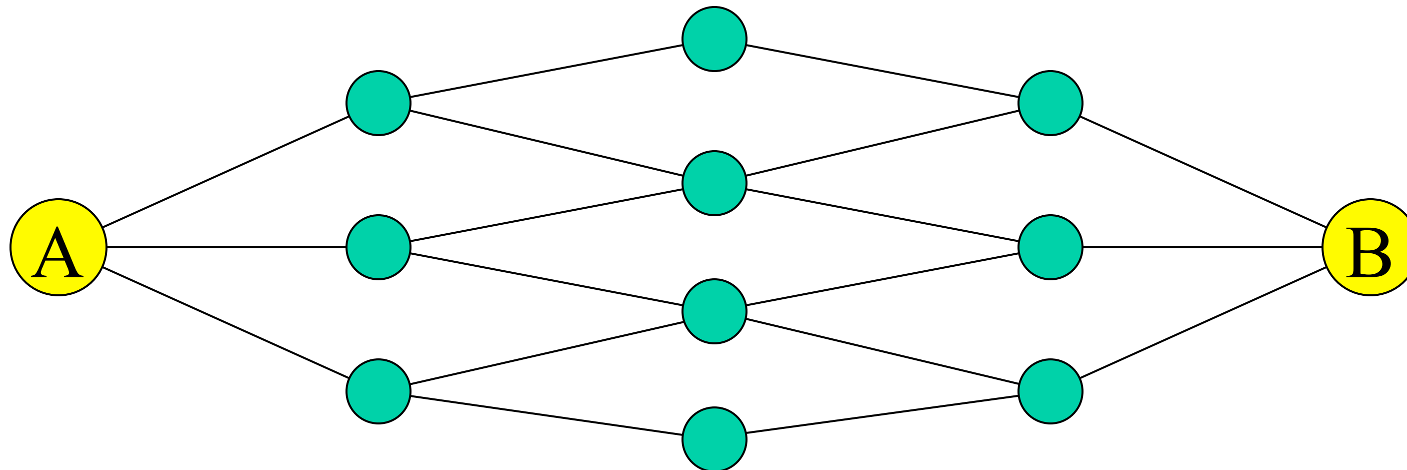
Example Problem

A company has all of its branches in a particular city all wired directly to each other to maximize the rate that data can be transferred around. If a critical operation needs to transfer information from A to B, what is the fastest it can be done given each connection has its own rate?

