ECM3412/ECMM409
Nature Inspired Computation
Lecture 2

What Evolutionary Algorithms are for
(About Search, Optimisation, Hard and Easy Problems, Exact and Approximate Algorithms)

Today's Plan

About the concept of *optimisation* (since this is what EAs are `for')

Complexity: Hard problems and Easy problems

Exact algorithms and Approximate Algorithms

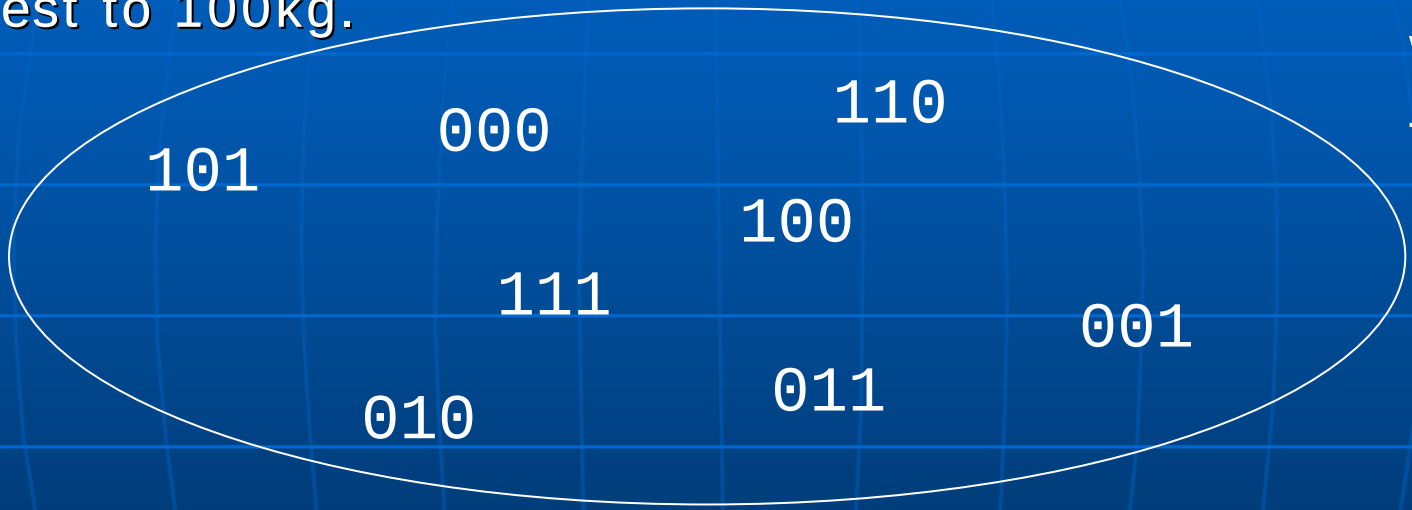
EAs are *approximate algorithms* applicable to *hard problems*.

What does this mean? Well ...

Search and Optimisation

We have 3 items as follows: (item 1: 20kg; item2: 75kg; item 3: 60kg)

Suppose we want to find the subset of items with total weight closest to 100kg.



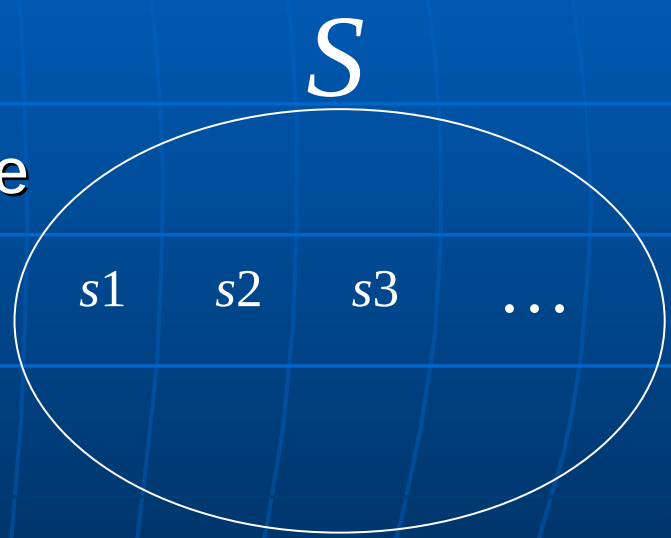
Which is the best?

Well done, you just searched the space of possible subsets. You also found the optimal one. If the above set of subsets is called *S*, and the subsets themselves are *s*1, *s*2, *s*3, etc …, you just optimised the function “closest_to_100kg(*s*)”; i.e. you found the *s* which minimises the function |(weight—100)| .

Search and Optimisation

- In general, *optimisation* means that you are trying to find the best solution you can (usually in a short time) to a given problem.

- We always have a set s of possible solutions



S may be small (as just seen)

S may be very, *very*, *very*, ***very*** large
(e.g all alignments of two 50-base sequences allowing 10 insertions/deletions, or all possible timetables for a 500-exam/3-week period)
… in fact something like 10^{30} is typical for real problem.
S may be infinitely large – e.g. all real numbers.

The Fitness function

- Every **candidate solution** *s* in *S* can be given a score, or a "fitness", by a so-called fitness function. We usually write *f*(*s*) to indicate the fitness of solution *s*. Obviously, we want to find the *s* in *S* which has the best score.

Examples

timetabling: *f could be no. of clashes*.
racing car setup: *f could be lap times*
design of something: *(electric circuits, water distribution networks, site layouts, antenna for satellites, …* :

- f will usually be a measure of closeness of fit to the design spec/requirements

Searching through *S*

When *S* is small (e.g. 10, 100, or only 1,000,000 or so items), we can simply do so-called *exhaustive search*.

