the variable holding that tree's smallest element; this update takes $O(\log n)$ time, and we are handling it in the insertion process so that we can meet the O(1) guarantee for returning the smallest odd/even element.

When we are asked for a smallest odd/even element, we return the value of the odd/even tree's variable.