Network Flows

Schedule

- Next Monday is midterm 2
- Next Thursday there is **no class**
- We'll start back up in May
- Only one problem set in May
- Final will be cumulative



Preregistration

- Starts this week
- Talk to an advisor and get them to release your hold
- CS majors aren't attached to a specific "advisor" any faculty member is OK
- Schedule is linked under "advising and majors information" on the website
- Jeannie sent an email with some more details (and a copy of the schedule)

Next steps (courses)



- 361 (theory of computation): focuses more on "lower bound" side of things; proof based
- 432 (operating systems): a systems/programming course. Things like how your computer manages memory, schedules processes, stores files. Tons of little algorithms (prereq: 237)
- 358 (applied algorithms): learn about how algorithms can make systems run faster. Some proofs, more coding focused. (prereq: 237 or experience coding in C)
- 270 (AI): Artificial intelligence. Will likely be a rereq for all other AI/ML courses in the future. Offered both in the fall and spring!
- 379 (Computational neuroscience): use python to analyze problems from neuroscience

Courses (next spring)

- 270 (AI)
- 335 (Databases)
- 331 (Security)
- 359 (Gray Codes): Algorithmic class, with a "puzzle"style focus
- 334 (Programming Languages): formal methods for analyzing and comparing programming languages.
 One of 361 or 334 required for majors

Midterm 2

- In class next Monday
- Come to your regular class time. Email me with accommodations
- 75 minutes
- Can bring a 2-sided "cheat sheet"; put whatever you want on it
 - Do not need the "dynamic programming recipe" I will give it to you
- Practice midterm; answers to practice midterm posted on Glow
- Q&A session in second half(?) of class on Thursday

Midterm 2 Proctoring

- I won't be here :(
 - I'll be at a workshop
- The TAs will proctor the midterm
- I'm hoping there won't be any questions/confusion.
 But if there are, they will help the best they can
- I'll also give them my phone and email so they can ask me about any questions that come up

Midterm 2 contents

- Recurrences (some "easy", some "hard")
- network flow/reduction question
- divide and conquer question
- 2 dynamic programming questions: one PS5-ish, one PS6-ish
- The practice midterm is quite similar to the actual midterm

Studying for Midterm 2

- What should you study/how should you study?
- What to do if you are not confident in some of the recent material?
 - Probably the majority of the class is in this boat

Studying for Midterm 2

- Go over recurrences and the recursion tree method. Go over divide and conquer: problems from class and on the problem sets
- Dynamic programming: I'd strongly recommend looking at the first few problems again if you haven't
 - Python code/video of knapsack example
 - Tips for DP handout
 - You can use outside resources for dynamic programming if you want
 - In my experience: eventually it just kind of clicks. Use what you can to get there

Actually Doing Midterm 2

- Show your knowledge!
- If you are stuck on a recurrence, write what you thought of
- AND fill in the other parts based on what you wrote!
- Bear in mind: these will not be as difficult as the most difficult ones we've seen

What's a Flow Network?

- A flow network is a directed graph G = (V, E) with a
 - A **source** is a vertex *s* with in degree 0
 - A **sink** is a vertex *t* with out degree 0
 - Each edge $e \in E$ has edge capacity c(e) > 0



Max-Flow Min-Cut Theorem

• Theorem. Given any flow network G, there exists an (s, t) -flow f and a (s, t)-cut (S, T) such that,

v(f) = c(S, T)

• Our algorithm will prove the theorem! (like with Gale-Shapley)

Ford Fulkerson: Idea

- Want to make "forward progress" while letting ourselves undo previous decisions if they're getting in our way
- Idea: keep track of where we can push flow
 - Can push more flow along an edge with remaining capacity
 - Can also push flow "back" along an edge that already has flow down it
- Need a way to systematically track these decisions

