Introduction to NP

Liu Ziwei

References: Baase & Van Gelder 13.1, 13.2, 13.4

Outline

1. P and NP problems
2. NP-Completeness
3. Greedy heuristic algorithms for TSP
4. Greedy heuristic algorithm for Knapsack

P and NP problems

Hard problems are those that the best-known algorithm for the problem is expensive in running time.

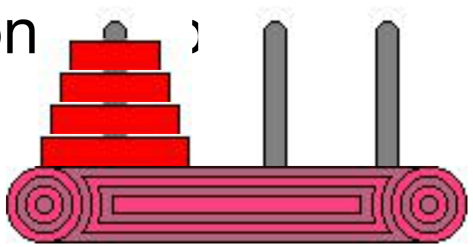
- It is not a problem that is hard to understand
- It is not a problem that is hard to code
- It is a problem that takes exponential time or more to compute – not practical to obtain a solution for a problem of a modest size n

It is important to know when a problem is hard.

- We should then focus on finding an efficient algorithm to obtain a good solution instead of an optimal solution

Problem: Towers of Hanoi

Compute a sequence of one-disk moves to transfer N disks from the first pole to the second pole. A third pole can be used in the process. Constraints: only one disk can be moved at a time and no disk is ever placed on



smaller one.

```
void TowersOfHanoi(int n, int x, int y, int z)
{
   if (n == 1)
       cout << "Move  disk from  " << x << " to  " << y << endl;
   else {
      TowersOfHanoi(n-1, x, z, y);
      cout << "Move  disk from  " << x << " to  " << y << endl;
      TowersOfHanoi(n-1, z, y, x);   }
}
```

The number of disk moves/lines to print:

$M_1 = 1;$

$M_n = 2M_{n-1} + 1$

Solution for the recurrence: $M_n = 2^n - 1$

For the computer program to solve this problem, the number of lines printed for N disks is $2^N - 1$. So with 64 disks it would print $2^{64} - 1$ lines. Assuming the printer can print 1 thousand lines a second, the program would require 140,000 hours to complete!!

What causes this lengthy processing time in Tower of Hanoi? The output?

A decision problem is an algorithmic problem that has two possible answers: yes, or no. The purpose is merely to decide whether a certain property holds for the problem's input.

Examples:

1. Can we travel from city A to city B within k hours?
2. Can we travel from city A, visit every other city exactly once and return to city A in k hours?
3. Is it possible to supply electricity to all homes in an area by a network of power lines of less than k kilometers?
4. Is there a subset of the n objects that fits in the knapsack of capacity weight C and returns a total profit of at least k?

An optimization problem is a problem of finding the best solution from all feasible solutions.

Examples:

1. What is the shortest path from city A to city B?
2. What is the shortest path to travel from city A, visit every other city exactly once and return to city A?
3. What is the network of minimum-length power lines to supply electricity to all homes in an area?
4. What is the maximum profit if a subset of the n objects are put into a knapsack of capacity weight C?

- In general, each decision problem has its corresponding optimization problem.
- Each optimization problem can be recast as a decision problem.
- If we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard.

$\boxed{\text{It is easy to solve the optimization problem}} \Rightarrow \boxed{\text{It is easy to solve the decision problem}}$

$\boxed{\text{It is not easy to solve the decision problem}} \Rightarrow \boxed{\text{It is not easy to solve the optimization problem}}$

Problems that are not hard – P problems

Definitions:

An algorithm is said to be polynomially bounded if its worst case complexity is bounded by a polynomial function of the input size.

A problem is said to be polynomially bounded if there is a polynomially bounded algorithm for it.

The class P problems is a class of decision problems that are Polynomially bounded.

Examples of **P** problems:

