Course Notes for

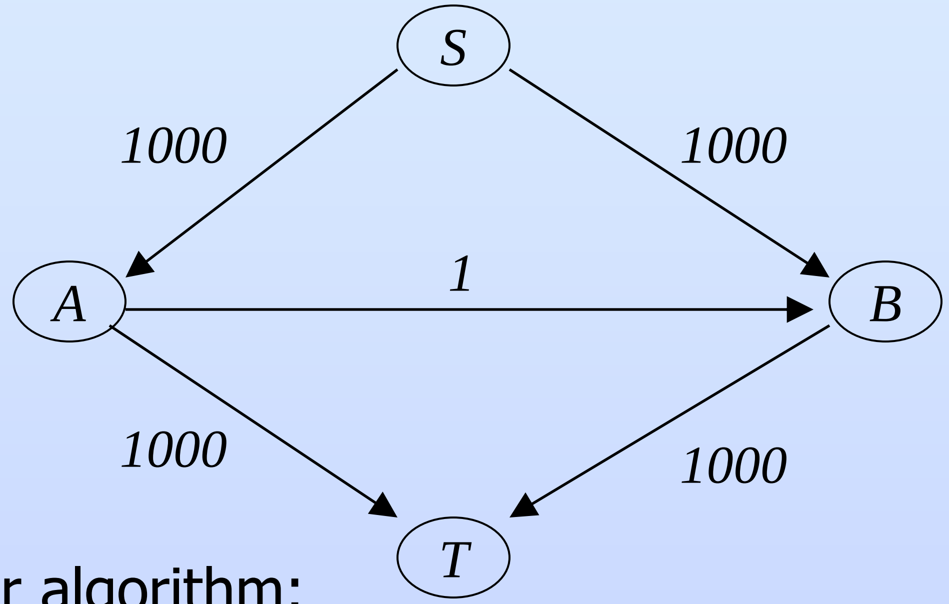
CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- In our previous lecture we discussed the Ford-Fulkerson algorithm for determining network flow
  - We looked at the basic approach of adding augmenting paths until a cut is formed in the graph
  - We also looked at how the graph would be represented
- However, we have not yet discussed way to determine an augmenting path for the graph
  - How that this be done in a regular, efficient way?
  - We need to be careful so that if an augmenting path exists, we will find it, and also so that "quality" of the paths is fairly good

- Ex: Consider the following graph



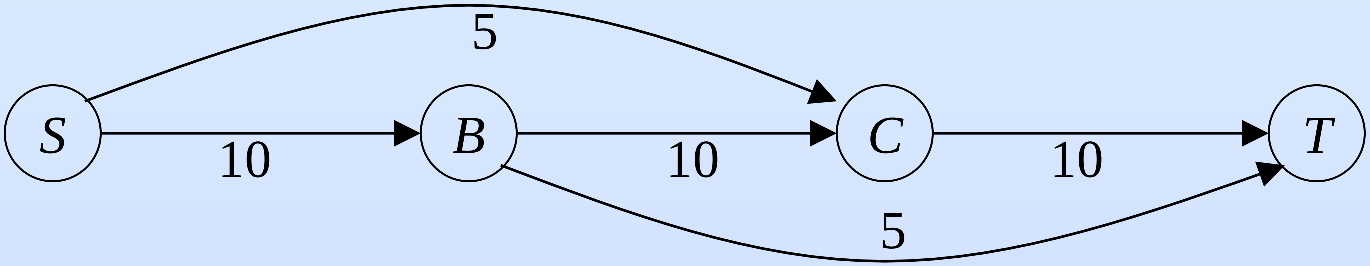
- Poor algorithm:
  - Aug. Paths SABT (1), SBAT (1), SABT (1), SBAT (1) …
  - Every other path goes along edge AB in the opposite direction, adding only 1 to the overall flow
    - > This is legal due to backward flow edges (see Lecture 19)
  - 2000 Aug. Paths would be needed before completion

Implementing FF approach

- Good algorithm:
  - 2 Aug. Paths SAT (1000), SBT (1000) and we are done
- In general, if we can find aug. paths using some optimizing criteria we can probably get good results

- Edmonds and Karp suggested two techniques:
  - Use BFS to find the aug. path with fewest edges
  - Use PFS to find the aug. path with largest augment
    - In effect we are trying to find the path whose segments are largest (maximum spanning tree)
    - Since amount of augment is limited by the smallest edge, this is giving us a "greatest" path
- Let's consider our second example from last class