Residual Graph

- Given flow network G = (V, E, c) and a feasible flow f on G, the residual graph $G_f = (V, E_f, c_f)$ is defined as:
 - Vertices in G_f same as G
 - (Forward edge) For $e \in E$ with residual capacity c(e) f(e) > 0, create $e \in E_f$ with capacity c(e) f(e)
 - (Backward edge) For $e \in E$ with f(e) > 0, create $e_{\text{reverse}} \in E_f$ with capacity f(e)



reverse edge

Flow Algorithm Idea

- Now we have a residual graph that lets us make forward progress or push back existing flow
- We will look for $s \thicksim t$ paths in G_f rather than G
- Once we have a path, we will "augment" flow along it similar to greedy
 - find bottleneck capacity edge on the path and push that much flow through it in G_f
- When we translate this back to G, this means:
 - We increment existing flow on a forward edge
 - Or we decrement flow on a backward edge

Ford-Fulkerson Example



Ford-Fulkerson Example



Why is Max Flow = Min Cut?

- We already saw: an (S, T)-cut has c(S, T) = f(v) if and only if:
 - There is no flow into ${\boldsymbol{S}}$
 - All outgoing edges from S have flow = capacity



Analysis: Ford-Fulkerson

Analysis Outline

- Feasibility and value of flow:
 - Show that each time we update the flow, we still get a flow (it satisfies the constraints: no edge has more flow assigned than capacity, and flow in = flow out).
 - And that value of this flow increases each time by that amount
- Optimality:
 - Final value of flow is the maximum possible
- Running time:
 - How long does it take for the algorithm to terminate?
- Space:
 - How much total space are we using

Feasibility of Flow

- Claim. Let *f* be a flow in *G* and let *P* be an augmenting path in *G_f* with bottleneck capacity *b*. Let *f* ← AUGMENT(*f*, *P*), then *f* is a flow.
- **Proof**. Only need to verify constraints on the edges of P (since f' = f for other edges). Let $e = (u, v) \in P$
 - If *e* is a forward edge: f'(e) = f(e) + b

$$\leq f(e) + (c(e) - f(e)) = c(e)$$

• If *e* is a backward edge: f'(e) = f(e) - b

$$\geq f(e) - f(e) = 0$$

- Conservation constraint hold on any node in $u \in P$:
 - $f_{in}(u) = f_{out}(u)$, therefore $f'_{in}(u) = f'_{out}(u)$ for both cases

Value of Flow: Making Progress

• **Claim**. Let f be a feasible flow in G and let P be an augmenting path in G_f with bottleneck capacity b. Let

 $f' \leftarrow \text{AUGMENT}(f, P)$, then v(f') = v(f) + b.

- Proof.
 - First edge $e \in P$ must be out of s in G_f
 - (*P* is simple so never visits *s* again)
 - e must be a forward edge (P is a path from s to t)
 - Thus f(e) increases by b, increasing v(f) by $b \blacksquare$
- Note. Means the algorithm makes forward progress each time!

Optimality

Ford-Fulkerson Optimality

- **Recall**: If *f* is any feasible *s*-*t* flow and (S, T) is any *s*-*t* cut then $v(f) \le c(S, T)$.
- We will show that the Ford-Fulkerson algorithm terminates in a flow that achieves equality, that is,
- Ford-Fulkerson finds a flow f^* and there exists a cut (S^*, T^*) such that, $v(f^*) = c(S^*, T^*)$
- Proving this shows that it finds the maximum flow (and the min cut)
- This also proves the max-flow min-cut theorem

Ford-Fulkerson Optimality

- Lemma. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_{f} , then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- Proof.
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V S^*$
- Is this an *s*-*t* cut?
 - Yes: $s \in S^*, t \in T^*$, $S^* \cup T^* = V$ and $S^* \cap T^* = \emptyset$
- Consider an edge $e = u \rightarrow v$ with $u \in S^*, v \in T^*$, then what can we say about f(e)?