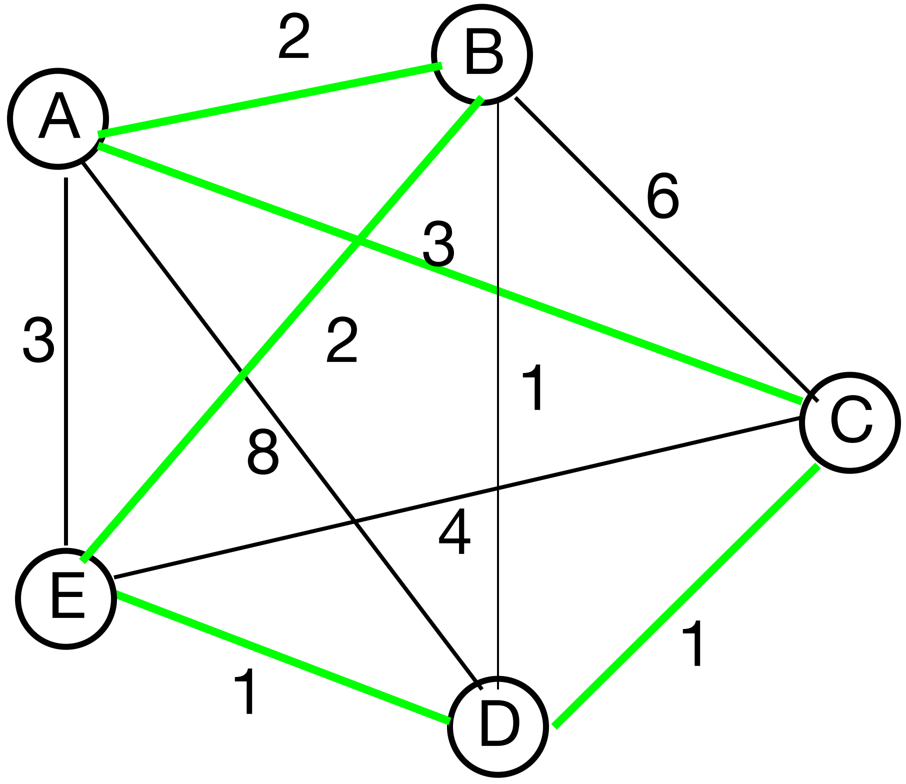
1. Can we travel from city A to city B within k hours?
2. Is it possible to supply electricity to all homes in an area by a network of power lines of less than k kilometers?

How about the problem of finding if we can travel from city A, visit every other city exactly once and return to city A in k hours?

The corresponding optimization problem is known as the <u>travelling salesman problem.</u>

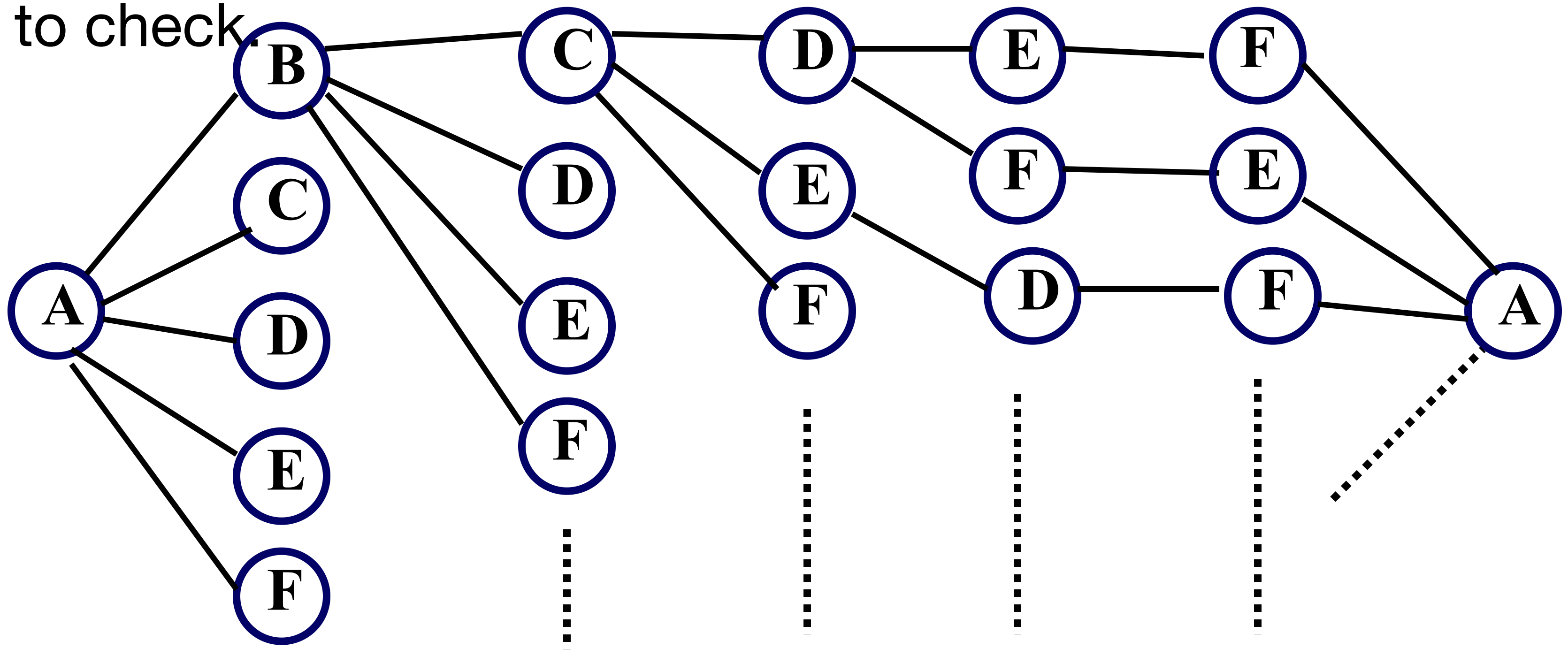
Travelling Salesman Problem

Example:
Is there a tour of all the cities with total cost of not more than 12?