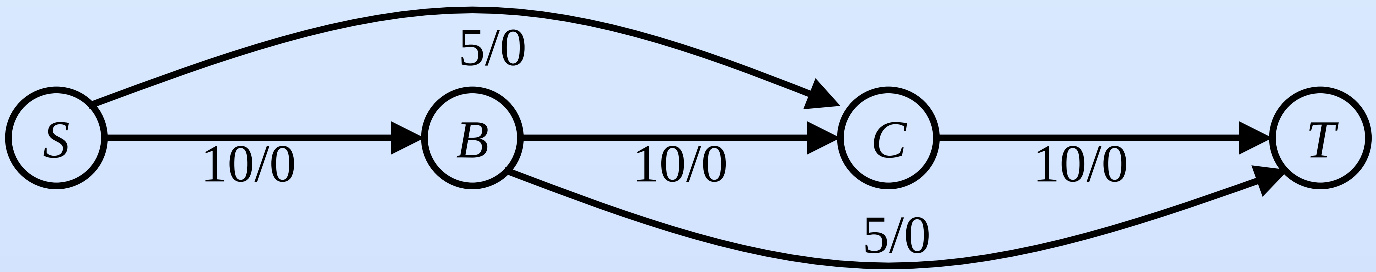
- Consider the following example with BFS



  - For each augmenting path, we form a BFS spanning tree from S
    - This is the same BFS algorithm we used previously
    - We can only consider edges with residual capacity, but we don't base our choice on the amount of that capacity
    - Rather we find the path from S to T with the fewest number of hops
    - Recall how we would do this (good review of BFS)

BFS FF Approach

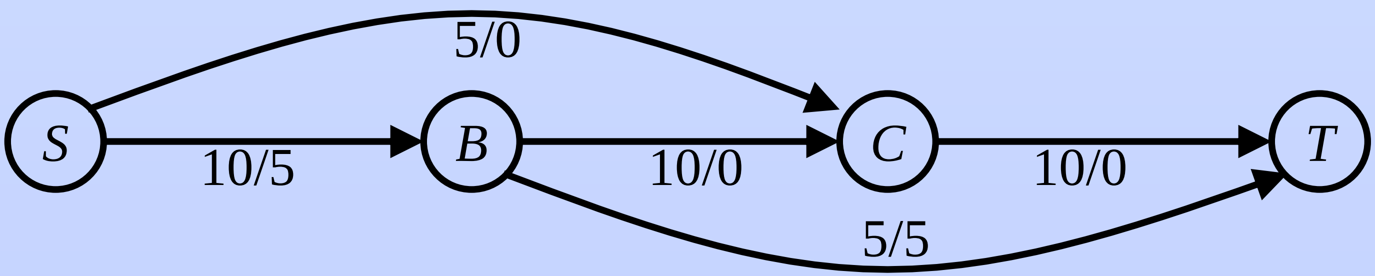
Q:	S	B	C	T



- BFS Tree shows path SBT (with weight 5)
  - > Use this as our first augmenting path

Q:	S	B	C	T

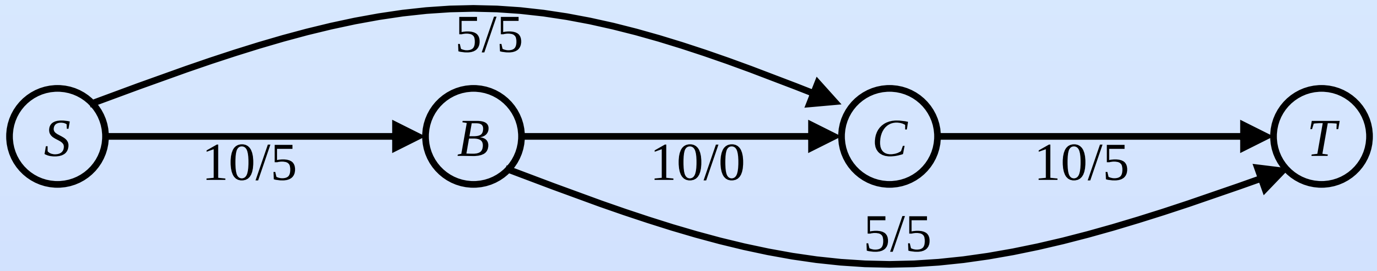
- BFS Tree shows path SCT (with weight 5)
  - > Use this as our second augmenting path