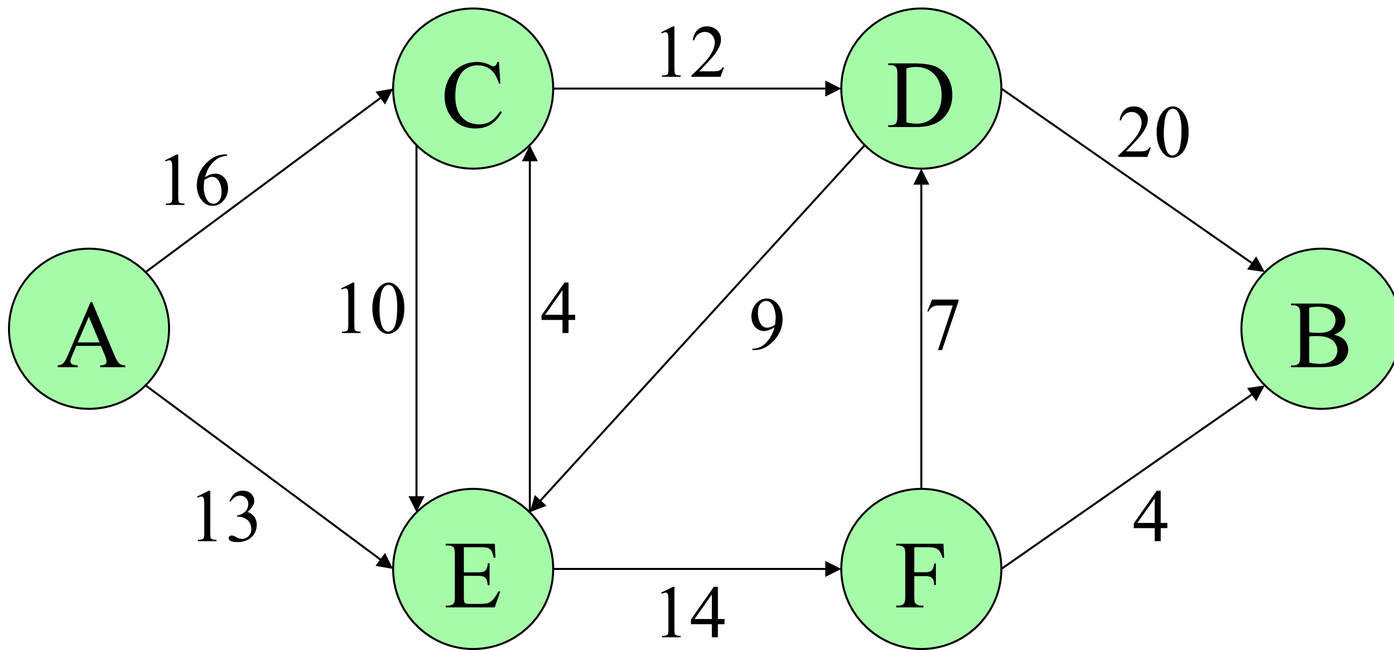


Network Flow

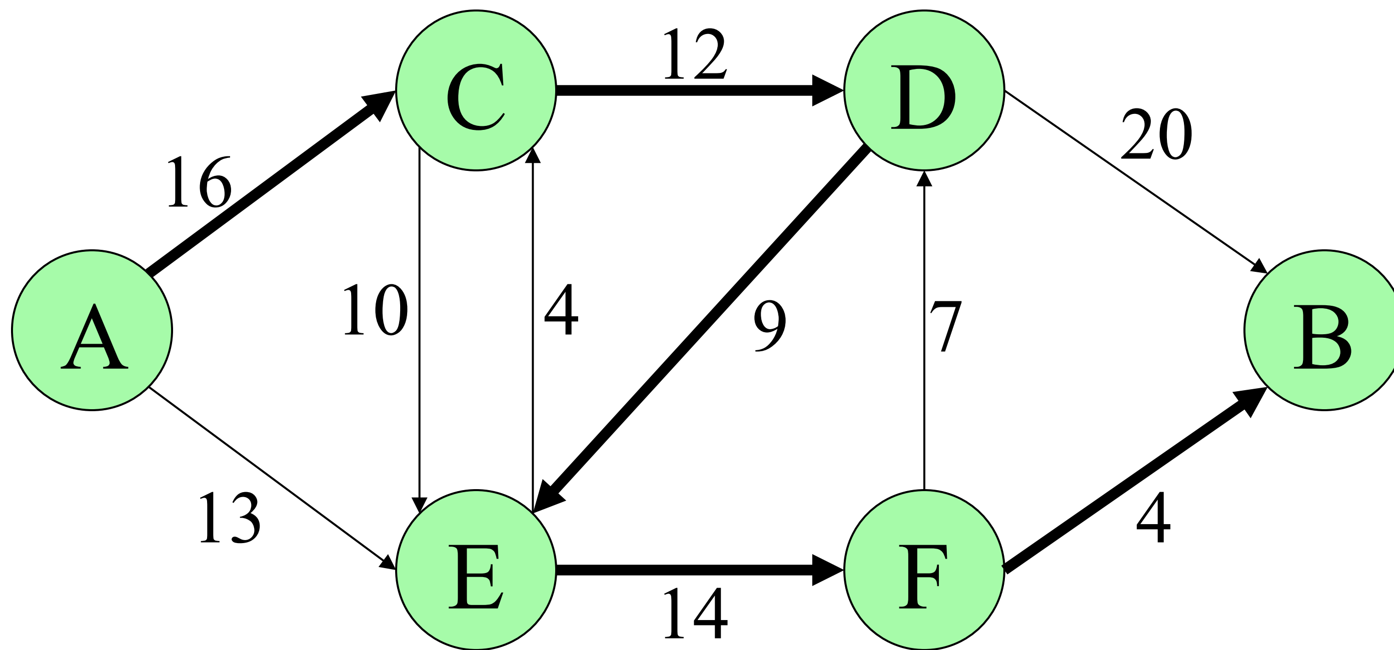
Inputs: A graph G, where each edge $e = (i, j)$ has a capacity c_e. A source node A, and a sink node B.

Problem: What is the maximum flow you can route from A to B while respecting the capacity constraint of each edge?

