Data Structures in Java

Lecture 25: Introduction to NP Completeness

Daniel Bauer

4/20/2022

Algorithms and Problem Solving

- Purpose of algorithms: find solutions to problems.
- Data Structures provide ways of organizing data such that problems can be solved more efficiently
  - Examples: Hashmaps provide (ideally) constant time access by key, Heaps provide a cheap way to explore different possibilities in order…
- When confronted with a new problem, how do we:
  - Get an idea of how difficult it is?
  - Develop an algorithm to solve it?

Problem Difficulty

- We can think of the *difficulty of a problem* in terms of *the best algorithm we can find to solve the problem.*

  - Most problems we discussed so far have linear time solutions O(N), or slightly more than linear O(N log N).

    - We often considered anything worse than $O(N^2)$ to be a **bad** solution.

    - For some problems we don't know efficient algorithms.

- Ideally, want lower bounds: For a given problem, what is the best algorithm we can hope for?
  (for instance, $\Omega(N \log N)$ for comparison based sorting).

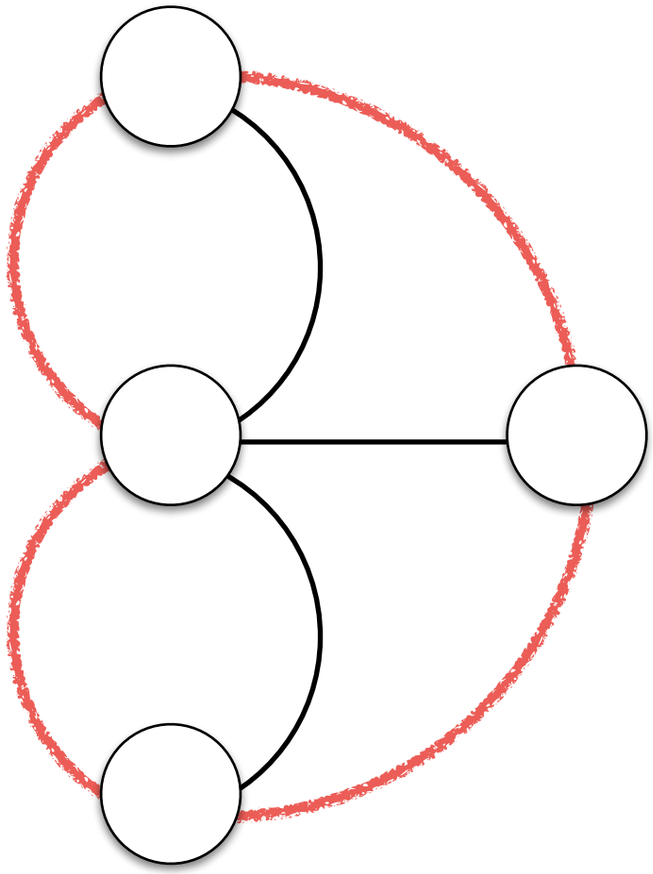
Polynomial and Exponential Time

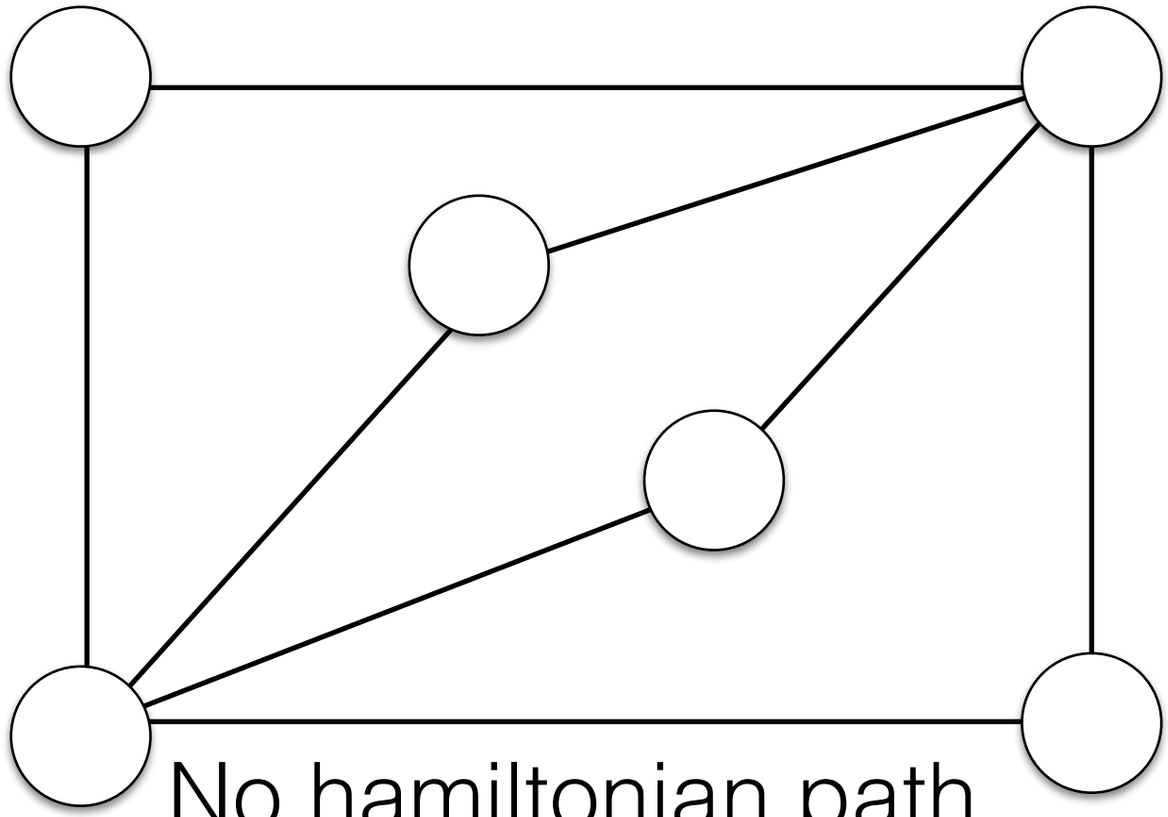
- Two common classes of running time for algorithms:
  - Polynomial: $O(N^k)$ for some constant k.
  - Exponential: $O(2^{N^k})$ for some constant k

"I can't find an efficient algorithm, but neither can all these famous people."

Hamiltonian Cycle

- A Hamiltonian Path is a path through an undirected graph that visits *every vertex* exactly once (except that the first and last vertex may be the same).
- A Hamiltonian Cycle is a Hamiltonian Path that starts and ends in the same node.

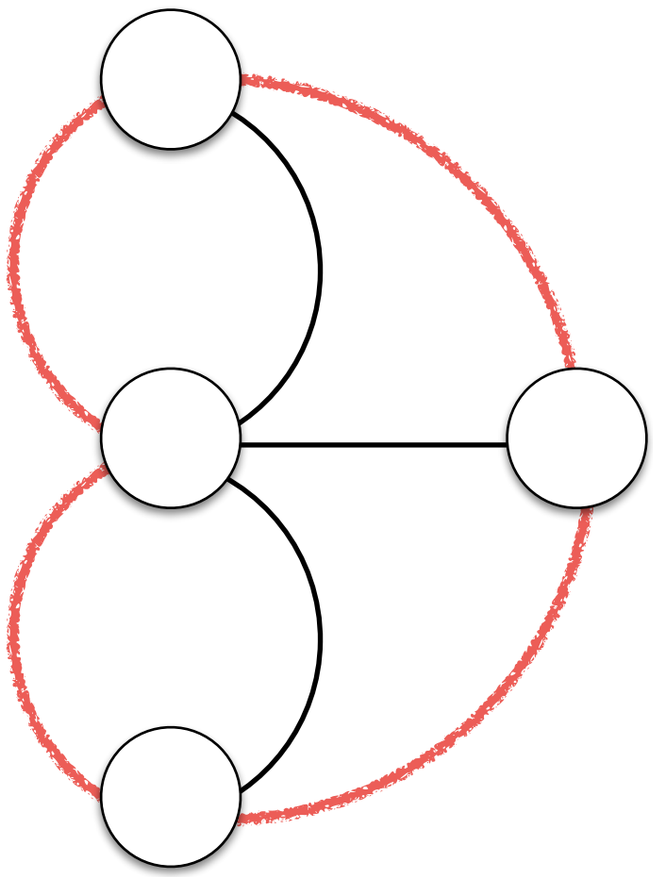


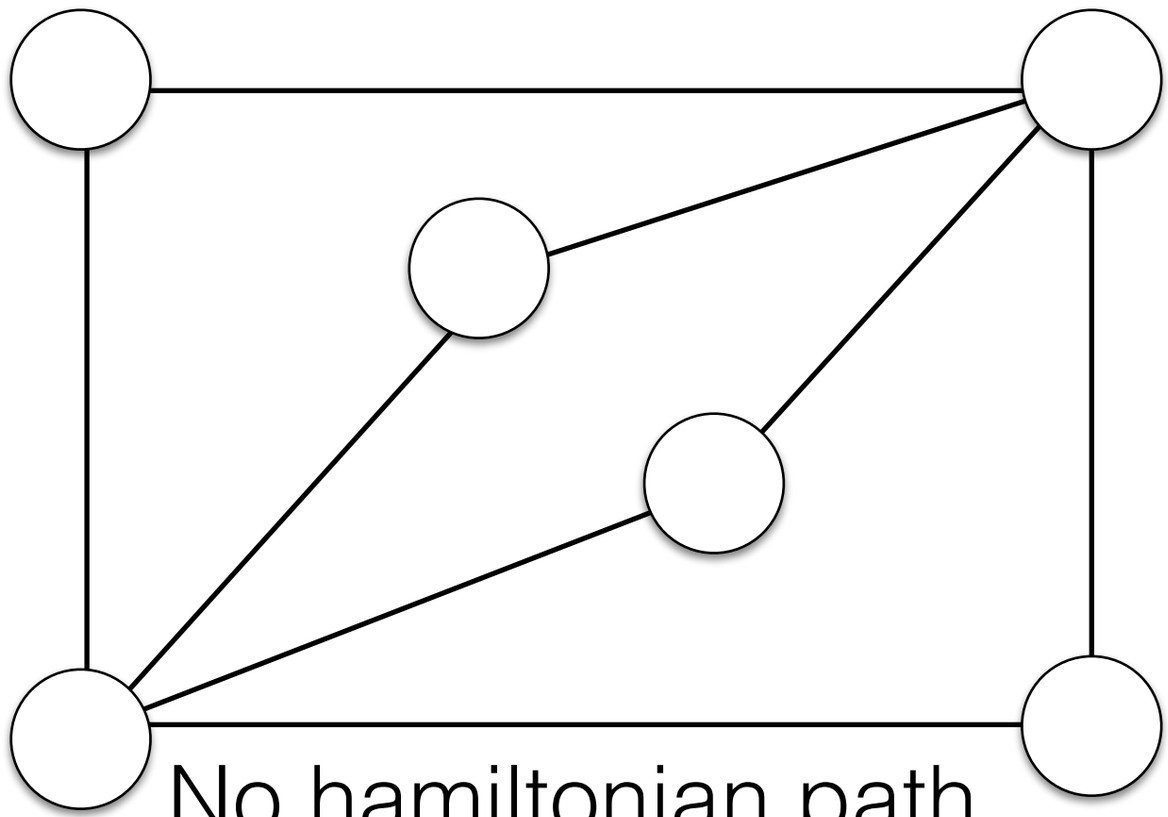


No hamiltonian path.

Hamiltonian Cycle

- Can check if a graph contains an Euler Circuit in linear time.
- Surprisingly, checking if a graph contains a Hamiltonian Path/Cycle is much harder!
- No polynomial time solution (i.e. $O(N^k)$) is known.

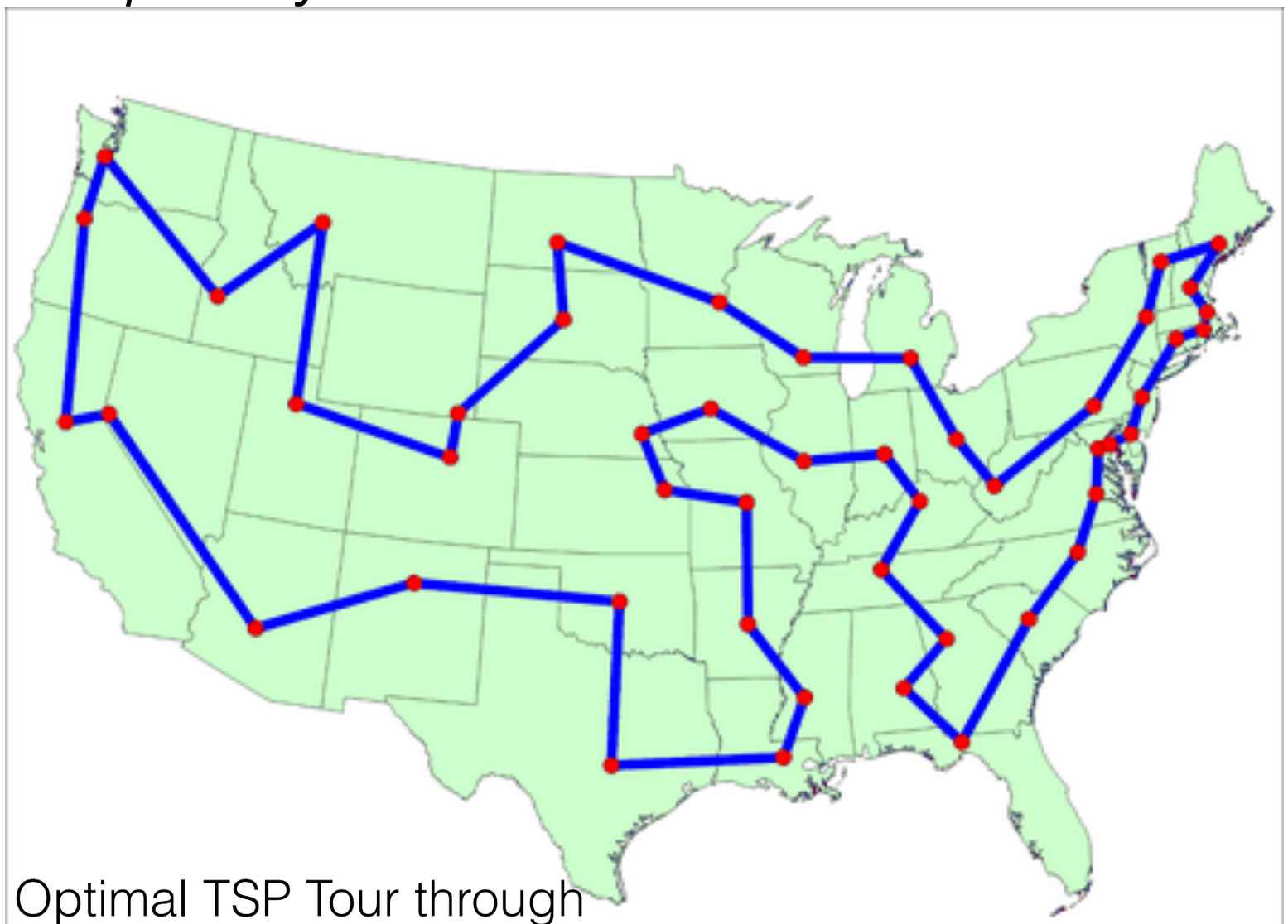




No hamiltonian path.