Exhaustive search: Generate every possible solution, work out its fitness, and hence discover which is best (or which set share the best fitness)

This is also called ***Enumeration***

However …

- *In **all** interesting/important cases*, *S* is much much much too large for exhaustive search (**ever**).

- There are two kinds of `too-big' problem:
  - easy (or `tractable', or `in *P*')
  - hard (or `intractable', or `not known to be in *P*')
- There are rigorous mathematical definitions of the two types.
- Important (for you) is that almost all important problems are technically hard.

About Optimisation Problems

To *solve* a problem means to find an *optimal* solution. i.e. to deliver an element of *s* whose fitness is ***guaranteed*** to be the best in *S*.

An *Exact algorithm* is one which can do this (i.e. solve a problem, guaranteeing to find the best).

Problem complexity

This is all about characterising how hard it is to solve a given problem. Statements are made in terms of functions of n, which is meant to be some indication of the size of the problem. E.g.:

Correctly sort a set of n numbers

- Can be done in *around* $n \log n$ *steps*

Find the closest pair out of n vectors

- Can be done in $O(n^2)$ steps

Find *best* multiple alignment of n sequences.

- Can be done in $O(2^n)$ steps …

Polynomial and Exponential Complexity

- Given some problem *Q*, with `size' *n*, imagine that *A* is the fastest algorithm known for solving that problem exactly. The complexity of problem *Q* is the time it takes *A* to solve it, as a function of *n*.

- There are two key kinds of complexity:

Polynomial: the dominant term in the expression is polynomial in *n*. E.g. n^{34}, $n.\log.n$, $\sin(n^{2.2})$, etc ...

Exponential: the dominant term is exponential in *n*. E.g. 1.1^n, n^{n+2} , 2^n, ...

Polynomial and Exponential Complexity

n	2	3	4	5	6	10	20	50	100
1.1^n	1.21	1.33	1.46	1.61	1.77	2.59	6.73	117	13,780
$n^{1.1}$	2.14	3.35	4.59	5.87	7.18	12.6	27.0	73.9	159

Problems with exponential complexity take too long to solve at large n

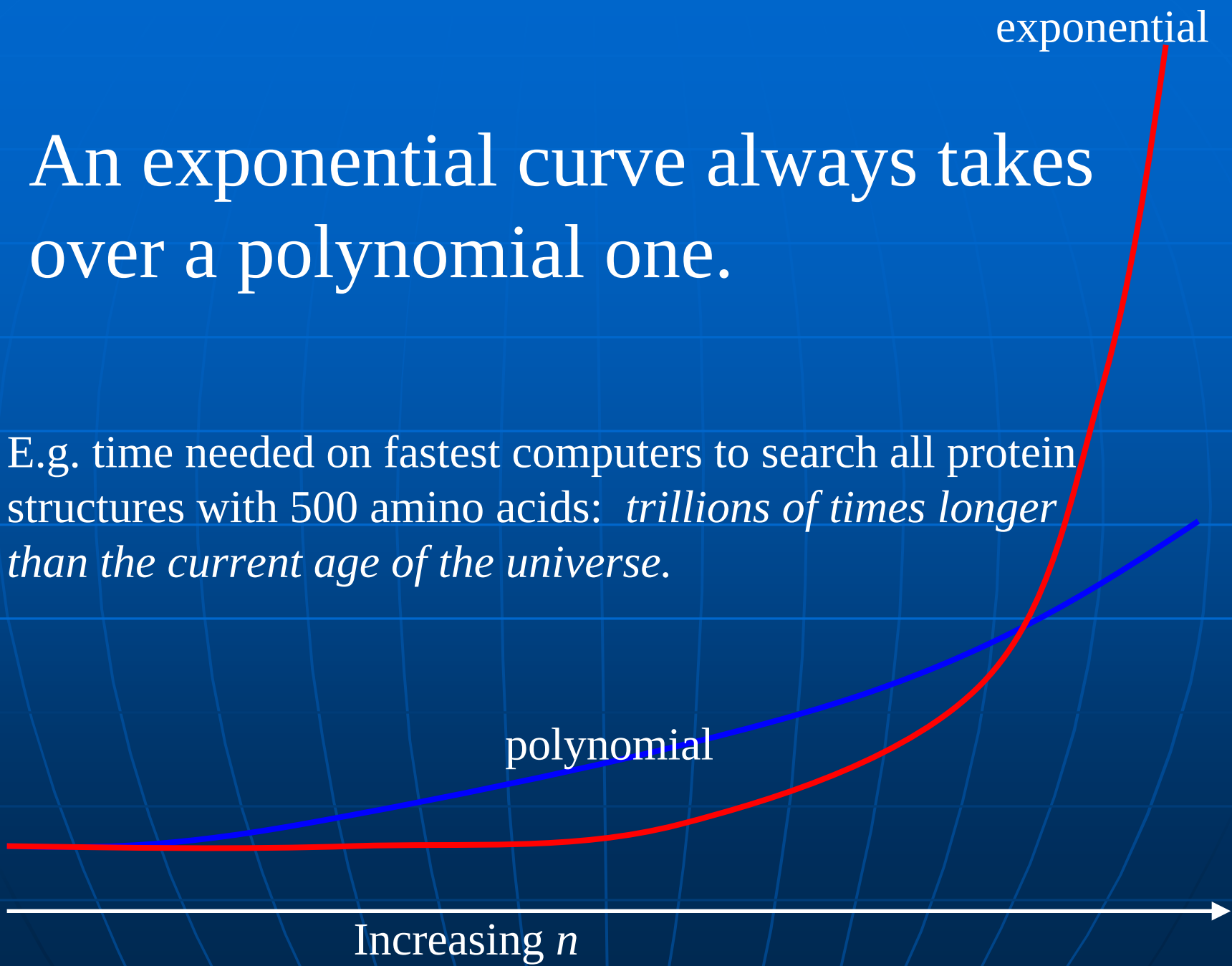
Hard and Easy Problems

***Polynomial Complexity*:** these are called *tractable*, and *easy* problems. Fast algorithms are known which provide the best solution. Pairwise alignment is one such problem. Sorting is another.