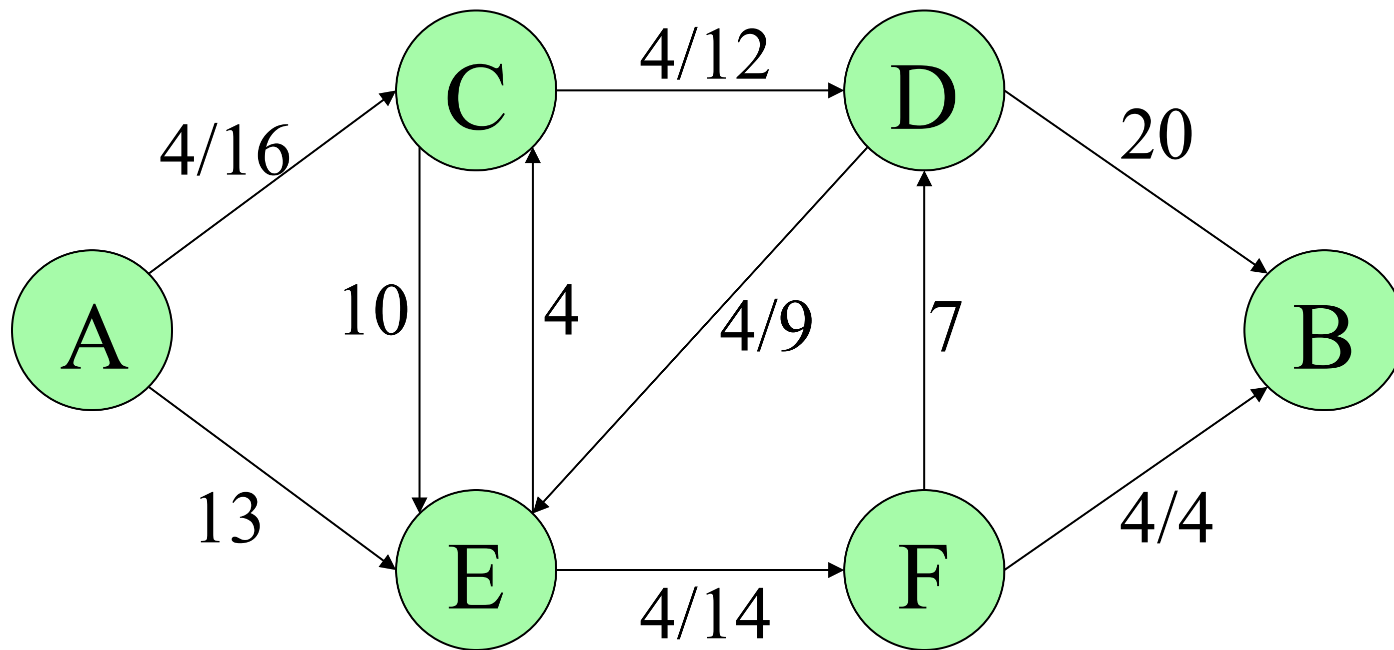
Example



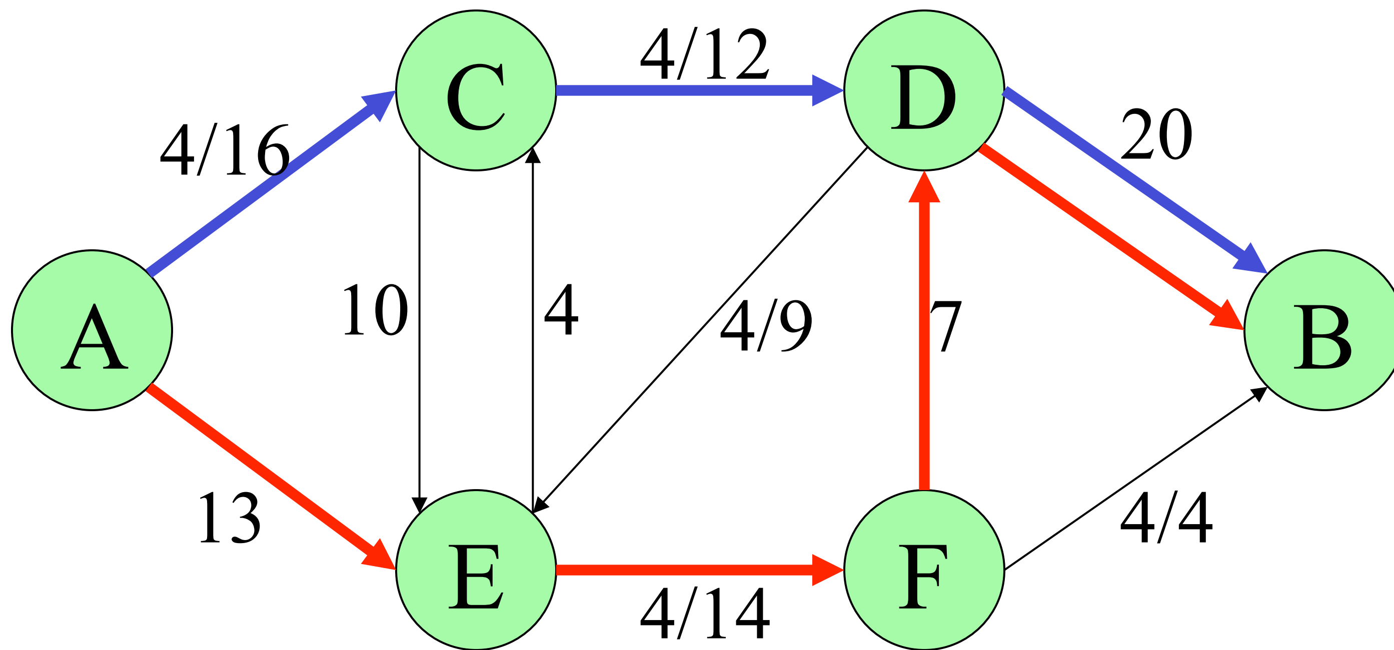
One Possible Path



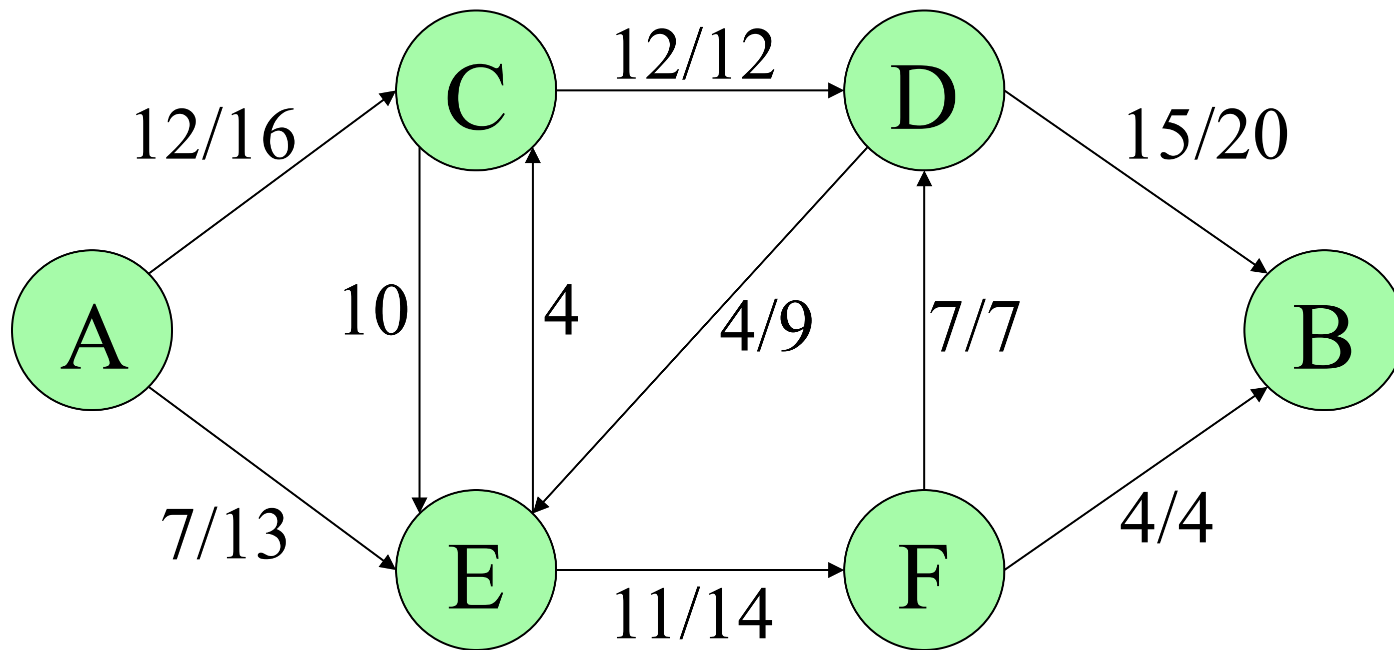
The capacity used so far...