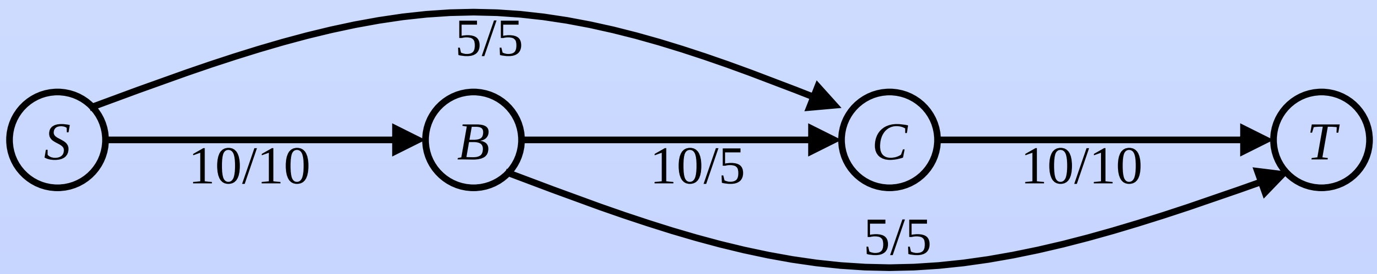
BFS FF Approach

Q:	S	B	C	T

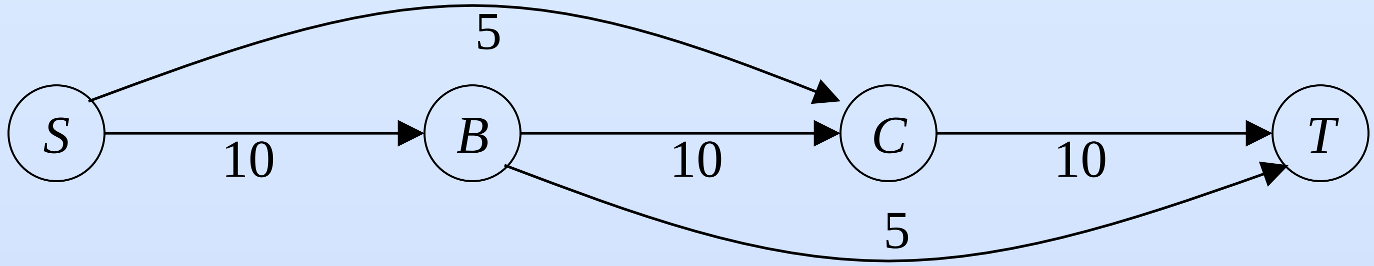
- BFS Tree shows path SBCT (with weight 5)
  - > Use this as our third augmenting path



- Graph now has a cut (SB, SC)
  - > BFS would not get to sink
  - > **Network flow is 5 + 5 + 5 = 15**



- Consider the same example with PFS



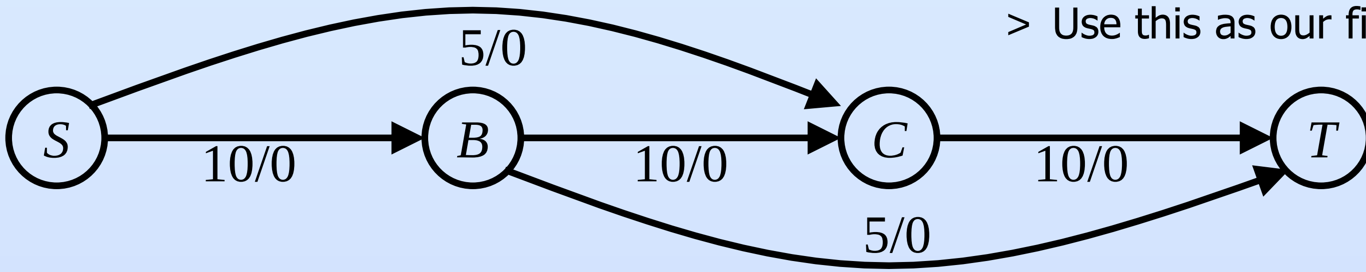
- For each augmenting path, we form a PFS spanning tree from S
    - Again, this is basically a maximum spanning tree
    - Now the amount of residual capacity available on each edge is key in building the tree
    - Recall how we would do this
        - Algorithm is Eager Prim with max instead of min
        - Let's see how it would work with a trace

PFS FF Approach

PQ:	S	B	C	C	T	T
	-	SB	SC	BC	BT	CT
	∞	10	5	10	5	10

– PFS Tree shows path SBCT (with weight 10)

