



**CS/COE 1501**  
**Algorithm Implementation**

**6-P vs NP**

Fall 2018

**Sherif Khattab**

**ksm73@pitt.edu**

**6307 Sennott Square**

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

# But first, something completely different...

- Some computational problems are *unsolvable*
  - No algorithm can be written that will always produce the correct output
  - One example is the *halting problem*
    - Given a program and an input, will the program halt?
      - Can we write an algorithm to determine whether any program/input pair will halt?

# Halting problem example

```
String test = “  
public boolean proof_sketch(String program) {  
    if (WILL_EVENTUALLY_HALT(program, program)){  
        while (true){}  
        return false;  
    }  
    else {  
        return true;  
    }  
}  
”;  
proof_sketch(test);
```

what can possibly happen?



# There are a number of other undecidable problems

- But the halting problem is all that we'll cover here

# Intractable problems

- Solvable, but require too much time to solve to be practically solved for modest input sizes
  - Listing all of the subsets of a set:
    - $\Theta(2^n)$
  - Listing all of the permutations of a sequence:
    - $\Theta(n!)$

# Polynomial time algorithms

- Most of the algorithms we've covered so far this term
  - Also the most practically useful of the three classes we've just covered...
- Largest term in the runtime is a simple power with a constant exponent
  - E.g.,  $n^2$
  - Or a power times a logarithm
    - E.g.,  $n \lg n$

# Consider the following

- The shortest path problem
  - Easily solved in polynomial time
- The longest path problem
  - How long would it take us to find the longest path between two points in a graph?

# What if a problem doesn't fall into one of our three categories?

- It can be solved
- There is no proof that a solution requires exponential time
  - ... yet
- There is no valid solution that runs in polynomial time
  - ... yet

# P vs NP

- P
  - The set of problems that can be solved by deterministic algorithms in polynomial time
- NP
  - The set of problems that can be solved by non-deterministic algorithms in polynomial time
    - I.e., solution from a non-deterministic algorithm can be verified in polynomial time

# Deterministic vs non-deterministic algorithms

- Deterministic
  - At any point during the run of the program, given the current instruction and input, we can predict the next instruction
  - Running the same program on the same input produces the same sequence of executed instructions
- Non-deterministic
  - A conceptual algorithm with more than one allowed step at certain times and which always takes the right or best step
    - Conceptually, could run on a deterministic computer with unlimited parallel processors
      - Would be as fast as always choosing the right step

# Non-deterministic algorithms

- Array search:
  - Linear search:
    - $\Theta(n)$
  - Binary search:
    - $\Theta(\lg n)$
  - Non-deterministic search algorithm:
    - $\Theta(1)$

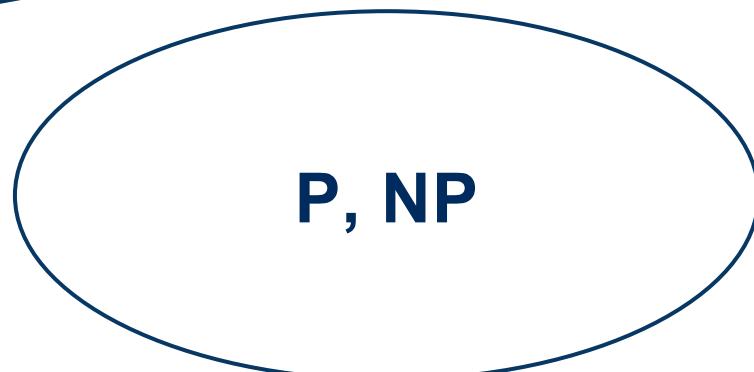
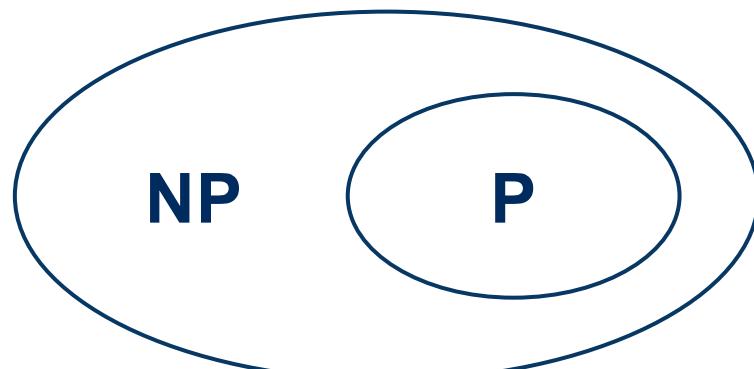
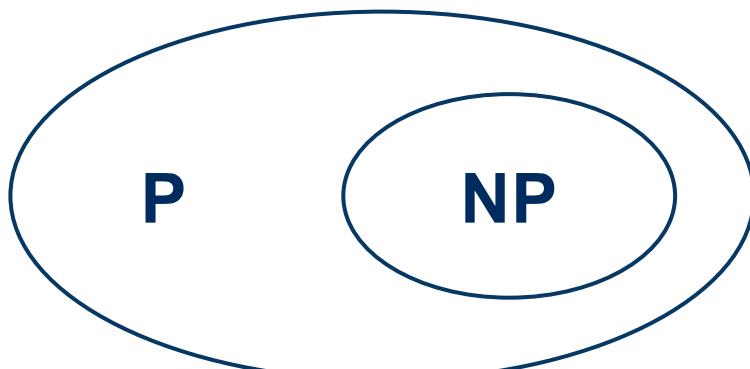
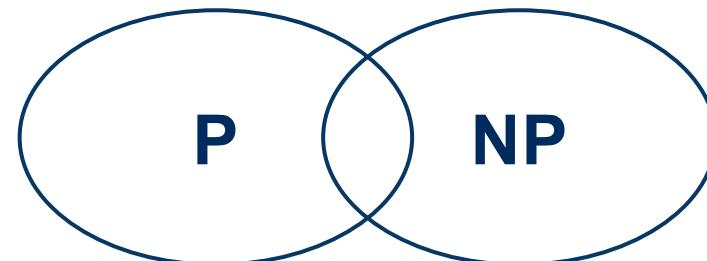
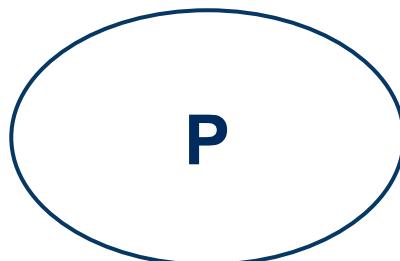


