Flow, P vs. NP, Reductions

CSE 417 Fall 22
Lecture 22

One More Example

A classic example

We’ll also be able to use the min-cut in addition to the flow!

Question: Can the Mariners still win* the division?

*or at least tie for first place.

And if they can’t, can you explain why.

Can The Mariners Win The Division?

It's late at night September 14, 1998.
You're working for the Seattle Times.
The Mariners won! But the Angels did too.
How do you frame the Mariners current situation in your postgame article?

Team	Wins (w)	Games Left
Angels	81	12
Rangers	80	12
Mariners	70	12
A's	69	12

MLB rules say all games will be played (if they end up mattering) so you can assume those will happen.

Can The Mariners Win The Division?

Team	Wins (w)	Games Left	Possible Wins (P)
Angels	81	12	93
Rangers	80	12	92
Mariners	70	12	82
A's	69	12	81

$P_{MARINERS} \geq w_i$ for all i, so the Mariners can win the division, right?

Well…No

The teams will play each other, here are the number of games to be played against each other.

g_{ij}	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-

Team	Wins (w)	Games Left	Possible Wins (P)
Angels	81	12	93
Rangers	80	12	92
Mariners	70	12	82
A's	69	12	81

Well…No

At least one of the Angels and Rangers is going to win at least 83 games

someone wins at least three of the five they play against each other.

The Mariners can only win 82 games.

Lessons

Comparing P_i to w_j is insufficient to tell if a team is eliminated.

The teams are interconnected by the games they play against each other.

Let's find a way to do this calculation…not by hand.

What do we need to assign?

Assignment

We need to assign who wins each of the remaining games.

Safe to assume the Mariners will win them all.

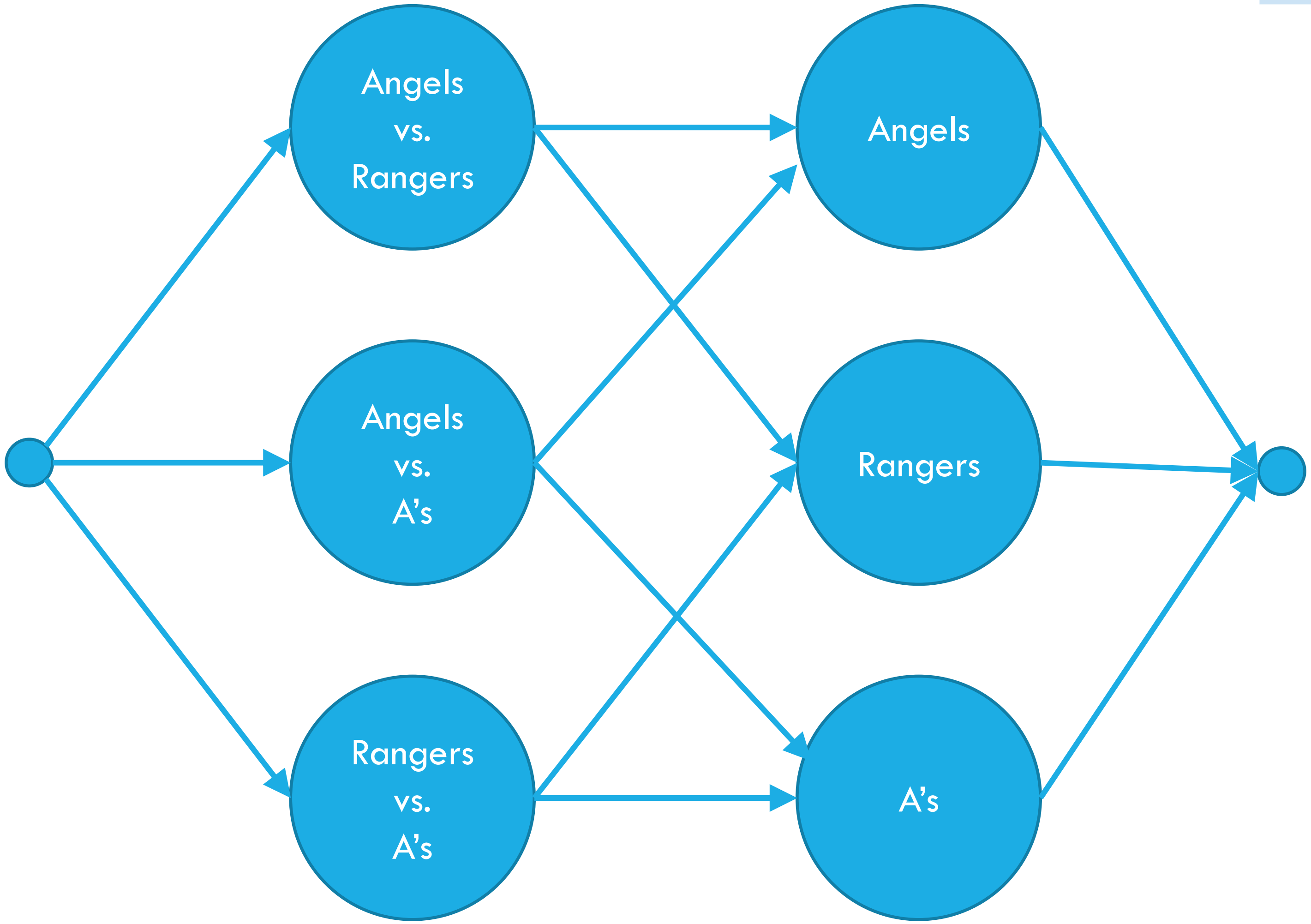
Just need to figure out the others.

One unit of flow represents one win.

Making a Network

	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-

Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81



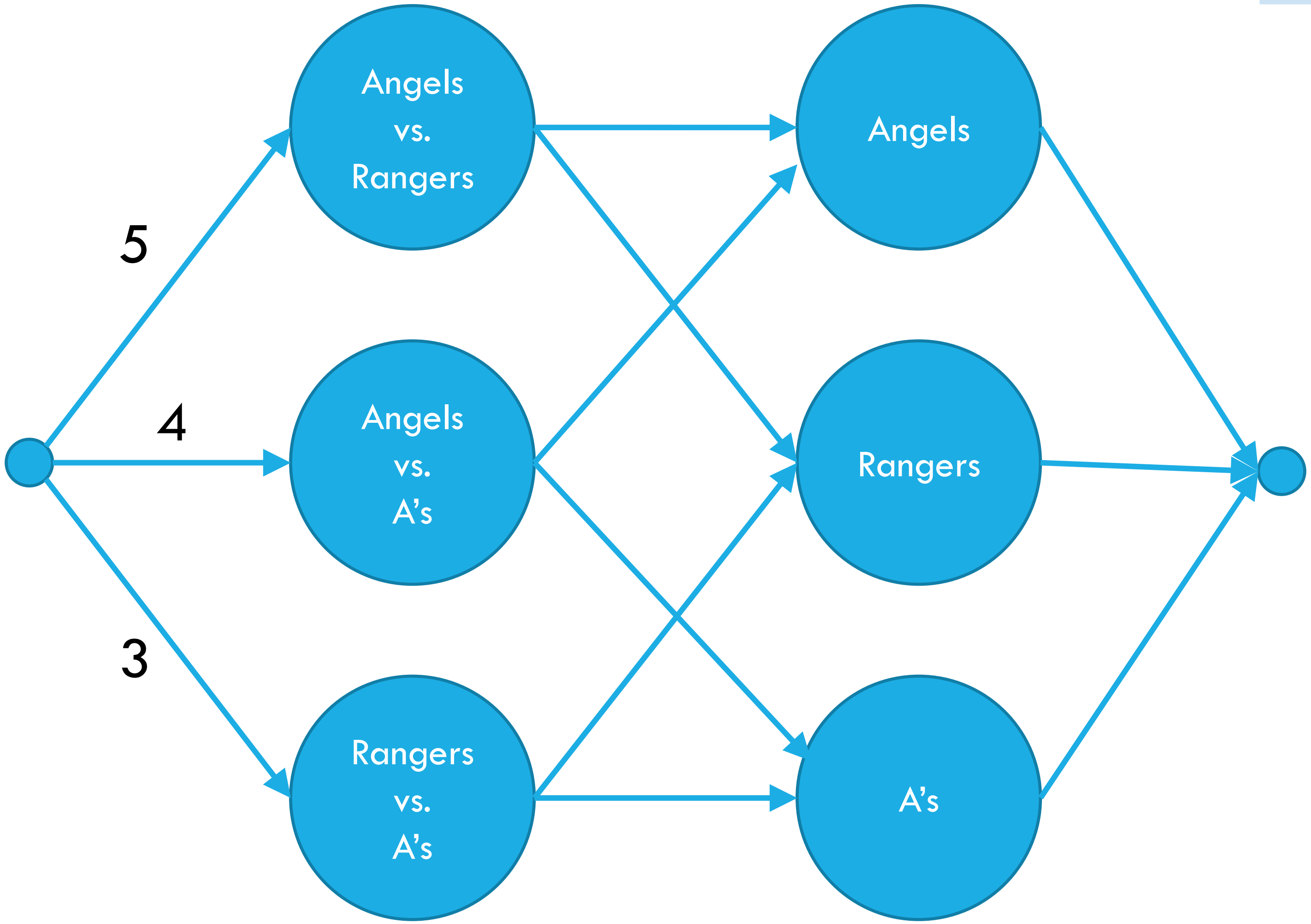
s, t on the ends

First layer is pairs of opponents (i.e. what game is being played)

Second layer is individual teams.