Recall: Ford-Fulkerson Example



Ford-Fulkerson Optimality

- Lemma. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_{f} , then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- Proof.
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V S^*$
- Is this an *s*-*t* cut?
 - Yes: $s \in S^*, t \in T^*$, $S^* \cup T^* = V$ and $S^* \cap T^* = \emptyset$
- Consider an edge $e = u \rightarrow v$ with $u \in S^*, v \in T^*$, then what can we say about f(e)?
 - f(e) = c(e)

Ford-Fulkerson Optimality

- Lemma. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_f , then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- Proof. (Cont.)
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V S^*$
- Is this an *s*-*t* cut?
 - Yes: $s \in S^*, t \in T^*$, $S^* \cup T^* = V$ and $S^* \cap T^* = \emptyset$
- Consider an edge $e = w \rightarrow v$ with $v \in S^*, w \in T^*$, then what can we say about f(e)?

Recall: Ford-Fulkerson Example



Ford-Fulkerson Optimality

- Lemma. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_{f} , then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- Proof. (Cont.)
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V S^*$
- Is this an *s*-*t* cut?
 - $s \in S, t \in T, S \cup T = V$ and $S \cap T = \emptyset$
- Consider an edge $e = w \rightarrow v$ with $v \in S^*, w \in T^*$, then what can we say about f(e)?
 - f(e) = 0

Ford-Fulkerson Optimality

- Lemma. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_{f} , then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- Proof. (Cont.)
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V S^*$
- Thus, all edges leaving S^{\ast} are completely saturated and all edges entering S^{\ast} have zero flow
- $v(f) = f_{out}(S^*) f_{in}(S^*) = f_{out}(S^*) = c(S^*, T^*) \blacksquare$
- **Corollary**. Ford-Fulkerson returns the maximum flow.

Ford-Fulkerson Algorithm Running Time

Ford-Fulkerson Performance

```
FORD-FULKERSON(G)
```

```
FOREACH edge e \in E : f(e) \leftarrow 0.

G_f \leftarrow residual network of G with respect to flow f.
```

```
WHILE (there exists an s\negt path P in G<sub>f</sub>)
```

```
f \leftarrow \operatorname{AUGMENT}(f, P).
```

Update G_f .

RETURN f.

- Does the algorithm terminate?
- Can we bound the number of iterations it does?
- Running time?

Ford-Fulkerson Running Time

- Recall we proved that with each call to AUGMENT, we increase value of flow by $b = \text{bottleneck}(G_f, P)$
- Assumption. Suppose all capacities c(e) are integers.
- Integrality invariant. Throughout Ford–Fulkerson, every edge flow f(e) and corresponding residual capacity is an integer. Thus $b \ge 1$.
- Let $C = \max_{u} c(s \rightarrow u)$ be the maximum capacity among edges leaving the source *s*.
- It must be that $v(f) \leq (n-1)C$
- Since, v(f) increases by $b \ge 1$ in each iteration, it follows that FF algorithm terminates in at most v(f) = O(nC) iterations.

Ford-Fulkerson Performance

```
FORD-FULKERSON(G)
```

```
FOREACH edge e \in E : f(e) \leftarrow 0.
```

 $G_f \leftarrow$ residual network of G with respect to flow f.

WHILE (there exists an s \neg t path *P* in *G*_{*f*})

 $f \leftarrow \text{AUGMENT}(f, P).$

Update G_f .

RETURN *f*.

- Operations in each iteration?
 - Find an augmenting path in G_f
 - Augment flow on path
 - Update G_{f}
Ford-Fulkerson Running Time

- **Claim.** Ford-Fulkerson can be implemented to run in time O(nmC), where $m = |E| \ge n 1$ and $C = \max_{u} c(s \to u)$.
- **Proof**. Time taken by each iteration:
- Finding an augmenting path in G_f
 - G_f has at most 2m edges, using BFS/DFS takes O(m + n) = O(m) time
- Augmenting flow in P takes O(n) time
- Given new flow, we can build new residual graph in O(m) time
- Overall, O(m) time per iteration

[Digging Deeper] Polynomial time?

- Does the Ford-Fulkerson algorithm run in time polynomial in the input size?
- Running time is O(nmC), where $C = \max c(s \rightarrow u)$
- What is the input size?
 - *n* vertices, *m* edges, *m* capacities
 - C represents the magnitude of the maximum capacity leaving the source node

U

• How many bits to represent *C*?