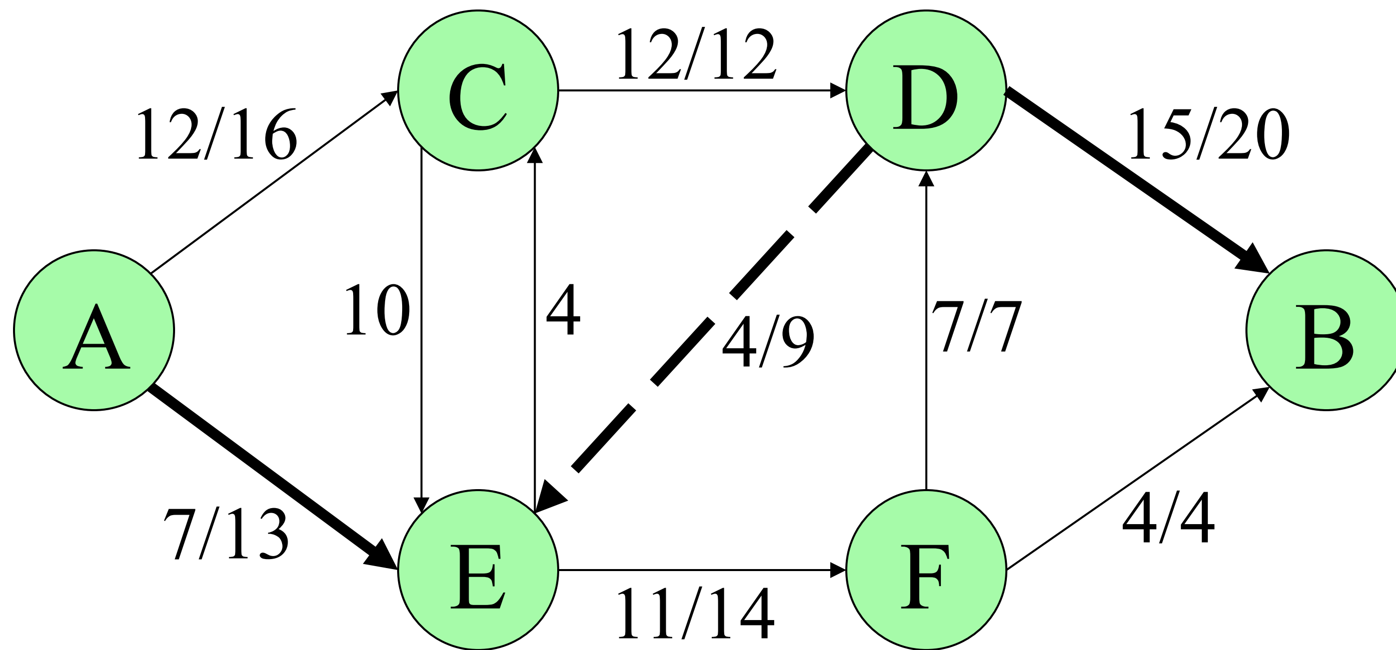
Two more paths...



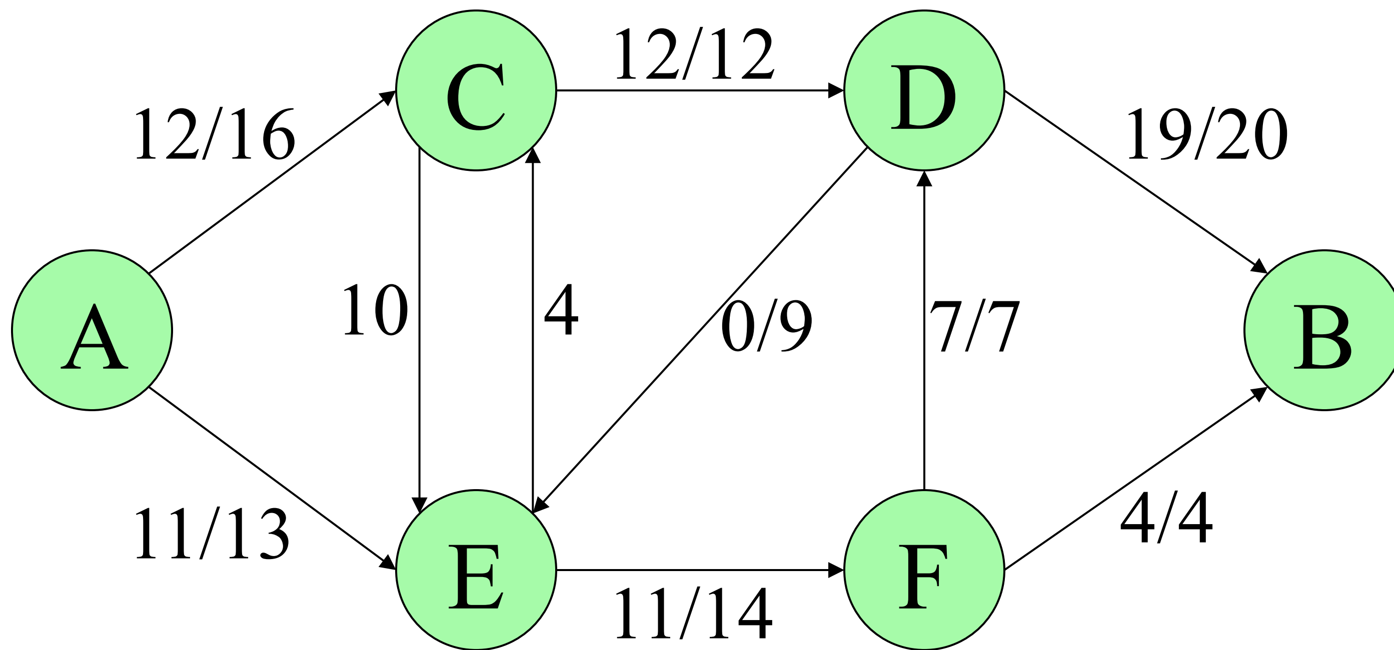
Final Capacities?