Making a Network

	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-

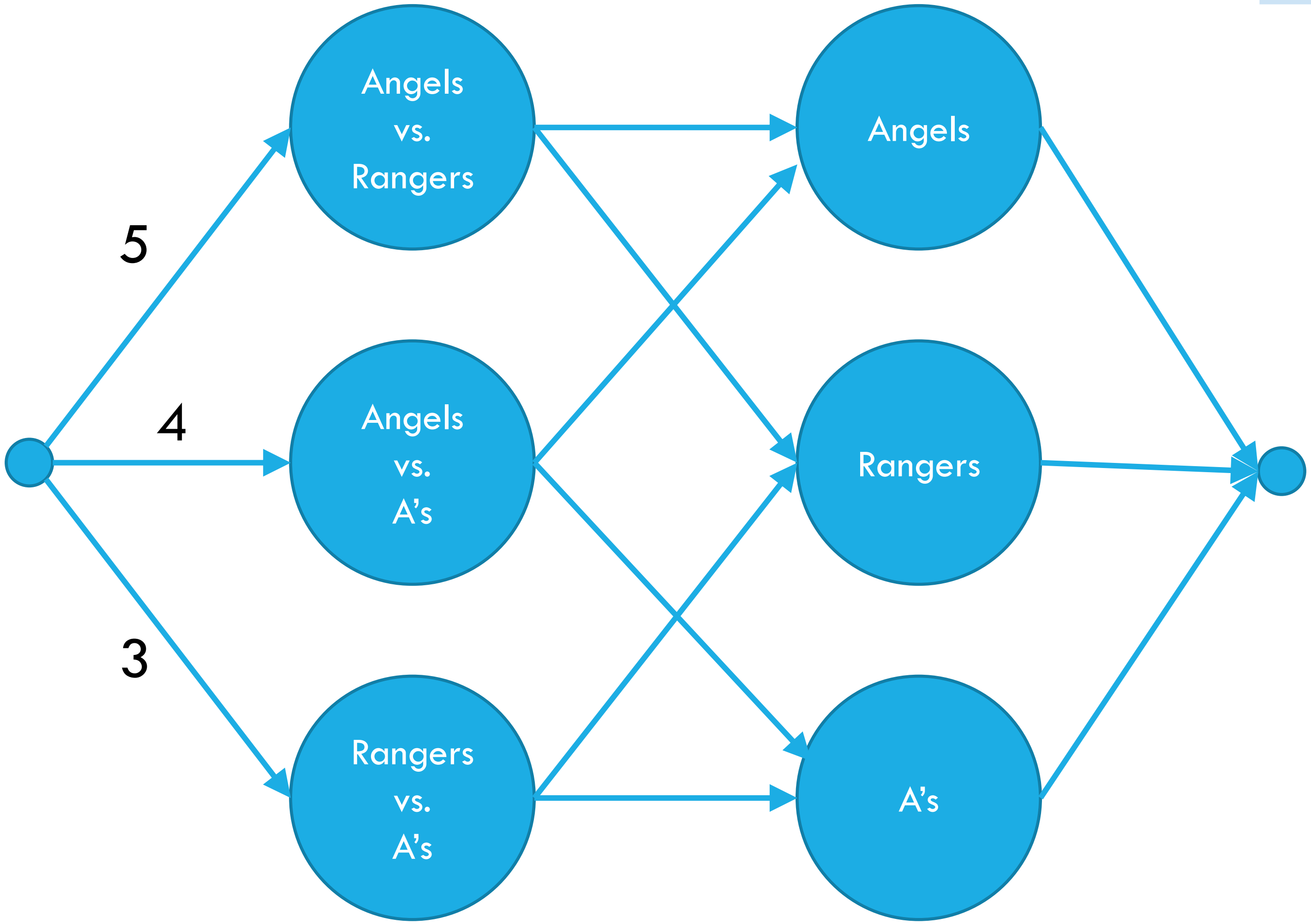


Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

Put number of games to be played from s to pairs

Making a Network

	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-



Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

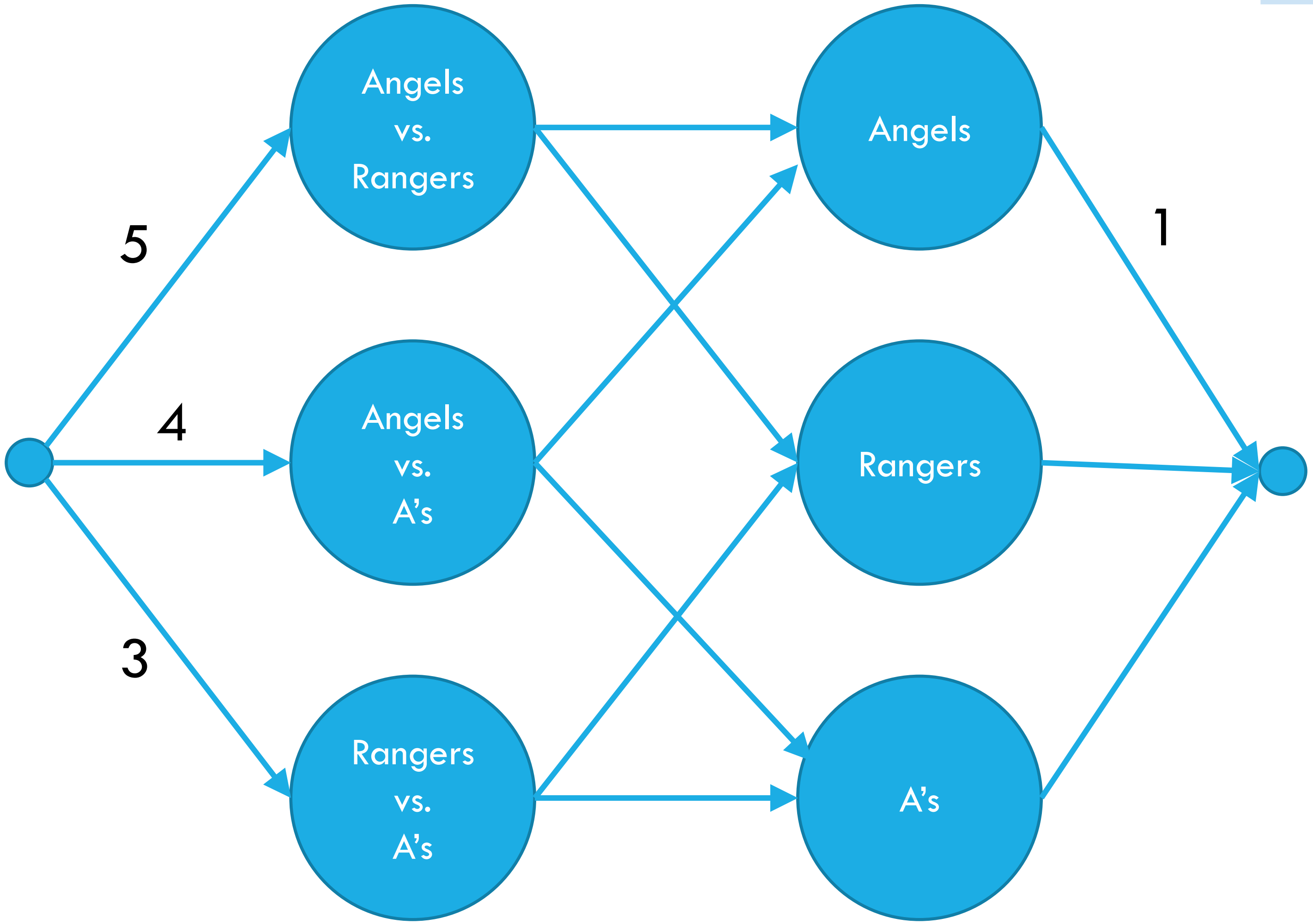
How do we make sure Mariners win? They'll end the season with 82 wins (current + games left).

How many more can each team win?

Mariners poss total – team current

Making a Network

Angels have 81 wins, 1 more is ok (total matches Mariners possible) 2 is not. Capacity is 1.