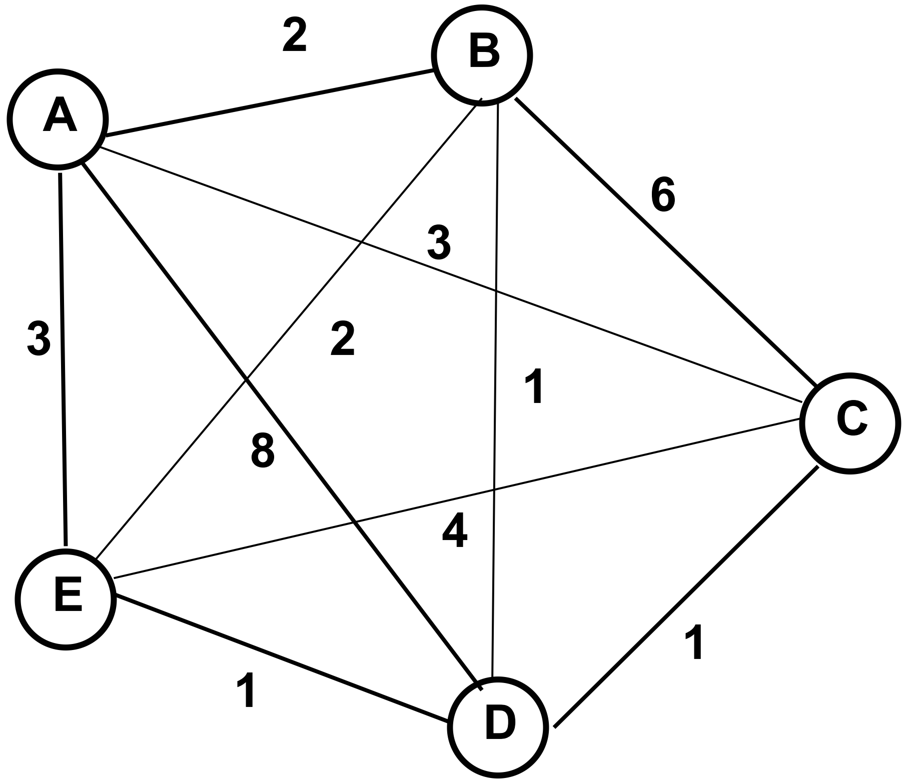


How do we find the answer to this decision problem?

Consider all possible tours, terminating after we find a tour that costs no more than 12. We have potentially 120 tours to check.



The worst case time complexity is O(N!). If N = 25, N! is a 26 digit number!! There is no known polynomial time algorithm.



Another characteristic of this problem:

Make a guess and check:
(1) route ABCDEA - total is 13 – answer is ‘no’.
(2) route ABDCEA - total is 11 – answer is ‘yes’
Problem solved.
May not get the best route which is ABEDCA - cost of 9

How do we check a solution: (1) it is a correct route (2) total length of the route. <u>The checking can be done in O(n^2) time</u>!

0/1 Knapsack problem

We have a knapsack of capacity weight C (a positive integer) and n objects with weights w_1, w_2, …w_n and profits p_1, p_2, …p_n (all w_i and all p_i are positive integers), is there a subset of the objects that fits in the knapsack and returns a total profit of at least k?

- There is no known polynomial time algorithm.
- There are 2^n subsets of n objects: in the worst case all subsets need to be examined which takes $O(2^n)$ time.
- Given a guess of a subset, it takes O(n) time to check whether it satisfies the requirements.

The dynamic programming algorithm has a complexity of O(nC).
It is pseudo-polynomial time

Example: C = 20, is there a subset of objects that can be put into the knapsack to make a profit of at least 21?