Traveling Salesperson Problem (TSP)

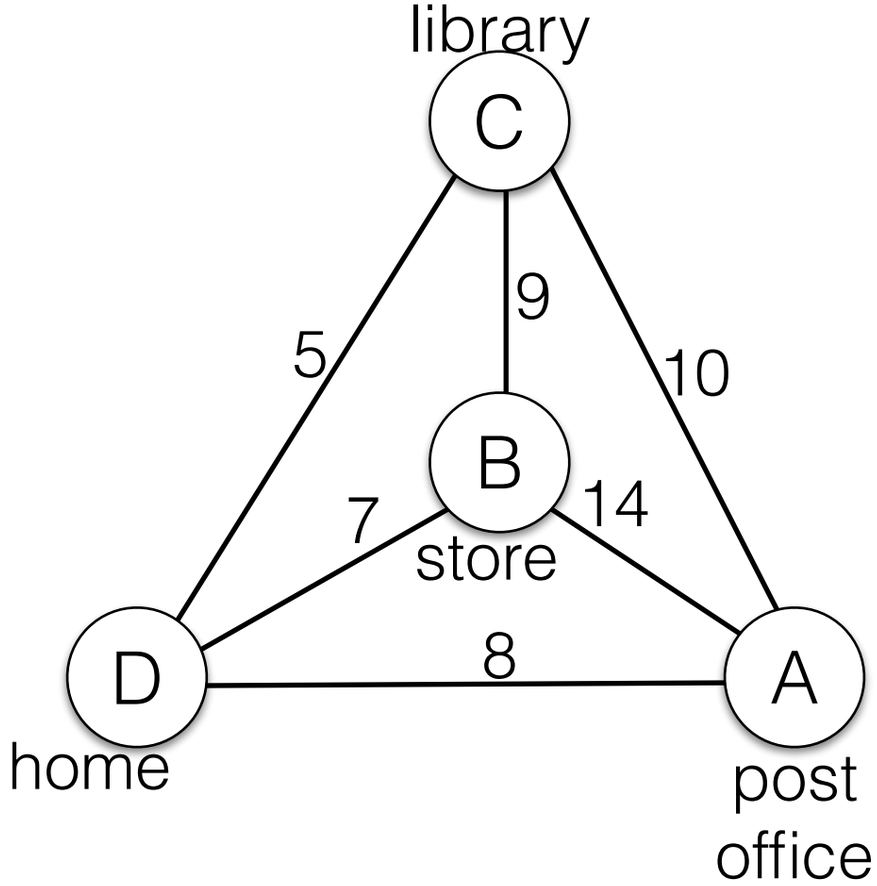
Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.



Optimal TSP Tour through
all 48 continental state capitals.

TSP - How many tours are there?

Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.

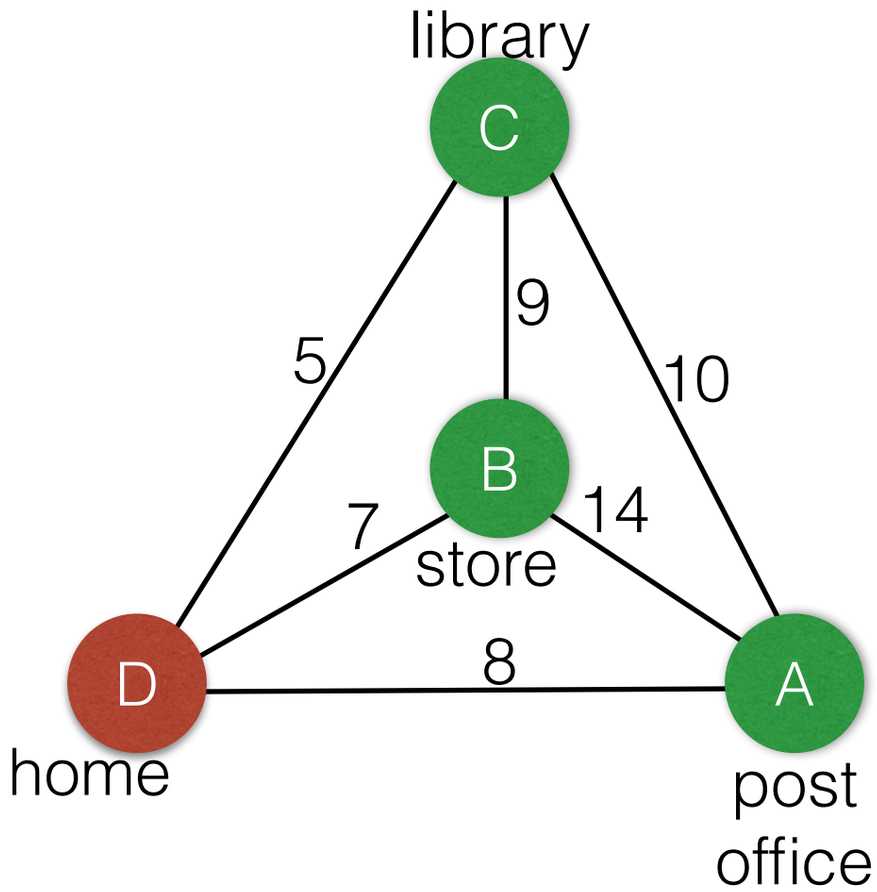


We can visit the vertices of the graph in ANY order.

How many possibilities are there?

TSP - How many tours are there?

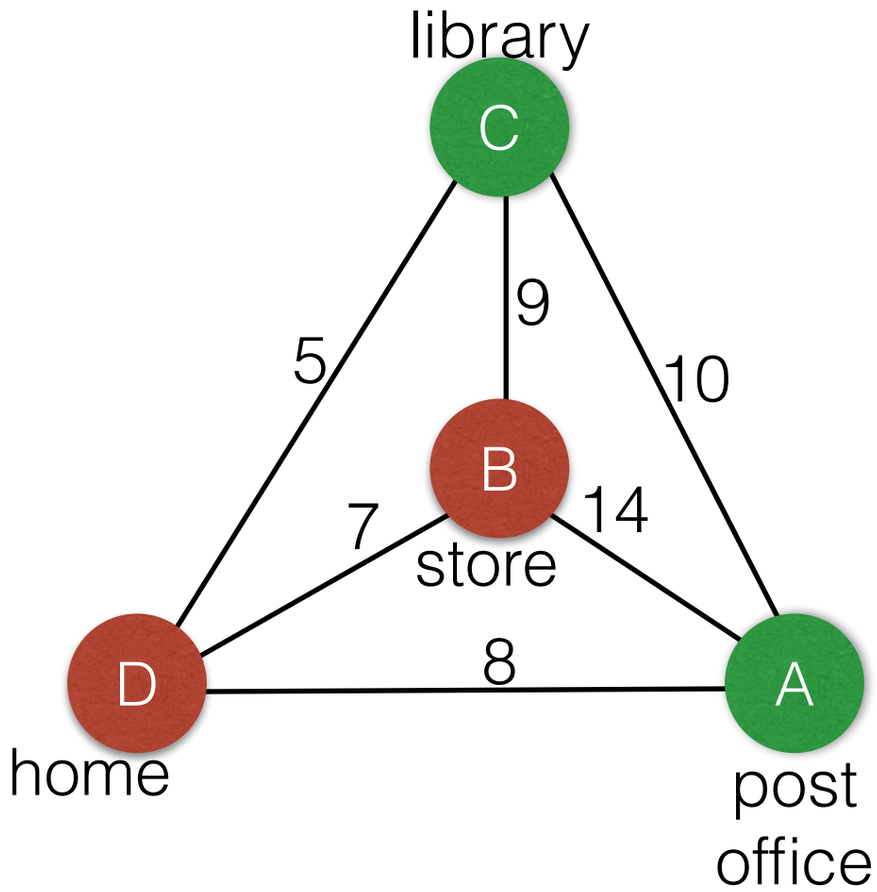
Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.



We start at D.
Because the graph is complete,
we can go to any of the other N-1 nodes.

TSP - How many tours are there?

Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.

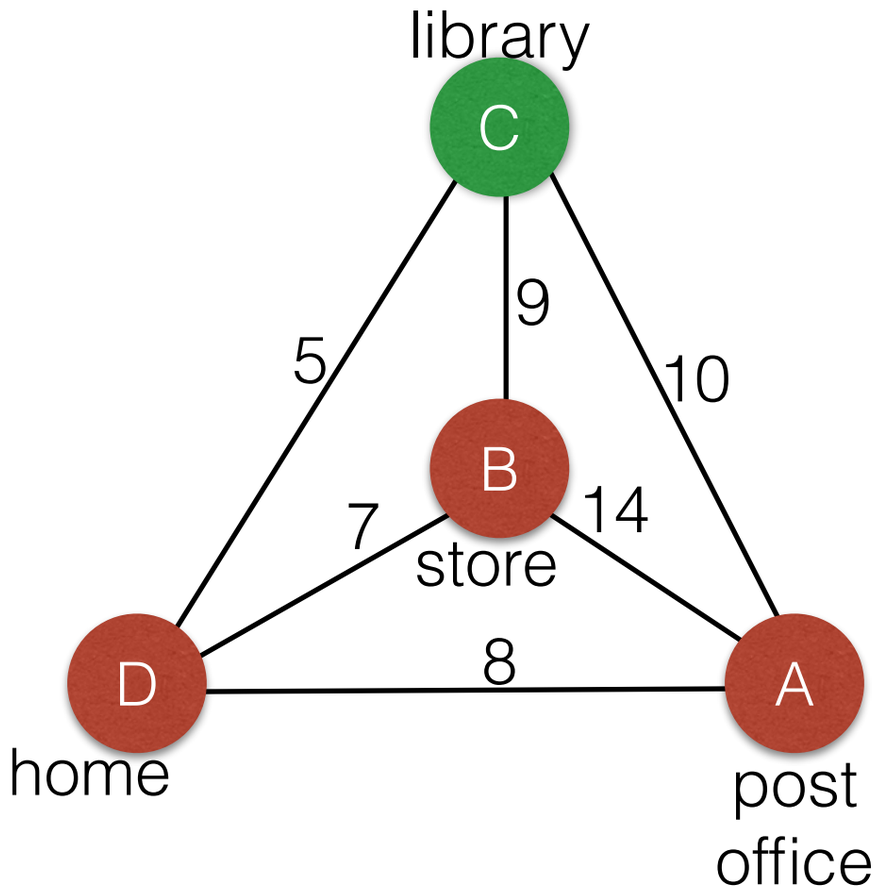


We start at D.
Because the graph is complete,
we can go to any of the other N-1 nodes.

Once we decide for a node, we can go to N-2 remaining nodes.

TSP - How many tours are there?

Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.



We start at D.
Because the graph is complete,
we can go to any of the other N-1 nodes.

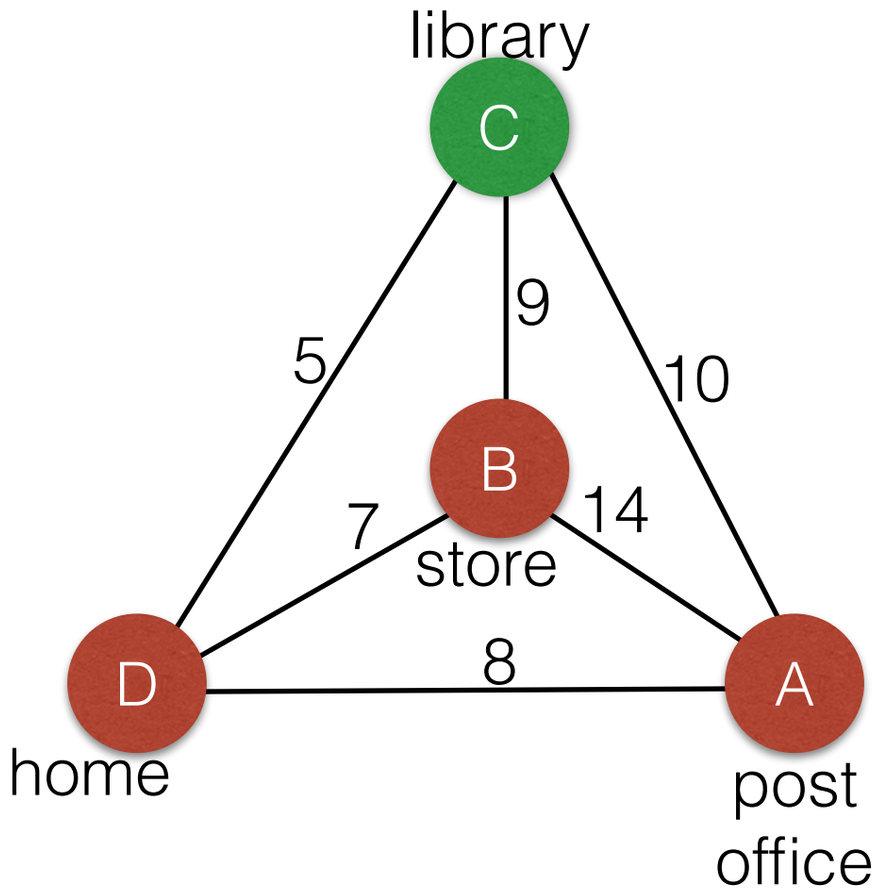
Once we decide for a node, we can go to N-2 remaining nodes.

Once we decide for a node, we can go to N-3 remaining nodes.