	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-

Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

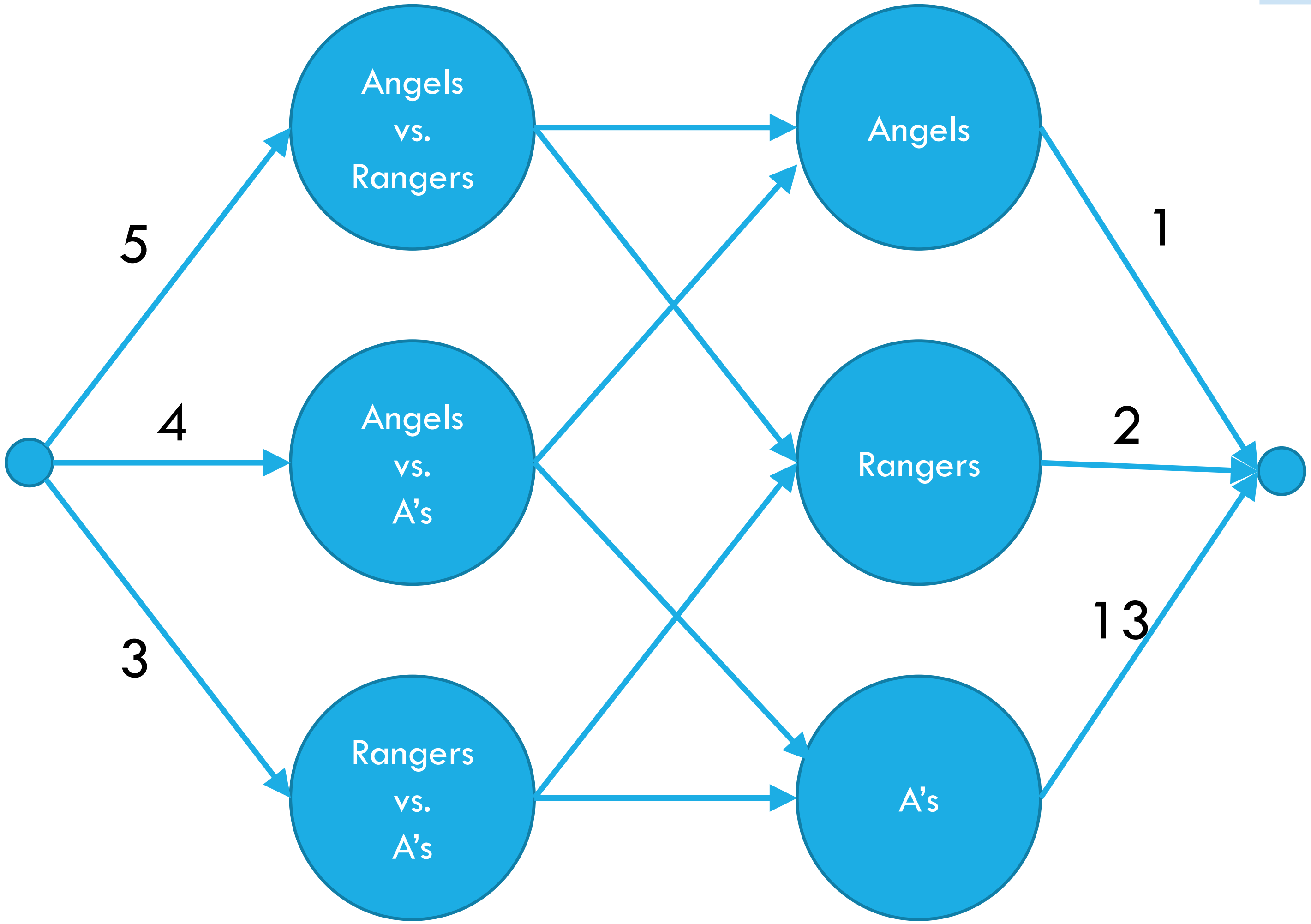
How do we make sure Mariners win? They'll end the season with 82 wins (current + games left).

How many more can each team win?

Mariners poss total – team current

Making a Network

	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-



Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

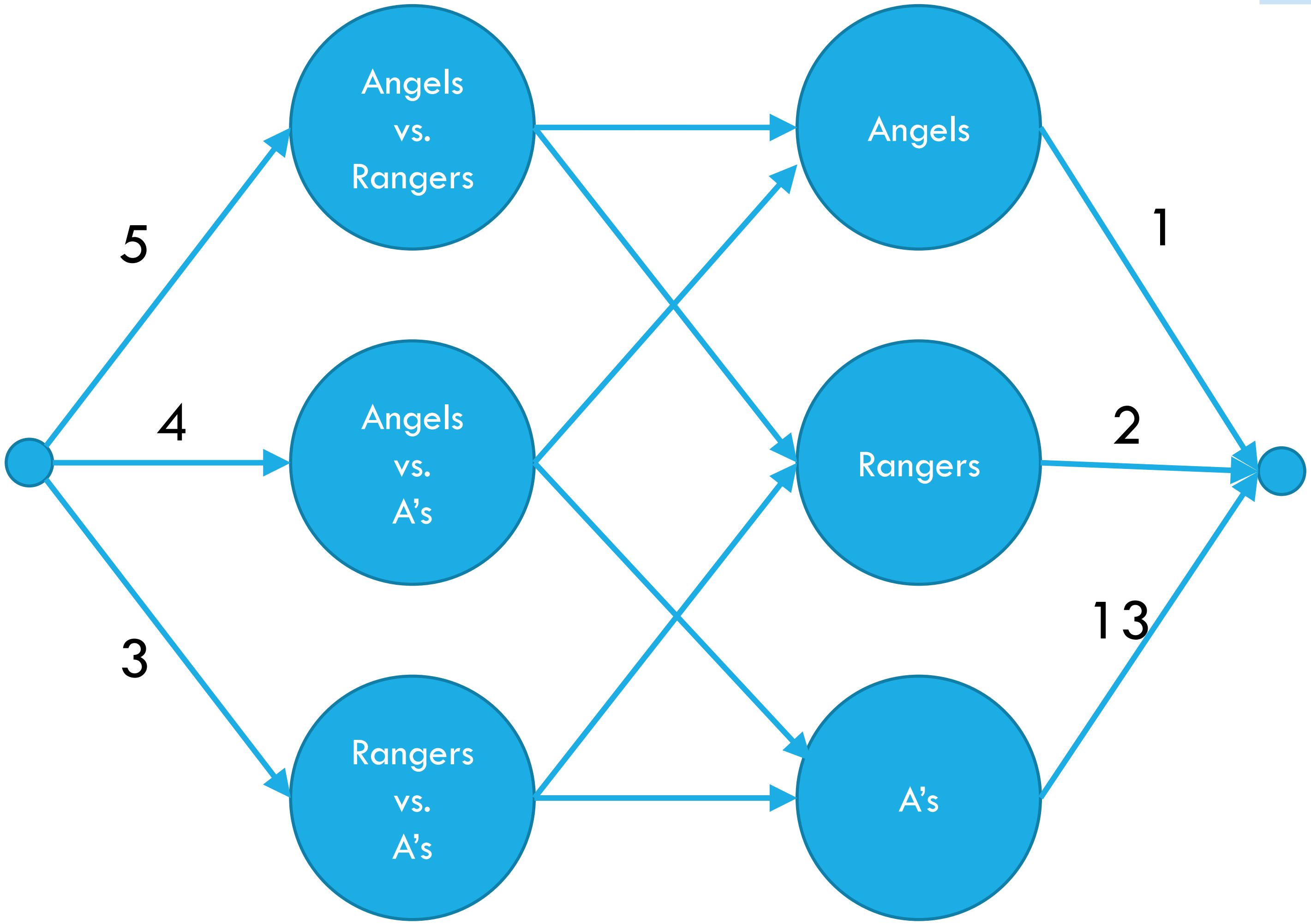
How do we make sure Mariners win? They'll end the season with 82 wins (current + games left).

How many more can each team win?