# Hamiltonian cycle problem

- A Hamiltonian cycle is a simple cycle through a graph that visits every vertex of the graph
- Can we determine if a given graph has a Hamiltonian cycle?
  - Yes, a brute-force deterministic algorithm would look at every possible cycle of the graph to see if one is Hamiltonian
    - How many possibilities for a complete graph?
      - $v!$
  - A non-deterministic algorithm would simply return a cycle
- Can we verify a result?
  - Yes! simply look through the returned cycle and verify that it visits every vertex
    - How long will this take?

# So we can group problems into P and NP...

- 5 options for how the sets P and NP intersect:



# Are any of these clearly impossible?

- Why?

# Remember how I kept saying “... yet”

- Either  $P \subset NP$  or  $P = NP$ 
  - One of the biggest unsolved problems in computer science
- Can prove that  $P \subset NP$  by:
  - Proving an NP problem to be intractable
- Can prove  $P = NP$  by:
  - Developing a polynomial time algorithm to solve an NP-Complete problem

# What if P = NP?

- Most widely-used cryptography would break
  - Efficient solutions would exist for:
    - Attacking public key crypto
    - Attacking AES/DES
    - Reversing cryptographic hash functions
- Operations research and management science would be greatly advanced by efficient solutions to the travelling salesman problem and integer programming problems
- Biology research would be sped up with an efficient solution to protein structure prediction
- Mathematics would be drastically transformed by advances in automated theorem proving

# What if P $\neq$ NP?

- meh
  - Mostly assumed to be the case

# OK, but wait...

- What exactly is NP-Complete?
  - That came out of nowhere on the last few slides
    - NP-Complete problems are the "hardest" problems in NP
      - They are all equally "hard"
        - So, if we find a polynomial time solution to one of them, we clearly have a polynomial time solution to all problems in NP

## Consider for a moment...

- You've just discovered a new computational problem
  - But you cannot find a polynomial time solution
    - If you can show that the problem is NP-Complete, you know that finding a polynomial time solution boils down to solving P vs NP

# Proving NP-Completeness

- Show the problem is in NP
- Show that your problem is at least as hard as every other problem in NP
  - Typically done by reducing an existing NP-Complete problem to your problem
    - In polynomial time