	1	2	3	4
w_i	4	6	8	6
p_i	7	6	9	5

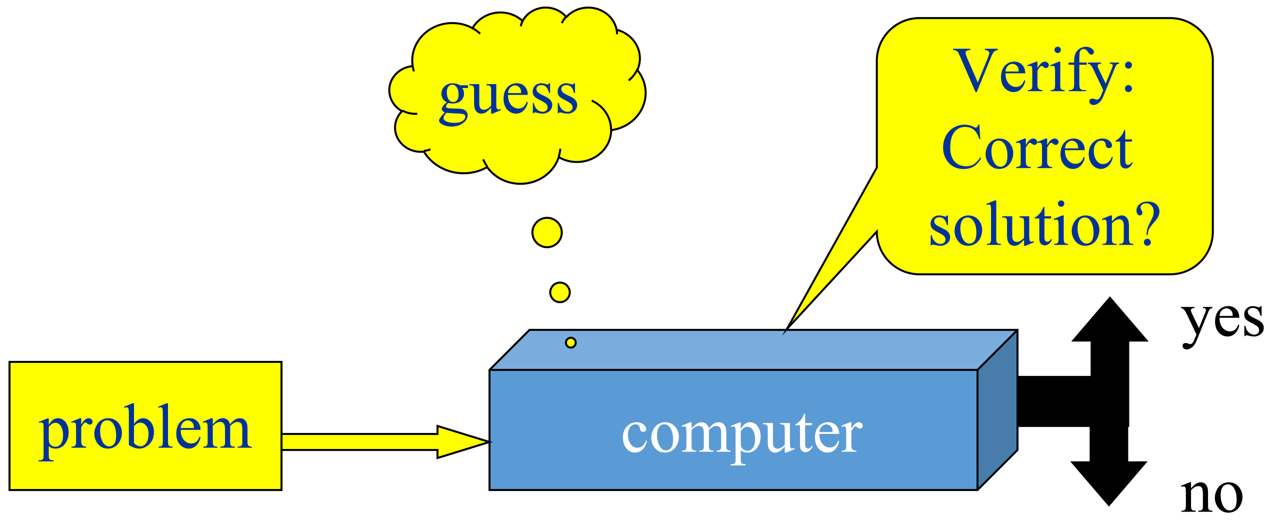
Make a guess and check: Subset: {1, 3, 4}.

Total weight = $w_1 + w_3 + w_4 = 18$.

Profit = $p_1 + p_3 + p_4 = 21$

Answer 'yes'. (optimal is 22)

- There is a class of problems for which
  - There is no known polynomial time algorithms
  - Checking of a solution can be done in polynomial time



A nondeterministic algorithm solves a problem in two phases, ‘guess’ then ‘verify’, and an output step, ‘yes’ or ‘no’ output.

A nondeterministic algorithm can also be considered as a machine that checks all possible solutions in parallel to see which is correct.

NP problems

Definition:

NP [**N**ondeterministic **P**olynomially bounded] : is the class of decision problems for which there is a polynomially bounded nondeterminitic algorithm. i.e. it is a problem that can be solved in polynomial time on a nondeterministic machine.

Alternatively, we say that the complexity class NP is the class of problems that can be verified by a polynomial time algorithm.

- $P \subseteq NP$. Every class P problem is also a class NP problem.
- There is widespread belief that $P \neq NP$.

- Examples of NP problems
  - Travelling salesman problem (a tour of n cities that has a total cost less than k)
  - Knapsack problem
- An example of non-NP problems: A variant of the travelling salesman problem is:

  Given a network of cities G and a number k. Does every tour have total cost of not more than k?

  - We cannot verify a solution in polynomial time.
  - In fact, it is a provable exponential time problem.