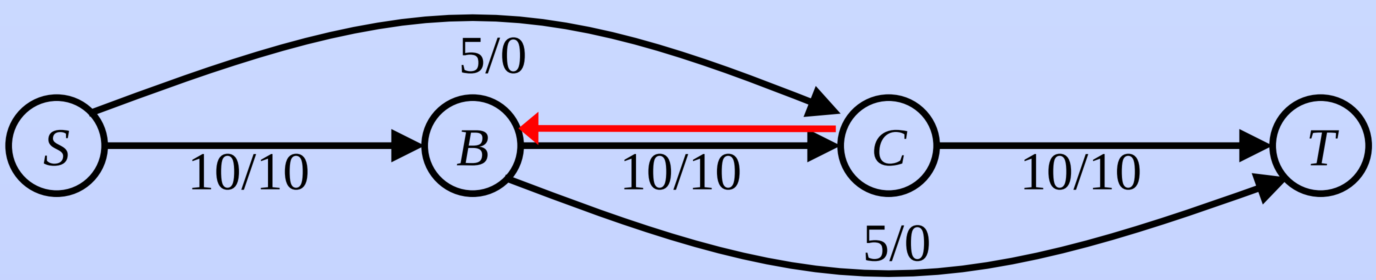
> Use this as our first aug path



PQ:	S	C	B	T
	-	SC	CB	BT
	∞	5	5	5

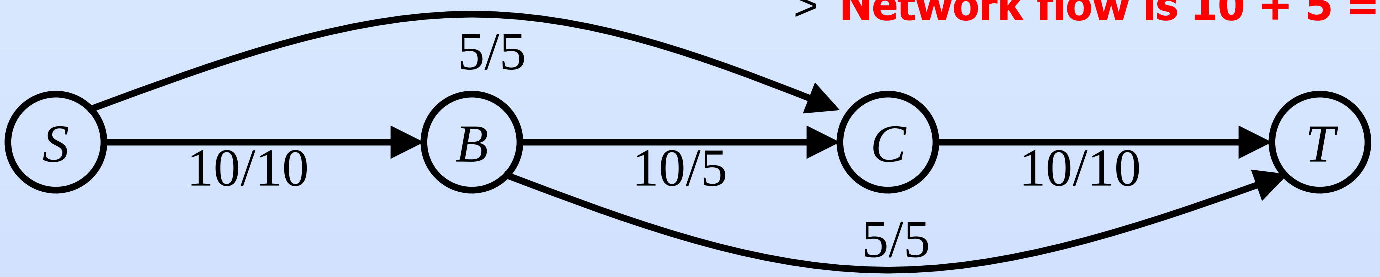
– PFS Tree shows path SCBT (with weight 5)

> Use this as our 2nd aug path



PFS FF Approach

- Graph now has a cut (SB, SC)
  - > PFS would not get to sink
  - > **Network flow is 10 + 5 = 15**



- Both approaches determine the same total flow
  - This is expected
- However, the augmenting paths chosen differ due to the different way that they are selected
- Look at the implementation in FordFulkerson.java

- Notes about the program:
  - Main algo (in constructor) will work with BFS or PFS – it uses hasAugmentingPath method to build spanning tree starting at source, and continuing until sink is reached
    - Total flow is updated by the value for the current path
    - Each path is then used to update residual graph
      - Path is traced from sink back to source, updating each edge along the way
    - If sink is not reached, no augmenting path exists and algorithm is finished
  - Sedgewick implementation uses BFS to find the augment with the fewest edges

- I have added code to allow the option of PFS to find the augmenting paths
- Run both versions on sample files (see comments at top of program for details)