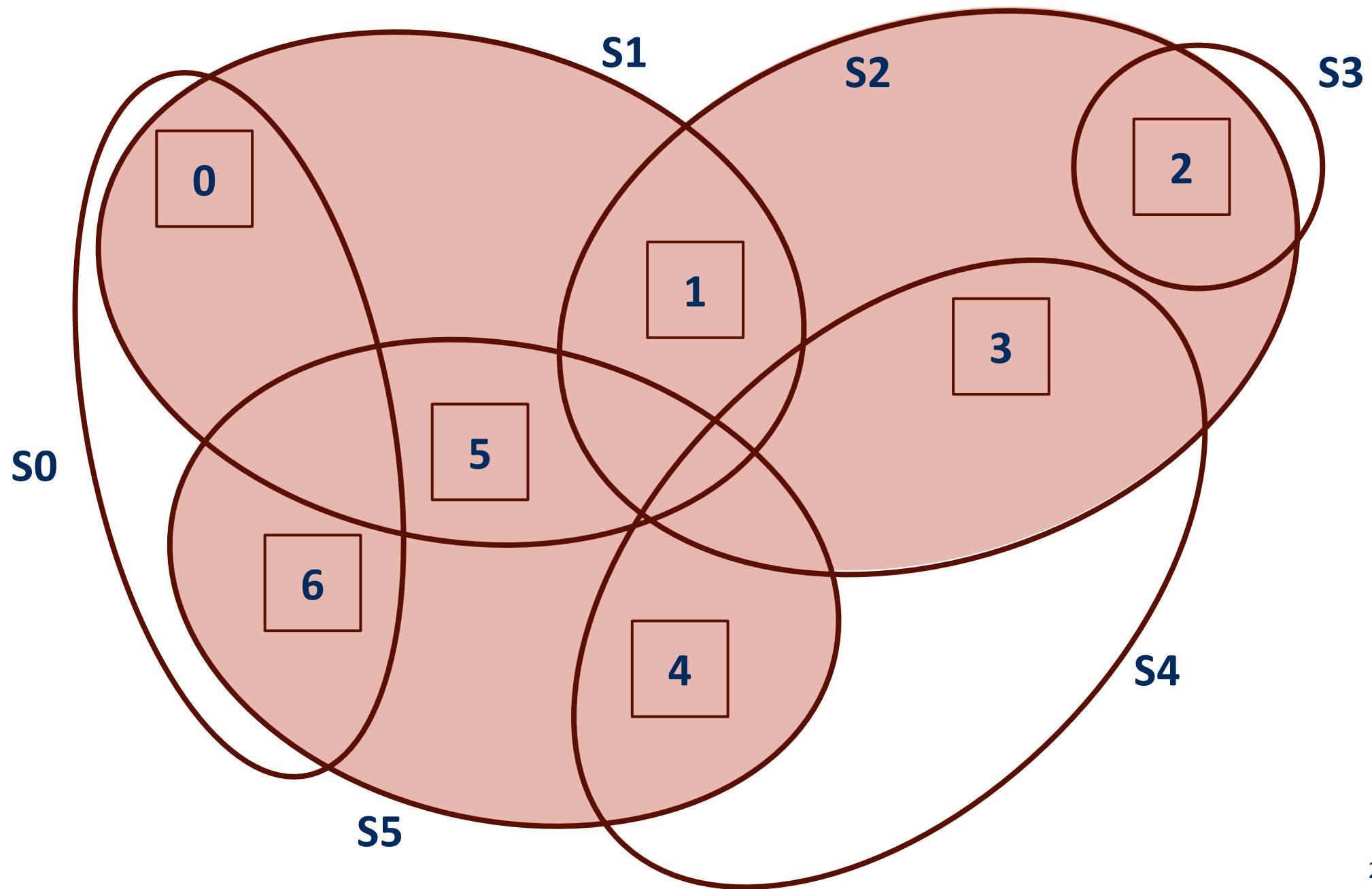
# Reduction to show NP-Completeness

- Goal: show that your problem can be used to solve an NP-Complete problem
  - And that the transformation of problem inputs can be performed in polynomial time
- Why does this work?
  - If your algorithm can solve an NP-Complete problem, then a polynomial time solution to your problem with a polynomial time transformation from the NP-Complete problem would mean a polynomial-time solution to an NP-Complete problem

# Reduction example

- Assume we've just come up with the set cover problem:
  - Given a set  $S$  of elements and a collection  $s_1 \dots s_m$  of subsets of  $S$  is there a collection of at most  $k$  of these sets whose union equals  $S$ ?

## Set cover example



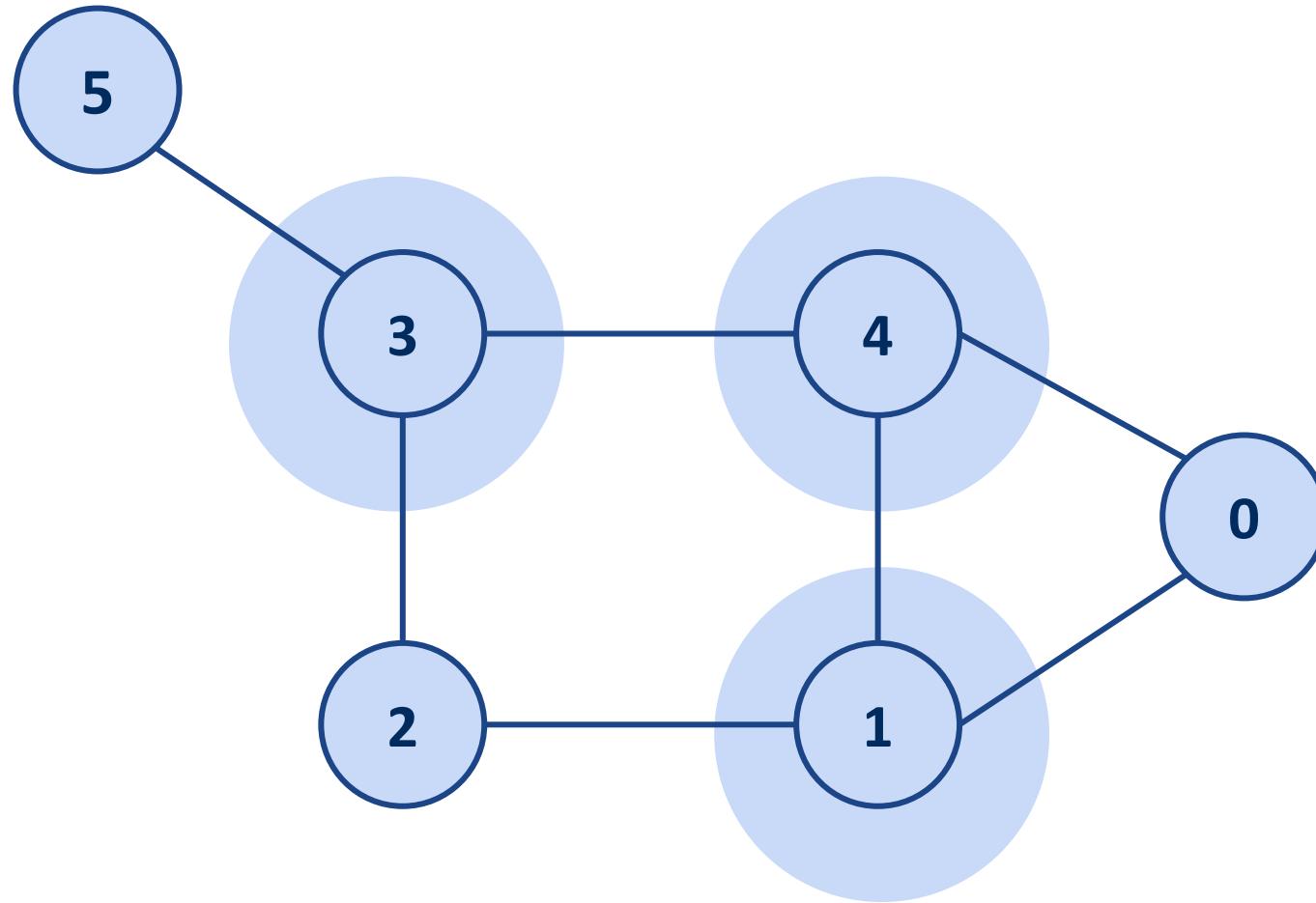
# Is set cover NP-Complete?