[Digging Deeper] Polynomial time?

- Question. Does the Ford-Fulkerson algorithm run in polynomialtime in the size of the input?
- Answer. No. if max capacity is C, the algorithm can take $\geq C$ iterations. Consider the following example.

each augmenting path

sends only 1 unit of flow

(# augmenting paths = 2C)



[Digger Deeper] Pseudo-Polynomial

- Input graph has n nodes and $m = O(n^2)$ edges, each with capacity c_e
- $C = \max_{e \in E} c(e)$, then c(e) takes $O(\log C)$ bits to represent
- Input size: $\Omega(n \log n + m \log n + m \log C)$ bits
- Running time: $O(nmC) = O(nm2^{\log C})$
 - Exponential in the size of C
- Such algorithms are called **pseudo-polynomial**
 - If the running time is polynomial in the magnitude but not size of an input parameter.
 - We saw this for knapsack as well!

Non-Integral Capacities?

- If the capacities are rational, can just multiply to obtain a large integer (massively increases running time)
- If capacities are irrational, Ford-Fulkerson can run infinitely!
 - Improvement at each step can be arbitrarily small
 - Can create bad instances where it doesn't terminate in finite steps

Network Flow: Beyond Ford Fulkerson

Edmond and Karp's Algorithms

- Ford and Fulkerson's algorithm does not specify which path in the residual graph to augment
- Poor worst-case behavior of the algorithm can be blamed on bad choices on augmenting path
- Better choice of augmenting paths. In 1970s, Jack Edmonds and Richard Karp published two natural rules for choosing augmenting paths (you don't need to remember this)
 - Widest path first: paths with largest bottleneck capacity
 - Running time $O(m^2 \log n \log C)$
 - Shortest (in terms of edges) augmenting paths first (Dinitz independently discovered & analyzed this rule)
 - Running time $O(m^2n)$

Progress on Network Flows

1951	$O(m n^2 C)$	Dantzig
1955	$O(m \ n \ C)$	Ford–Fulkerson
1970	$O(m n^2)$	Edmonds-Karp, Dinitz
1974	$O(n^3)$	Karzanov
1983	$O(m \ n \log n)$	Sleator-Tarjan
1985	$O(m \ n \log C)$	Gabow
1988	$O(m n \log (n^2 / m))$	Goldberg–Tarjan
1998	$O(m^{3/2}\log(n^2 / m)\log C)$	Goldberg-Rao
2013	O(m n)	Orlin
		Best among "combinatorial" approaches that push flow through the graph

Progress on Network Flows

- More recently: [Chen et al. 2022] achieve running time better than $O(m^{1+\epsilon})$ for any constant ϵ

• Specifically:
$$O\left(m^{1+1/\log^{1/168}m}\right)$$

- (don't worry about this running time)
- Not combinatorial: uses "interior point methods"
 - "Jumps" between solutions with drastically different, non-integral flow values
 - (Very intense math)



Progress on Network Flows

- Let's say that the best known: O(nm)
- For the purpose of this class, network flows can be solved in O(nm) time
- Some of these algorithms do REALLY well in "practice"; basically O(n + m)

Applications of Network Flow: Solving Problems by Reduction to Network Flows

- Data mining
- Bipartite matching
- Network reliability
- Image segmentation
- Baseball elimination
- Network connectivity
- Markov random fields
- Distributed computing
- Network intrusion detection
- Many, many, more.



liver and hepatic vascularization segmentation

- Network flows model a variety of optimization problems
- These optimization problems look complicated with lots of constraints; on the face of it seem to have nothing to do with networks or flows

Clients to base stations. Consider a set of mobile computing clients who each need to be connected to one of several possible base stations. We'll suppose there are n clients and k base stations; the position of each of these is specified by their (x, y) coordinates in the plane.

For each client, we wish to connect it to exactly one of the base stations, constrained in the following ways: a client can only be connected to a base station that is within distance r, and no more than L clients can be connected to any single base station. Design a polynomial time algorithm for the problem.