…

TSP - How many tours are there?

Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.



We start at D.
Because the graph is complete,
we can go to any of the other N-1 nodes.

Once we decide for a node, we can go to N-2 remaining nodes.

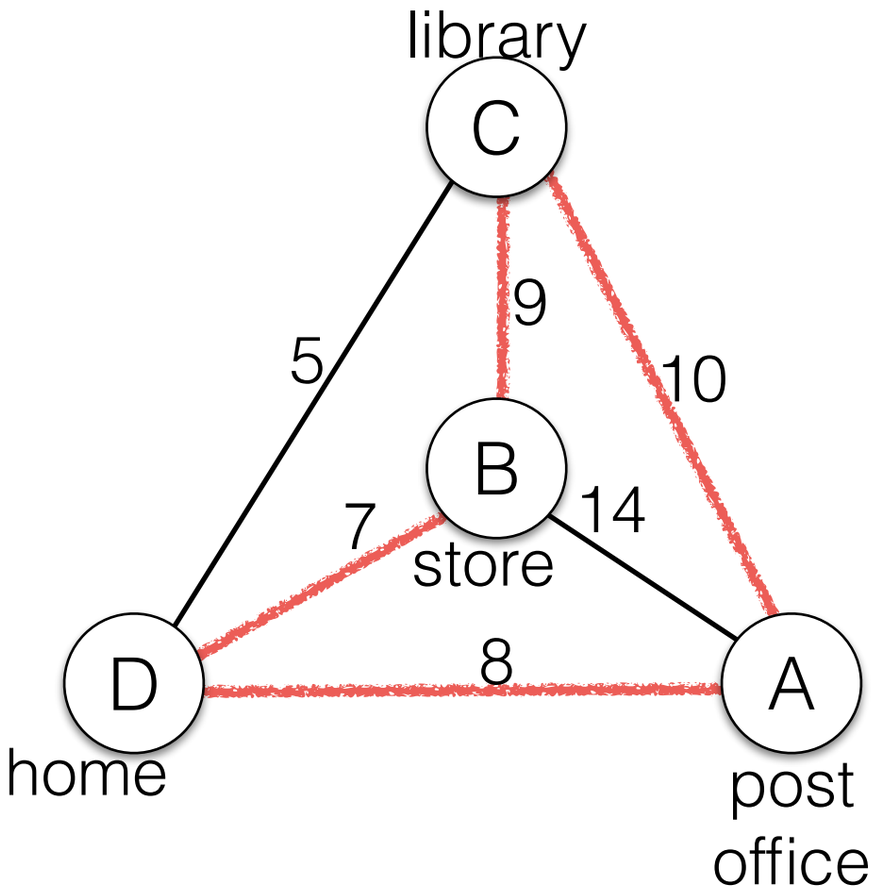
Once we decide for a node, we can go to N-3 remaining nodes.

…

$$(N-1)\cdot(N-2)\cdots(1) = (N-1)!$$

TSP - How many tours are there?

Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.



D A C B = D B C A

There are

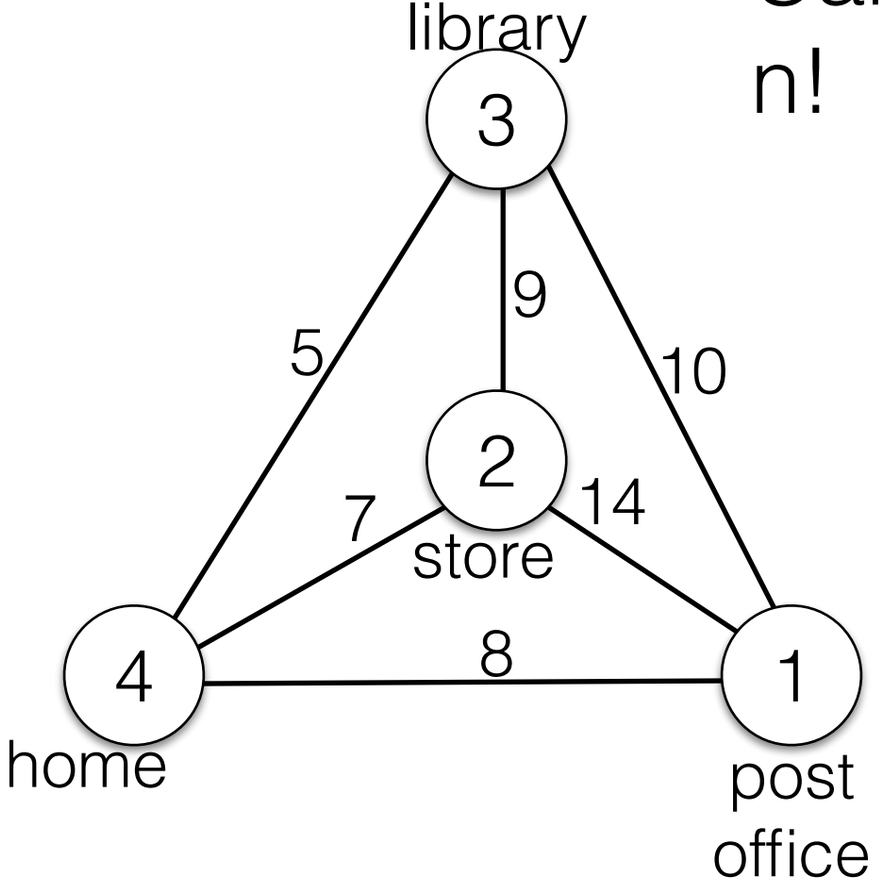
$$(N-1)\cdot(N-2)\cdots(1) = (N-1)!$$

possibilities, but we can traverse complete tours in either direction.

There are $\frac{(N-1)!}{2}$ complete tours.

TSP - How many tours are there?

Given a *complete*, undirected graph G = (V,E), find the shortest *simple* cycle that visits all vertices.



Can also think of this as enumerating all n! *permutations* of the vertex set.

(1, 2, 3, 4)
(1, 2, 4, 3)
(1, 3, 2, 4)
(1, 3, 4, 2)
(1, 4, 2, 3)
(1, 4, 3, 2)
(2, 1, 3, 4)

…

TSP - Brute Force Approach

Try all possible tours and return the shortest one.

Obviously this algorithm runs in $O(N!)$

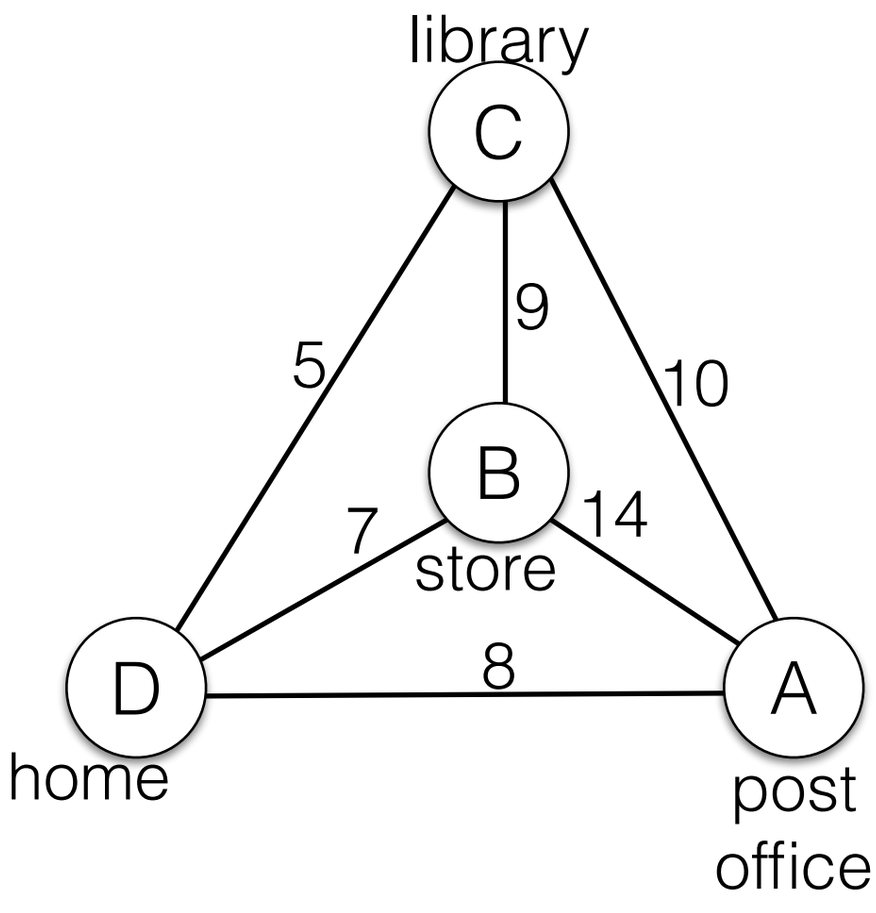
Better algorithm:
Dynamic Programming algorithm by Held-Karp (1962)

$$O(2^N N^2)$$

No exact polynomial time algorithm is known!

TSP - Nearest Neighbor

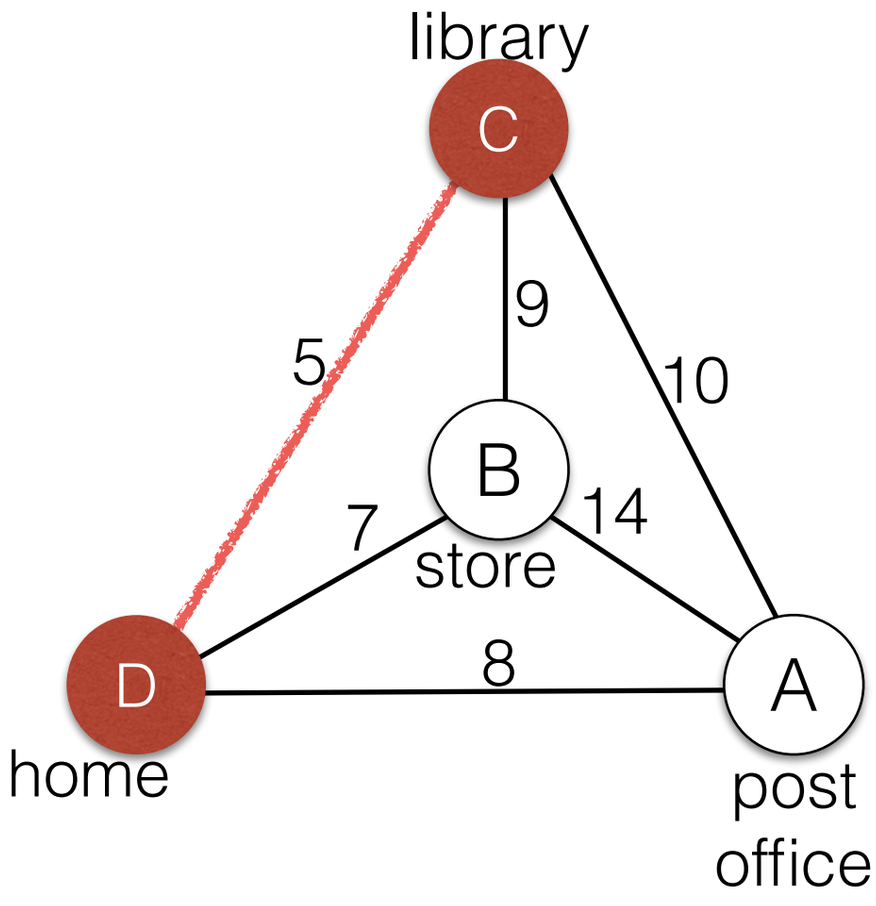
How about a greedy approximation?



Start with node D, always follow the lowest cost edge until all vertices have been visited.

TSP - Nearest Neighbor

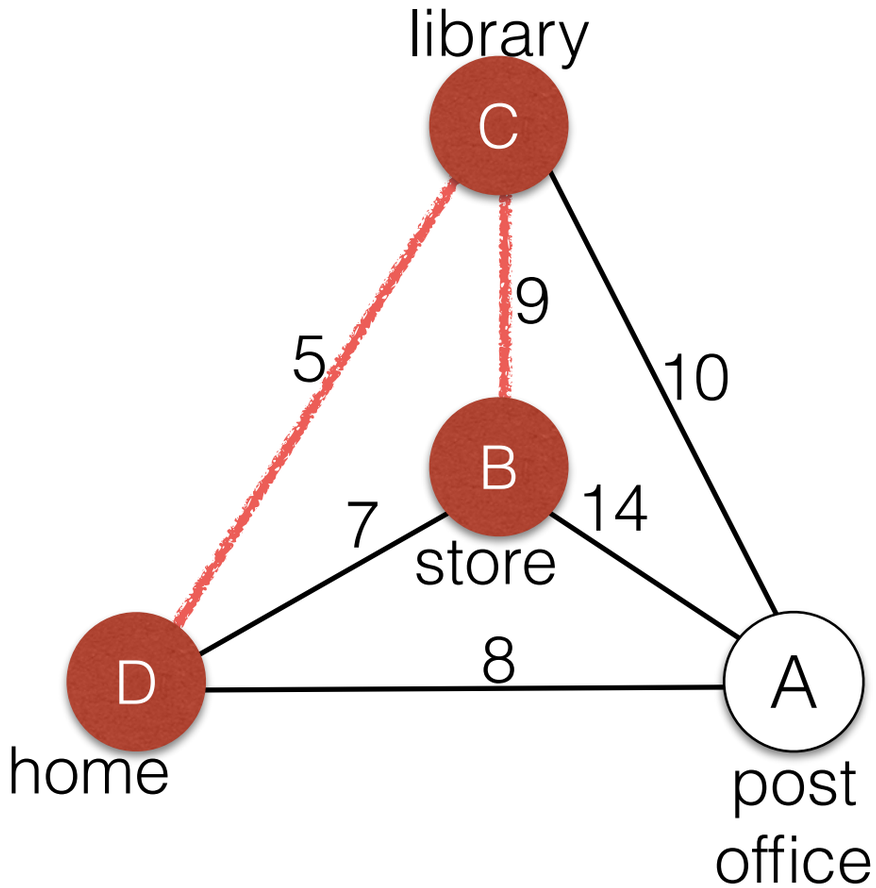
How about a greedy approximation?



Start with node D, always follow the lowest edge until all vertices have been visited.