- First of all is it in NP?
- OK, next step is to find a problem that is known to be NP-Complete and reduce it to an instance of set cover

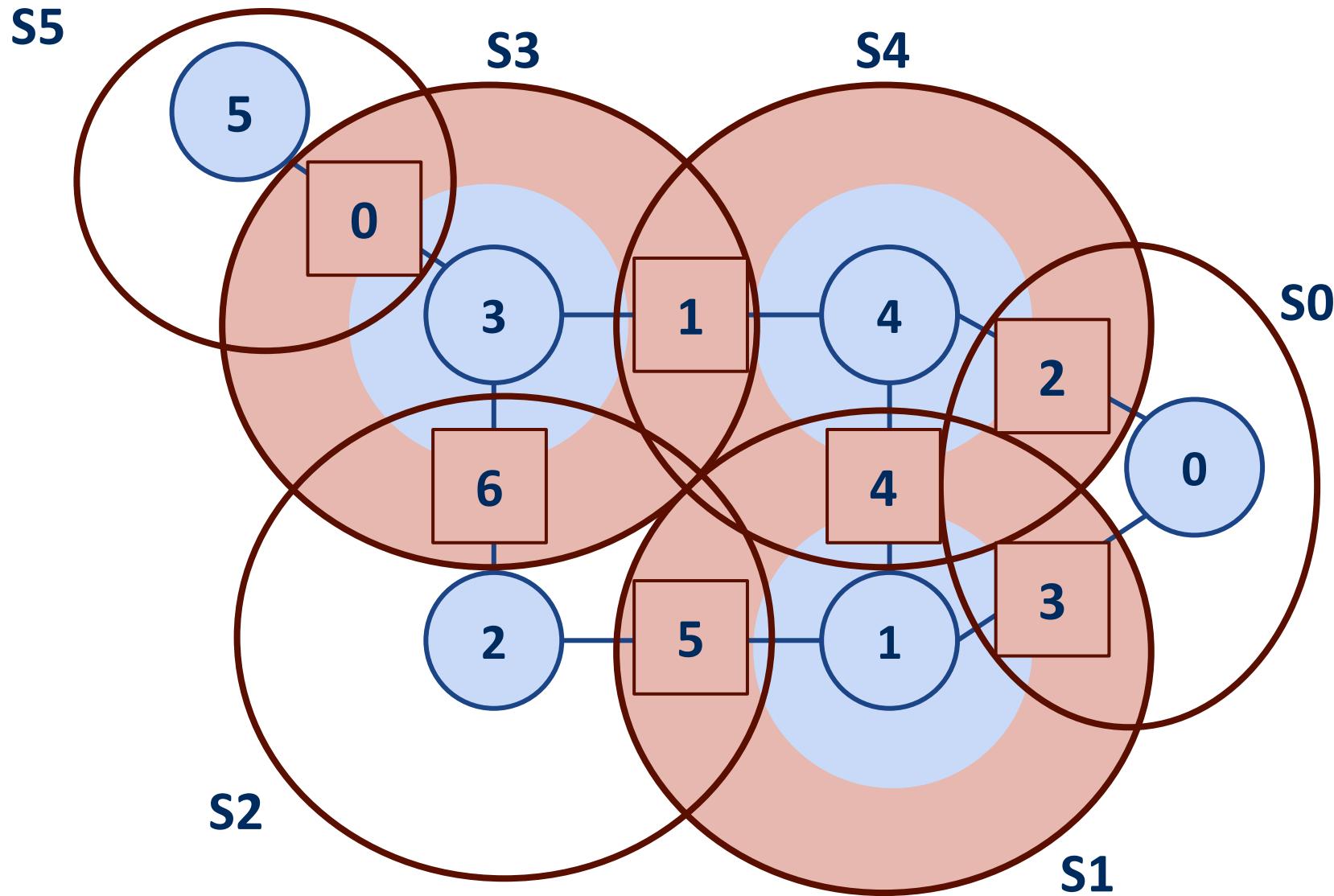
# The vertex cover problem

- A vertex-cover of an undirected graph  $G = (V, E)$  is a subset  $V'$  of  $V$  such that if edge  $(u, w)$  is an edge of  $E$ , then  $u$  is in  $V'$ ,  $w$  is in  $V'$ , or both
- Does a vertex-cover exist for a graph  $G$  that consists of at most  $k$  vertices?

## Vertex cover example



# Reducing vertex cover to set cover



# Reducing vertex cover to set cover

- Given  $k$  and a graph to be vertex-covered:
  - Let  $S = E$
  - For each  $u_i \in V$ : create  $s_i$  such that  $s_i$  contains all edges in  $E$  with  $u_i$  as an endpoint
  - Solve the set cover problem for  $k$ ,  $S$ , and  $s_1 \dots s_v$
- Runtime to transform inputs to vertex cover into inputs for set cover?