One final possible path...



And the resulting flow...



What to do with Hard Problems?

Sometimes we end up with a situation where our boss tells us to write a fast algorithm to solve a problem, but we try and fail. We will learn how to prove that the problem is hard, and that we're *not* going to find one…

…but, will your boss be happy with this? Such problems will come up, and they need an answer, and we need to write *something* to get that answer.

So what are our options?

Our Options:

1. Try to do as much simplification as possible, and hope that we can get it to a point where we can solve the problem "fast enough". Often, if you know enough about your instances, you can write an algorithm that is near polynomial time in the *average* case.

2. Use a heuristic to approximate the solution. Rather than worrying about getting a correct answer in pretty good worst-case time, we worry about getting a worst-case pretty good answer in polynomial time!

Approximating TSP

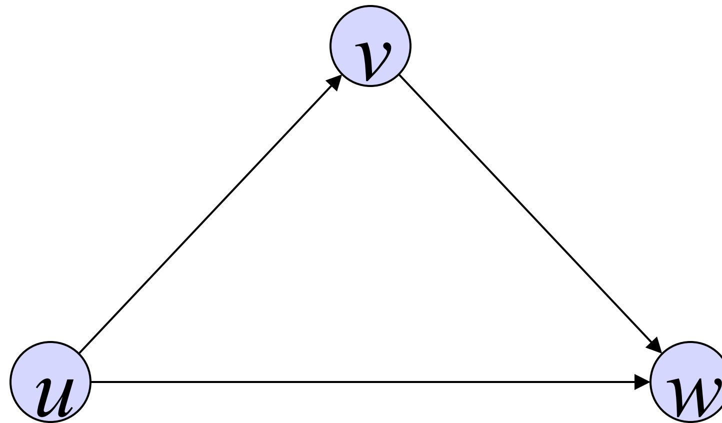
The ***Euclidean* Traveling Salesman** is the traditional version of this problem: a salesman wants to plan a drive to visit all his customers exactly once and get back home. All weights of the edges represent the actual distance between the points.

TSP remains hard even when the distances are Euclidean distances in the plane. However, this version does allow us an easy method to obtain a *reasonable* solution.

Properties of Euclidean TSP

Euclidean geometry satisfies the triangle inequality:

$$d(u, w) \leq d(u, v) + d(v, w)$$

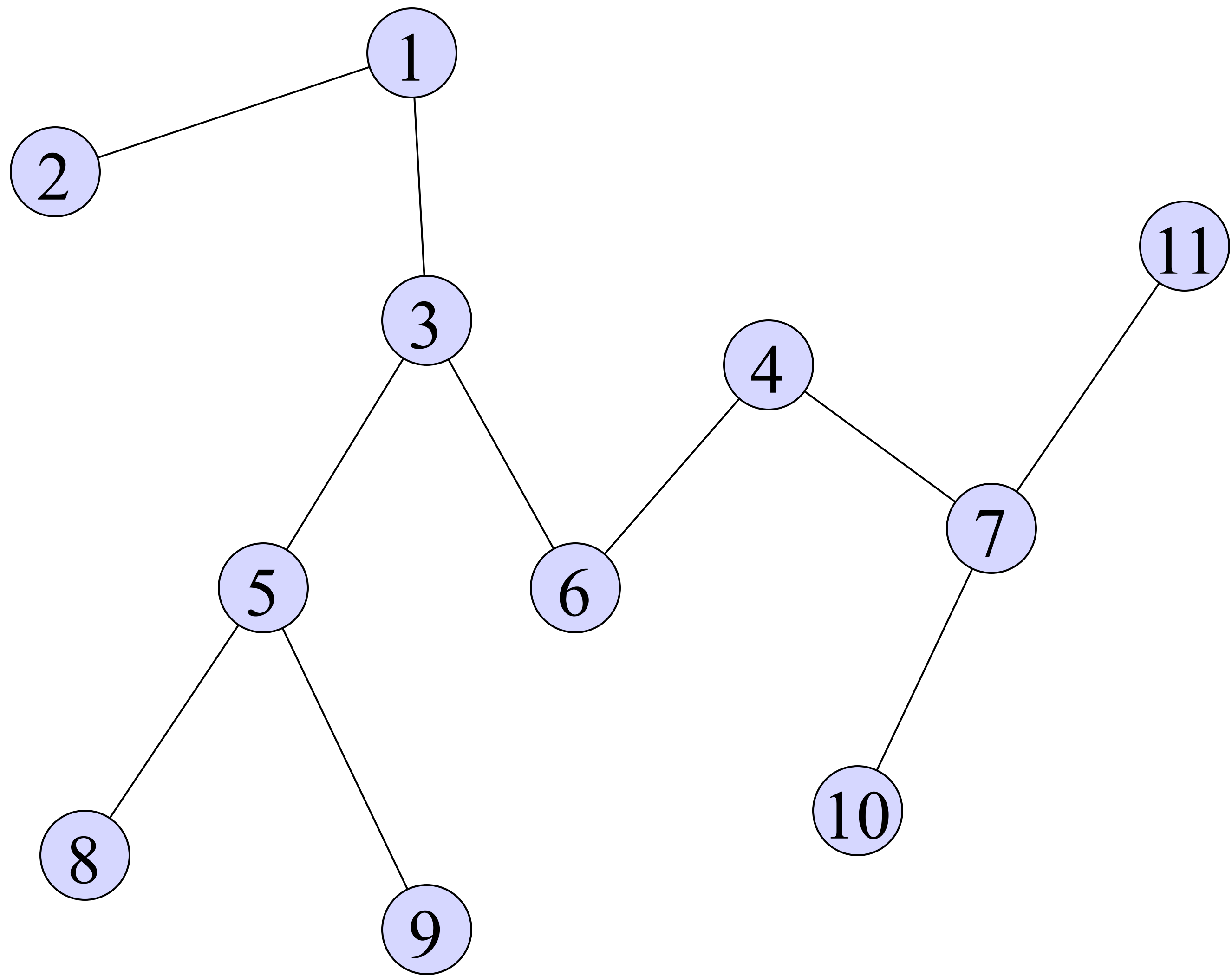


A lower bound...

We can approximate the optimal Euclidean TSP tour using minimum spanning trees.

Claim: the cost of a MST is a lower bound on the cost of a TSP tour.

Why? Deleting any edge from a TSP tour leaves a path, which is a tree of weight at least that of the minimum spanning tree!

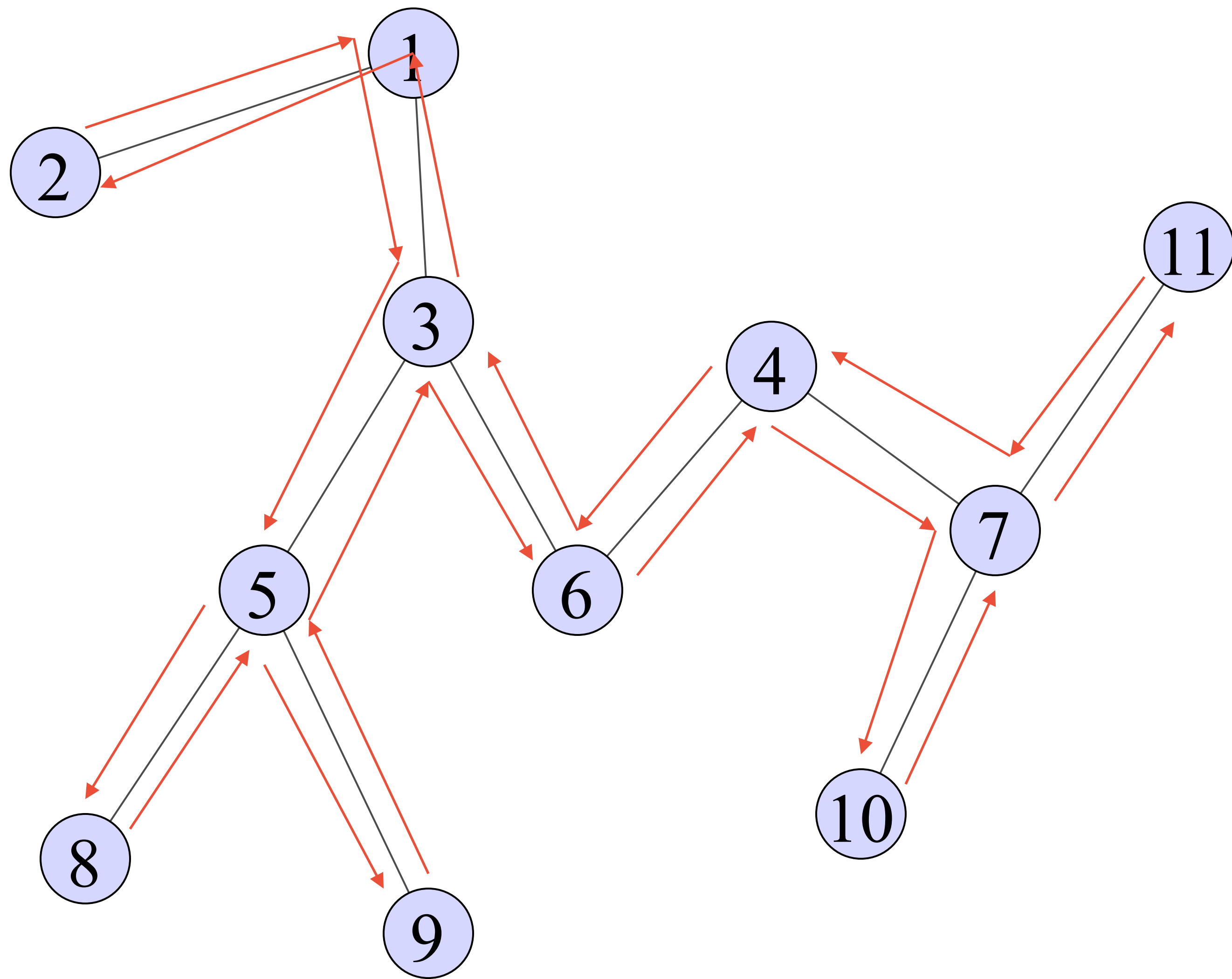

1
2
3
4
5
6
7
8
9
10
11

An upper bound...

Claim: *Twice* the minimum spanning tree is an upper bound on the length of the cost of a TSP tour.

If we were allowed to visit cities more than once, doing a depth-first traversal of a MST, and then walking out the tour specified is at most twice the cost of MST.