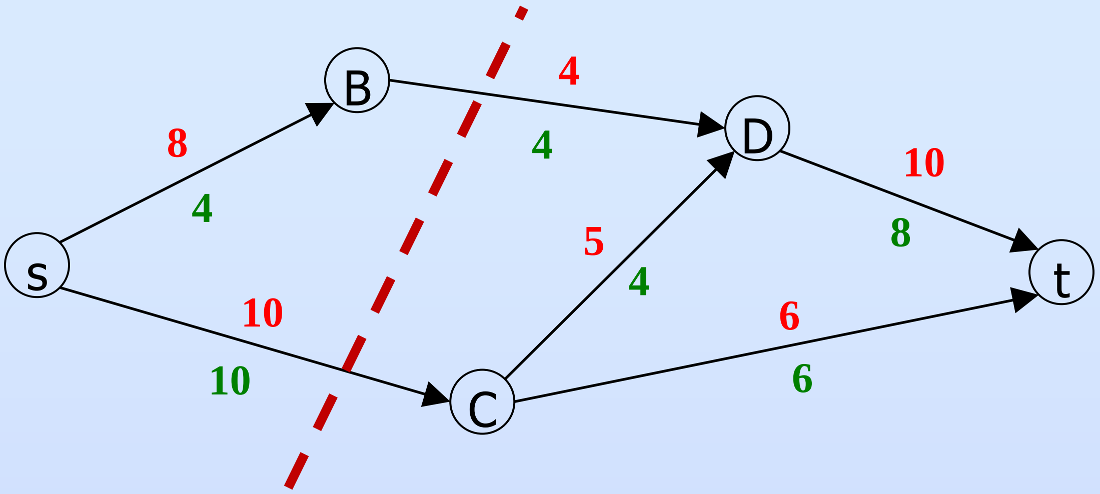
Which is better?

- It depends
- Intuitively we would expect to require fewer augmenting paths with PFS, since it is maximizing the augment
- However, with an adjacency list, PFS (Theta(ElgV)) takes longer than BFS (Theta(E+V)), so each augment requires more time
- Both approaches are reasonable

- Consider again a graph, G, as defined for the Network Flow problem
  - An st-cut in the graph is a set of edges, that, if removed, partitions the vertices of G into two disjoint sets
    - One set contains s, the other contains t
  - We can call the set of edges a cut set for the graph
    - We have already discussed this in previous slides
  - For a given graph there may be many cut sets
  - The minimum cut is the st-cut such that the capacity of no other cut is smaller

- Consider now a residual graph in which no augmenting path exists
  - One or more edges that had allowed a path between the source and sink have been used to capacity
  - These edges comprise the **min cut**
    - May not be unique
    - The sum of the weights of these edges is equal to the maximum flow of the graph
      - – In other words, calculating the maximum flow and the minimum cut are equivalent problems