- Network flows model a variety of optimization problems
- These optimization problems look complicated with lots of constraints; on the face of it seem to have nothing to do with networks or flows

Survey design: Design survey asking n_1 consumers about n_2 products.

- Can survey consumer i about product j only if they own it.
- Ask consumer i between c_i and c'_i questions.
- Ask between p_j and p'_j consumers about product j.

Goal. Design a survey that meets these specs, if possible.

- Network flows model a variety of optimization problems
- These optimization problems look complicated with lots of constraints; on the face of it seem to have nothing to do with networks or flows

Airline scheduling: A very complicated scheduling problem but we can turn it into a simplified one:

Every day we have k flights and flight i leaves origin o_i at time s_i and arrives at destination d_i at time f_i . **Goal**. Minimize number of flight crews.

Reductions

- We will solve all these problems by reducing them to a network flow problem
- We'll focus on the concept of **problem reductions**

Problem Reductions

Reduction: Median Finding

- Here's a correct algorithm to find the median of an array
 - We saw a faster algorithm earlier in the class. But this one is reasonably fast, and does work

```
1. Sort A
2. If n is even:
    return A[n/2] + A[n/2 + 1]
3. else:
    return A[(n+1)/2]
```

Reduction: Median Finding

- With this algorithm I can make the following statement for any f(n):
- "If I can sort in O(f(n)) time, then I can find the median in O(f(n) + 1) time"

```
1. Sort A
2. If n is even:
    return A[n/2] + A[n/2 + 1]
3. else:
    return A[(n+1)/2]
```

Reduction

- Idea: solve a problem using an algorithm for a (slightly) *different* problem
- So: using sorting, we can do median-finding

Reduction 2: Negative Cycle Finding

• We saw this algorithm last week



Reduction 2: Negative Cycle Finding

- We saw this algorithm last week
- Simplified:

The answer for G' is the *same* as the answer for G

Construct G':

 Add a vertex s to G'
 Add an edge of weight 0 from s to every other vertex in G'

 Return if there exists a neg. cycle reachable from s in G'

Reduction 2: Negative Cycle Finding

If there exists an O(nm) time algorithm to find a negative cycle reachable from a specific vertex *s* in a graph, **then** there is an O(nm) time algorithm to find a negative cycle in any graph

1. Construct G':

- 1. Add a vertex s to ${\tt G'}$
- 2. Add an edge of weight 0 from s to every other vertex in G'
- 2. Return if there exists a neg. cycle reachable from s in G'

Reduction

- Idea: solve a problem using an algorithm for a (slightly) different problem
- So: using sorting, we can do median-finding; using negative cycle finding from a specific vertex, we can do negative cycle finding anywhere in the graph
- From now on: our reductions will always have this special property that we **immediately** return the answer
- From now on: we will only do reductions for "yes/no" problems

Steps of Reduction

First: transform the input to an input for the new problem

1. Construct G':

- 1. Add a vertex s to G'
- 2. Add an edge of weight 0 from s to every other vertex in G'
- 2. Return if there exists a neg. cycle reachable from s in G'

Steps of Reduction

Construct G':

 Add a vertex s to G'
 Add an edge of weight 0 from s to every other vertex in G'

 Return if there exists a neg. cycle reachable from s in G'

Then: run the algorithm once, and immediately return the solution

Anatomy of Problem Reductions

- At a high level, a problem X reduces to a problem Y if an algorithm for Y can be used to solve X
- **Reduction.** Convert an arbitrary instance x of X to a special instance y of Y such that there is a 1-1 correspondence between them

First: construct the input to the new problem



Proving Reductions Correct

- Goal: we want to show that we give a correct answer
- Usually we will do the following:
 - Prove that if the answer to the original problem input is "yes", then the answer to our transformed input is "yes"
 - If the answer to the original problem input is "no", then the answer to our transformed input is "no"

Proving Reductions Correct

- Prove that if the answer to the original problem input is "yes", then the answer to our transformed input is "yes"
- If the answer to the original problem input is "no", then the answer to our transformed input is "no"



Anatomy of Problem Reductions

Remember: must give the same answer!