Exponential Complexity: these are called *intractable*, and *hard* problems. The fastest known algorithm which *exactly* solves it is usually not significantly faster than exhaustive search.

An exponential curve always takes over a polynomial one.

E.g. time needed on fastest computers to search all protein structures with 500 amino acids: *trillions of times longer than the current age of the universe.*

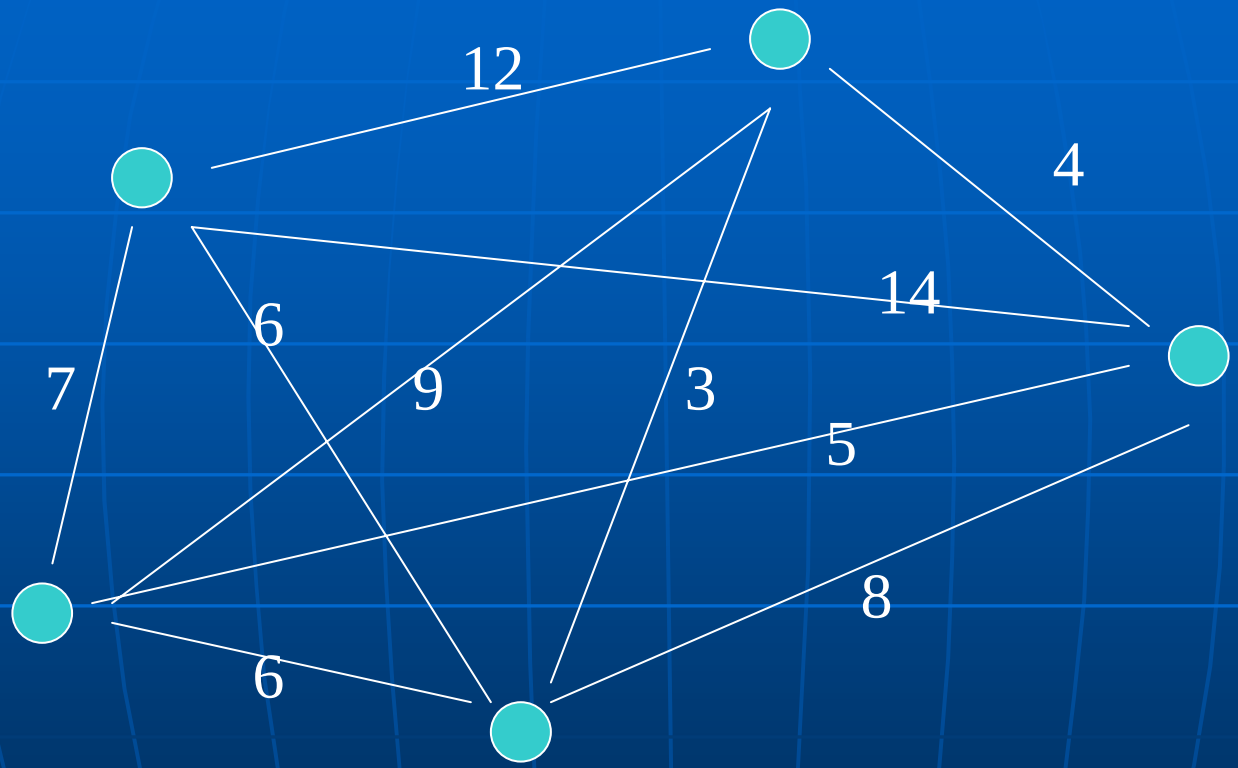


Example: Minimum Spanning Tree Problems

Find the cheapest tree which connects all (i.e. spans) the nodes of a given graph.

Applications: Comms network backbone design; Electricity distribution networks, water distribution networks, etc ...

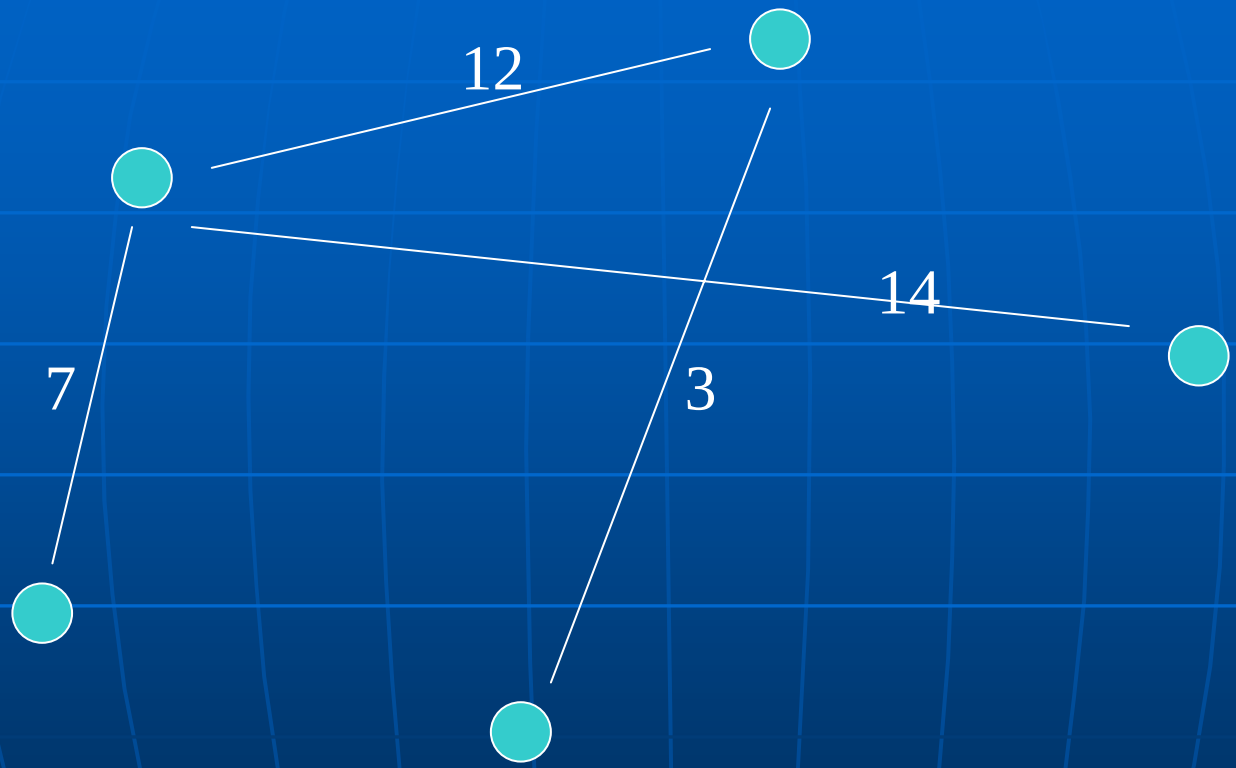
A graph, showing the costs of building each pair-to-pair link