NP-Completeness

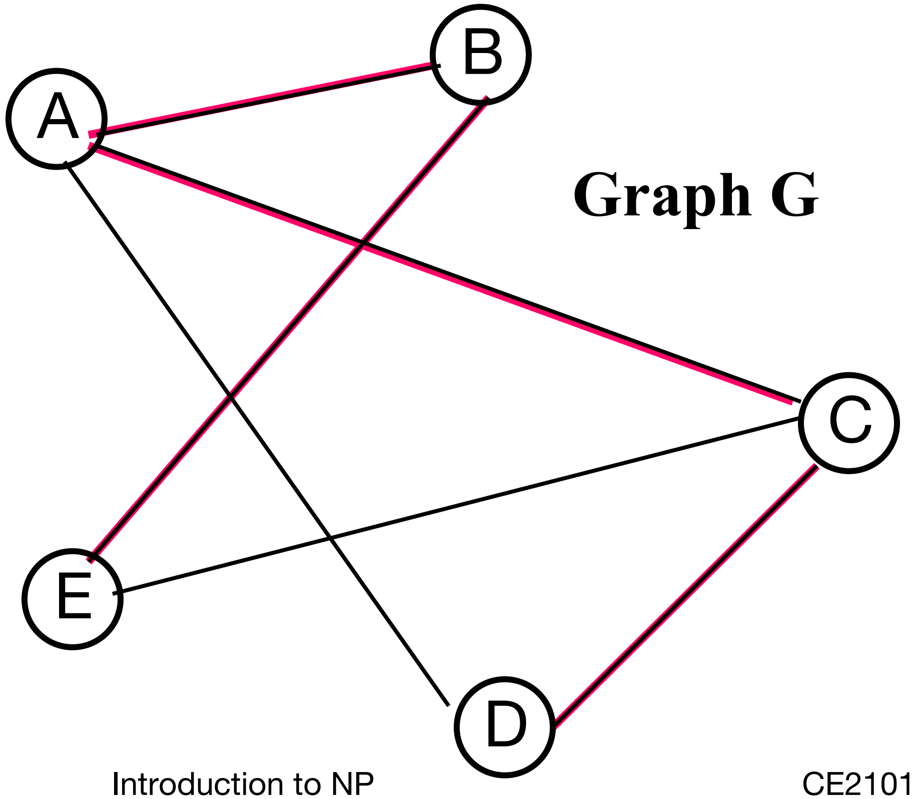
Problem Reduction

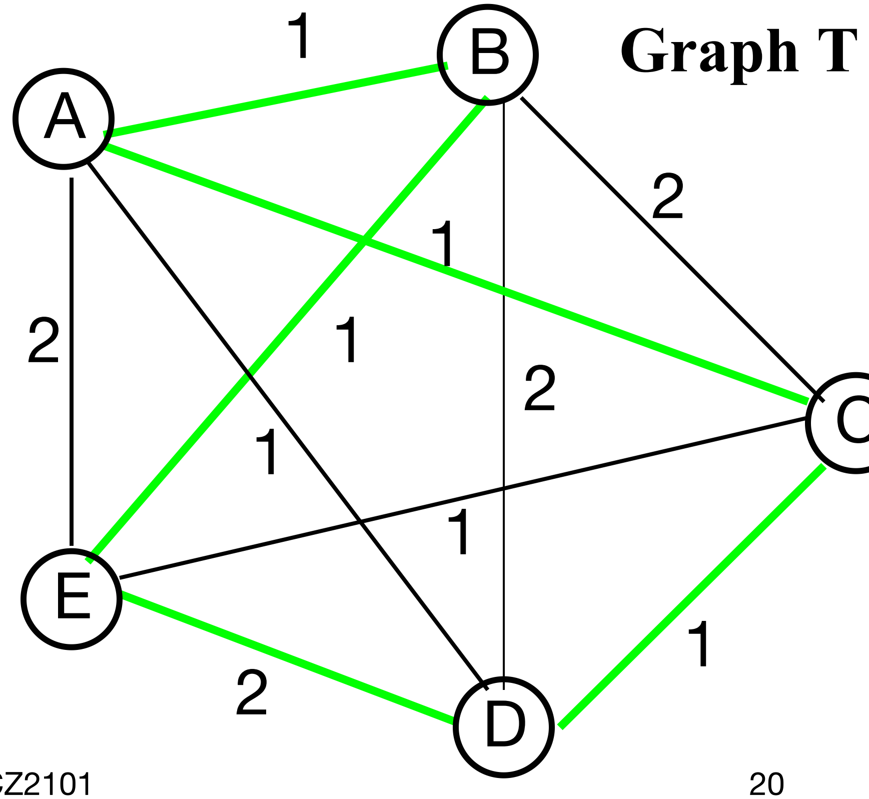
- How do we show that a problem is hard? We reduce a known hard problem to this problem
- A reduction is a way of converting one problem P_1 to another problem P_2
- If P_2 has an algorithm, reduction gives us a way to solve P_1.
- It also means that P_1 is no harder than P_2, i.e. P_2 is as hard as P_1
- Example: FactorAll(n) finds all the factors of a number n. Factor1(n) finds one factor.
  - FactorAll() reduces to Factor1().
  - We use Factor1(n) to find one factor m of n. Then divide n by m, and run FactorAll on the result. Keep repeating, and we get all the factors.

- Another example of reduction

  Hamiltonian path problem:

  Given a graph G, is there a path, any path, that passes through all the vertices of the graph exactly once?





The transformed problem: Does graph T, have a travelling salesman tour that is no longer than N+1, where N is the number of vertices in G?