- Claim. *x* satisfies a property iff *y* satisfies a corresponding property
- Proving a reduction is correct: prove both directions
- x has a property (e.g. has matching of size k) \implies y has a corresponding property (e.g. has a flow of value k)
- *x* does not have a property (e.g. does not have matching of size *k*) ⇒ *y* does not have a corresponding property (e.g. does not have a flow of value *k*)
- Or equivalently (and this is often easier to prove):
 - y has a property (e.g. has flow of value k) \implies x has a corresponding property (e.g. has a matching of value k)

Bipartite Matching

Review: Matching in Graphs

- **Definition.** Given an undirected graph G = (V, E), a matching $M \subseteq E$ of G is a subset of edges such that no two edges in M are incident on the same vertex.
 - In other words, each node appears in at most one edge in M



Review: Matching in Graphs

- **Definition.** Given an undirected graph G = (V, E), a matching $M \subseteq E$ of G is a subset of edges such that no two edges in M are incident on the same vertex.
 - In other words, each node appears in at most one edge in M
- A perfect matching matches all nodes in G
- **Max matching problem.** Find a matching of maximum cardinality for a given graph, that is, a matching with maximum number of edges
 - A perfect matching if it exists is maximum!

Review: Bipartite Graphs

- A graph is bipartite if its vertices can be partitioned into two subsets X, Y such that every edge e = (u, v) connects u ∈ X and v ∈ Y
- **Bipartite matching problem.** Given a bipartite graph $G = (X \cup Y, E)$ find a maximum matching.



Bipartite Matching Examples

- Models many assignment problems
 - A is a set of jobs, B as a set of machines
 - Edge (a_i, b_j) indicates where machine b_j is able to process job a_i
 - Perfect matching: way to assign each job to a machine that can process it, such that, each machine is assigned exactly one job
- Assigning customers to stores, students to dorms, etc
- Note. This is a different problem than the one we studied for Gale-Shapely matching!

Maximum & Perfect Matchings

- One of the oldest problems in combinatorial algorithms:
 - Determine the largest matching in a bipartite graph
- This doesn't seem like a network flow problem
 - But we will turn it into one
- Given arbitrary instance *x* of bipartite matching problem (*X*): *A*, *B* and edges *E* between *A* and *B*
- **Goal.** Create a special instance y of a max-flow problem (Y): flow network: G(V, E, c), source s, sink $t \in V$ s.t.
- 1-1 correspondence. There exists a matching of size k iff there is a flow of value k



- Create a new directed graph $G' = (A \cup B \cup \{s, t\}, E', c)$
- Add edge $s \to a$ to E' for all nodes $a \in A$
- Add edge $b \to t$ to E' for all nodes $b \in B$
- Direct edge $a \rightarrow b$ in E' if $(a, b) \in E$
- Set capacity of all edges in E' to 1



• Claim (\Rightarrow).

If the bipartite graph (A, B, E) has matching M of size k then flow-network G' has an integral flow of value k.



• Claim (\Rightarrow).

If the bipartite graph (A, B, E) has matching M of size k then flow-network G' has an integral flow of value k.

• Proof.

- For every edge $e = (a, b) \in M$, let f be the flow resulting from sending 1 unit of flow along the path $s \to a \to b \to t$
- *f* is a feasible flow (satisfies capacity and conservation) and integral
- v(f) = k

• Claim (\Leftarrow).

If flow-network G' has an integral flow of value k, then the bipartite graph (A, B, E) has matching M of size k.



G

• Claim (\Leftarrow).

If flow-network G' has an integral flow of value k, then the bipartite graph (A, B, E) has matching M of size k.

