CSE 332

AUGUST 18TH – ALGORITHM DESIGN

ASSORTED MINUTIAE

- Tokens

ASSORTED MINUTIAE

- **Tokens**
    - Due by Friday at midnight

ASSORTED MINUTIAE

- **Tokens**
  - Due by Friday at midnight
  - Submit the solution and a half-page reflection showing what lesson you’ve learned from the exercise

ASSORTED MINUTIAE

- **Tokens**
  - Due by Friday at midnight
  - Submit the solution and a half-page reflection showing what lesson you’ve learned from the exercise
  - Indicate you will use a token using grinch, then submit to canvas.

ASSORTED MINUTIAE

- **Tokens**
  - Due by Friday at midnight
  - Submit the solution and a half-page reflection showing what lesson you’ve learned from the exercise
  - Indicate you will use a token using grinch, then submit to canvas.
  - If you use a token for multiple things, make it into a single document

ASSORTED MINUTIAE

- **Course evaluations**
  - Very important to this class and this department
  - Above all, they’re very important to me
  - Should only take ~5 minutes, and it’s very valuable feedback
  - Only 8 have filled out so far, which is only 5 away from 50% completion… a particularly low bar in my opinion

ALGORITHM DESIGN

- Solving well known problems is great, but how can we use these lessons to approach new problems?

ALGORITHM DESIGN

- **Solving well known problems is great, but how can we use these lessons to approach new problems?**
  - Guess and Check

ALGORITHM DESIGN

- **Solving well known problems is great, but how can we use these lessons to approach new problems?**
    - Guess and Check (Brute Force)

ALGORITHM DESIGN

- **Solving well known problems is great, but how can we use these lessons to approach new problems?**
  - Guess and Check (Brute Force)
  - Linear Solving

ALGORITHM DESIGN

- **Solving well known problems is great, but how can we use these lessons to approach new problems?**
  - Guess and Check (Brute Force)
  - Linear Solving
  - Divide and Conquer

ALGORITHM DESIGN

- **Solving well known problems is great, but how can we use these lessons to approach new problems?**
  - Guess and Check (Brute Force)
  - Linear Solving
  - Divide and Conquer
  - *Randomization and Approximation*

ALGORITHM DESIGN

- **Solving well known problems is great, but how can we use these lessons to approach new problems?**
  - Guess and Check (Brute Force)
  - Linear Solving
  - Divide and Conquer
  - *Randomization and Approximation*
  - *Dynamic Programming*

LINEAR SOLVING

- **Basic linear approach to problem solving**

LINEAR SOLVING

- Basic linear approach to problem solving
- If the decider creates a set of correct answers, find one at a time

LINEAR SOLVING

- **Basic linear approach to problem solving**
- **If the decider creates a set of correct answers, find one at a time**
  - Selection sort: find the lowest element at each run through
- **Sometimes, the best solution**
  - Find the smallest element of an unsorted array

ALGORITHM DESIGN

- Which approach should be used comes down to how difficult the problem is

ALGORITHM DESIGN

- **Which approach should be used comes down to how difficult the problem is**
- **How do we describe problem difficulty?**
  - P : Set of problems that can be solved in polynomial time

ALGORITHM DESIGN

- **Which approach should be used comes down to how difficult the problem is**
- **How do we describe problem difficulty?**
  - P : Set of problems that can be solved in polynomial time
  - NP : Set of problems that can be verified in polynomial time

ALGORITHM DESIGN

- **Which approach should be used comes down to how difficult the problem is**
- **How do we describe problem difficulty?**
  - P : Set of problems that can be solved in polynomial time
  - NP : Set of problems that can be verified in polynomial time
  - EXP: Set of problems that can be solved in exponential time

ALGORITHM DESIGN

- Some problems are provably difficult

ALGORITHM DESIGN

- **Some problems are provably difficult**
  - Humans haven’t beaten a computer in chess in years, but computers are still far away from “solving” chess

ALGORITHM DESIGN

- **Some problems are provably difficult**
  - Humans haven't beaten a computer in chess in years, but computers are still far away from "solving" chess
  - At each move, the computer needs to approximate the best move

ALGORITHM DESIGN

- **Some problems are provably difficult**
  - Humans haven't beaten a computer in chess in years, but computers are still far away from "solving" chess
  - At each move, the computer needs to approximate the best move
  - Certainty always comes at a price