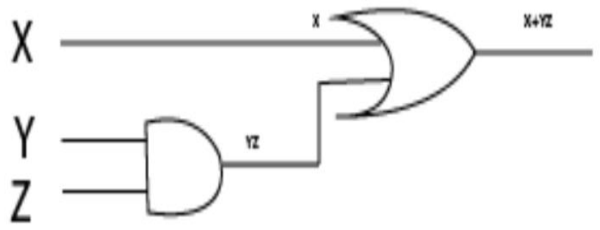
- o The answer to the Hamiltonian path problem is “yes” precisely when the answer to the TSP is “yes”.
- o The transformation can be done <u>in polynomial time</u>.
- o So if TSP has a polynomial time algorithm, Hamiltonian path problem can also be solved in polynomial time!

- Reducibility plays an important role in classifying problems.
- The point of reducibility of P1 to P2 is that P2 is at least as ‘hard’ to solve as P1.

NP-Completeness

Definition: A problem D is NP-complete if it is in NP and every problem Q in NP is reducible to D in polynomial time.

The circuit satisfiability problem (CIRCUIT-SAT) is the decision problem of determining whether a given Boolean circuit has an assignment of its inputs that makes the output true.



SAT = satisfiable Boolean formulas.
Given a Boolean formula, is there any setting for the variables which makes the Boolean formula true?
$(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee \neg B \vee \neg C \vee \neg D)$

Setting A=B=C=true, D=false makes the formula true.

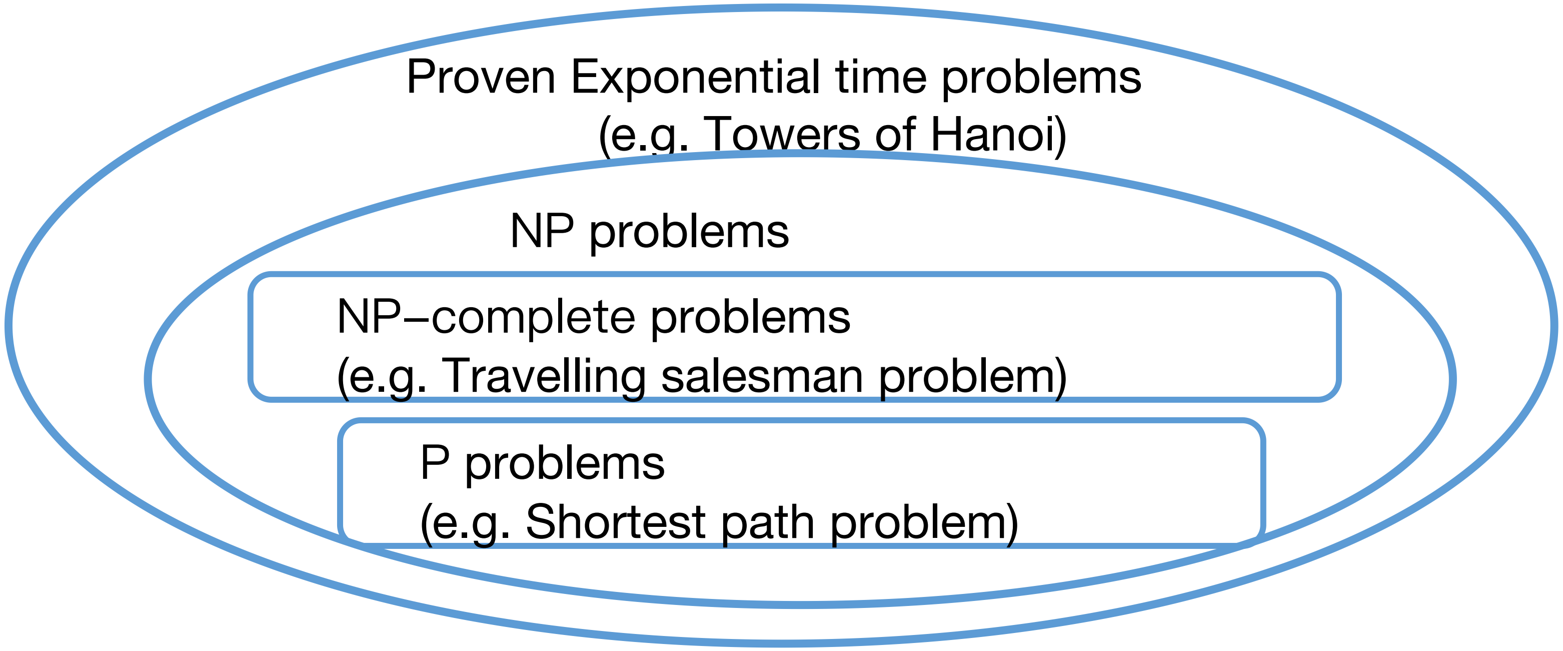
- Cook and Levin proved that CIRCUIT-SAT is NP-complete.
- This means every NP problem can be solved by reducing it to CIRCUIT-SAT.
- CIRCUIT-SAT can be reduced to SAT and SAT is in NP. So SAT is NP-complete.
- All the problems below and many others are NP-Complete.