- Proof.
 - Let M = set of edges from A to B with f(e) = 1.
 - No two edges in *M* share a vertex, why?
 - |M| = k
 - $v(f) = f_{out}(S) f_{in}(S)$ for any (S, V S) cut
 - Let $S = A \cup \{s\}$

Summary & Running Time

- Proved matching of size k iff flow of value k
- Thus, max-flow iff max matching
- Running time of algorithm overall:
 - Running time of reduction + running time of solving the flow problem (dominates)

Can also use Orlin's algorithm

- What is running time of Ford–Fulkerson algorithm for a flow network with all unit capacities?
 - *O*(*nm*)
- Overall running time of finding max-cardinality bipartite matching: O(nm)

Disjoint Paths Problem

Disjoint Paths Problem

• **Definition.** Two paths are edge-disjoint if they do not have an edge in common.

• Edge-disjoint paths problem.

Given a directed graph with two nodes *s* and *t*, find the max number of edge-disjoint $s \sim t$ paths.



Towards Reduction

- Given: arbitrary instance x of disjoint paths problem (X): directed graph G, with source s and sink t
- Goal. create a special instance y of a max-flow problem (Y): flow network G'(V', E', c) with s', t' s.t.
- 1-1 correspondence. Input graph has k edge-disjoint paths iff flow network has a flow of value k



- Reduction. G': same as G with unit capacity assigned to every edge
- Claim [Correctness of reduction]. G has k edge disjoint $s \sim t$ paths iff G' has an integral flow of value k.
- Proof. (\Rightarrow)
- Set f(e) = 1 if e in some disjoint $s \sim t$, f(e) = 0 otherwise.
- Consider cut ($\{s\}, V \{s\}$), we get that v(f) = k
- Why is *f* feasible? Capacity constraint? Conservation?
 - We only ever send 1 unit of flow, so capacity is never violated
 - Say node u is part of $k' \leq k$ paths, then $f_{in}(u) = f_{out} = k'$

- Reduction. G': same as G with unit capacity assigned to every edge
- Claim [Correctness of reduction]. G has k edge disjoint $s \sim t$ paths iff G' has an integral flow of value k.
- Proof. (\Rightarrow)
- Set f(e) = 1 if e in some disjoint $s \sim t$, f(e) = 0 otherwise.
- Consider cut ($\{s\}, V \{s\}$), we get that v(f) = k
- We argued that f is a feasible flow with v(f) = k
- (\Leftarrow) Need to show: If G' has a flow of value k then there are k edge-disjoint $s \leadsto t$ paths in G

Correction of Reduction

- Claim. (⇐) If f is a 0-1 flow of value k in G', then the set of edges where f(e) = 1 contains a set of k edge-disjoint s ~ t paths in G.
- **Proof** [By induction on the # of edges k' with f(e) = 1]
- If k' = 0, no edges carry flow, nothing to prove
- IH: Assume claim holds for all flows that use < k' edges
- Consider an edge $s \to u$ with $f(s \to u) = 1$
- By flow conservation, there exists an edge $u \rightarrow v$ with $f(u \rightarrow v) = 1$, continue "tracing out the path" until
- Case (a) reach *t*, Case (b) visit a vertex *v* for a 2nd time

Correction of Reduction

- Case (a) We reach t, then we found a $s \sim t$ path P
 - f': Decrease the flow on edges of P by 1
 - v(f') = v(f) 1 = k 1
 - Number of edges that carry flow now < k': can apply IH and find k 1 other $s \sim t$ disjoint paths
- Case (b) visit a vertex v for a 2nd time: consider cycle C of edges visited btw 1st and 2nd visit to v
 - f': decrease flow values on edges in C to zero
 - v(f') = v(f) but # of edges in f' that carry flow < k', can now apply IH to get k edge disjoint paths

Summary & Running Time

- Proved k edge-disjoint paths iff flow of value k
- Thus, max-flow iff max # of edge-disjoint $s \sim t$ paths
- Running time of algorithm overall:
 - Running time of reduction + running time of solving the max-flow problem (dominates)
- What is running time of Ford–Fulkerson algorithm for a flow network with all unit capacities?
 - *O*(*nm*)
- Overall running time of finding max # of edge-disjoint s ~ t
 paths: O(nm)