APPROXIMATION DESIGN

- **What is approximated in the chess game?**

APPROXIMATION DESIGN

- **What is approximated in the chess game?**
  - Board quality – If you could easily rank which board layout in order of quality, chess is simply choosing the best board

APPROXIMATION DESIGN

- **What is approximated in the chess game?**
  - Board quality – If you could easily rank which board layout in order of quality, chess is simply choosing the best board
  - It is very difficult, branching factor for chess is ~35

APPROXIMATION DESIGN

- **What is approximated in the chess game?**
  - Board quality – If you could easily rank which board layout in order of quality, chess is simply choosing the best board
  - It is very difficult, branching factor for chess is ~35
  - Look as many moves into the future as time allows to see which move yields the best outcome

APPROXIMATION DESIGN

- Recognize what piece of information is costly and useful for your algorithm

APPROXIMATION DESIGN

- **Recognize what piece of information is costly and useful for your algorithm**
    - Consider if there is a cheap way to estimate that information

APPROXIMATION DESIGN

- **Recognize what piece of information is costly and useful for your algorithm**
    - Consider if there is a cheap way to estimate that information
    - Does your client have a tolerance for error?
    - Can you map this problem to a similar problem?
    - “Greedy” algorithms are often approximators

RANDOMIZATION DESIGN

- **Randomization is also another approach**

RANDOMIZATION DESIGN

- **Randomization is also another approach**
  - Selecting a random pivot in quicksort gives us more certainty in the runtime

RANDOMIZATION DESIGN

- **Randomization is also another approach**
  - Selecting a random pivot in quicksort gives us more certainty in the runtime
  - This doesn’t impact correctness, a randomized quicksort still returns a sorted list

RANDOMIZATION DESIGN

- **Randomization is also another approach**
  - Selecting a random pivot in quicksort gives us more certainty in the runtime
  - This doesn’t impact correctness, a randomized quicksort still returns a sorted list
- **Two types of randomized algorithms**
  - Las Vegas – correct result in random time

RANDOMIZATION DESIGN

- **Randomization is also another approach**
    - Selecting a random pivot in quicksort gives us more certainty in the runtime
    - This doesn't impact correctness, a randomized quicksort still returns a sorted list
- **Two types of randomized algorithms**
    - Las Vegas – correct result in random time
    - Montecarlo – estimated result in deterministic time

RANDOMIZATION DESIGN

- Can we make a Montecarlo quicksort?

RANDOMIZATION DESIGN

- **Can we make a Montecarlo quicksort?**
  - Runs O(n log n) time, but not guaranteed to be correct

RANDOMIZATION DESIGN

- **Can we make a Montecarlo quicksort?**
  - Runs O(n log n) time, but not guaranteed to be correct
  - Terminate a random quicksort early!

RANDOMIZATION DESIGN

- **Can we make a Montecarlo quicksort?**
  - Runs O(n log n) time, but not guaranteed to be correct
  - Terminate a random quicksort early!
  - If you haven’t gotten the problem in some constrained time, just return what you have.

RANDOMIZATION DESIGN

- How *close* is a sort?
- If we say a list is 90% sorted, what do we mean?

RANDOMIZATION DESIGN

- **How *close* is a sort?**
- **If we say a list is 90% sorted, what do we mean?**
  - 90% of elements are smaller than the object to the right of it?

RANDOMIZATION DESIGN

- **How *close* is a sort?**
- **If we say a list is 90% sorted, what do we mean?**
  - 90% of elements are smaller than the object to the right of it?
  - The longest sorted subsequence is 90% of the length?

RANDOMIZATION DESIGN

- **How *close* is a sort?**
- **If we say a list is 90% sorted, what do we mean?**
  - 90% of elements are smaller than the object to the right of it?
  - The longest sorted subsequence is 90% of the length?
- **Analysis for these problems can be very tricky, but it's an important approach**

RANDOMIZATION

- Guess and check

RANDOMIZATION

- **Guess and check**
  - How bad is it?