TSP - Nearest Neighbor

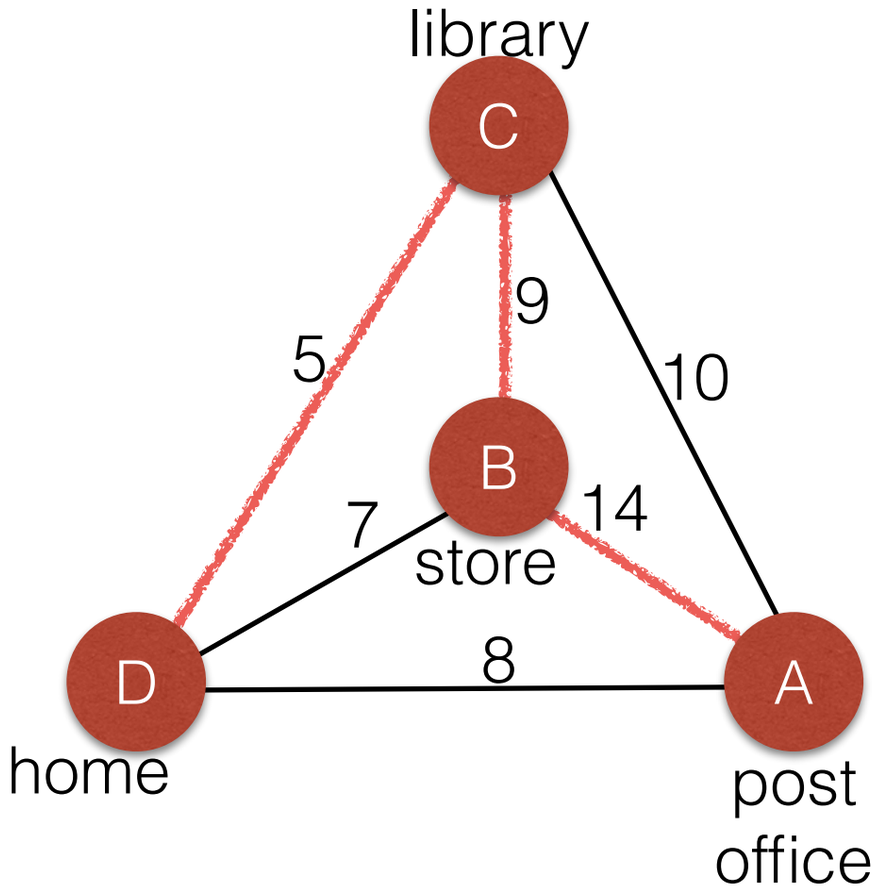
How about a greedy approximation?



Start with node D, always follow the lowest edge until all vertices have been visited.

TSP - Nearest Neighbor

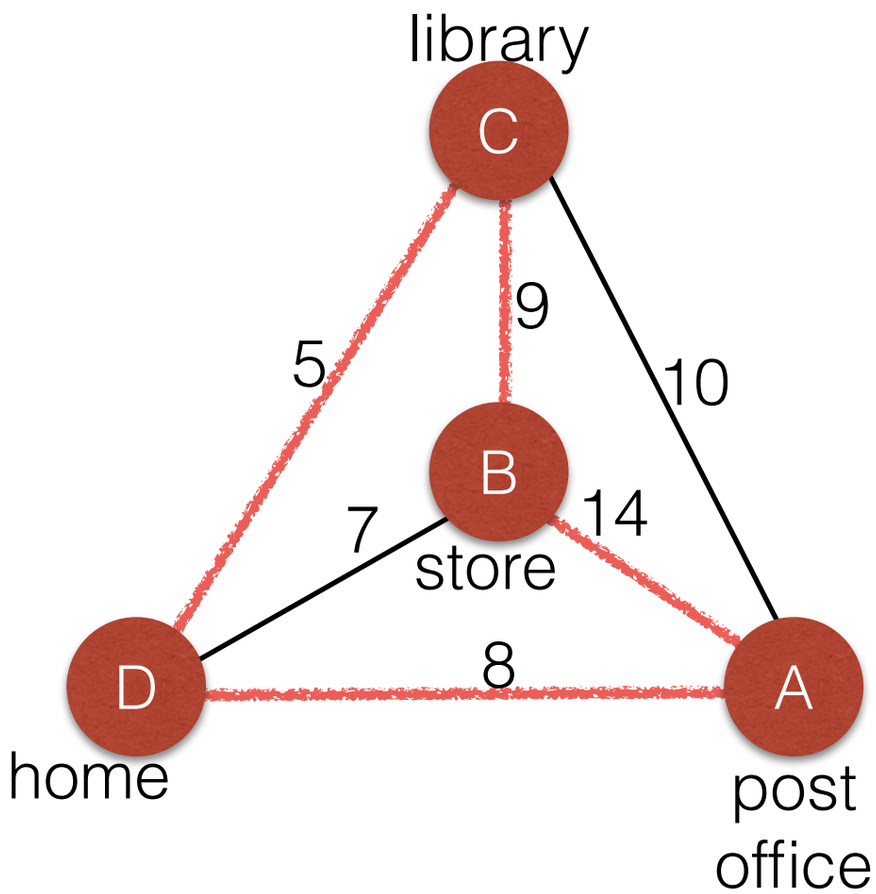
How about a greedy approximation?



Start with node D, always follow the lowest edge until all vertices have been visited.

TSP - Nearest Neighbor

How about a greedy approximation?



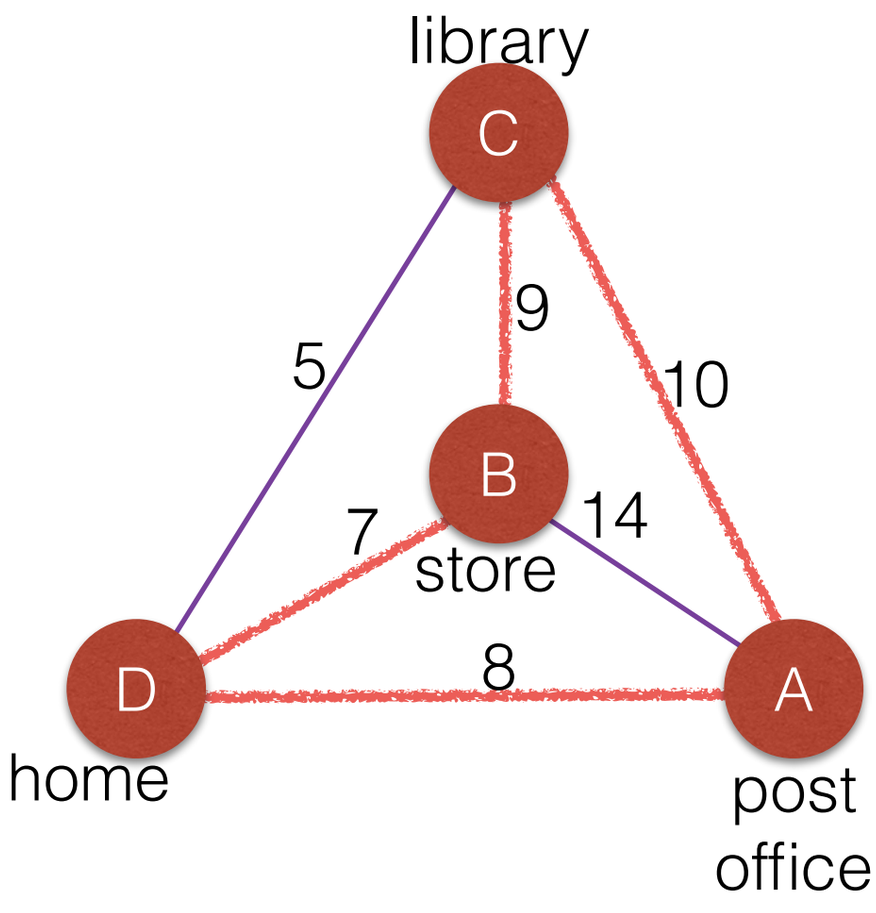
Start with node D, always follow the lowest edge to an unvisited vertex until all vertices have been visited.

Unfortunately, this is not guaranteed to find an optimal solution.

cost = 36

TSP - Nearest Neighbor

How about a greedy approximation?



cost = 34

Start with node D, always follow the lowest edge to an unvisited vertex until all vertices have been visited.

Unfortunately, this is not guaranteed to find an optimal solution.

Greedy Algorithms

"Take what you can get now"

- Algorithm uses multiple "phases" or "steps". In each phase a local decision is made that appears to be good.

- Making a local decision is fast (often O(log N) time). Examples: Dijkstra's, Prim's, Kruskal's

- Greedy algorithms assume that making ***locally optimal decisions*** leads to a ***global optimum.***

  - This works for some problems.

  - For many others it doesn't, but greedy algorithms are still useful to find approximate solutions.

Decision Problems

- A decision problem has, for each input, exactly two possible outcomes, YES or NO.

- *“Does this Graph contain an Euler Circuit”*
  “Does this Graph contain a Hamiltonian Cycle”

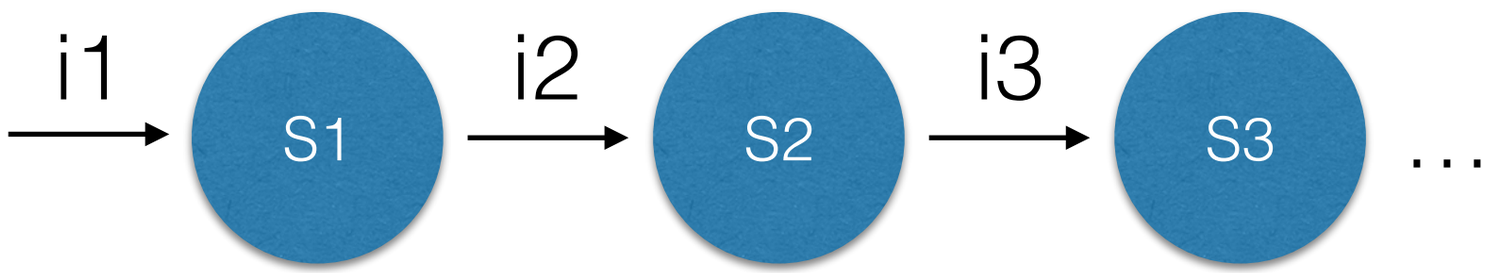
From Combinatorial Optimization to Decision Problems

- Any combinatorial optimization problem can be re-phrased as a decision problem by asking if a decision that is better than a certain threshold exists.
  - For instance, for TSP:
    “Is there a simple cycle that visits all vertices and has total cost ≤ K ?”

- Observation:
  Solving the optimization problem is at least as hard as solving the decision problem.

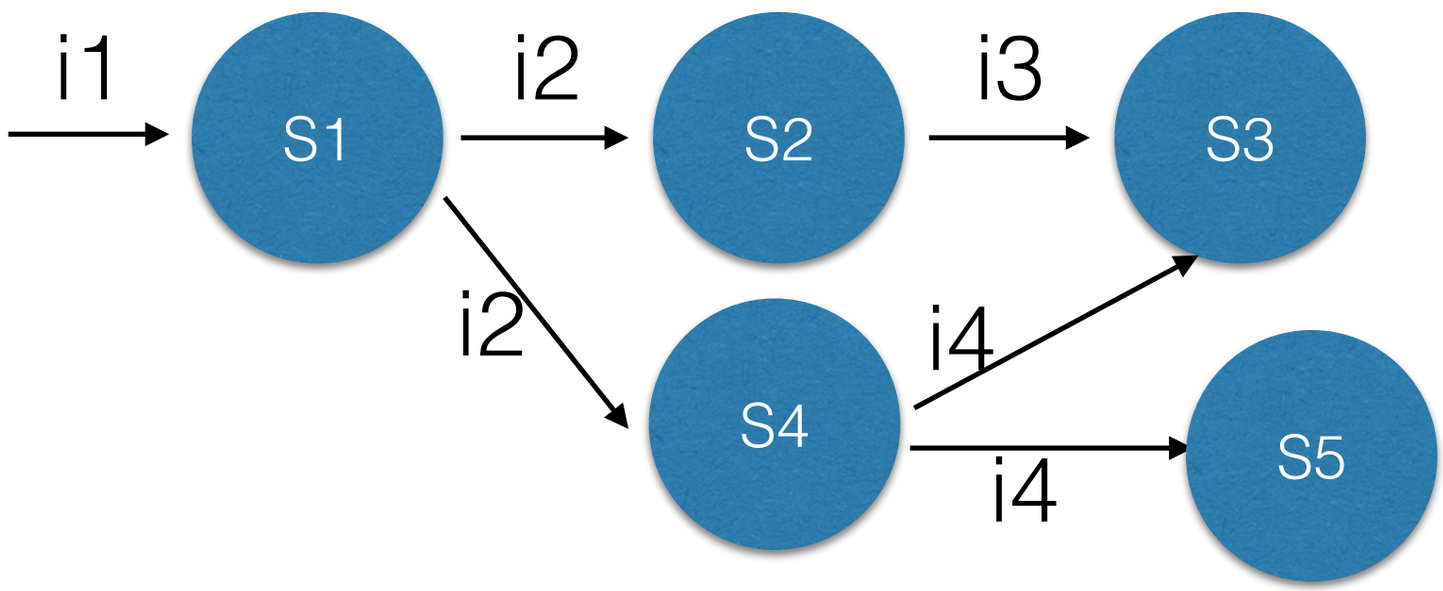
Deterministic and Non-Deterministic Machines

- The “state” of a computation consists of all current data (input, memory, CPU registers,…) and the last program instruction.
- Given a state and one instruction, a deterministic machine goes to a unique successor state.