Mariners poss total – team current

Making a Network

	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-



Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

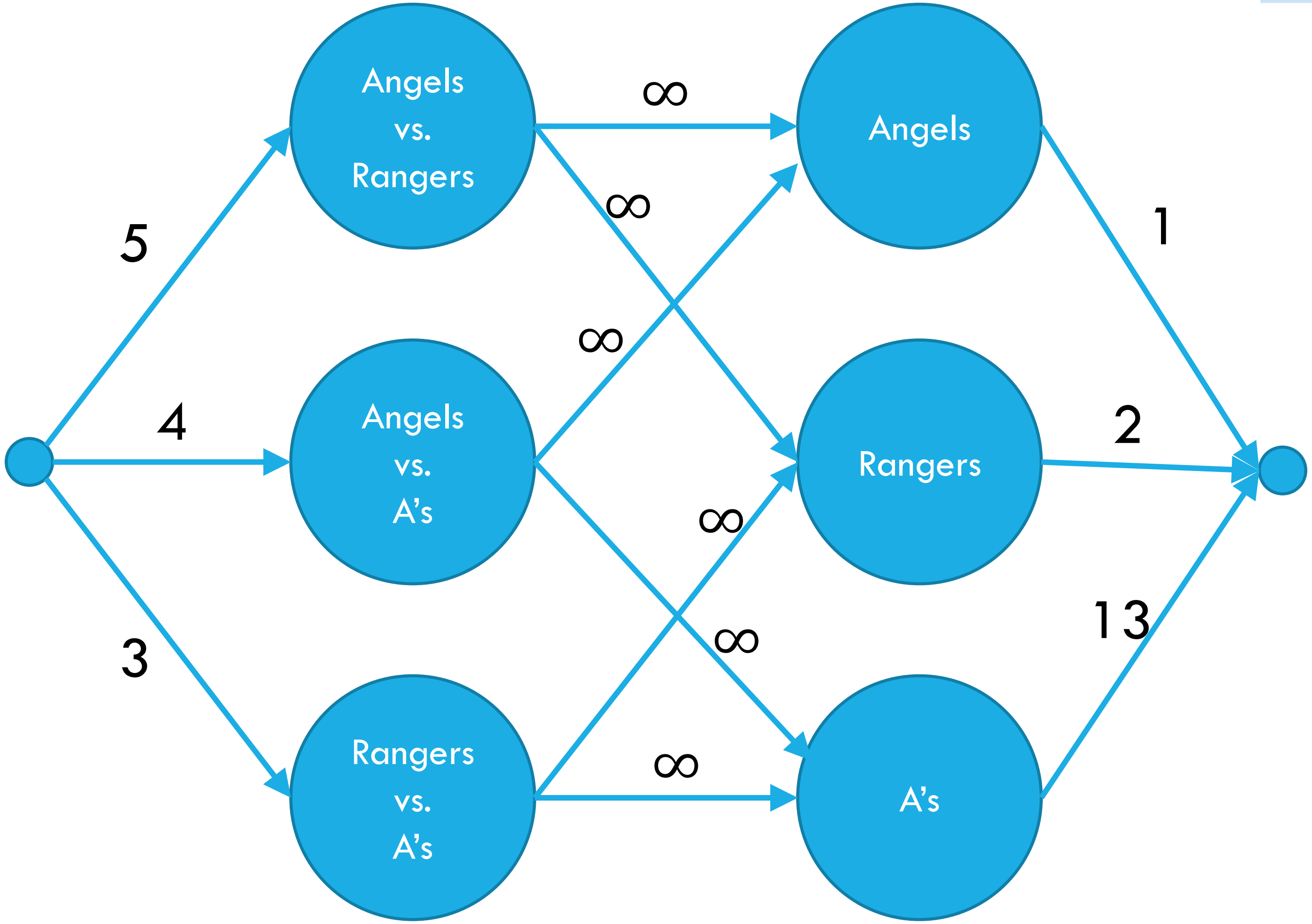
Edges in the middle?
Only to the two teams playing.

We've handled are constraints, can leave capacities at ∞.

Making a Network

	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-

Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

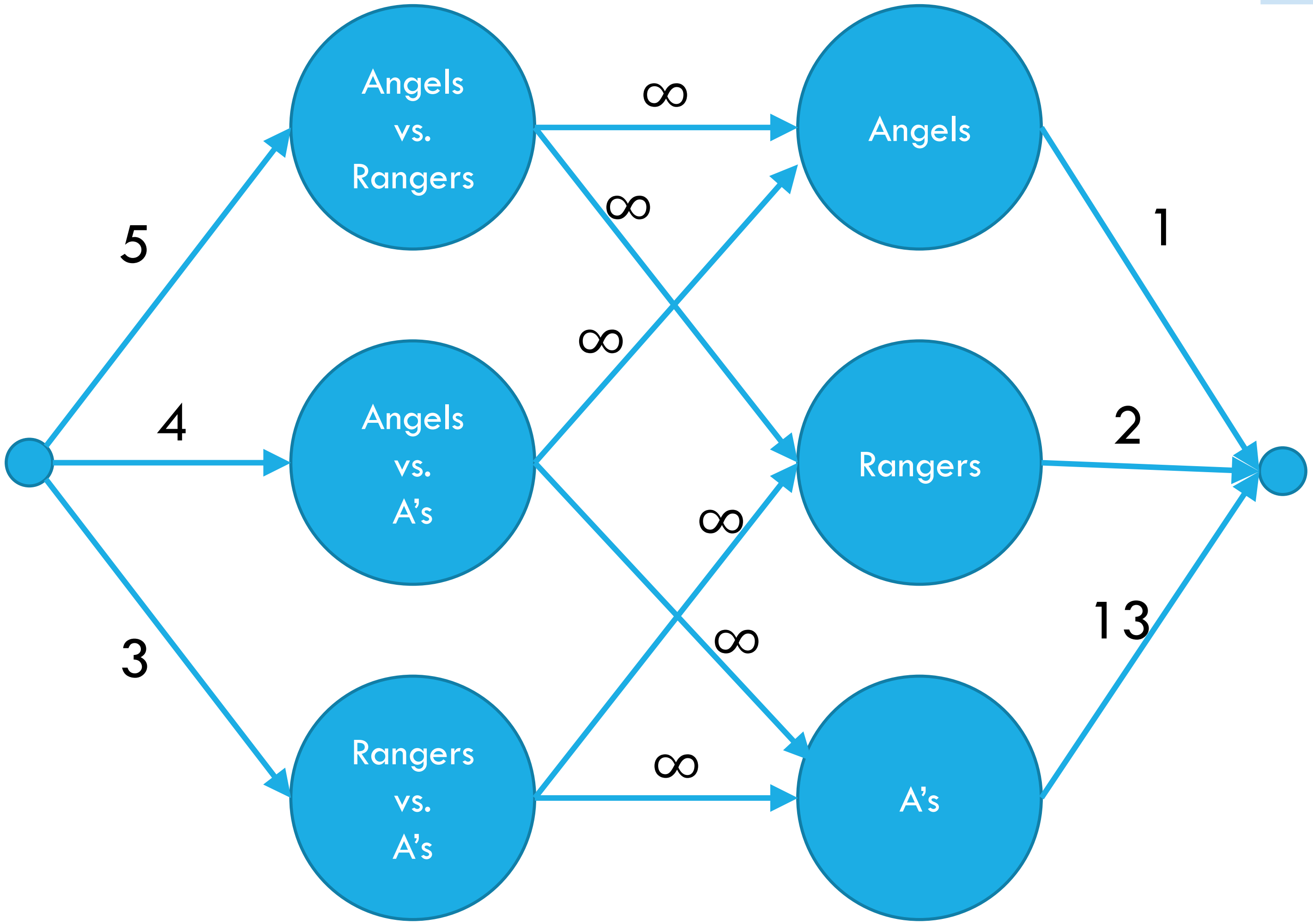


Edges in the middle?
Only to the two teams playing.

We've handled are constraints, can leave capacities at ∞.

Making a Network

	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-



Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

We're done!

Why are all the constraints met?

How many games are there to play? Equal to the capacities leaving s.

So if we have a flow of at least that value, we'll assign winners to all the games.

Why will the Mariners win with this assignment?

The capacity from team A to t ensures A will not end with more wins.

No "half-wins" or anything weird?

All capacities are integers, so we'll get an integer solution!