RANDOMIZATION

- **Guess and check**
  - How bad is it?
    - Necessary for some hard problems

RANDOMIZATION

- **Guess and check**
  - How bad is it?
    - Necessary for some hard problems
    - Still can be useful for some easier problems

RANDOMIZATION

- If an algorithm has a chance P of returning the correct answer to an NP-complete problem in $O(n^k)$ time

RANDOMIZATION

- **If an algorithm has a chance P of returning the correct answer to an NP-complete problem in O(n^k) time**
  - P is our success probability

RANDOMIZATION

- **If an algorithm has a chance P of returning the correct answer to an NP-complete problem in $O(n^k)$ time**
  - P is our success probability
  - NP-complete means we can check a solution in $O(n^k)$ time, but we can find the exact solution in $O(k^n)$ time – very bad

RANDOMIZATION

- **If an algorithm has a chance P of returning the correct answer to an NP-complete problem in $O(n^k)$ time**
  - P is our success probability
  - NP-complete means we can check a solution in $O(n^k)$ time, but we can find the exact solution in $O(k^n)$ time – very bad
  - Suppose we want to have a confidence equal to α, how do we get this?

RANDOMIZATION

- Even if P is low, we can increase our chance of finding the correct solution by running our randomized estimator multiple times

RANDOMIZATION

- **Even if P is low, we can increase our chance of finding the correct solution by running our randomized estimator multiple times**
  - We can verify solutions in polynomial time, so we can just guess-and-check.

RANDOMIZATION

- **Even if P is low, we can increase our chance of finding the correct solution by running our randomized estimator multiple times**
  - We can verify solutions in polynomial time, so we can just guess-and-check.
  - How many times do we need to run our algorithm to be sure our chance of error is less than α?

RANDOMIZATION

- **Even if P is low, we can increase our chance of finding the correct solution by running our randomized estimator multiple times**
  - We can verify solutions in polynomial time, so we can just guess-and-check.
  - How many times do we need to run our algorithm to be sure our chance of error is less than α?

RANDOMIZATION

$$(1-p)^k = \alpha$$

RANDOMIZATION

$$(1-p)^k = \alpha$$

$$k*\ln(1-p) = \ln \alpha$$

$$k = \frac{(\ln \alpha)}{(\ln(1-p)}$$

$$k = \log_{(1-p)} \alpha$$

RANDOMIZATION

- Cool, I guess… but what does this mean?

RANDOMIZATION

- Cool, I guess… but what does this mean?
- Suppose P = 0.5 (we only have a 50% chance of success on any given run) and $\alpha = 0.001$, we only tolerate a 0.1% error

RANDOMIZATION

- Cool, I guess… but what does this mean?
- Suppose P = 0.5 (we only have a 50% chance of success on any given run) and $\alpha = 0.001$, we only tolerate a 0.1% error
- How many runs do we need to get this level of confidence?

RANDOMIZATION

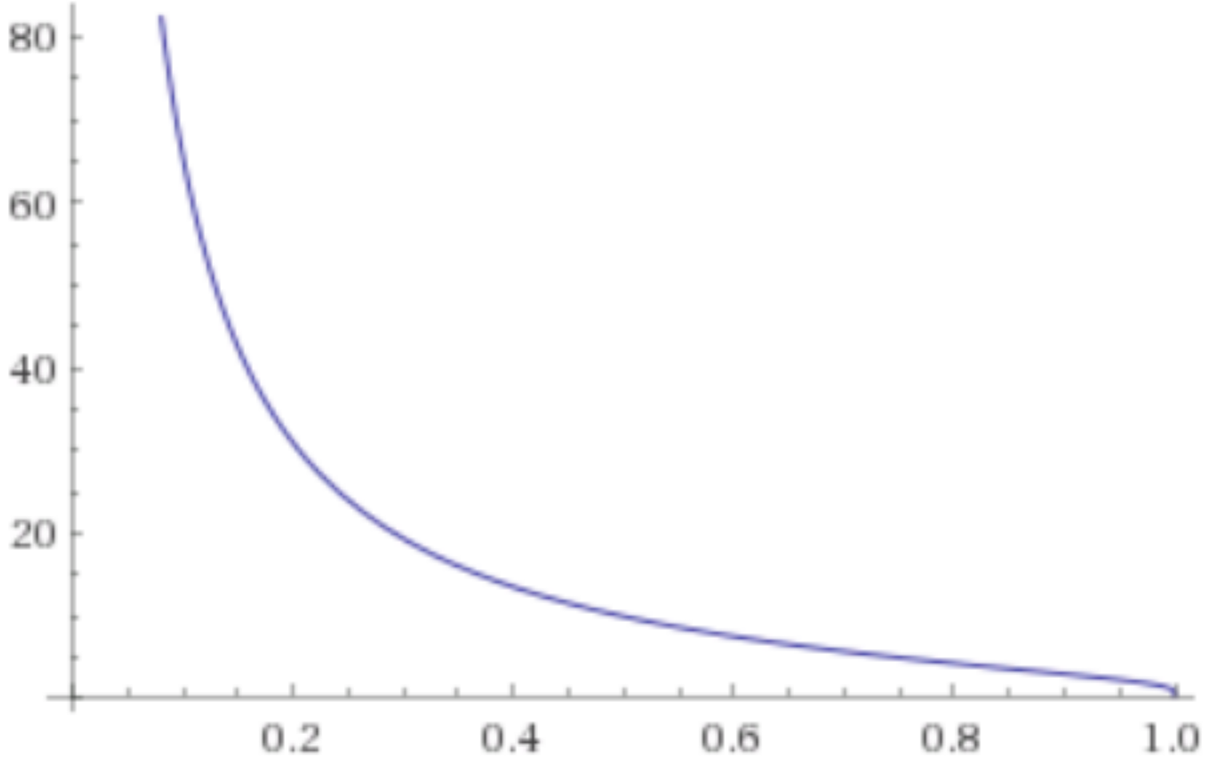
- **Cool, I guess… but what does this mean?**
- **Suppose P = 0.5 (we only have a 50% chance of success on any given run) and $\alpha = 0.001$, we only tolerate a 0.1% error**
- **How many runs do we need to get this level of confidence?**
  - Only 10! This is a constant multiple