$$\begin{tikzcd}
 & \text{CIRCUIT-SAT} \arrow[d] & \\
 & \text{SAT} \arrow[d] & \\
 & \text{3-CNF-SAT} \arrow[dl] \arrow[dr] & \\
\text{CLIQUE} \arrow[d] & & \text{SUBSET-SUM} \\
\text{VERTEX-COVER} \arrow[d] & & \\
\text{HAM-CYCLE} \arrow[d] & & \\
\text{TSP} & &
\end{tikzcd}$$

Note that reducibility is transitive.

- NP-complete problems are equal to each other in difficulty and they are the hardest problem in NP.
- So if we want to show that a problem is hard, prove it is a NP_complete problem.
- If anyone finds a polynomial time solution to one of these NP-complete problems, by a series of reductions, every other problem that is in NP can also be solved in polynomial time !! Then P = NP.

Classification of problems

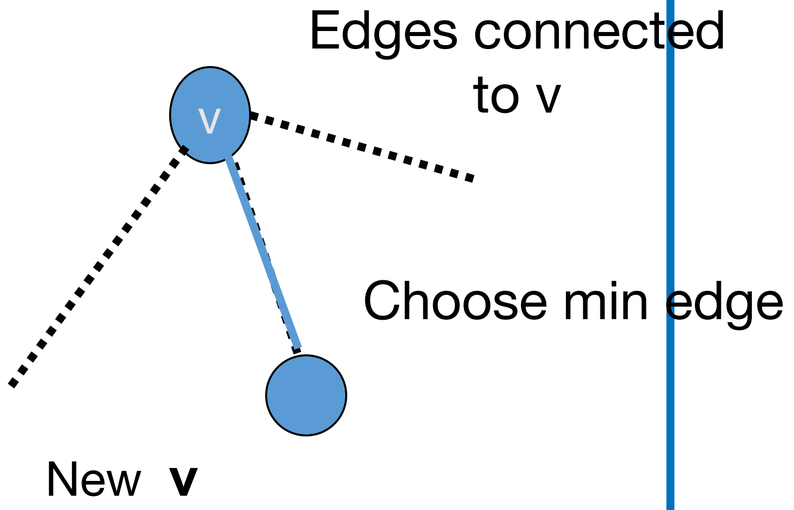

Proven Exponential time problems
(e.g. Towers of Hanoi)
NP problems
NP−complete problems
(e.g. Travelling salesman problem)
P problems
(e.g. Shortest path problem)

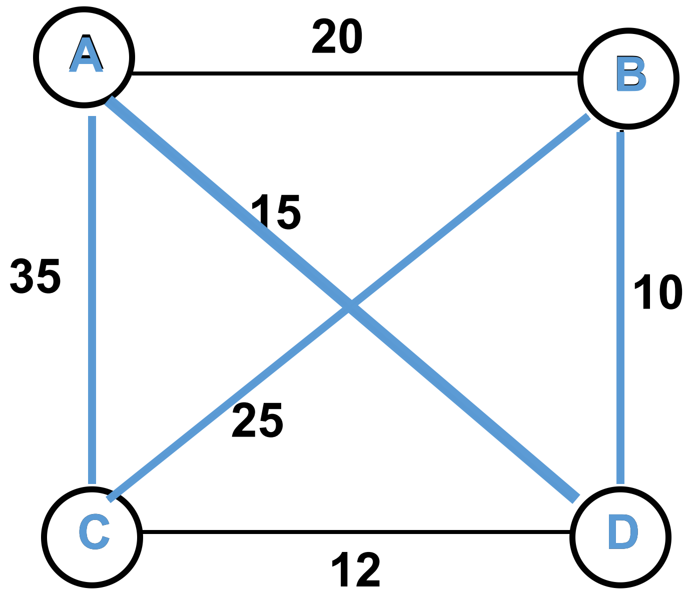
With so many NP- complete problems that are important applications, how do we solve them?