# A final note on reduction

- Be careful about the ordering
  - Always solve the known NP-Complete problem using your new problem
    - You WILL get this mixed up at some point

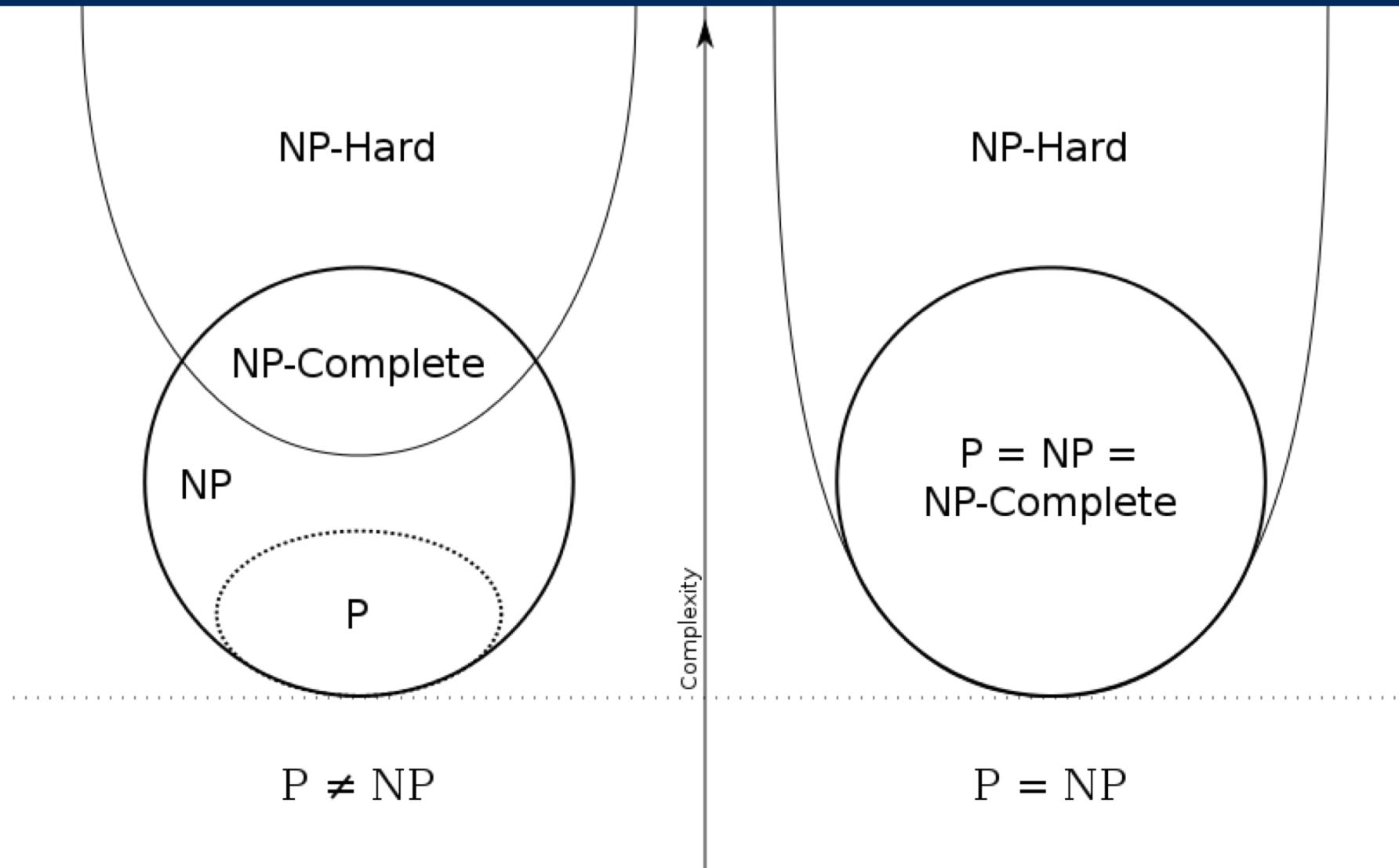
# A timeline of P/NP

- 1971 - Cook presents the Cook-Levin theorem, which shows that the boolean satisfiability problem is as hard as every other problem in NP
  - It is NP-Complete, but this term appears nowhere in paper
- 1972 - Karp presents 21 NP-Complete problems via reduction from boolean satisfiability
- Thousands have since been discovered by reducing from those 21

# Karp's 21 problems

- Boolean Satisfiability
  - 0–1 integer programming
  - Clique (see also independent set problem)
    - Set packing
    - Vertex cover
      - Set covering
      - Feedback node set
      - Feedback arc set
      - Directed Hamilton circuit
        - Undirected Hamilton circuit
  - Satisfiability with at most 3 literals per clause
    - Chromatic number (aka Graph Coloring Problem)
      - Clique cover
      - Exact cover
        - Hitting set
        - Steiner tree
        - 3-dimensional matching
        - Knapsack
          - Job sequencing
          - Partition
            - Max cut

# The landscape



# To review

- What are P problems?
- What are NP problems?
- What are NP-Complete problems?
- What about NP-Hard?