Min Cut

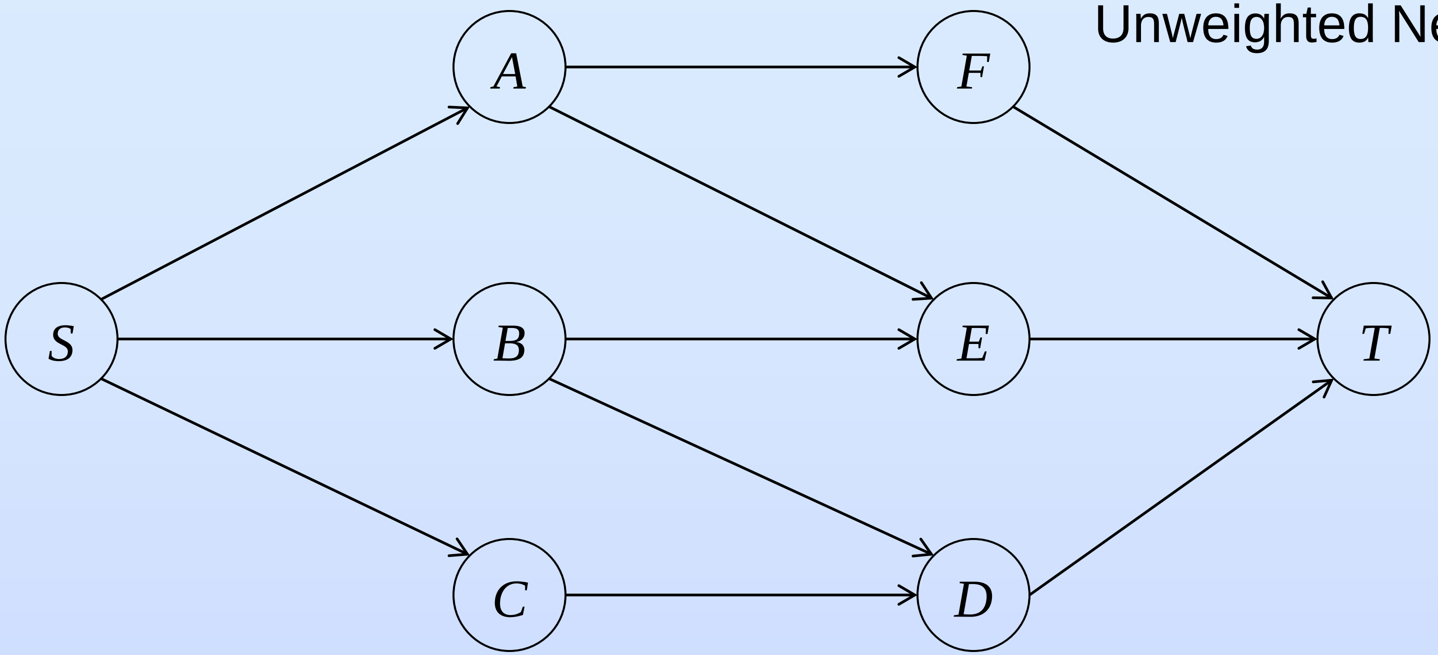


- Min Cut is shown in this graph: { sC, BD }
  - Note that there are other cuts in this graph
    - Ex: { sB, sC } ➔ weight 18
    - Ex: { BD, CD, Ct } → weight 15
  - Finding a cut is not that difficult
    - Trivial cut is to remove all outgoing edges from source or all ingoing edges to sink
  - Finding the Min Cut is equivalent to finding the Max Flow

- How to determine the min cut?
  - Do Ford-Fulkerson max flow algorithm
  - When no more augmenting paths can be found:
    - Consider the set of all vertices that are still reachable from the source (including the source itself)
      - Note that this includes vertices reachable via back edges
    - Edges that have one endpoint within this set are in the min cut
      - How can we determine this? -- Discuss
- Does Network Flow make sense in an unweighted graph?
  - Yes – in fact we can still use the Ford Fulkerson algorithm

- However, now edges are either used or not in any augmenting path – we can not use part of their capacity as we do for weighted graphs
- PFS approach for augmenting paths doesn't make sense here – BFS is best approach
  - We can still have back edges, however – going backward would "restore" that edge for another path
- In this case the maximum flow is the maximum number of distinct paths from S to T
- The min cut in an unweighted graph is the min number of edges that, when removed, disconnect S and T
  - Idea is that each edge in the cut would disconnect one path

Unweighted Network Flow



- Assume edges for each vertex are stored in **alphabetical order.**
  - What are the augmenting paths?
  - What is the min cut?
  - We will do this and discuss in our synchronous lecture
    - But you can see the bottom of this slide for the answers if you wish

- Some computational problems are unsolvable
  - No algorithm can be written that will always produce the "right" answer
  - Most famous of these is the "Halting Problem"
    - Given a program P with data D, will P halt at some point?
      - It can be shown (through a clever example) that this cannot be determined for an arbitrary program
      - http://en.wikipedia.org/wiki/Halting_problem
  - Other problems can be "reduced" to the halting problem
    - Indicates that they too are unsolvable

- Some problems are solvable, but require an exponential amount of time
  - We call these problems intractable
    - For even modest problem sizes, they take too long to solve to be useful
  - Ex: List all subsets of a set
    - We know this is Theta(2^N)
  - Ex: List all permutations of a sequence of characters
    - We know this is Theta(N!)

Polynomial Problems

- Most useful algorithms run in polynomial time
  - Largest term in the Theta run-time is a simple power with a constant exponent
    - Or a power times a logarithm (ex: NlgN is considered to be polynomial)
  - Most of the algorithms we have seen so far this term fall into this category