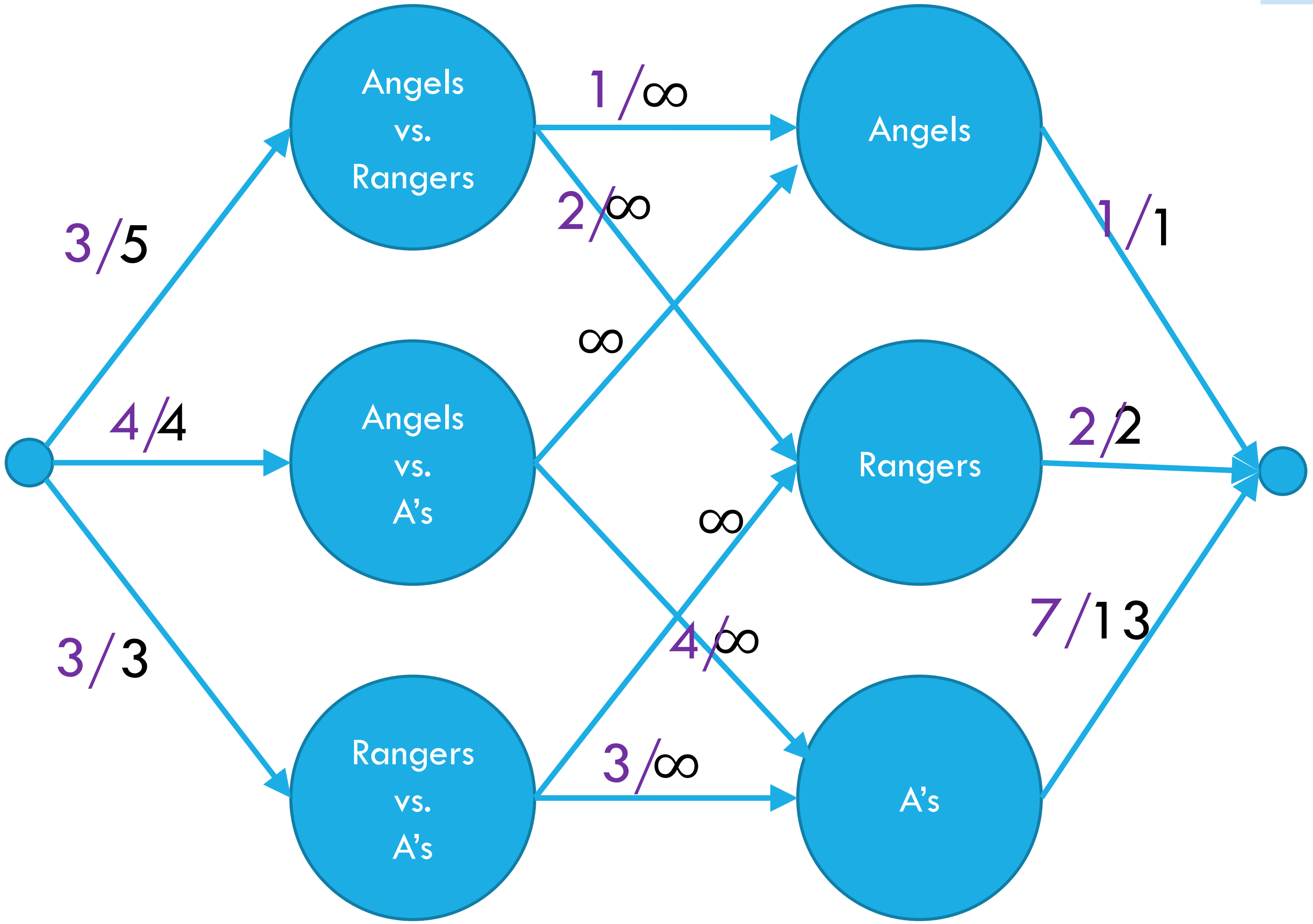
Interpreting the answer

If the max flow has value equal to number of games, we know how the Mariners can still win the division.

If the max flow is less than that, the Mariners can't win the division!

(if they could win the division, then there is a way that the remaining games could play out with the mariners having as many wins as anyone else, but then we could make a feasible flow by assigning a unit of flow for each winner).

Max Flow



	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-

Team	Wins (w)	Possible Wins (P)
Angels	81	93
Rangers	80	92
Mariners	70	82
A's	69	81

This is the maximum flow. What's the min-cut?

{s, Angels vs. Rangers, Angels, Rangers} is one side of the cut.

The Angels and Rangers were enough to prove that the Mariners couldn't win!

Generating Proof that you're eliminated

How do you describe to the general public that the Mariners are eliminated.

People are going to say "the Mariners can still win 82 games, no one has one 82, it's not over yet!"

Of the Angels and Rangers, they will win (combined) at least

$81 + 80 + 5$ games (Angels wins, Rangers wins, games to be played among these teams)

On average they win $\frac{166}{2} = 83$ games. That's more than 82. Someone is beating that average, and whoever that is the Mariners won't catch them.

In General

Find the max flow. If its value is the number of games remaining, great!

Mariners can still win.

If its value is less than that, find the min cut. The set of all teams reachable from s in the residual graph will show you **why** the Mariners are eliminated.

Takeaways

If you want to "assign" things, max-flow might be a good option.

If you say "at most" you can probably just make a capacity constraint

Once you can do an "exactly equal" or "at most" by checking the value of the max-flow.

Sometimes you want an extra layer or two if you have a multiple types of assignments.

Sometimes you can convert an "at least" in one group into an "at most" on another group.

Optional – Why is there always an explanation?

An Explanation Always Exists

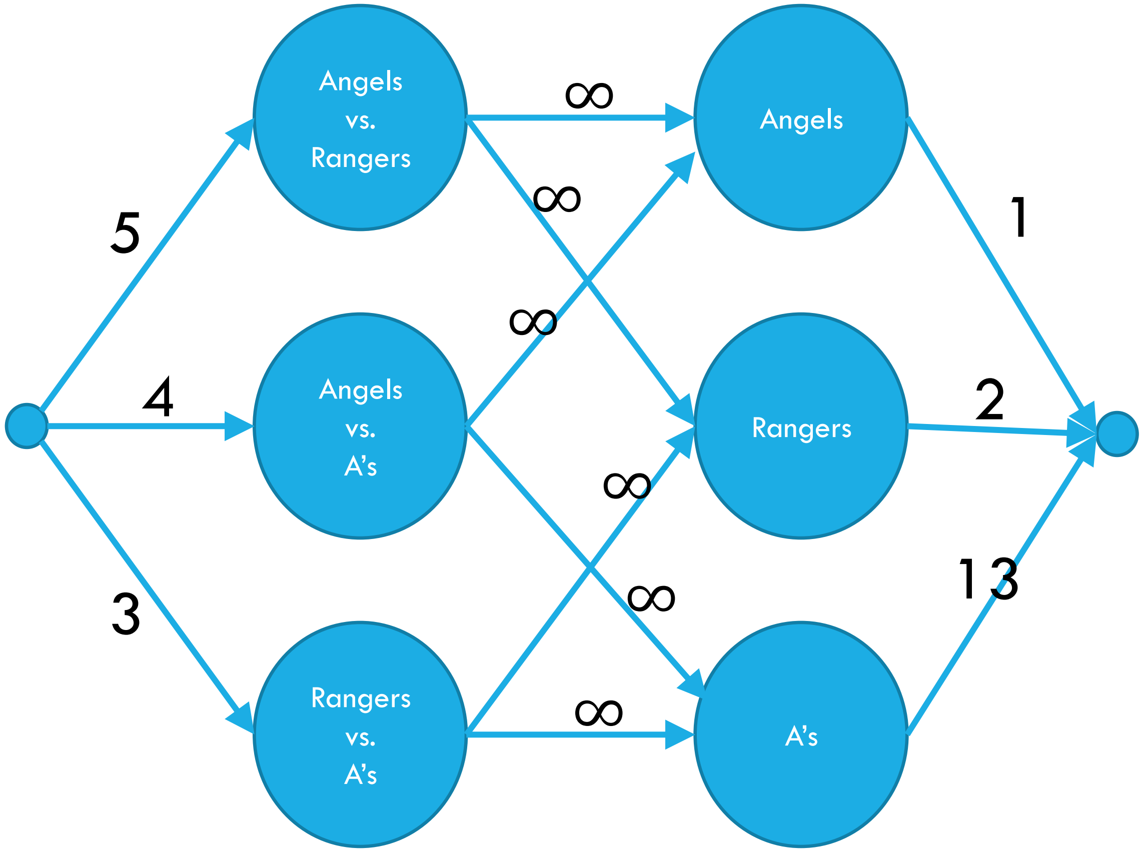
g_{ij} is games to be played between i and j
P is number of wins possible for Mariners
w_i is current number of wins for team i.

Let $(S, \bar{S})$ be a min-cut.
There's a lot of structure in the min-cut.
Let R be the set of teams whose vertices are reachable from s after the edges have been cut.

The capacity of the cut is
$\sum_{i \notin R \text{ or } j \notin R} g_{ij} + \sum_{i \in R} P - w_i$

And the capacity of the cut is less than $\sum_{i,j} g_{ij}$ (because that is a cut, and we can't have a flow of that value).

If R is a set of teams, let $a(R) = \frac{\sum_{i \in R} w_i + \sum_{i,j \in R} g_{i,j}}{|R|}$ the average number of games won be a team in R.



An Explanation Always Exists

g_{ij} is games to be played between i and j
P is number of wins possible for Mariners
w_i is current number of wins for team i.

$$\sum_{i \notin R \text{ or } j \notin R} g_{ij} + \sum_{i \in R} P - w_i < \sum_{i,j} g_{ij}$$

$$\sum_{i \in R} P - w_i < \sum_{i \in R, j \in R} g_{ij}$$

After subtracting pairs where at least one of i, j are not in R all that remains are pairs where both i, j are in R.

$$|R|P < \sum_{i \in R, j \in R} g_{ij} + \sum_{i \in R} w_i$$

Move w_i to the other side. P is a constant, so we just add $|R|$ copies of P.

$$P < \frac{\sum_{i \in R, j \in R} g_{ij} + \sum_{i \in R} w_i}{|R|}$$

That is, the average number of wins for a team in R (after all games are played) is strictly more than the possible number of wins for the Mariners.