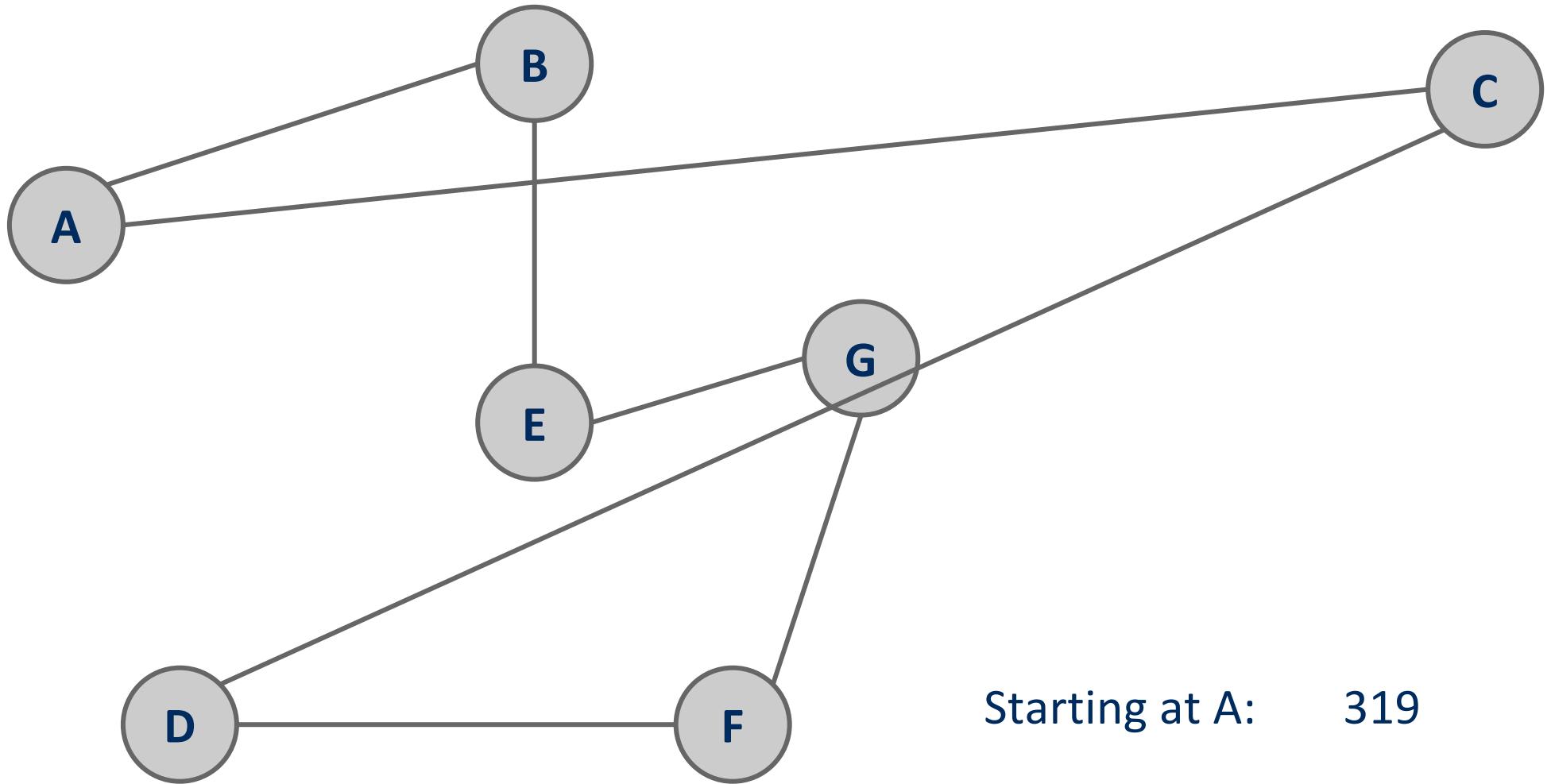
# What if you still need to solve an NP-C problem?

- Can't get an exact solution in a reasonable amount of time.
  - What about an approximate solution?
    - Can we devise an algorithm that runs in a reasonable amount of time and gives a close to optimal result?
- Let's look at some heuristics for approximating solutions to the Travelling Salesman Problem:
  - Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