NP-Complete Problems

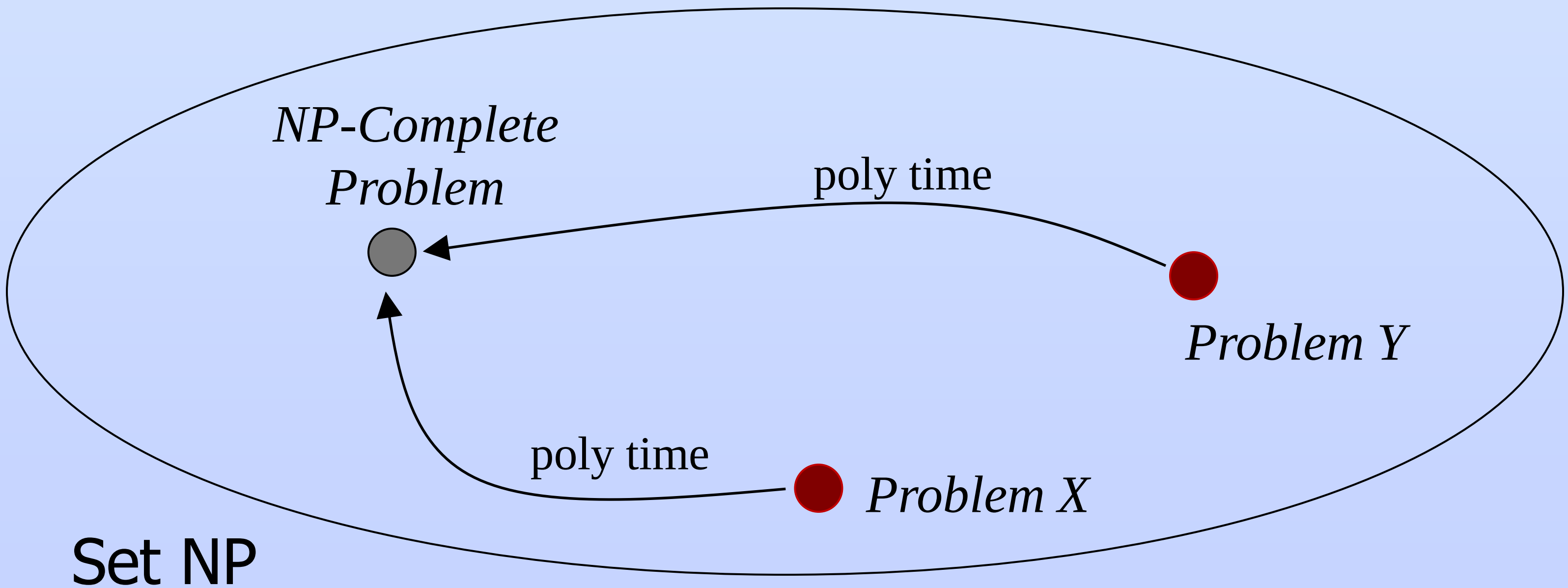
- Background
  - Some problems don't (yet) fall into any of the previous 3 categories:
    - They can definitely be solved
    - We have not proven that any solution requires exponential execution time
    - No one has been able to produce a valid solution that runs in polynomial time
  - It is from within this set of problems that we produce NP-complete problems

- More background:
  - Define $\mathbb{P}$ = set of problems that can be solved by deterministic algorithms in polynomial time
    - What is deterministic?
      - At any point in the execution, given the current instruction and the current input value, we can predict (or determine) what the next instruction will be
      - If you run the same algorithm twice on the same data you will have the same sequence of instructions executed
      - Most algorithms that we have discussed this term fall into this category

- Define **NP** = set of problems that can be solved by non-deterministic algorithms in polynomial time
  - What is non-deterministic?
    - Formally this concept is tricky to explain
      - Involves a Turing machine
    - Informally, we allow the algorithm to "cheat"
      - We can "magically" guess the solution to the problem, but we must verify that it is correct in polynomial time
    - Naturally, our programs cannot actually execute in this way
      - We simply define this set to categorize these problems
  - http://www.nist.gov/dads/HTML/nondetermAlgo.html
  - http://en.wikipedia.org/wiki/NP_(complexity)

- Ex: TSP (Traveling Salesman Problem)
  - Instance: Given a finite set $C = \{c_1, c_2, \dots c_m\}$ of cities, a distance $d(c_I, c_J)$ for each pair of cites, and in integer bound, B (positive)
  - Question: Is there a "tour" of all of the cities in C (i.e. a simple cycle containing all vertices) having length no more than B?
- In non-deterministic solution we "guess" a tour (ex: try the "best" choice at each step) and then verify that it is valid and has length <= B or not within polynomial time
- In deterministic solution, we need to actually find this tour, requiring quite a lot of computation
  - No known algo in less than exponential time