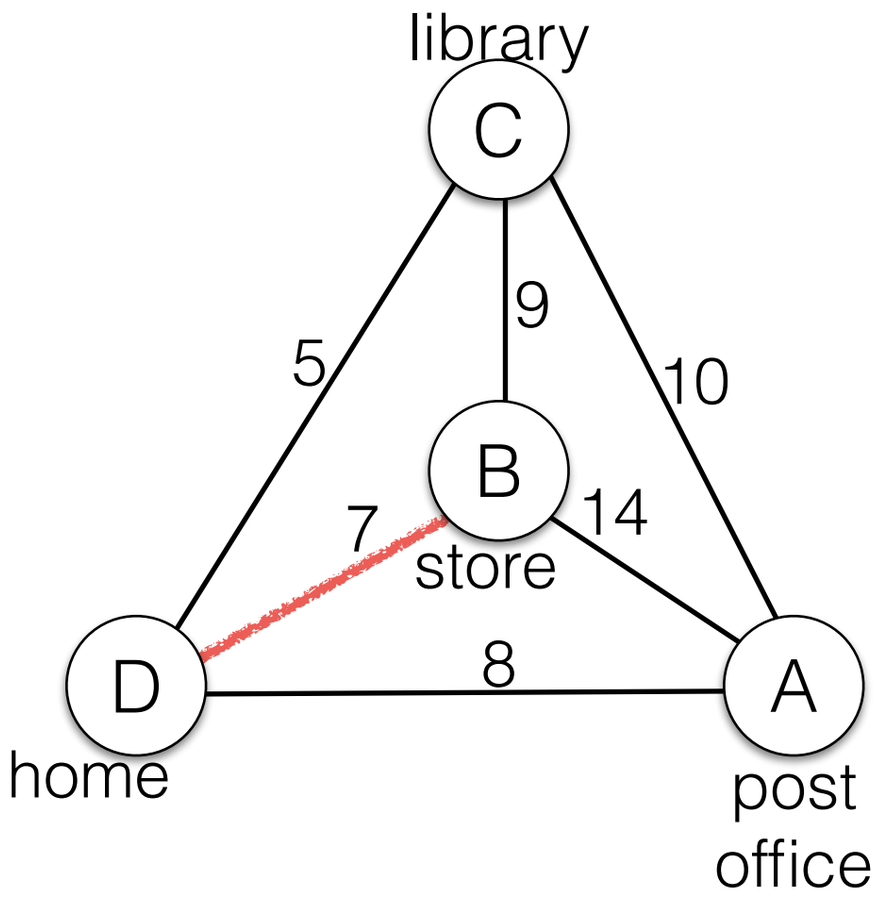
Deterministic and Non-Deterministic Machines

- A non-deterministic machine could perform more than one instruction in each state.
- Equivalently, a non-deterministic machine contains an “oracle” that tells it the optimal instruction (of several multiple instructions) to execute in each state.



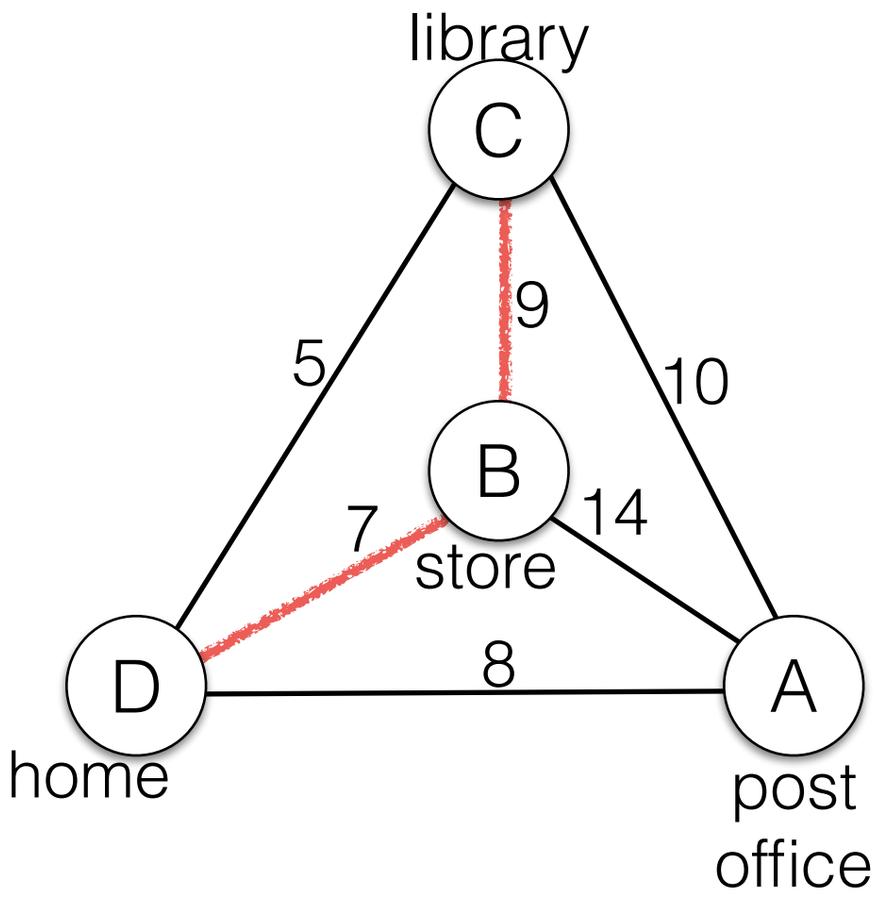
TSP with an Oracle

- State of the computation: Visited nodes, previous path.
- Same algorithm as greedy algorithm,
  but now the oracle tells us which edge to follow next.



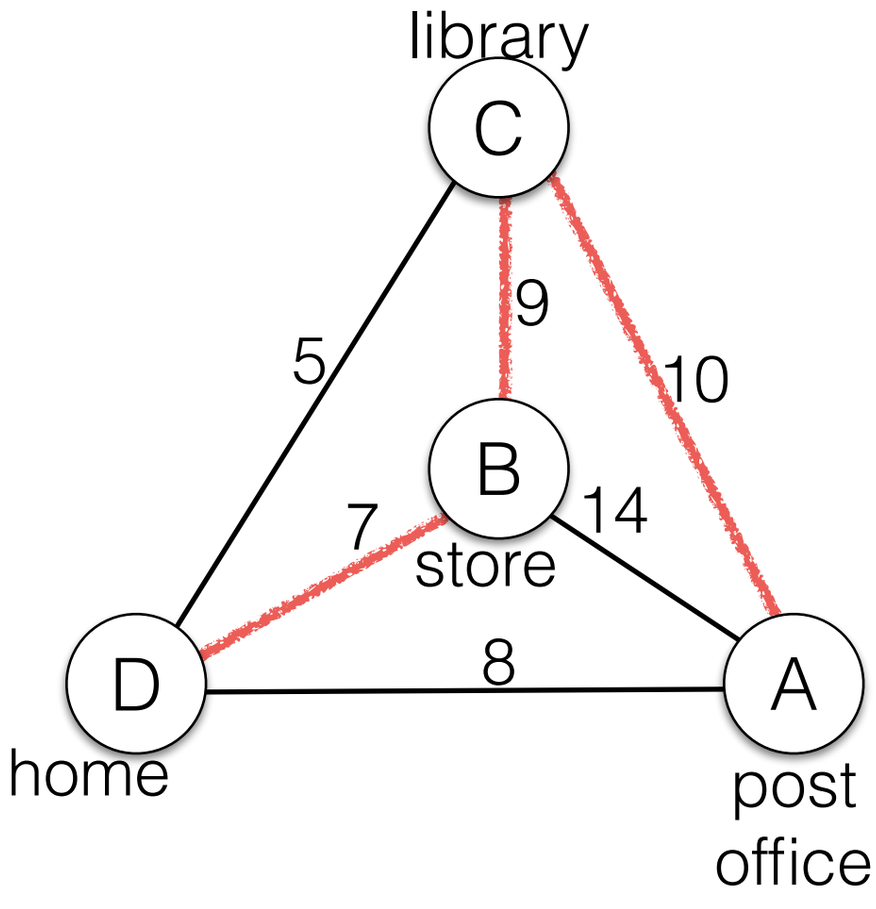
TSP with an Oracle

- State of the computation: Visited nodes, previous path.
- Same algorithm as greedy algorithm,
  but now the oracle tells us which edge to follow next.



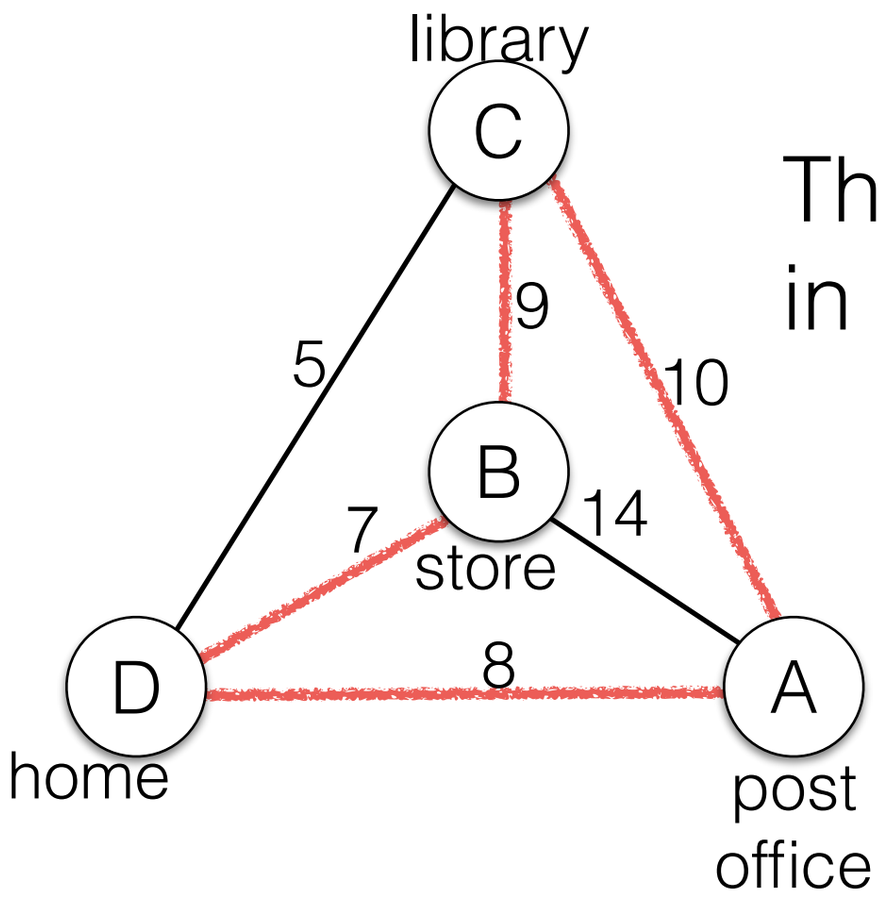
TSP with an Oracle

- State of the computation: Visited nodes, previous path.
- Same algorithm as greedy algorithm,
  but now the oracle tells us which edge to follow next.



TSP with an Oracle

- State of the computation: Visited nodes, previous path.
- Same algorithm as greedy algorithm,
  but now the oracle tells us which edge to follow next.



This algorithm produces an optimal tour in linear time!

Unfortunately, a real oracle is not realistic.
(But we can have a limited amount of parallelism).

The Class of NP Problems

- NP (*Nondeterministic Polynomial Time*) is the the class of problems for which a polynomial running time algorithm is known to exist on a non-deterministic machine.

- How do we know that a problem is in NP.

- Are there problems that are not in NP?

How Do We Know If a Problem Is in NP?

- Assume a decision problem produces YES on some input and some proof/“certificate” for this result.

- A decision problem is in NP if we can verify, in *deterministic polynomial time*, that the proof for a YES instance is correct.

- Examples:
  - An algorithm determines that a graph contains a Hamiltonian cycle and provides such a cycle as proof.
  - A spanning tree of cost < K is proof that such a spanning tree exists in a graph.

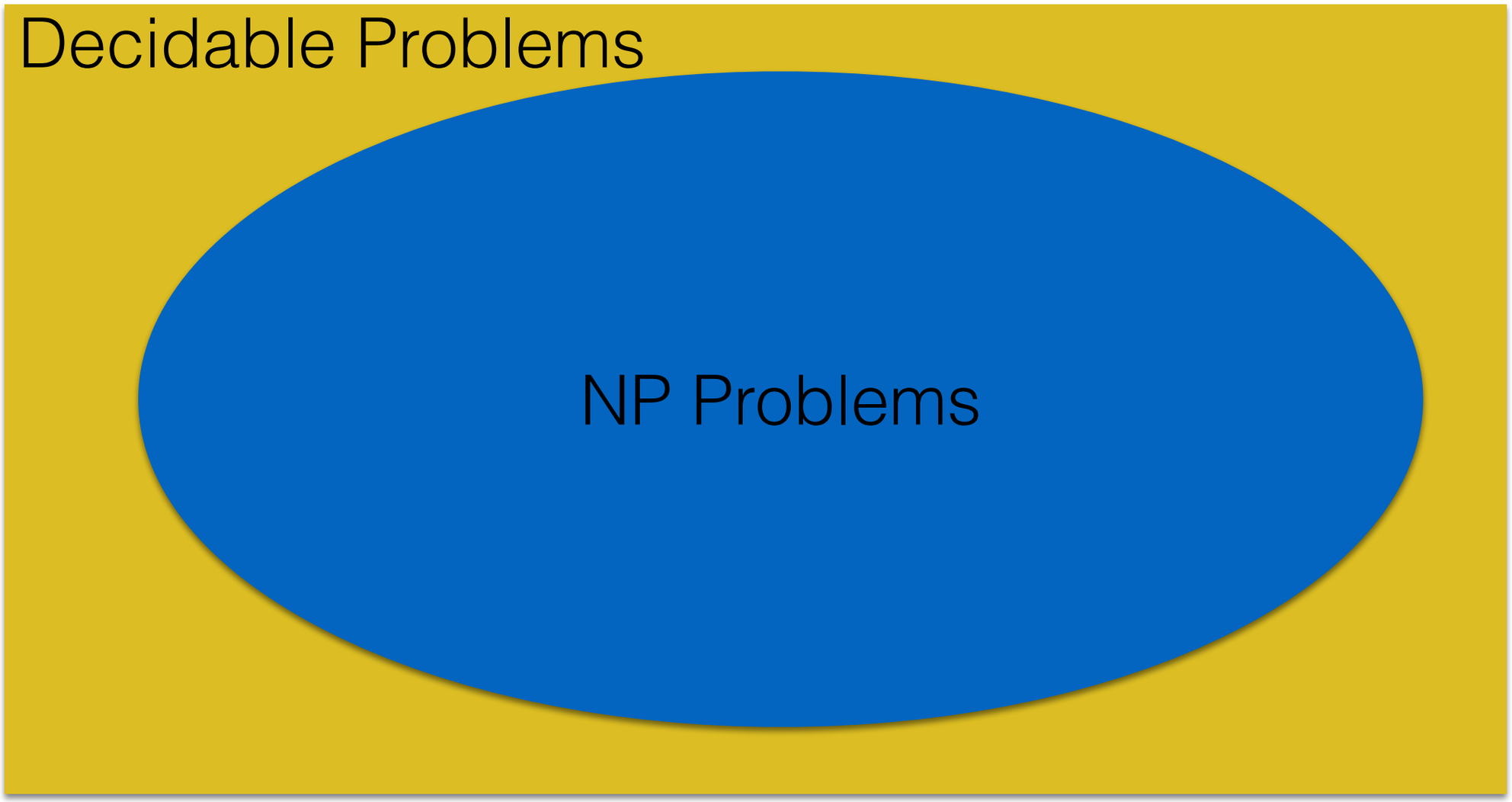
A decidable problem that is (probably) not in NP

- Consider the problem of deciding if a graph does NOT have a hamiltonian cycle.