RANDOMIZATION

- In fact, suppose we always want our error to be 0.1%, how does this change with p?

RANDOMIZATION

- In fact, suppose we always want our error to be 0.1%, how does this change with p?



RANDOMIZATION

- Even if p is 0.1, only a 10% chance of success, we only need to run the algorithm 80 times to get a 0.001 confidence level

RANDOMIZATION

- Even if p is 0.1, only a 10% chance of success, we only need to run the algorithm 80 times to get a 0.001 confidence level
- What does this mean?

RANDOMIZATION

- **Even if p is 0.1, only a 10% chance of success, we only need to run the algorithm 80 times to get a 0.001 confidence level**
- **What does this mean?**
  - Randomized algorithms don’t have to be complicated, if you can create a *reasonable* guess and can verify it in a short amount of time, then you can get good performance just from running repeatedly.

MINCUT

- **Suppose there is a graph G(V,E)**

MINCUT

- Suppose there is a graph G(V,E)
- Find the two non-empty subgraphs V_1 and V_2 such that V_1 U V_2 = V and the set of edges connecting them are minimal

MINCUT

- Suppose there is a graph G(V,E)
- Find the two non-empty subgraphs V_1 and V_2 such that V_1 U V_2 = V and the set of edges connecting them are minimal

- Why do we even care?