Summary

To tell whether your favorite team is eliminated, you can run a max-flow computation on a graph with $O(n^2)$ vertices and $O(n^2)$ edges.

If your team is eliminated, there is a witness set of teams that must average more wins than is possible for your team.

P vs. NP and Reductions

How do we know a problem is hard?

At this point in the quarter, you’ve probably at least once been banging your head against a problem.

For so long that you began to thought “there’s no way there’s actually an efficient algorithm for this problem.”

That wasn’t true for any of the problems we gave you so far.

But it **is** true for some problems. At least we think it is.

The next few lectures are: what problems do we think there aren’t efficient algorithms for, and how do we tell?

Some definitions

A problem is a set of inputs and the correct outputs.

"Find a Minimum Spanning Tree" is a problem.

Input is a graph, output is the MST.

"Tell whether a graph is bipartite" is a problem.

Input is a graph, output is "yes" or "no"

"Find the 'maximum subarray sum'" is a problem.

Input is an array, output is the number that represents the largest sum of a subarray.

Some definitions

An instance is a single input to a problem.

A single, particular graph is an instance of the MST problem

A single, particular graph is an instance of the bipartite-checking problem.

A single, particular array is an instance of the maximum subarray sum problem.

Decision Problems

Our goal is to divide problems into solvable/not solvable.
We're going to talk about decision problems.

Problems that have a "yes" or "no" answer. (a correct algorithm has a Boolean return type)

Why?

Theory reasons (ask me later).

But it's not too bad

most problems can be rephrased as very similar decision problems.

E.g. instead of "find the shortest path from s to t" ask
Is there a path from s to t of length at most k?

“Ranking” difficulty of problems

We’ll use “reductions” to tell whether one problem is harder than another.

Reduction (informally)
Using an algorithm for Problem B to solve Problem A.

In that case, we’ll say “A reduces to B”

In difficulty (for us, as algorithm designers), $A \leq B$

ANY algorithm for B solves A. A is no harder to solve than B.

A might be easier (maybe there’s another way to solve A without B) or they might be about the same (maybe $B \leq A$ too!)

Reductions

Even less formally:

Calling a library.

If you wrote a library to solve problem B

And your algorithm for A calls that library,

Then $A \leq B$ (A reduces to B).

Does the name feel backwards?

When we think of B as already solved, the name makes sense. We reduced our job to someone else's

When we think of neither A nor B as solved then $A \leq B$ notation makes sense.

Baseball Elimination

g_{ij}	Angels	Rangers	Mariners	A's
Angels	-	5	3	4
Rangers	5	-	4	3
Mariners	3	4	-	5
A's	4	3	5	-

Team	w	g	P
Angels	81	12	93
Rangers	80	12	92
Mariners	70	12	82
A's	69	12	81

Transform Input

Max-Flow Algorithm

Transform Output

More Previous Examples

On Homework 1, you reduced “stable matchings with unacceptable pairs and unequal numbers of agents“ to “[standard] stable matching”

On Homework 6, you (might have) reduced “finding a labeling with the minimum number of 0s on an unlabeled tree” to “2-coloring”

Today, we reduced “telling whether the Mariners could win the division” to “finding a maximum flow”

A Formal Definition

We need a formal definition of a reduction.

We will say "A reduces to B in polynomial time" (or "A is polynomial time reducible to B" or "A reduces to B" or "$A \leq B$") if:

There is an algorithm to solve problem A, which, if given access to a polynomial-time algorithm for problem B, runs in polynomial time overall (**including** the library's running time!!!).

Let's Do A Reduction

4 steps for reducing (decision problem) A to problem B.

1. Describe the reduction itself (i.e. the algorithm, with a call to a library for problem B)

2. Make sure the running time would be polynomial (usually skip writing out this step).

3. Argue that if the correct answer (to the instance for A) is YES, then our algorithm answers YES.

4. Argue that if the correct answer (to the instance for A) is NO, then our algorithm answers NO.

Reduce 2-coloring to 3-coloring

What's 3-coloring?

3-coloring

Input: Undirected Graph G

Output: `True` if the vertices of G can be labeled with red,green, and blue so that no edge has both of its endpoints colored the same color. `False` if it cannot.

Reduce 2-coloring to 3-coloring

Given a graph G, figure out whether it can be 2-colored, by using an algorithm that figures out whether it can be 3-colored.

Usual outline:

Transform G into an input for the 3-coloring algorithm

Run the 3-coloring algorithm

Transform the answer from the 3-coloring algorithm into the answer for G for 2-coloring

Reduction

If we just ask the 3-coloring algorithm about G, it might use 3 colors…we can't get it to use just 2…

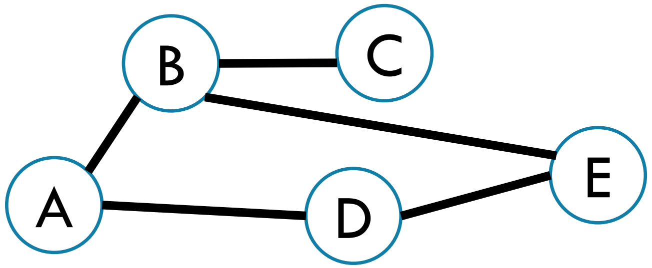
…unless…

Unless we force it not to, by adding extra vertices that **force** the 3-coloring algorithm to "use up" one color on the extra vertices, leaving only two colors for the "real" vertices.

Add an extra vertex v, and attach it to **everything** in G.

Reduction

```
2ColorCheck(Graph G)
    Let H be a copy of G
    Add a vertex to H, attach it to all other
      vertices.
    Bool answer = 3ColorCheck(H)
    return answer //don't need any modification!
```



Transform Input

3ColorCheck algorithm

Transform Output

Correctness?

```
2ColorCheck(Graph G)
    Let H be a copy of G
    Add a vertex to H, attach it to all
      other vertices.
    Bool answer = 3ColorCheck(H)
    return answer
```

TWO statements to prove: (“two directions”)

If the correct answer for G is YES, then we say YES

If the correct answer for G is NO, then we say NO

Correctness?

```
2ColorCheck(Graph G)
    Let H be a copy of G
    Add a vertex to H, attach it to all
      other vertices.
    Bool answer = 3ColorCheck(H)
    return answer
```

TWO statements to prove: (“two directions”)

If the correct answer for G is YES, then we say YES

If G is 2-colorable, then H will be 3-colorable – you can extend a 2-color labeling of G to 3 colors on H by making the new vertex the new color. All the edges in G have different colors (because we started with a 2-coloring) and any added edge has different endpoints (because v is a new color) so 3ColorCheck returns True and we return True!

If the correct answer for G is NO, then we say NO

Correctness?

```
2ColorCheck(Graph G)
     Let H be a copy of G
     Add a vertex to H, attach it to all
       other vertices.
     Bool answer = 3ColorCheck(H)
     return answer
```

TWO statements to prove: (“two directions”)

If the correct answer for G is YES, then we say YES

The new vertex can be a new color!

If the correct answer for G is NO, then we say NO

So we **can’t 2-color G**. That’s going to be hard to work with.
Take the contrapositive!!

Correctness?

```
2ColorCheck(Graph G)
    Let H be a copy of G
    Add a vertex to H, attach it to all
      other vertices.
    Bool answer = 3ColorCheck(H)
    return answer
```

TWO statements to prove: ("two directions")

If the correct answer for G is YES, then we say YES

The new vertex can be a new color!

If the correct answer for G is NO, then we say NO

We want to show instead: If we say YES, then the correct answer is YES.
If we say YES, then 3ColorCheck(H) must have returned YES, what does a 3-coloring of H look like? The added vertex must be a different color than all the other vertices (otherwise it's not a valid coloring – there's an edge between the added vertex and all others). So deleting the added vertex we get a 2-coloring of G. So the right answer is YES!!

Correctness

Two DIFFERENT statements

Correct Answer YES → Our algorithm says YES

If G is 2-colorable, then H will be 3-colorable – you can extend a 2-color labeling of G to 3 colors on H by making the new vertex the new color. All the edges in G have different colors (because we started with a 2-coloring) and any added edge has different endpoints (because v is a new color) so 3ColorCheck returns True and we return True!

Our algorithm says YES → Correct Answer YES

We want to show instead: If we say YES, then the correct answer is YES.