- No NP algorithm is known for this problem.

- Intuitively, a proof would require to list all possible cycles. Verifying the proof means to show that none of them is Hamiltonian, one by one.

NP Problems



Undecidable Problems

P and NP Problems

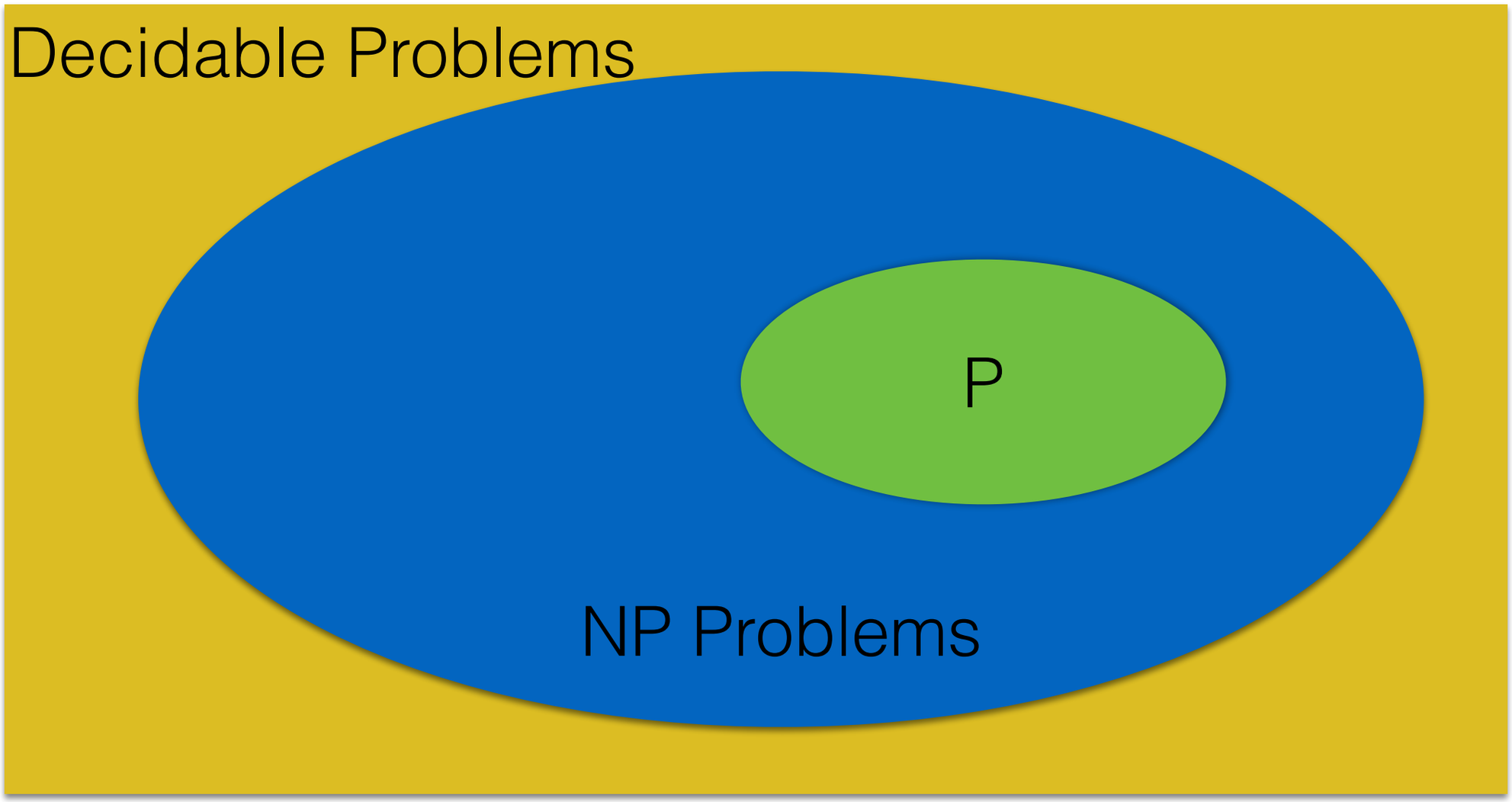
- The class P contains all problems that are solvable in polynomial time on a deterministic machine (most of the problems discussed in this course are in P).

- Clearly, all problems in P are also in NP.

- Surprisingly, it is **unknown** if there are problems in NP (i.e. with proofs that can be verified in polynomial time), that cannot be SOLVED in polynomial time.

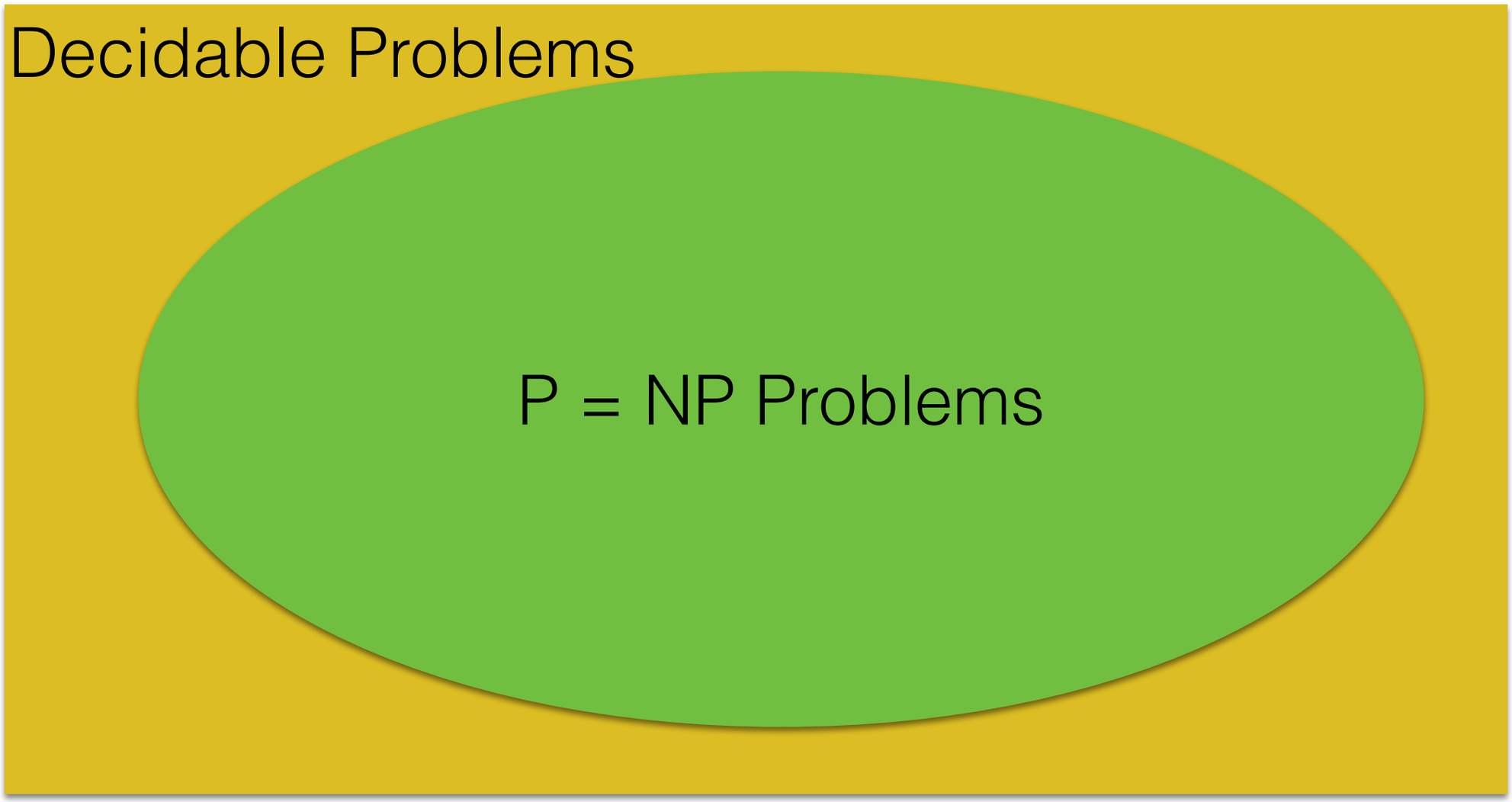
$$P = NP \qquad \text{vs.} \qquad P \subsetneq NP$$

P and NP Problems



if $P \subsetneq NP$

P and NP Problems



if $P = NP$

NP-Complete Problems

- An NP problem is NP-Hard if it is at least as hard as any problem in NP. It is NP-Complete if it is additionally in NP.

- How de we know that a given problem p is NP-Complete (NPC)?

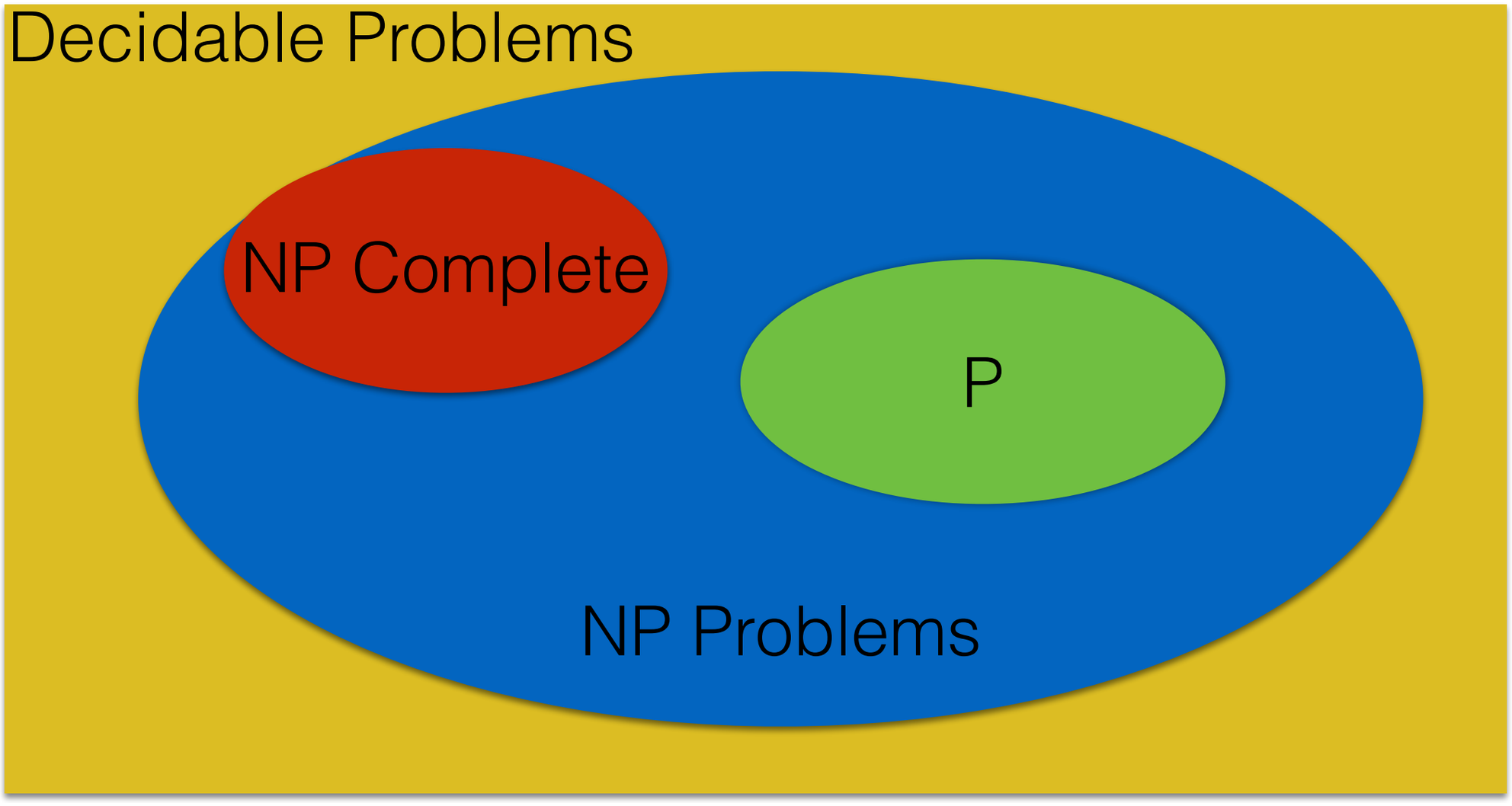
  - Any instance of any problem q in NPC can be *transformed* into an instance of p in polynomial time.