MINCUT

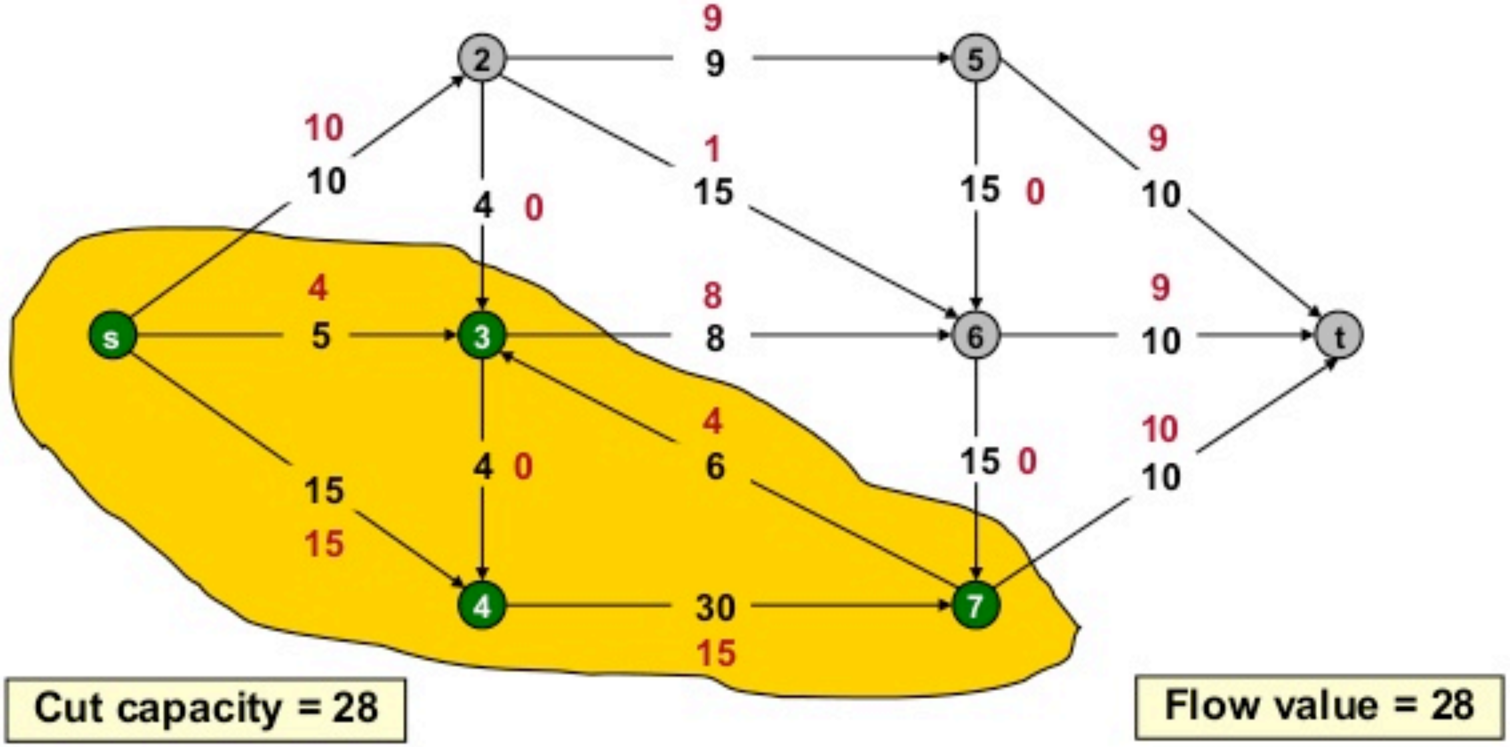
- **Suppose there is a graph G(V,E)**
- **Find the two non-empty subgraphs V_1 and V_2 such that $V_1 \cup V_2 = V$ and the set of edges connecting them are minimal**

- **Why do we even care?**
  - The min-cut is the maximum flow, if we are trying to connect two cities, the limit of traffic flow between nodes in the network