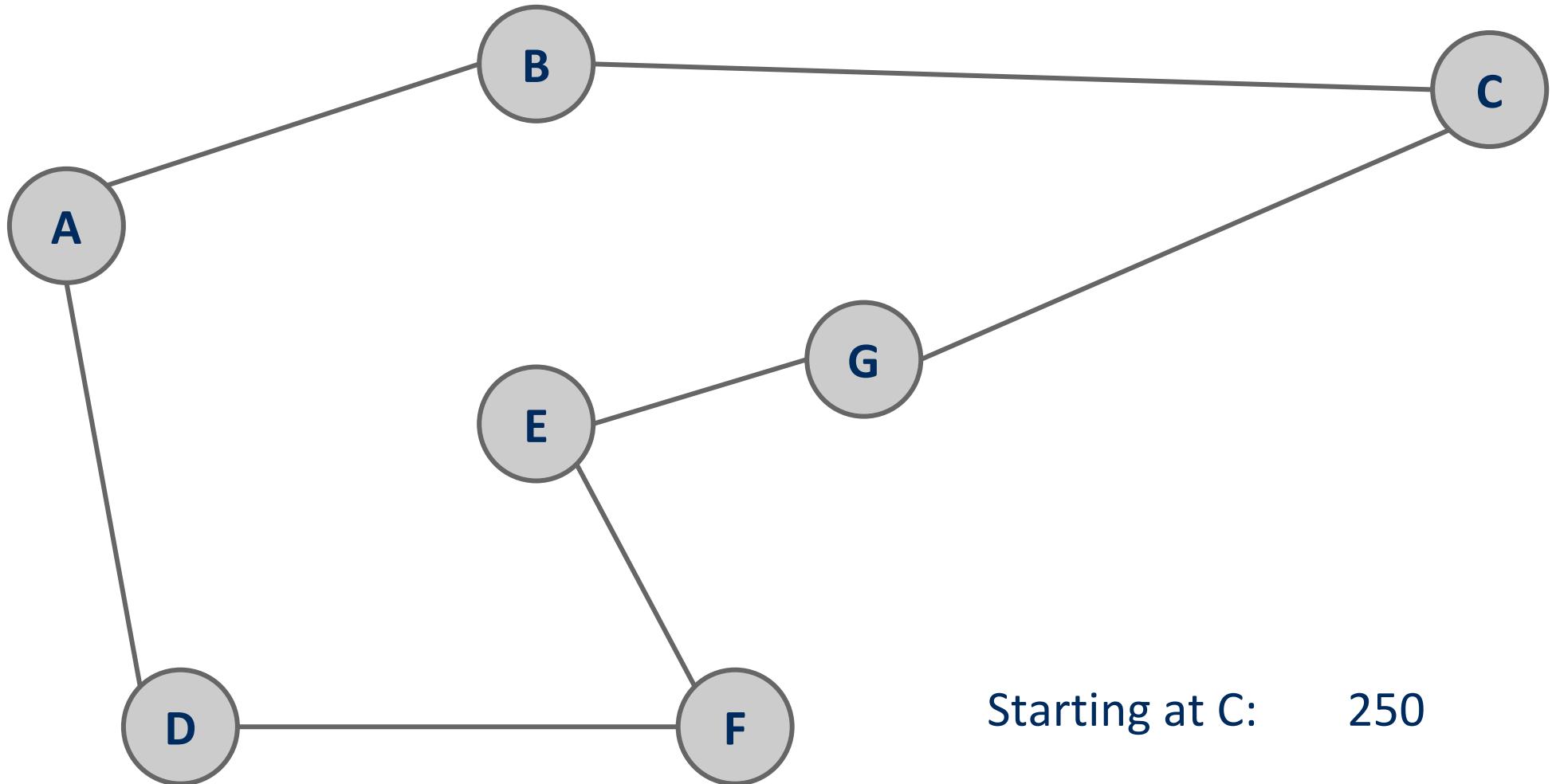
# Nearest neighbor heuristic

- From each city, visit the nearest city until a circuit of all cities is completed
- Runtime?
  - $v^2$
- Any other issues?
- How good are solutions generated by this heuristic compared to optimal solutions?