What is the minimal-cost spanning tree?

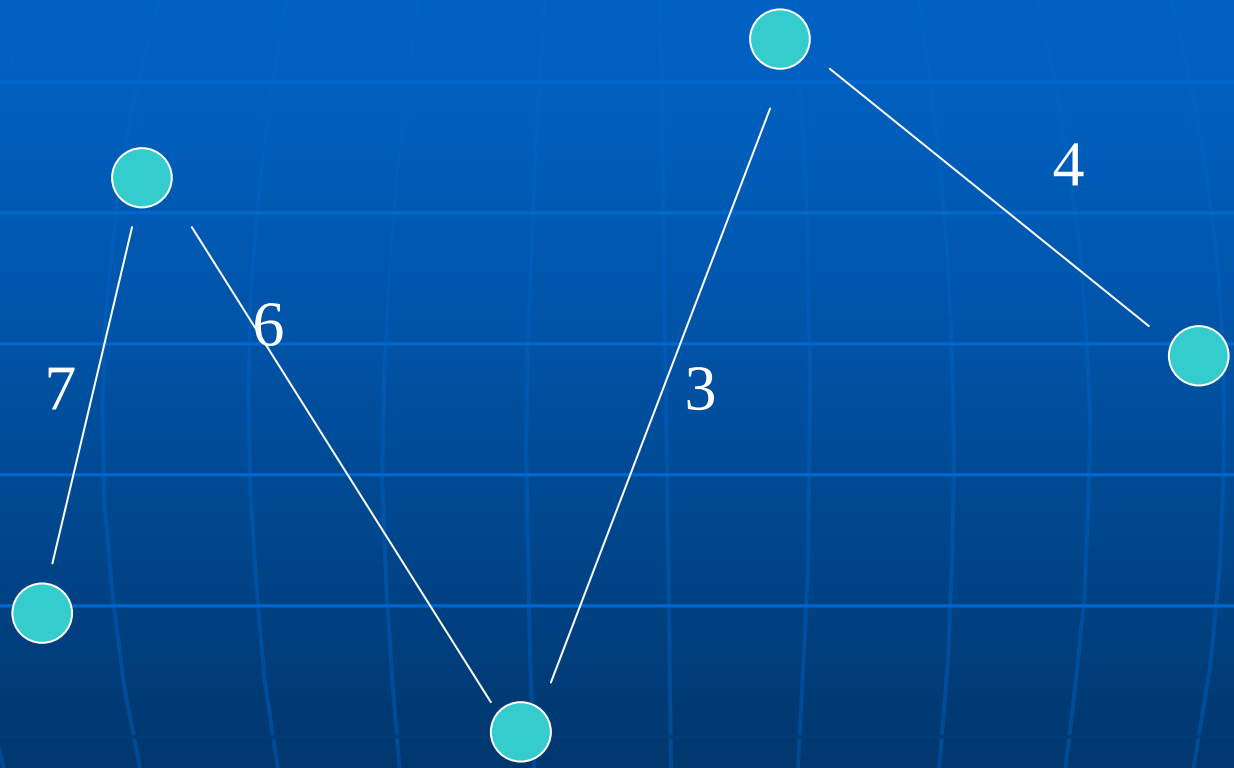
(Spanning Tree = visits all nodes, has no cycles; cost is sum of costs of edges used in the tree)

Here's one tree:



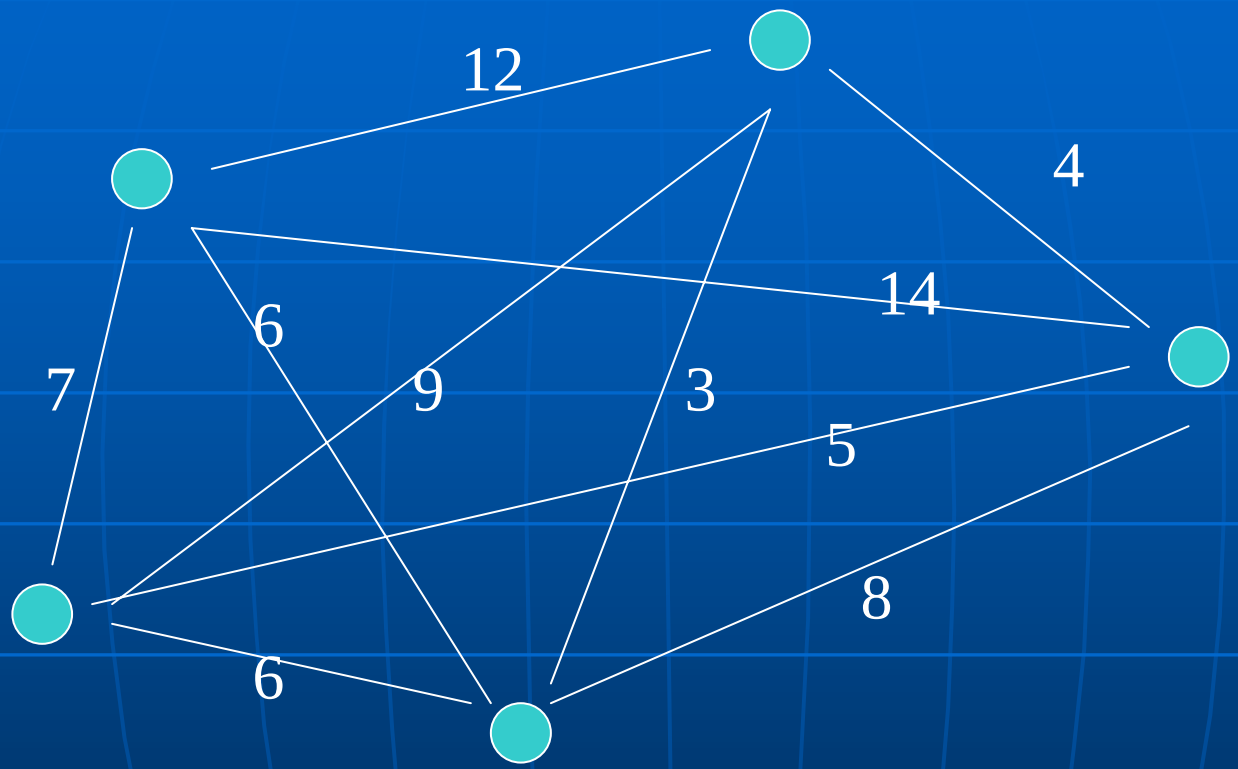
With cost = 36

Here's a cheaper one



With cost 20 …

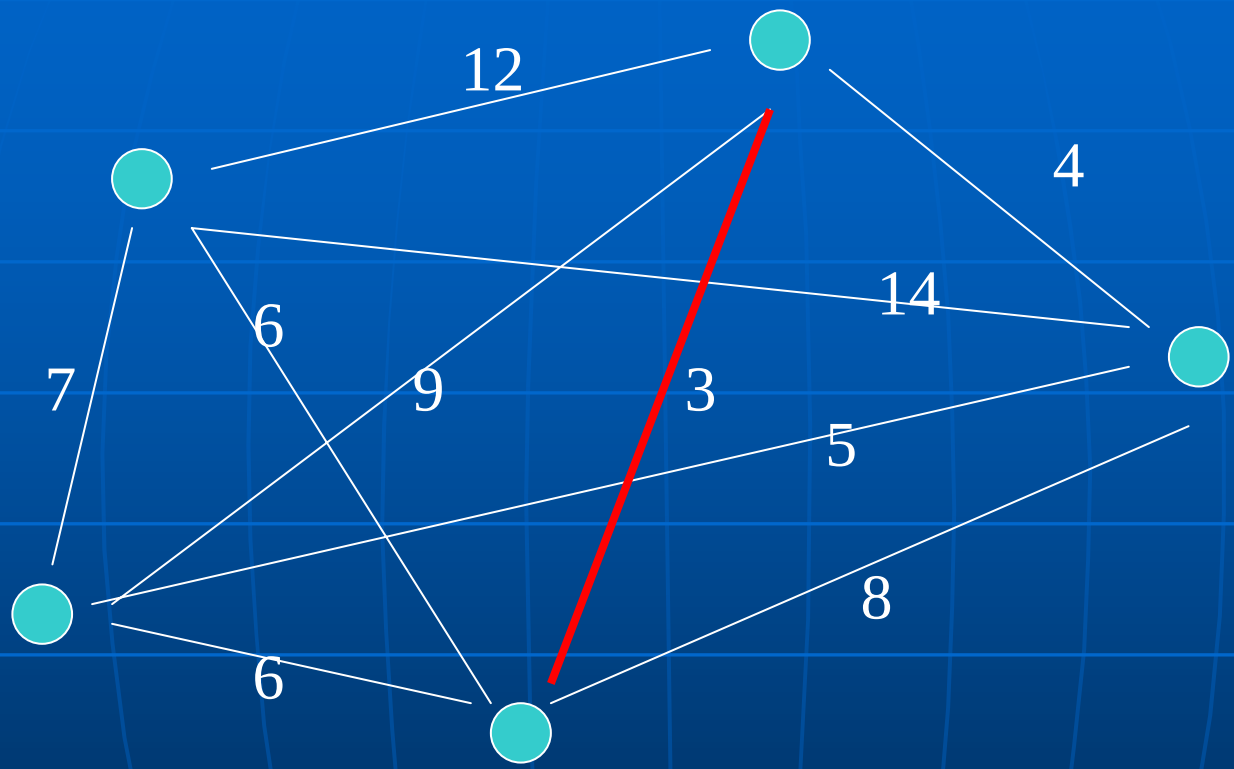
The problem *find the minimal cost spanning tree* (aka the `MST') is *easy* in the technical sense.