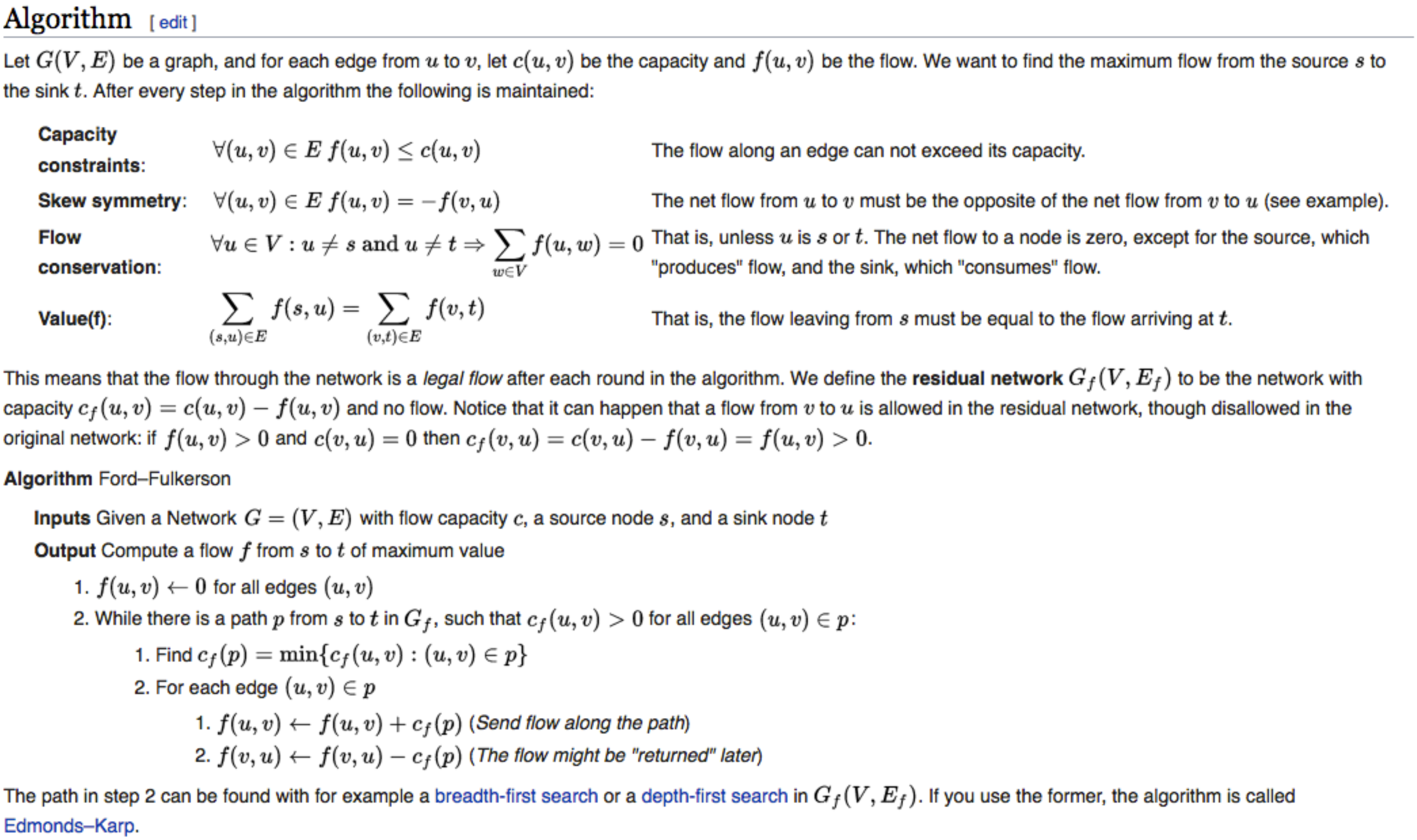
Max-Flow Min-Cut Theorem

MAX-FLOW MIN-CUT THEOREM (Ford-Fulkerson, 1956): In any network, the value of the max flow is equal to the value of the min cut.

- "Good characterization."
- Proof IOU.



FORD-FULKERSON



Algorithm [edit]

Let $G(V,E)$ be a graph, and for each edge from u to v, let $c(u,v)$ be the capacity and $f(u,v)$ be the flow. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

Capacity constraints:	$\forall (u,v) \in E\ f(u,v) \le c(u,v)$	The flow along an edge can not exceed its capacity.
Skew symmetry:	$\forall (u,v) \in E\ f(u,v) = -f(v,u)$	The net flow from u to v must be the opposite of the net flow from v to u (see example).
Flow conservation:	$\forall u \in V : u \neq s \text{ and } u \neq t \Rightarrow \sum_{w \in V} f(u,w) = 0$	That is, unless u is s or t. The net flow to a node is zero, except for the source, which "produces" flow, and the sink, which "consumes" flow.
Value(f):	$\sum_{(s,u) \in E} f(s,u) = \sum_{(v,t) \in E} f(v,t)$	That is, the flow leaving from s must be equal to the flow arriving at t.

This means that the flow through the network is a *legal flow* after each round in the algorithm. We define the **residual network** $G_f(V,E_f)$ to be the network with capacity $c_f(u,v) = c(u,v) - f(u,v)$ and no flow. Notice that it can happen that a flow from v to u is allowed in the residual network, though disallowed in the original network: if $f(u,v) > 0$ and $c(v,u) = 0$ then $c_f(v,u) = c(v,u) - f(v,u) = f(u,v) > 0$.

```
Algorithm Ford–Fulkerson
    Inputs Given a Network $G = (V,E)$ with flow capacity $c$, a source node $s$, and a sink node $t$
    Output Compute a flow $f$ from $s$ to $t$ of maximum value
        1. $f(u,v) \leftarrow 0$ for all edges $(u,v)$
        2. While there is a path $p$ from $s$ to $t$ in $G_f$, such that $c_f(u,v) > 0$ for all edges $(u,v) \in p$:
            1. Find $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
            2. For each edge $(u,v) \in p$
                1. $f(u,v) \leftarrow f(u,v) + c_f(p)$ (Send flow along the path)
                2. $f(v,u) \leftarrow f(v,u) - c_f(p)$ (The flow might be "returned" later)
```

The path in step 2 can be found with for example a breadth-first search or a depth-first search in $G_f(V,E_f)$. If you use the former, the algorithm is called Edmonds–Karp.