## Nearest neighbor example



## What about a different starting point?



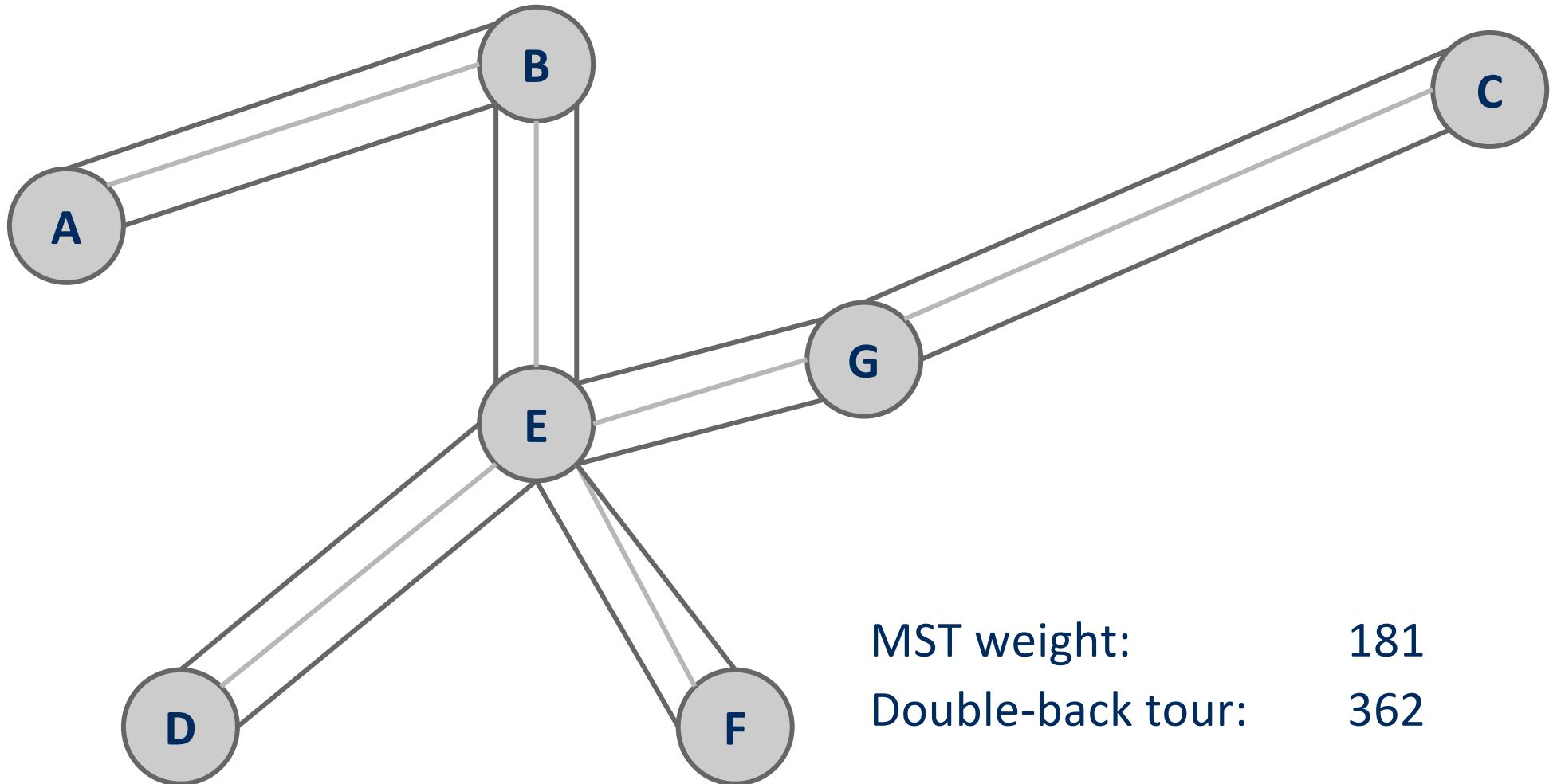
# Measuring heuristic algorithm quality

- Let's consider  $H_{NN}(C)$  be the length of the tour of the set of cities described by  $C$  found by the nearest neighbor heuristic
- Let  $OPT(C)$  be the optimal tour for  $C$
- The approximation ratio of the nearest neighbor heuristic is then  $H_{NN}(C)/OPT(C)$ 
  - I.e, how much worse than optimal is nearest neighbor
    - For nearest neighbor, this approximation ratio grows according to  $\log(v)$

# Let's aim for a constant approximation ratio

- Consider minimum spanning trees
  - Creates a fully connected graph with minimum edge weight
    - Optimal TSP solution must be more than MST weight
  - Consider the tour produced by a DFS traversal of the MST
    - Travels every edge twice
    - Since MST weight is less than optimal solution, this tour must be less than 2x the optimal solution!

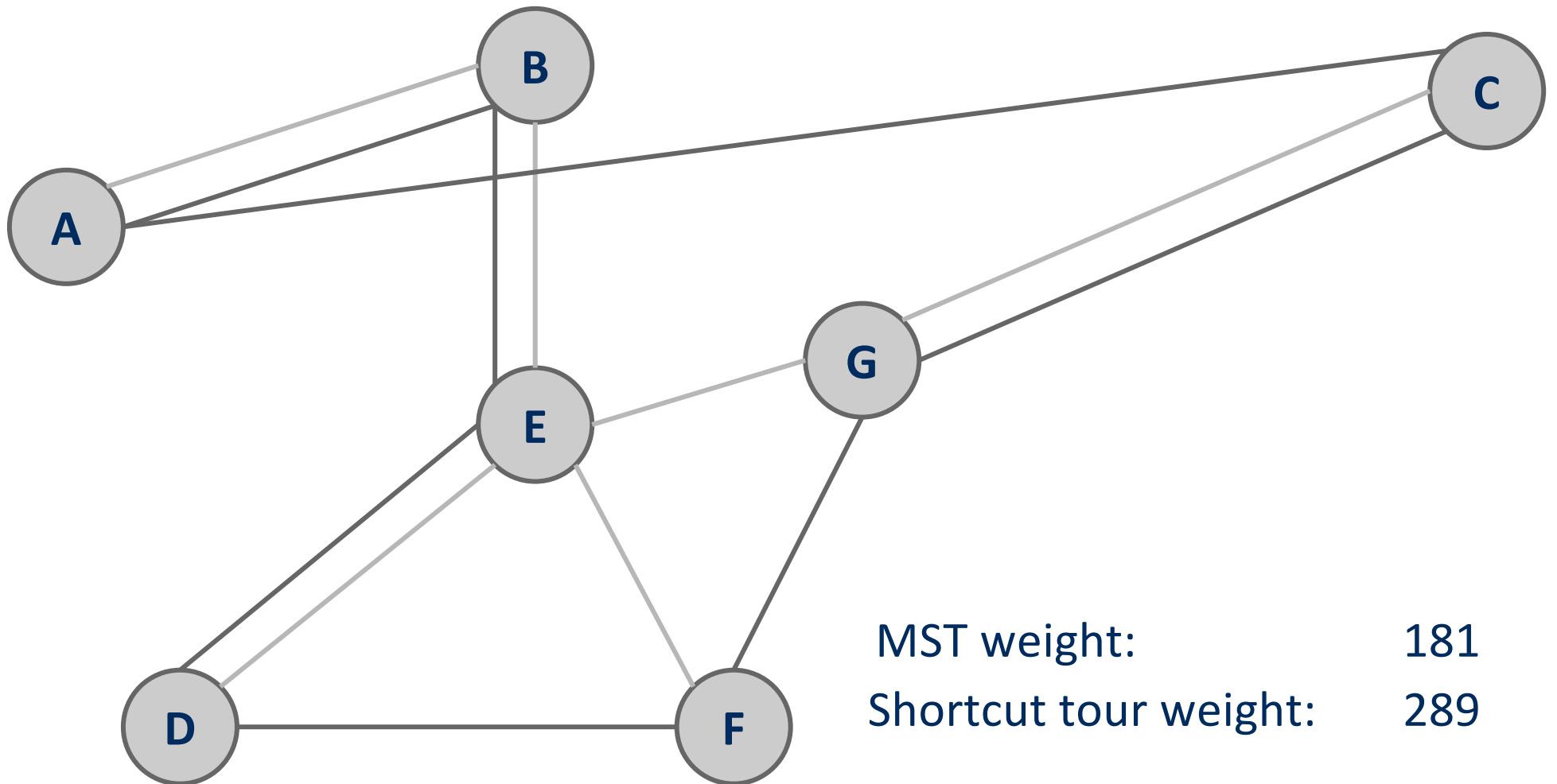
# MST Heuristic example



## But it visits some cities twice...

- Violates a condition of being a TSP solution
- What about this:
  - Find MST
  - Determine traversal order
  - At each backtrack, simply take the direct route to the next city

# MST Heuristic example



# Does this maintain our approximation ratio?

- Yes, if we make an additional assumption
  - Distances between "cities" have to abide by Euclidean geometry
    - Specifically, they need to uphold the triangle inequality
      - A direct path between two cities must be shorter than going through a third city