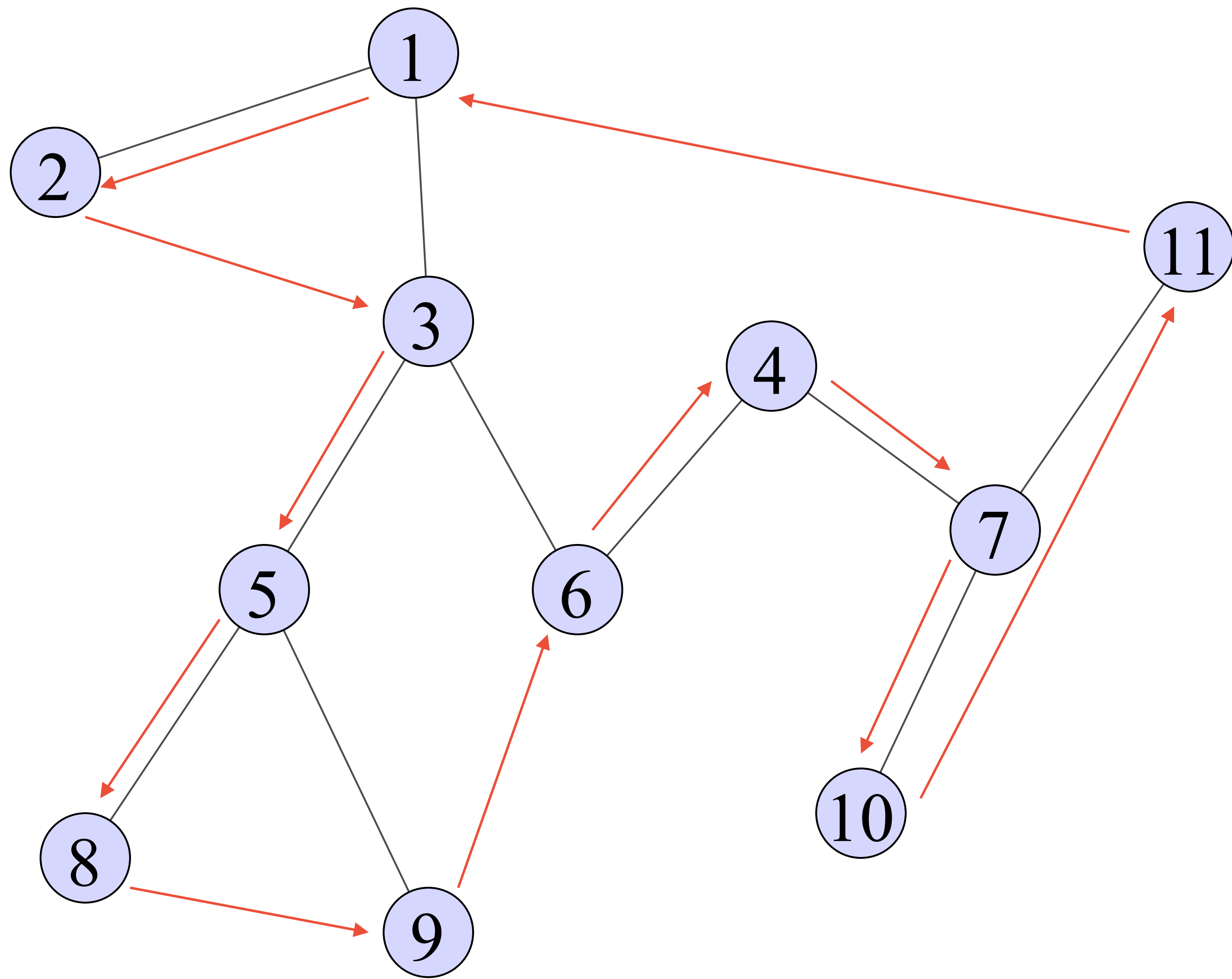
Why? We will be using each edge exactly twice.