FORD-FULKERSON

- Bleh. Garbage. Who has the time?

FORD-FULKERSON

- Bleh. Garbage. Who has the time?
- Can we estimate the min-cut?

FORD-FULKERSON

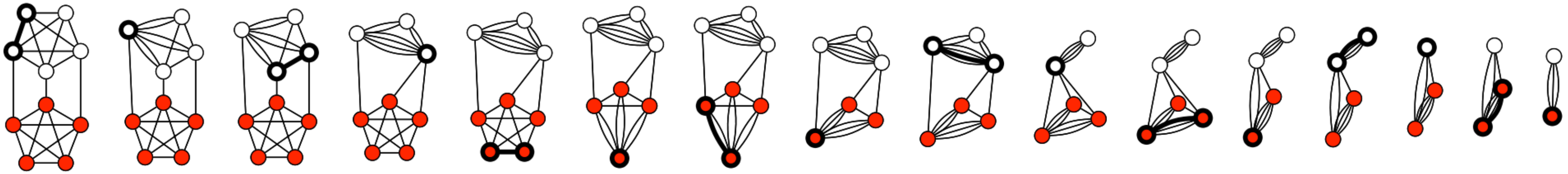
- **Bleh. Garbage. Who has the time?**
- **Can we estimate the min-cut?**
  - What might be an easy estimator?

FORD-FULKERSON

- **Bleh. Garbage. Who has the time?**
- **Can we estimate the min-cut?**
  - What might be an easy estimator?

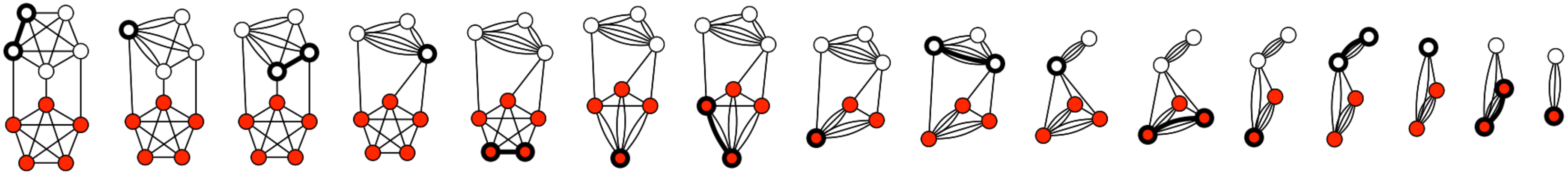
KARGER'S ALGORITHM

- **Bleh. Garbage. Who has the time?**
- **Can we estimate the min-cut?**
  - What might be an easy estimator?
- **Contract edges at random!**
  - How many edges will you contract to get two subgraphs?



KARGER'S ALGORITHM

- **Bleh. Garbage. Who has the time?**
- **Can we estimate the min-cut?**
  - What might be an easy estimator?
- **Contract edges at random!**
  - How many edges will you contract to get two subgraphs?
  - Only |V|-2



KARGER'S ALGORITHM

- Does this work?

KARGER'S ALGORITHM

- **Does this work?**
  - Success probability of 2/|E|

KARGER'S ALGORITHM

- **Does this work?**
  - Success probability of 2/|E|
  - Run it O(E) times, and you have a bounded success rate!

RANDOMIZATION CONCLUSION

- Good for estimating difficult problems in constrained time

RANDOMIZATION CONCLUSION

- Good for estimating difficult problems in constrained time
- Relies on the quality of the guess

RANDOMIZATION CONCLUSION

- Good for estimating difficult problems in constrained time
- Relies on the quality of the guess
- Important approach to consider in modern computing

EXAM FORMAT

- **Two one-hour portions**
- **Material before midterm and from the projects are acceptable both days**
- **First, Thursday 9:40 – 10:40**
  - Parallelism and Sorting
- **Second, Friday 9:40 – 10:40**
  - Graphs and Algorithms

EXAM FORMAT

- We will be our most strict grading yet, don’t make any assumptions that aren’t explicit
- Analysis work needs to be thorough and concrete, recurrences and summations will likely be required
- Show all of your work. Many algorithms are trivial to solve by hand. Just providing “the solution” will not earn points. Algorithms are about process.