If we say YES, then 3ColorCheck(H) must have returned YES, what does a 3-coloring of H look like? The added vertex must be a different color than all the other vertices (otherwise it's not a valid coloring – there's an edge between the added vertex and all others). So deleting the added vertex we get a 2-coloring of G. So the right answer is YES!!

Write two separate arguments

You need to show **both** “we won’t get any false positives” and “we won’t get any false negatives.”

To make sure you handle both directions, I **strongly** recommend:

1. Always do two separate proofs (don’t try to prove both directions at once, don’t refer back to the prior proof and say “for the same reason”).
2. Don’t use contradiction (it’s easy to start from the wrong spot and accidentally prove the same direction twice without realizing it).
3. Follow one of the four pairs on the next slide (don’t accidentally take a contrapositive wrong)

Argument Outlines

Most common

If the correct answer is YES, then our algorithm says YES.
And If our algorithm says YES, then the correct answer is YES

Less common but sometimes:

If our algorithm says NO, then the correct answer is NO.
And If our algorithm says YES, then the correct answer is YES

OR

If the correct answer is YES, then our algorithm says YES.
And If the correct answer is NO, then our algorithm says NO

Works, but rarely the best:

If our algorithm says NO, then the correct answer is NO.
And If the correct answer is NO, then our algorithm says NO

Back to Problem Ranking

P (can be solved efficiently)

P (stands for "Polynomial")

The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k.

The decision version of all problems we've solved in this class are in P.

P is an example of a "complexity class"
A set of problems that can be solved under some limitations (e.g. with some amount of memory or in some amount of time).

Problems go in complexity classes. Not algorithms.
We're comparing problem difficulty, not algorithm quality.

NP

Our second set of problems have the property that "I'll know it when I see it"
We're looking for **something,** and if someone shows it to me, we can recognize it quickly (it just might be hard to find)

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that for every YES-instance, there is a certificate for that instance which can be verified in polynomial time.

A "verifier" takes in: an instance of the NP problem, and a "proof"
And returns "true" if it received a valid proof that the instance is a YES instance, and "false" if it did not receive a valid proof

NP problems have "verifiers" that run in polynomial time.
Do they have **solvers** that run in polynomial time? The definition doesn't say.

NP

Our second set of problems have the property that “I’ll know it when I see it”
We’re looking for **something,** and if someone shows it to me, we can recognize it quickly (it just might be hard to find)

NP (stands for “nondeterministic polynomial”)

The set of all decision problems such that for every YES-instance, there is a certificate for that instance which can be verified in polynomial time.

If you have a “YES” instance, a little birdy can magically find you this certificate-thing, and you’ll say “Oh yeah, that’s totally a yes instance!”

What if it’s a NO instance? No guarantee.

NP

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

Decision Problems such that:
If the answer is YES, you can prove the answer is yes by
  Being given a "proof" or a "certificate"
  Verifying that certificate in polynomial time.
What certificate would be convenient for short paths?
  The path itself. Easy to check the path is really in the graph and really short.

Light Spanning Tree:
Is there a spanning tree of graph G of weight at most k?

The spanning tree itself.
Verify by checking it really connects every vertex and its weight.

3-Coloring:
Can you color vertices of a graph red, blue, and green so every edge has differently colored endpoints?

The coloring.
Verify by checking each edge.

Large flow:
Is there a flow from s to t in G of value at least k?

The flow itself.
Verify the capacity constraints, conservation, and that flow value at least k.

NP

Our second set of problems have the property that “I’ll know it when I see it”
We’re looking for **something**, and if someone shows it to me, we can recognize it quickly (it just might be hard to find)

NP (stands for “nondeterministic polynomial”)

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

It’s a common misconception that NP stands for “not polynomial”

Never, ever, ever, ever say “NP” stands for “not polynomial”
Please
Every time someone says that, a theoretical computer scientist sheds a single tear
(That theoretical computer scientist is me)

P vs. NP

P vs. NP
Are P and NP the same complexity class? **That is, can every problem that can be** verified **in polynomial time also be** solved **in polynomial time.**

If you'll know it when you see it, can you also search to find it efficiently?

No one knows the answer to this question.

In fact, it's the biggest unsolved question in Computer Science.

Some New Problems

Here are some new problems. Are they in NP?

If they're in NP, what is the "certificate" when the answer is yes?

COMPOSITE – given an integer n is it composite (i.e. not prime)?

MAX-FLOW – find a maximum flow in a graph.

VERTEX-COVER – given a graph G and an integer k, does G have a vertex cover of size at most k?

NON-3-Color – given a graph G, is it not 3-colorable?

Some New Problems

COMPOSITE – given an integer n is it composite (i.e. not prime)?

In NP (certificate is factors).

MAX-FLOW – find a maximum flow in a graph.

Not in NP (not a decision problem)

VERTEX-COVER – given a graph G and an integer k, does G have a vertex cover of size at most k?

In NP (certificate is cover)

NON-3-Color – given a graph G, is it not 3-colorable?

Not known to be in NP.

Hard Problems

Let’s say we want to figure out if every problem in NP can actually be solved efficiently.

We might want to start with a really hard problem in NP.

What is the hardest problem in NP?

What does it mean to be a hard problem?

Reductions are a good definition:

If A reduces to B then “$A \leq B$” (in terms of difficulty)

- Once you have an algorithm for B, you have one for A automatically from the reduction!

NP-hardness

NP-hard

The problem B is NP-hard if
for all problems A in NP, A reduces to B.

An NP-hard problem is “hard enough” to design algorithms for that if you write an efficient algorithm for it, you’ve (by accident) designed an algorithm that works for every problem in NP.

What does it look like? Let A be in NP, and let B be the NP-hard problem you solved, on an input to A, “run the reduction” and plug in your actual algorithm for B!

NP-Completeness

NP-Complete

The problem B is NP-complete if B is in NP and B is NP-hard

An NP-complete problem is a “hardest” problem in NP.

If you have an algorithm to solve an NP-complete problem, you have an algorithm for **every** problem in NP.

An NP-complete problem is a **universal language** for encoding “I’ll know it when I see it” problems.

Why is being NP-hard/-complete interesting?

Let B be an NP-hard problem. Suppose you found a polynomial time algorithm for B. Why is that interesting?

You now have for free a polynomial time algorithm for **every** problem in NP. (if A is in NP, then $A \leq B$. So plug in your algorithm for B!)

So $P = NP$. (if you find a polynomial time algorithm for an NP-hard problem).

On the other hand, if any problem in NP is not in P (any doesn't have a polynomial time algorithm), then no NP-complete problem is in P.

NP-Completeness

An NP-complete problem does exist!

Cook-Levin Theorem (1971)

3-SAT is NP-complete

Theorem 1: If a set S of strings is accepted by some nondeterministic Turing machine within polynomial time, then S is P-reducible to {DNF tautologies}.

This sentence (and the proof of it) won Cook the Turing Award.

What’s 3-SAT?

Input: A list of Boolean variables $x_1, \ldots, x_n$

A list of constraints, all of which must be met.

Each constraint is of the form:

```
( (xi == <T,F>) || (xj == <T,F>) || (xk == <T/F>) )
```

ORed together, always exactly three variables, you can choose T/F independently for each.

Output: true if there is a setting of the variables where all constraints are met, false otherwise.

Why is it called 3-SAT? 3 because you have 3 variables per constraint
SAT is short for “satisfiability” can you satisfy all of the constraints?

More Starting Points

We have one NP-hard problem (3-SAT). It'd be nice if we had more…

I'm just going to give us more (if you're interested in proving these NP-complete, many are here)

3-coloring is NP-complete.

Hamiltonian Path (given a directed graph, is there a path that visits every vertex exactly once?) is NP-complete.

More Reduction Facts

I have a problem

My problem C is hard.

So hard, it's probably NP-hard. How do I show it?

What does it mean to be NP-hard?

We need to be able to reduce any problem A to C.