  - This is also called a **reduction of q to p.**

Reductions

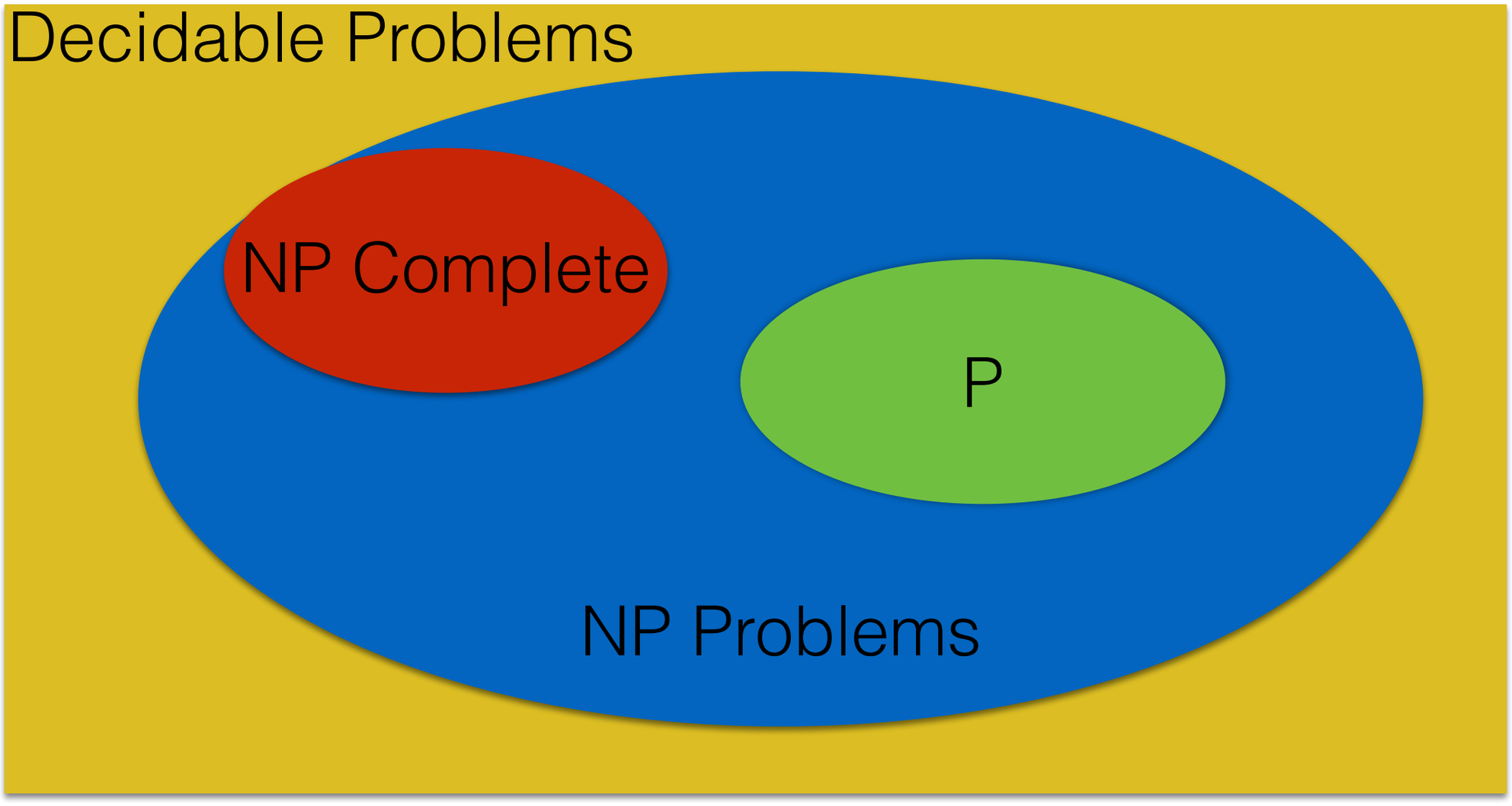
- Provide a mapping so that any instance of q can be transformed into an instance of p.
- Solve p and then map the result back to q.
- These mappings must be computable in (deterministic) polynomial time.

Problem Classes



if P ≠ NP

Problem Classes



if P ≠ NP

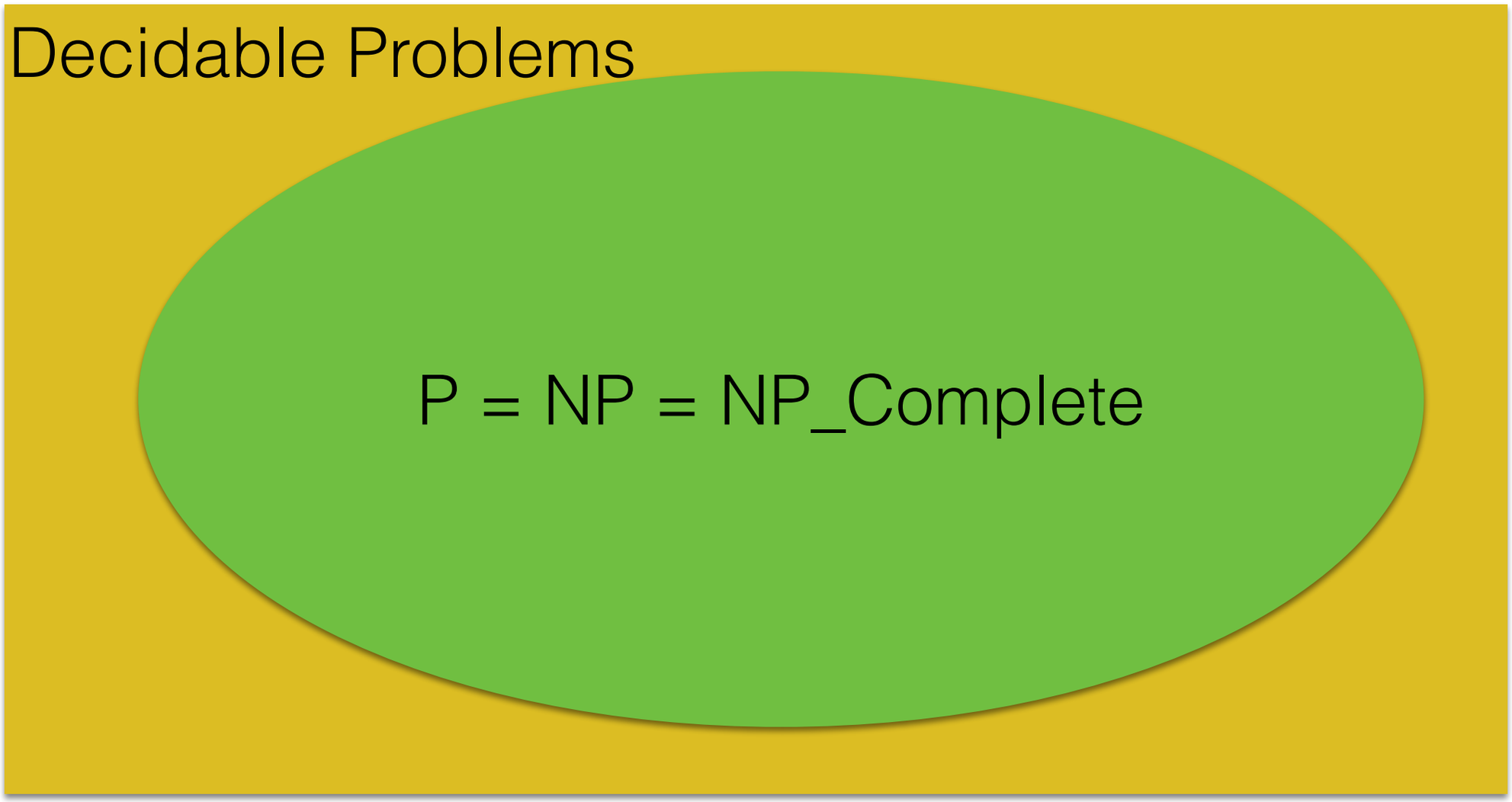
Importance of the NP-Complete Class

- Any other problem in NP can be transformed into an NP-Complete problem.

- If a polynomial time solution exists for any of these problems, there is a polynomial time solution for all problems in NP!

- To show that a new problem is NP-Complete, we show that another NP-complete problem can be reduced to it.

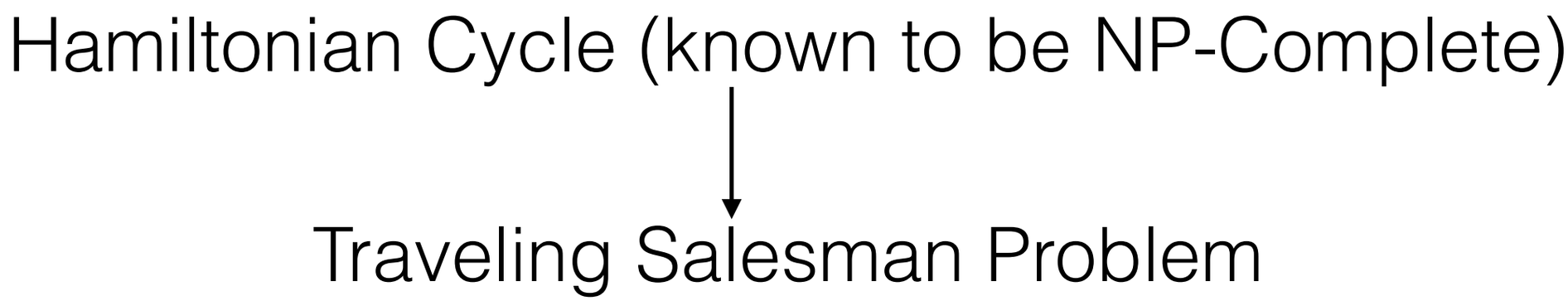
P and NP Problems



if P = NP

Example Reduction

- Assume we know the Hamiltonian cycle problem is NP-Complete.
- To show that TSP is NP-Complete, we can reduce Hamiltonian Cycle to it.



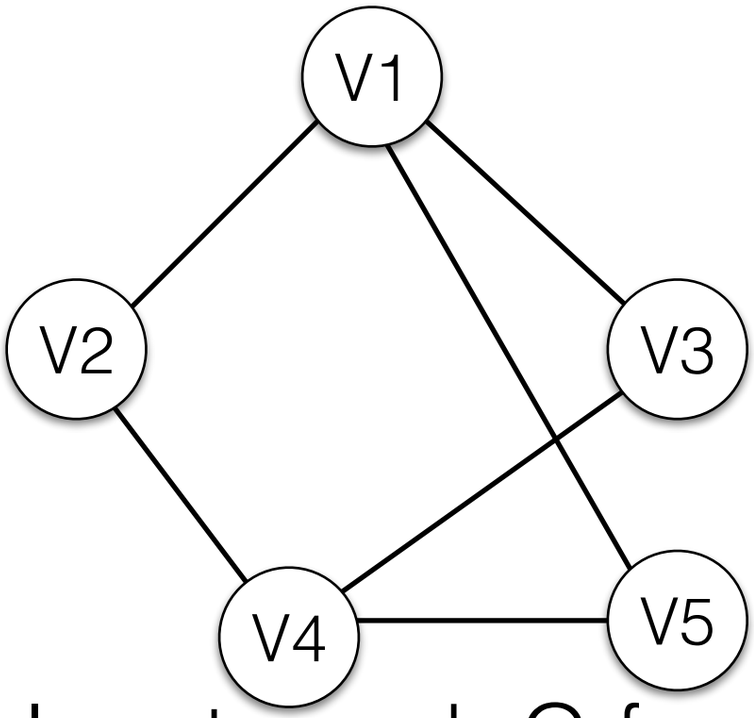
Reducing Hamiltonian Cycle to TSP

- We want to know if the input graph G = (V,E) contains a Hamiltonian Cycle.

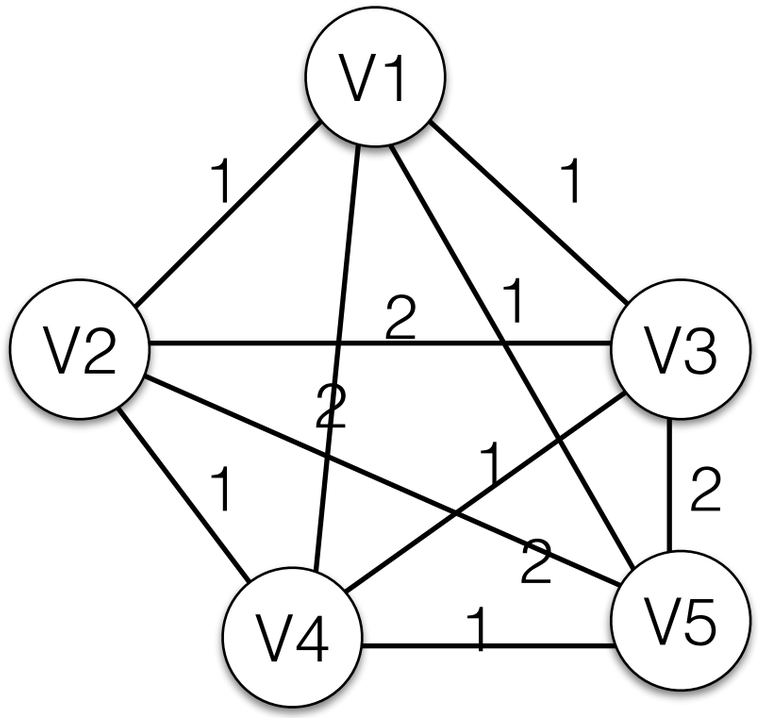
- Construct a complete graph G’ over V.

- Set the cost of all edges in G’ that are also in E to 1.0. Set the cost of all other edges to 2.0.

Reducing Hamiltonian cycle to TSP



Input graph G for Hamiltonian Cycle



Input graph G' for TSP

- Resulting TSP decision problem:
  - Does G' contain a TSP tour with cost ≤ |V|
- G contains a Hamiltonian Cycle if and only if G' contains a TSP tour with cost = |V|

Approximating TSP with MST

- A better solution than the greedy solution can be achieved using Minimum Spanning Trees as an approximation.

- The MST approximation only works if the edge costs obey the triangle inequality.