EXAM FORMAT

- **A time crunch is likely**
  - There are many topics that need to be covered
  - Get down things that you know, and if you don’t make progress move on and come back

TOPICS

- **Definitions**
  - ADT – Abstract Data Type – Describes a certain set of functionality and behavior
    - e.g. PriorityQueue
  - Data structure – Theoretical storage method that implements an ADT.
    - e.g. Heap
  - Implementation – Low-level design decisions that are often language dependent
    - e.g. Using an array for the heap

TOPICS

- **Stacks and Queues**
  - LIFO and FIFO ordered storage respectively
  - Can be implemented with arrays or linked lists
  - Understand the desired behavior and how to implement these structures

TOPICS

- **Priority Queues**
  - Insert(key, priority)
  - findMin()
  - deleteMin()
  - changePriority(key, newPriority)

TOPICS

- **Heaps**
  - Usually array implementations
  - Heap property
  - Complete trees
  - Runtimes and buildHeap()

TOPICS

- **Algorithm analysis**
  - bigO, bigOmega, bigTheta
  - c and n_0
  - Asymptotic behavior
  - Memory analysis
  - Recurrences
  - Summations
  - Work and Span

TOPICS

- **Dictionary**
  - ADT- insert(k,v), find(k) delete(k)
  - Many possible underlying data structures
  - Different runtimes (and support)

TOPICS

- **Binary search trees**
  - Best and worst case
  - Traversals
- **Balance property – AVL**
  - Rotations and correctness

TOPICS

- **Hashtables**
  - Linear, quadratic, secondary hashing
  - Separate chaining
  - Load factor and resizing
  - Primary and Secondary clustering
  - Runtime and memory constraints

TOPICS

- **B+-trees**
  - Temporal and Spatial localities
  - Pages and their use
  - Tiered caching
  - Basic rules and implementations
  - Signposts and Leaves

TOPICS

- **Parallelism**
  - ForkJoinPool
  - Work and Span
  - Speed-up
  - Debugging
  - Parallel primitives

TOPICS

- **Synchronization**
    - Critical Sections
    - Mutual Exclusion
    - Deadlock resolution
    - Course v. Fine-grained locking
    - Race conditions

TOPICS

- **Project Material**
  - Minimax
  - Alphabeta
  - Iterators
  - Debugging
  - Tries
  - N-grams

TOPICS

- **Graphs**
  - Notation G(V,E)
  - Traversals
  - Topological Sorts
  - Properties
    - Directed v. Undirected
    - Dense v. Sparse
    - Weighted v. Unweighted
    - Cyclic v. Acyclic

TOPICS

- **Graphs**
  - Algorithms
    - Dijkstra’s – path finding
    - Prim’s and Kruskal’s – Minimum spanning trees
  - Know their runtimes and the data structures they rely on for those runtimes…

TOPICS

- **Union find**
    - ADT – Disjoint sets
    - Partitions
    - Weighted Union
    - Path compression
    - Uptree – single array representation

TOPICS

- **Sorting**
  - Insertion and Selection
  - Heap, Merge and Quick
  - Bucket and Radix
- **Properties**
  - Comparison sorts
  - Stable
  - In place
  - Interruptible (top k)

TOPICS

- **Analysis**
  - Lower bound for comparison sorts
  - Memory usages for sorting
  - Best and worst case runtimes
  - Work and Span for parallel algorithms

TOPICS

- **Algorithm Design**
  - How can you approach the problem?
    - Guess and check (Approximation)
    - Brute Force (Linear Work)
    - Divide and Conquer
    - *Greedy algorithms (make best decision for a local sub-problem)*
    - Randomization, Las Vegas and Monte Carlo
    - Preprocessing

FINAL WORDS

- **Great quarter!**
- **Stressful week**
  - Nothing feels better than walking out of an exam and…
  - Filling out course evaluations!
- **Course has been tough**
  - But you have learned a lot

FINAL WORDS

- Good luck!
- Have a nice “summer”!