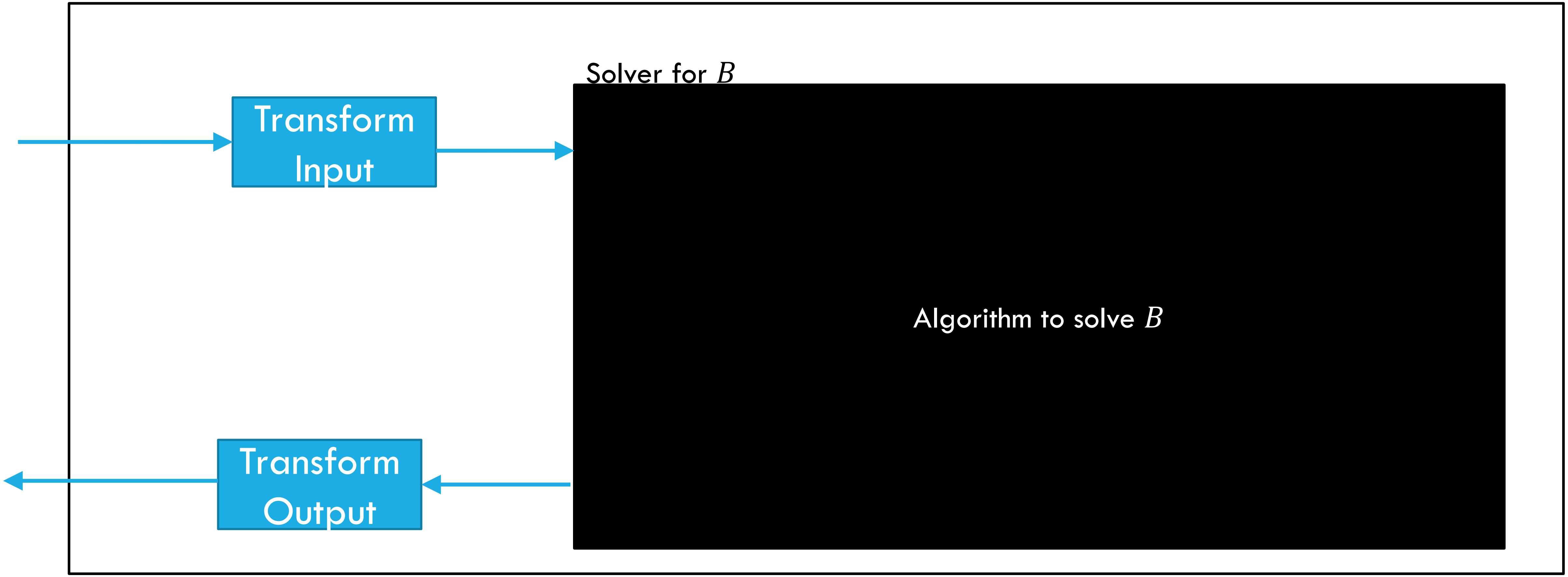
Let's choose B to be a **known** NP-hard problem. Since B **is known** to be NP-hard, $A \leq B$ for every possible A. So if **we show** $B \leq C$ too then $A \leq B \leq C \rightarrow A \leq C$ so every NP problem reduces to C!

$$A \leq B \leq C \rightarrow A \leq C$$

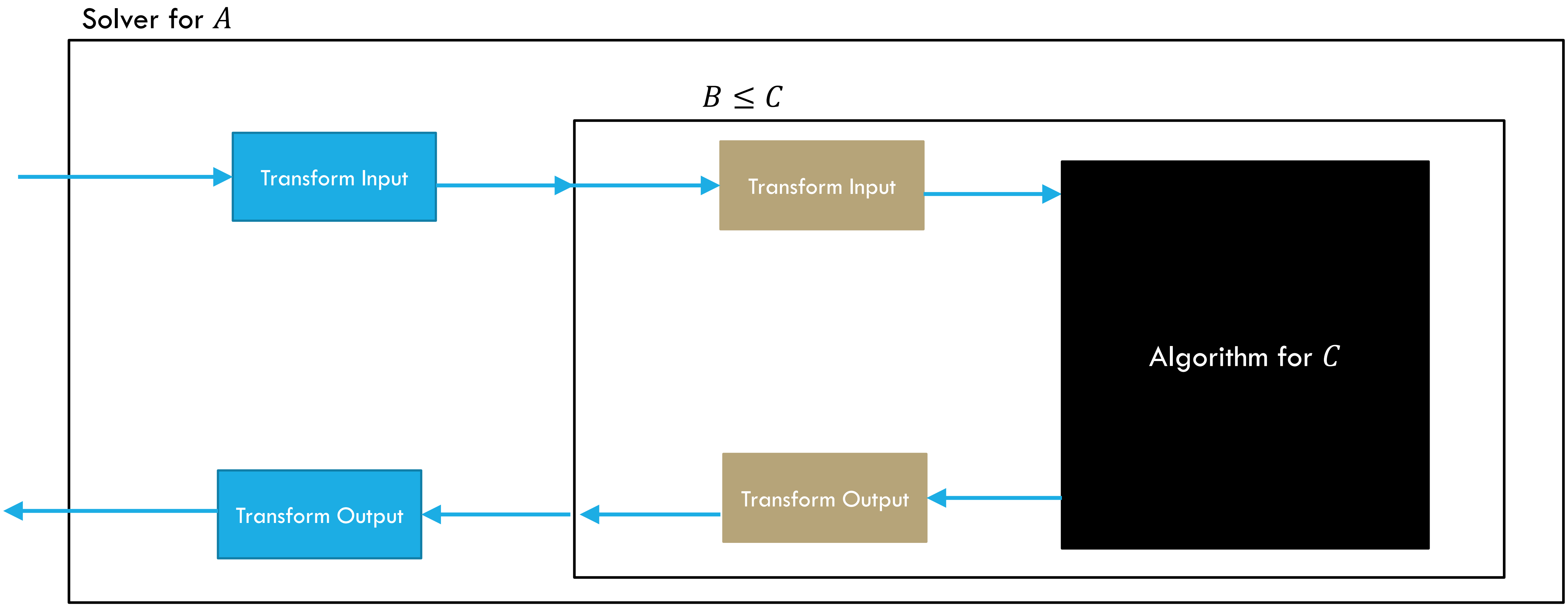
Is that true?

Solver for A

Because $A \leq B$, we have this reduction.



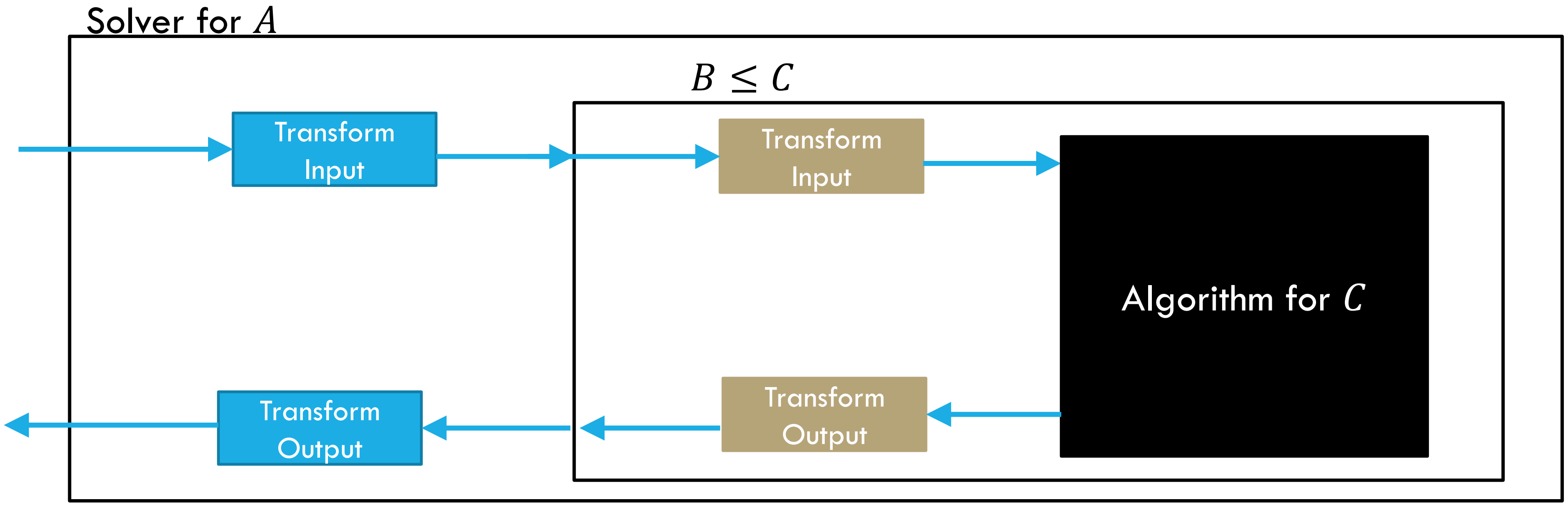
$$A \leq B \leq C \rightarrow A \leq C$$



$A \le B \le C \rightarrow A \le C$

Why does it work? Because our reductions work!

How long does it take? Still polynomial time! (Even if the input gets bigger at each step, it still can't get bigger than a polynomial). And we don't need a B solver, the reduction is the solver! We only use a C solver so it's "really" a reduction.



Said Differently

$A \leq B$

If I know B is not hard [I have an algorithm for it] then A is also not hard.

This is how we usually use reductions

$A \leq B$

If I know A is hard, then B also must be hard.

(contrapositive of the last statement)

Want to prove your problem is hard?

To show B is hard,

Reduce **FROM** the known hard problem **TO** the problem you care about

A reduction **From** an NP-hard problem A to B, shows B is also NP-hard.