- Use small problem sizes
- Solve for a special instance of the problem
- heuristic algorithms
    - These are fast (polynomially bounded) algorithms
    - not guaranteed to give the best solution
    - will give one that is close to the optimal in many cases
    - But could return a very bad solution

Greedy heuristic algorithm for TSP – Nearest Neighbou

```
nearestTSP(V, E, W)
{   Select an arbitrary vertex s to start the cycle C
     v = s;
     while there are vertices not in C  {
        select an edge vw of minimum weight
            where w is not in C
        add edge vw to C;
        v = w;    }
    add edge vs to C;
    return C;     }
```





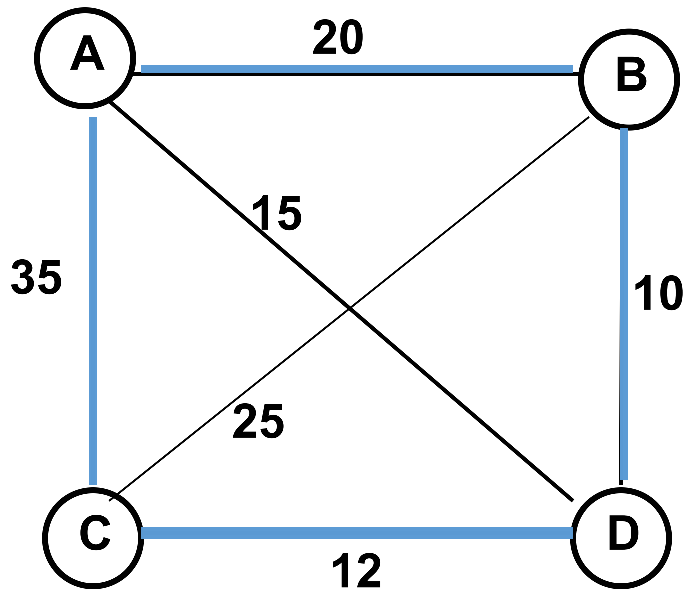
Starting at A, nearest-neighbour strategy results in:

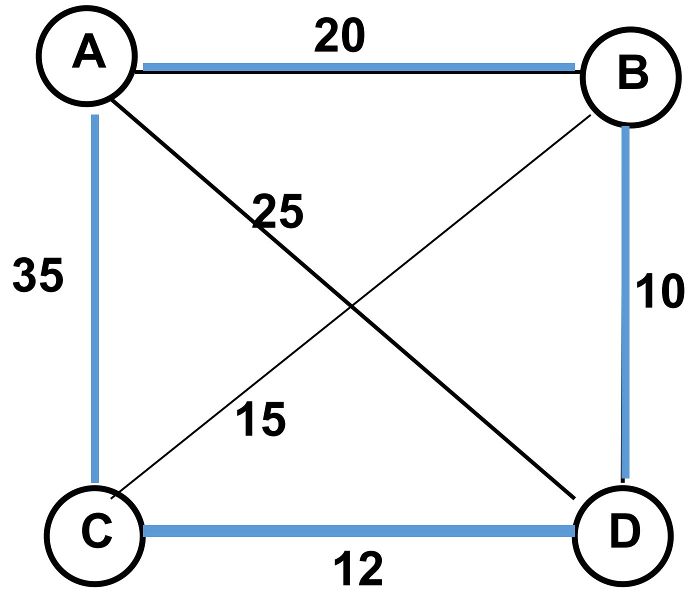
Total distance of 85.
This is NOT the minimum.

- Example of a greedy strategy that does not always give an optimum solution
- Will result in a very bad solution if the weight of AC is, e.g, 10,000

Greedy heuristic algorithm for TSP – Shortest Li

```
shortestLinkTSP(V, E, W)
{   R = E;
     C = empty;   // C is a forest
     while (no. of edges in C < |V| - 1)  {
        remove the lightest edge vw from R;
        if (vw does not form a cycle in C and vw
            would not be the third edge in C
            incident on v or w)
          add edge vw to C;   }
    add edge connecting the end points to C;
   return C;     }
```





Using shortestLinkTSP, edges chosen are: BD,DC,AB,CA. This solution has total distance of 77.
Though better than the nearest-neighbour, it is still not optimal.
Optimal is A→D→C→B→A [72]

Greedy heuristic algorithm for Knapsack

```
Function greedy-knap(w[1..n], p[1..n], C)
  Sort the list of objects in descending order of  p[i]/w[i];
  initialize weight and value of knapsack to 0;
  for i = 1 to n
        if (weigth+w[i] ≤ C)
                value + = v[i];
                weight += w[i];
  return value;
```

Example: C = 20

	1	2	3	4
w_i	4	6	8	6
p_i	7	6	9	5

	1	2	3	4
p_i/w_i	7/4 = 1.75	1	9/8 = 1.125	5/6 = 0.83

Answer : choose objects 1, 3, 2 so subset = {1, 2, 3}, profit = 22

THE END