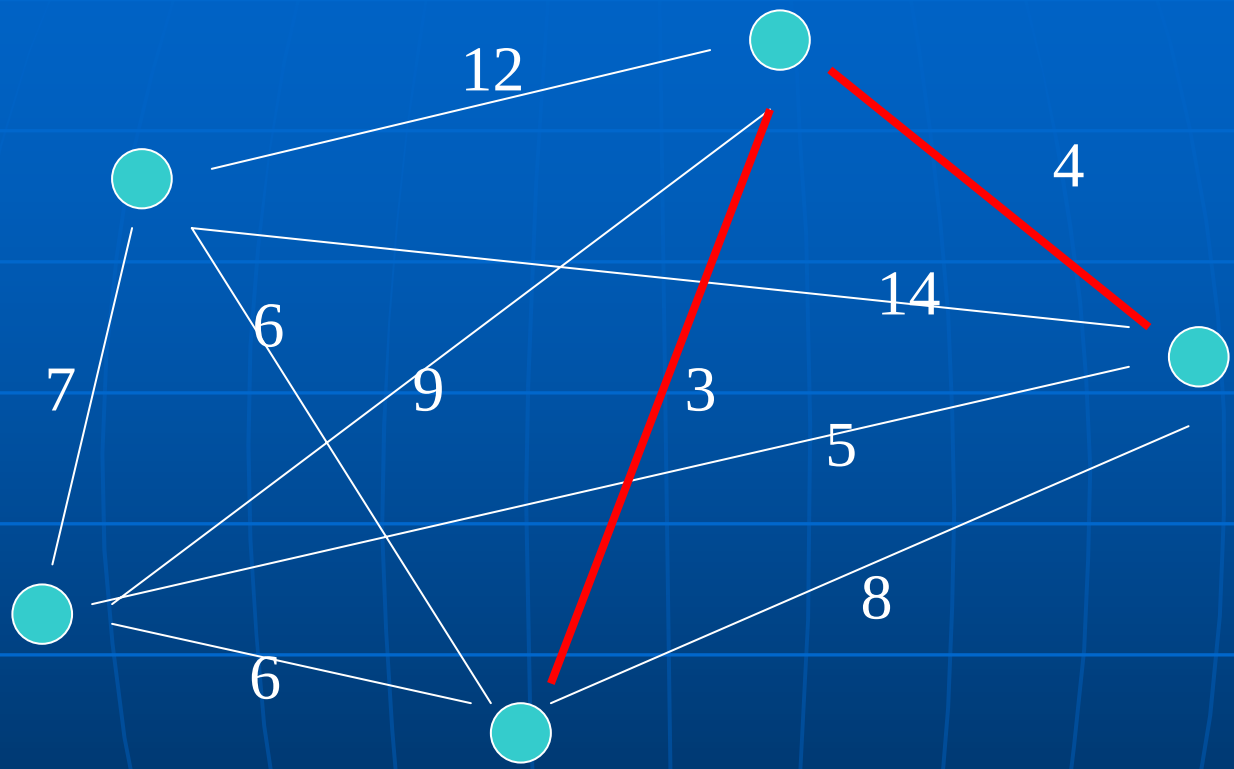
Several fast algorithms are known which solve this in polynomial time;
Here is the classic one: Prim's algorithm:

```
Start with empty tree (no edges)
Repeat: choose cheapest edge which feasibly extends the tree
Until:   n – 1  edges have been chosen.
```

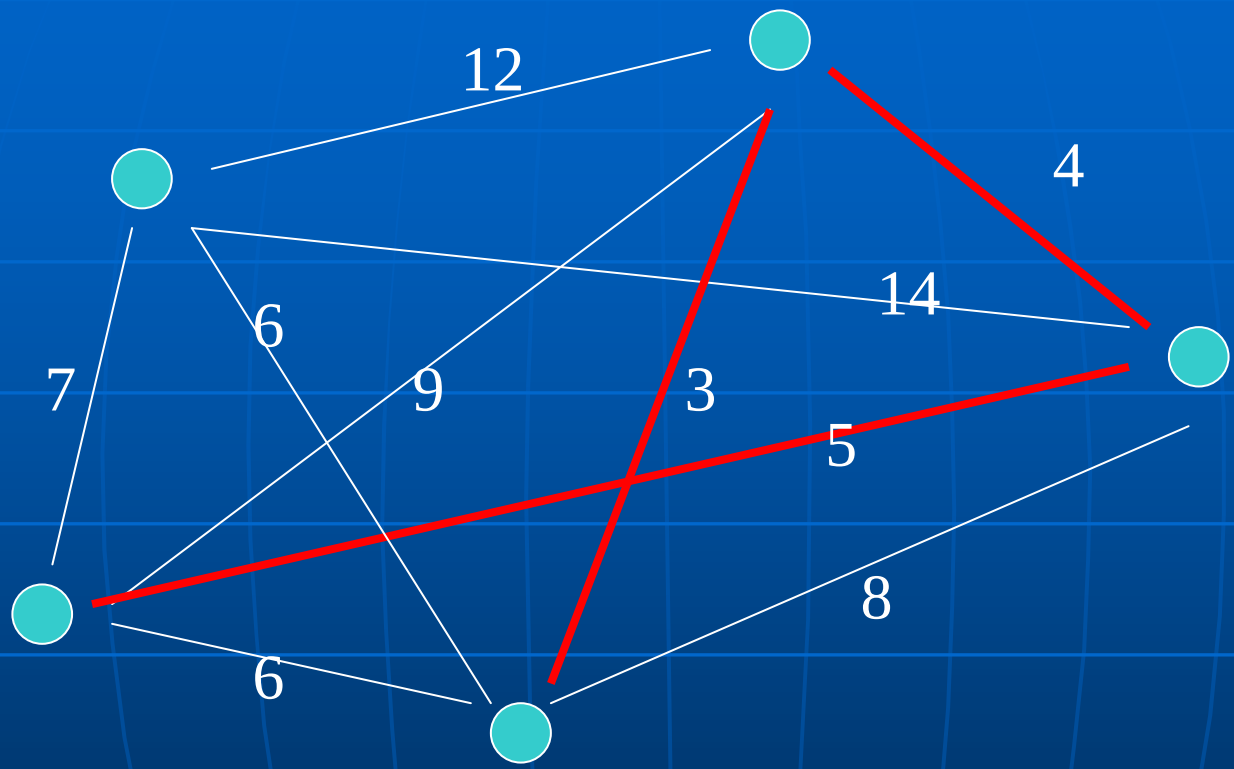
Prim’s step 1:



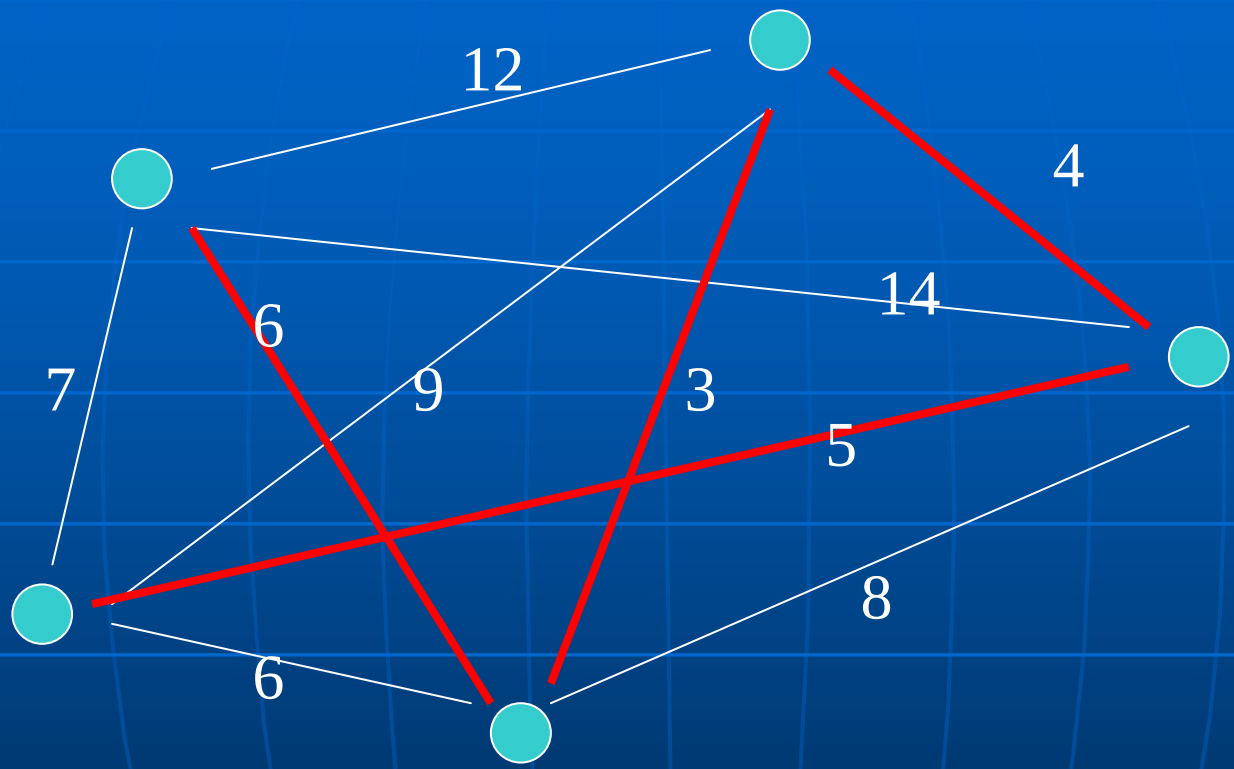
Prim's step 2:



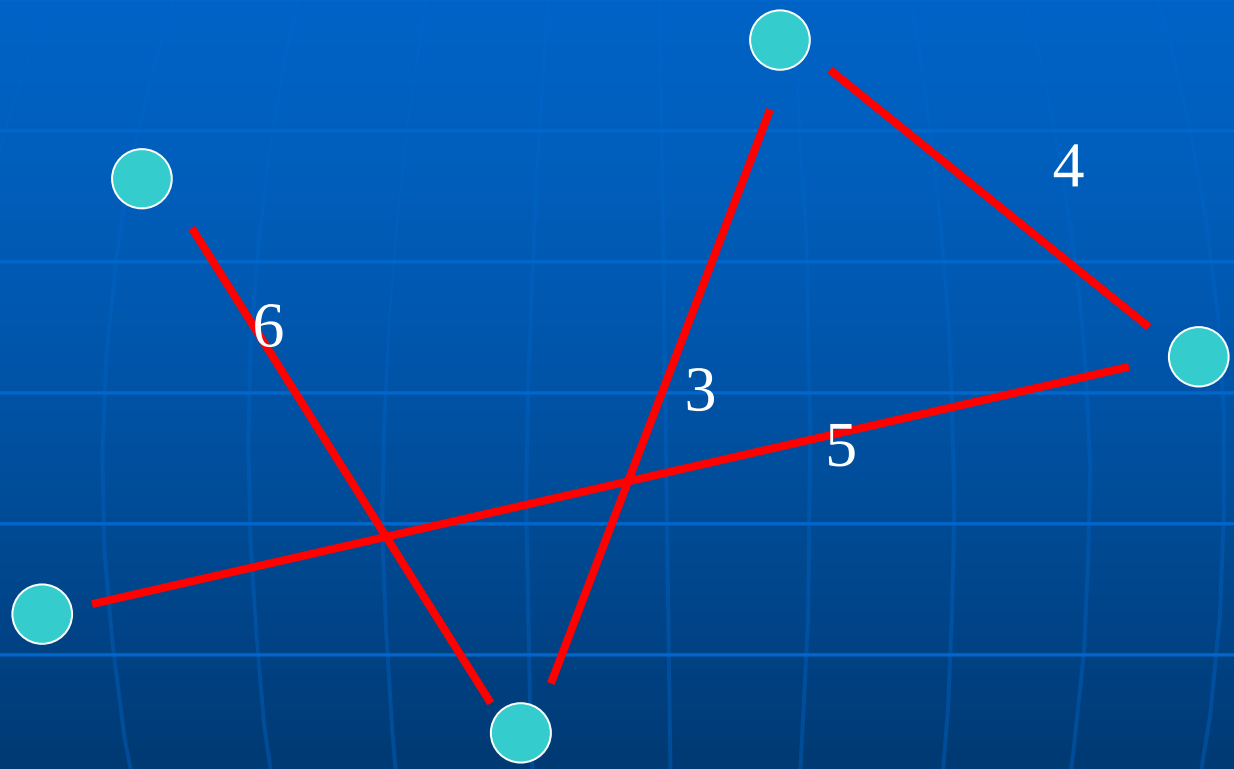
Prim's step 3:



Prim's step 4:



Prim's step 4:



Guaranteed to have minimal possible cost for this graph; i.e. this is the (or a) MST in this case.

But change the problem slightly:

We may want the **degree** constrained – MST (I.e. the MST, but where no node in the tree has a degree above 4)

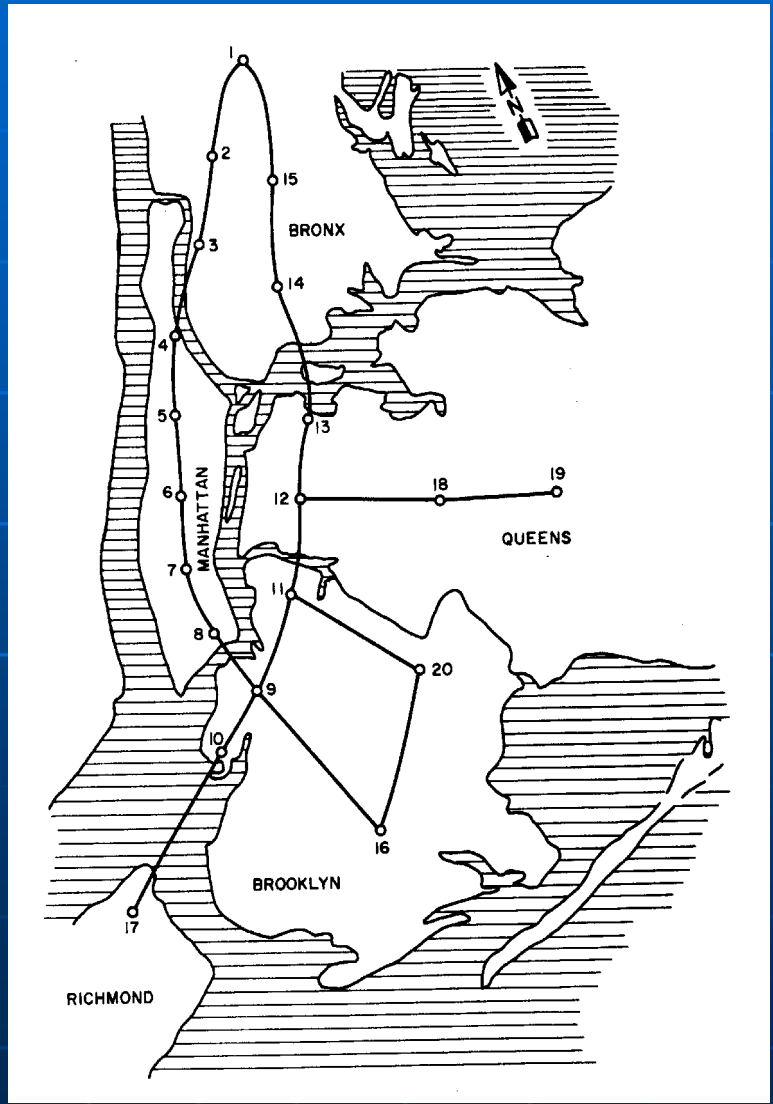
Or we may want the optimal communication spanning tree – which is the MST, but constrained among those trees which satisfy certain bandwidth requirements between certain pairs of nodes

There are many constrained/different forms of the MST. These are essentially problems where we seek the cheapest tree structure, but where many, or even most, trees are not actually feasible solutions.

Here's the thing: *These constrained versions are almost always technically hard*. a**nd** *Real-world MST-style problems are invariably of this kind.*

Real World Problems

- Tend to be hard
- New York Tunnels (highly simplified) WDN optimisation problem
  - 21 pipes
  - 16 possible diameters
  - How many potential solutions?



Well….

- Number of possible solutions = 16^{21}

Or….

$1.93 * 10^{25}$

Or….

19,342,813,113,834,066,795,298,816

Approximate Algorithms

For **hard** optimisation problems (again, which turns out to be nearly all the important ones), we need *Approximate algorithms* .

These:

- deliver solutions in reasonable time
- try to find pretty good (`near optimal') solutions, and often get optimal ones.
- do not (cannot) *guarantee* that they have delivered the optimal solution.

Typical Performance of Approximate Methods

Evolutionary Algorithms turn out to be the most successful and generally useful approximate algorithms around. They often take a long time though – it's worth getting used to the following curve which tends to apply across the board.