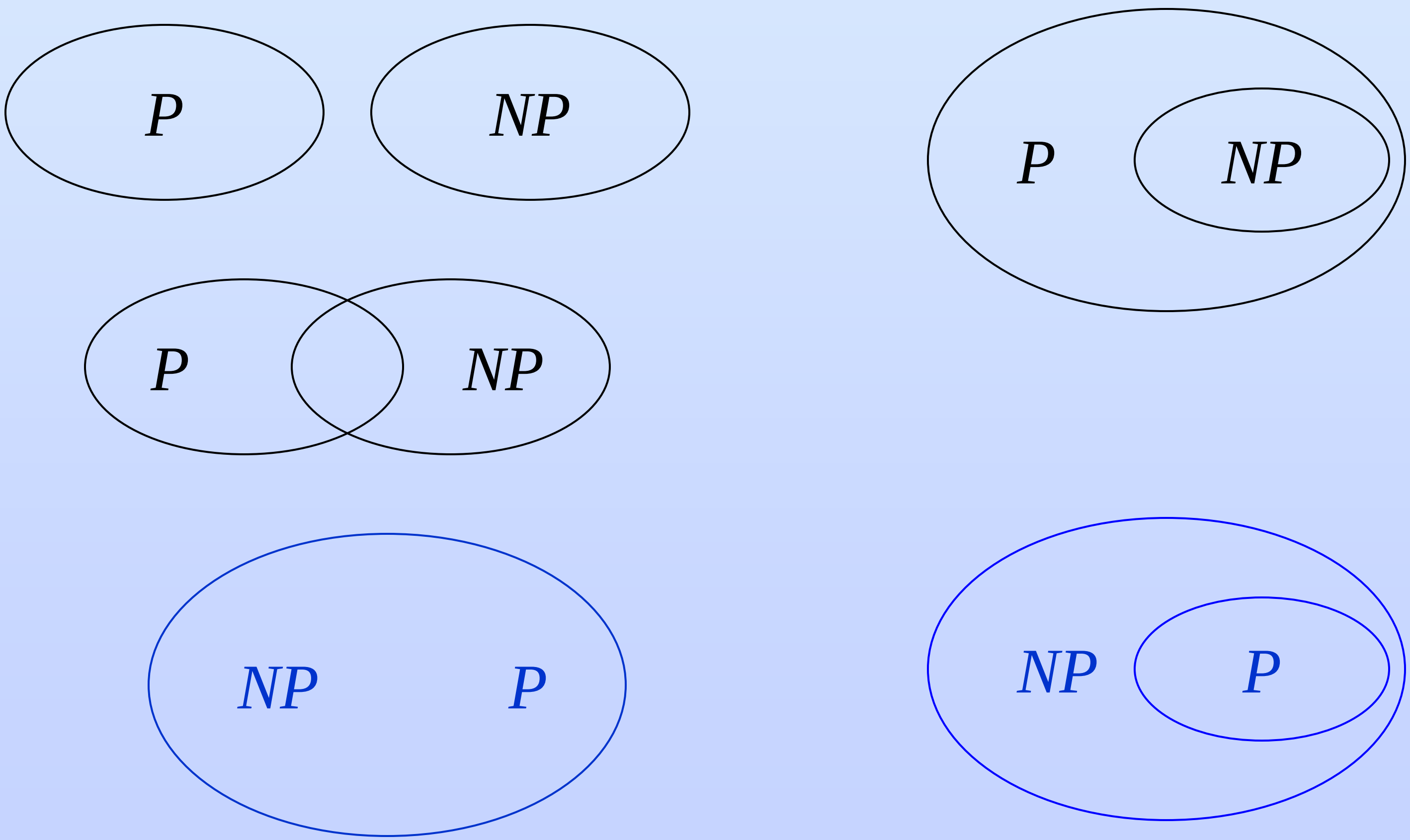
- So what are NP-Complete Problems?
  - Naturally they are problems in set NP
  - They are the "hardest" problems in NP to solve
    - All other problems in NP can be transformed into these problems in polynomial time



NP-Complete Problems

- **If** any NP-complete problem can be solved in deterministic polynomial time, then all problems in NP can be
  - Since the transformation takes polynomial time, we would have
    - A polynomial solution of the NP-Complete problem
    - A polynomial transformation of any other NP problem into the NP-Complete problem
    - Total time is still polynomial

- Consider sets P and NP:
  - We have 5 possibilities for these sets:



- 3 of these can be easily dismissed
  - We know that any problem that can be solved deterministically in polynomial time can certainly be solved non-deterministically in polynomial time
- Thus the only real possibilities are the two in blue:
  - $P \subset NP$
    - > P is a proper subset of NP, as there are some problems solvable in non-deterministic polynomial time that are NOT solvable in deterministic polynomial time
  - $P = NP$
    - > The two sets are equal – all problems solvable in non-deterministic polynomial time are solvable in deterministic polynomial time

- Right now, we don't know which of these is the correct answer
  - We can show P $\subset$ NP if we can prove an NP-Complete problem to be intractable
  - We can show P = NP if we can find a deterministic polynomial solution for an NP-Complete problem
- Most CS theorists believe the P $\subset$ NP
  - If not, it would invalidate a lot of what is currently used in practice
    - Ex: Some security tools that are secure due to computational infeasibility of breaking them may not actually be secure
- But prove it either way and you will be famous!