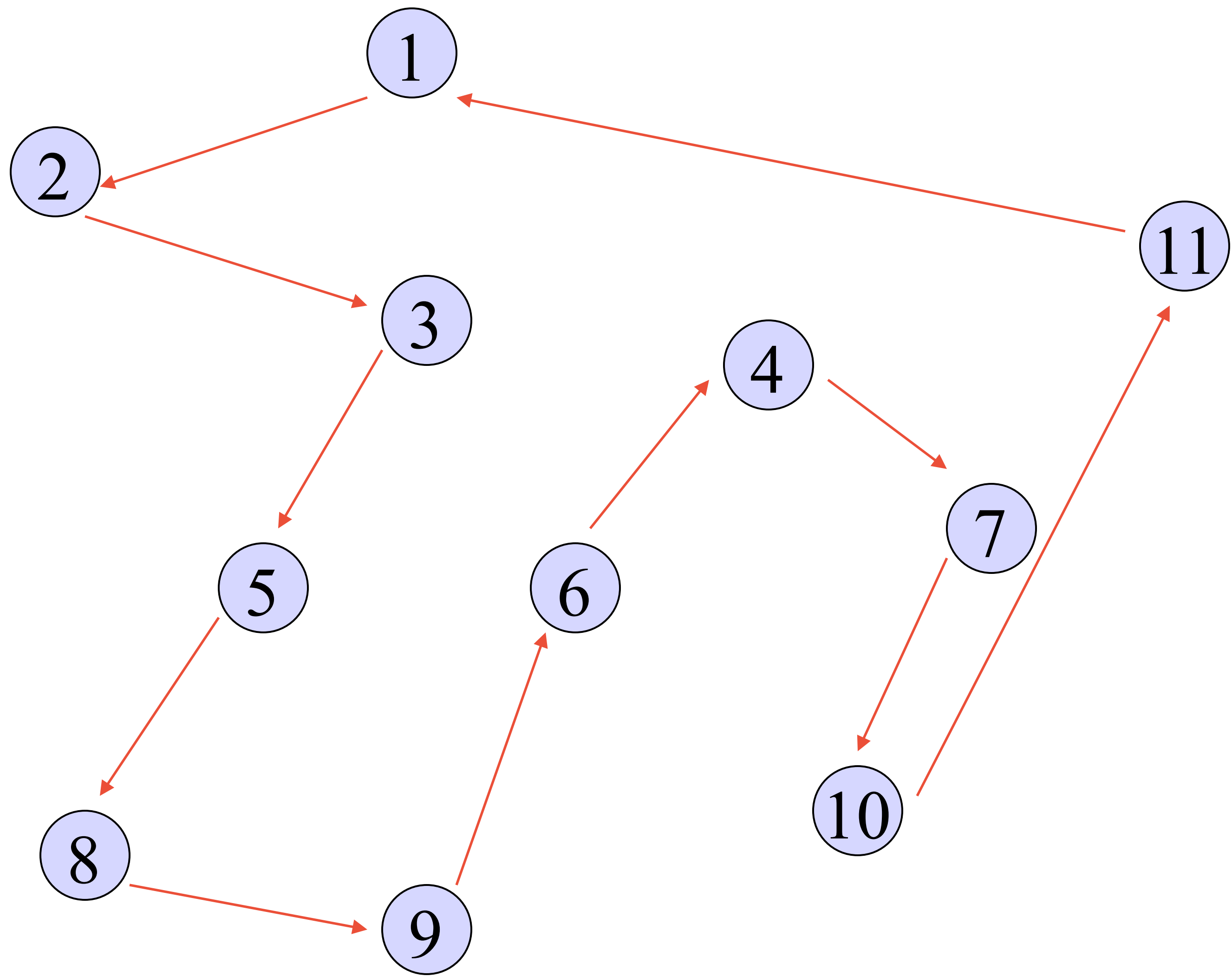

1
2
3
4
5
6
7
8
9
10
11

Preventing multiple visits...

One of the key elements about the Traveling Salesman Problem is that we're explicitly *not* allowed to visit a single vertex multiple times. If we do take into account this restriction, can we still guarantee the upper bound of twice the minimum spanning tree?

Yes! In fact, it will almost always be less than this, thanks to the triangle inequality. We can jump immediately to the next *new* vertex that we would encounter.


1
2
3
4
5
6
7
8
9
10
11


1
2
3
4
5
6
7
8
9
10
11

Do I really understand the problem?

- What exactly does the input consist of?
- What exactly are the desired results or output?
- Can I construct an example input small enough to solve by hand? What happens when I try to solve it?
- How important is it to my application that I always find an exact, optimal answer? Can I settle for something that is usually pretty good?
- How large will a typical instance of my problem be? Will I be working on 10 items? 1,000 items? 1,000,000 items?

- How important is speed in my application? Must the problem be solved within one second? One minute? One hour? One day?

- How much time and effort can I invest in implementing my algorithm? Will I be limited to simple algorithms that can be coded up in a day, or do I have the freedom to experiment with a couple of approaches and see which is best?

- Am I trying to solve a numerical problem? A graph problem? A geometric problem? A string problem? A set problem? Might my problem be formulated in more than one way? Which formulation seems easiest?

Can I find a simple algorithm or heuristic for the problem?

- Can I find an algorithm to solve my problem *correctly* by searching through all subsets or arrangements and picking the best one?
- How do I measure the quality of a solution I construct?
- Does this simple, slow solution run in polynomial or exponential time? Is my problem small enough that this brute-force solution will suffice?
- If I can't find a slow, *guaranteed* correct algorithm, why am I certain that my problem is sufficiently well-defined to have a correct solution?

- Can I find an acceptable solution to my problem by repeatedly trying some simple rule, like picking the biggest item first? The smallest item first? A random item first? (A *greedy* Algorithm!)

- If so, on what types of inputs does this heuristic work well? Do these correspond to the data that might arise in my application?

- On what types of inputs does this heuristic work poorly? If no such examples can be found, can I prove that it will always work well?

- How fast does my heuristic come up with an answer? Does it have a simple implementation?

Have I utilized all of the resources that are available?

- Is my problem in the catalog of algorithmic problems in the textbook? If so, what is known about the problem? Is there an implementation available that I can use?

- Are there relevant resources available on the World-Wide Web? Did I use algorithm specific web pages as well as general search engines?

- If I don't see my problem, did I look in the right place? Can I easily transform it into another problem that I might have better luck researching?

Does the problem have special cases that I know how to solve exactly?

- Can I solve the problem efficiently when I ignore some of the input parameters, or set them to trivial values, such as 0 or 1?
- Can I simplify the problem to the point where I *can* solve it efficiently? Is the problem now trivial or still interesting?
- Once I know how to solve a certain special case, why can't this be generalized to a wider class of inputs?
- Is my problem itself a special case of a more general problem?

Which design paradigm is most relevant to my problem?

- Is there a set of items that can be sorted by size or a key? Does a sorted order make the answer easier to find?
- Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?
- Do the input objects or desired solution have a natural left-to-right order, such as characters in a string, elements of a permutation, or the leaves of a tree? If so, can I use dynamic programming to exploit this order?

- Are there certain operations being repeatedly done on the same data, such as searching, minimum/maximum, or predecessor/successor? If so, can I use a data structure to speed up these queries?

- Can I use random sampling to select which object to pick next? What about constructing many random configurations and picking the best one? Can I use some kind of directed randomness like simulated annealing or genetic algorithms to approximate the best solution?

- Does my problem seem something like satisfiability, the traveling salesman problem, or some other NP-complete problem? If so, might the problem be NP-complete and thus not have an efficient algorithm?

More Approximation Methods

- Solve a restricted version of the problem.
- Solve the problem within a limited factor (i.e., guaranteed max of twice minimum)
- Determine a heuristic that will give a near-optimal answer *on average*.
- Search randomly for a solution, picking the best ones and assuming that “neighboring” solutions will be of similar quality.