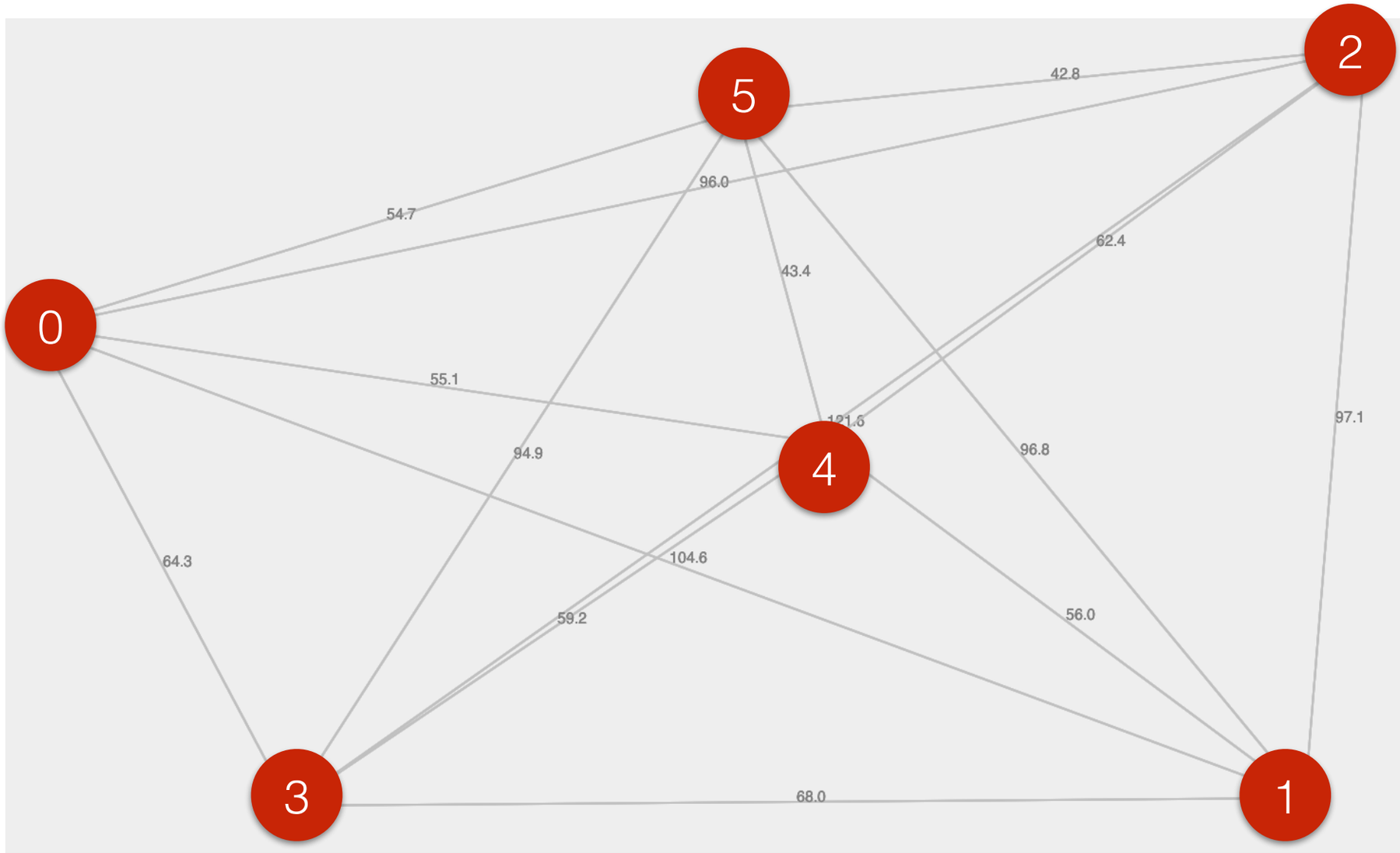
Triangle Inequality

$$c \leq a + b$$
$$b \leq a + c$$
$$a \leq c + b$$

- Euclidean distances obey the triangle inequality.

Approximating TSP with MST

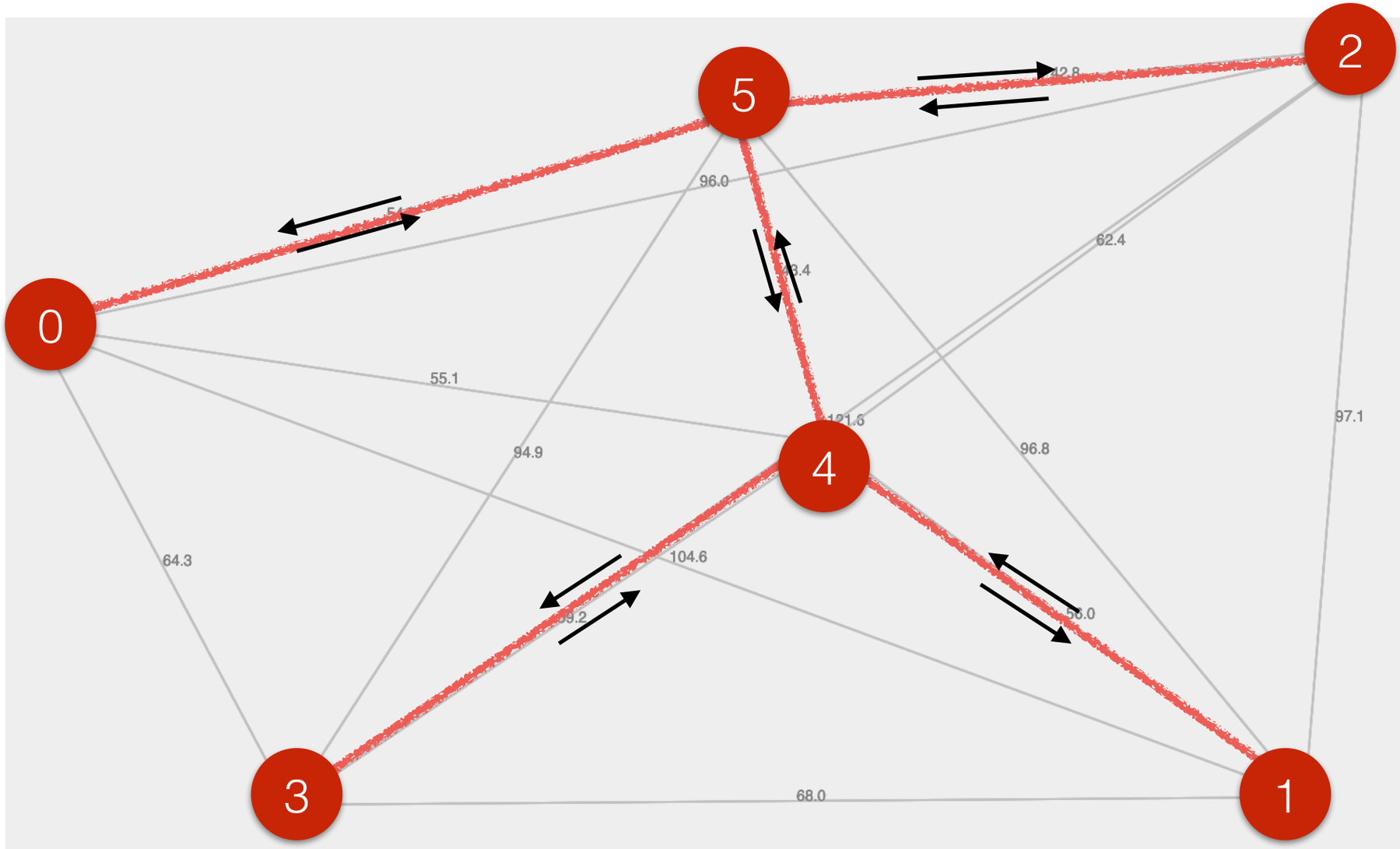
Step 1: Compute MST.



Approximating TSP with MST

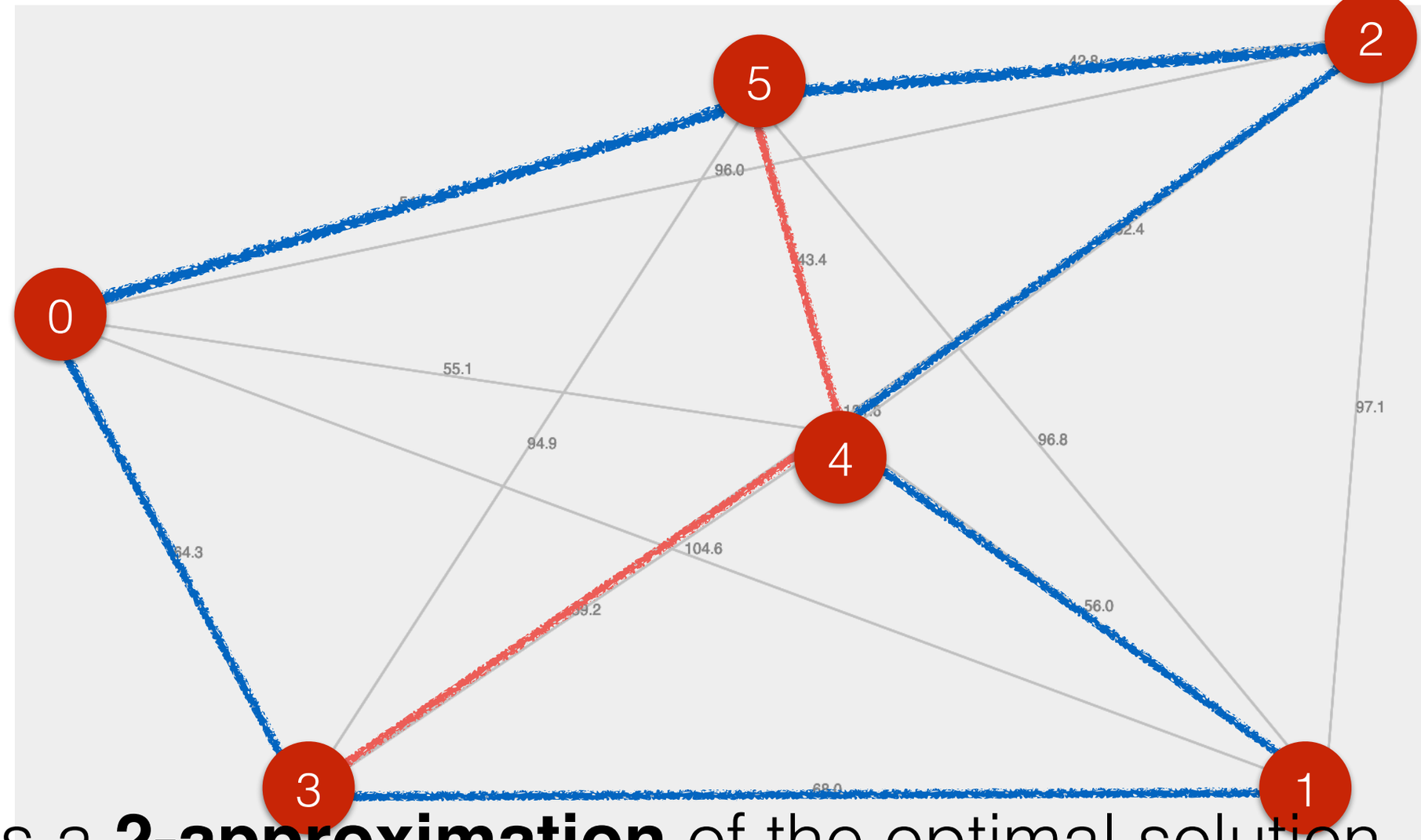
Step 1: Compute MST.

If we relax the condition that every vertex can only be visited once, we have a tour of cost **2 x cost of MST**.



Approximating TSP with MST

Step 2: Shortcut the tree to create a cycle.
Can show that because of the triangle inequality
cost(cycle) ≤ 2 x cost(MST)



The MST solution is a **2-approximation** of the optimal solution.

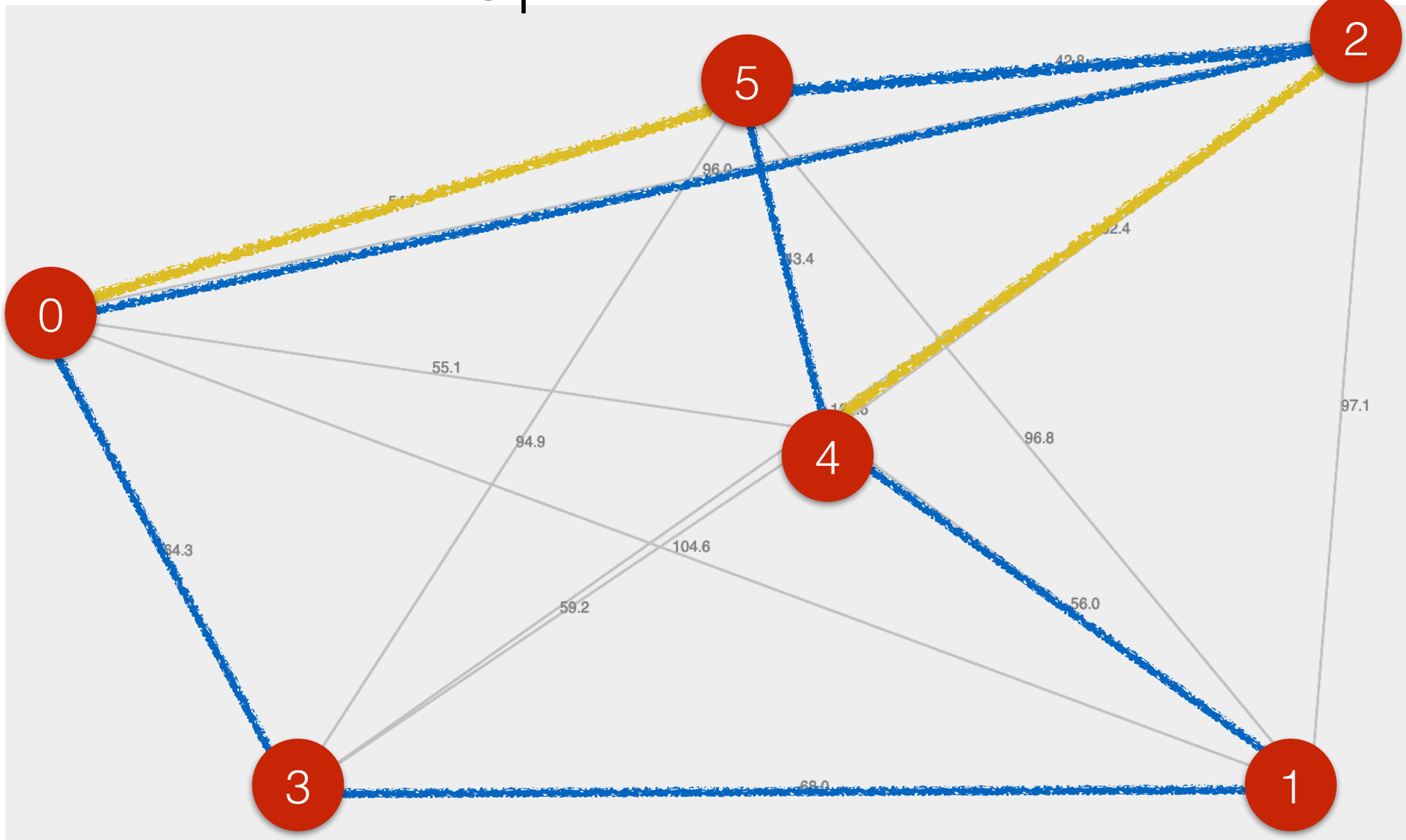
Iterative improvement of TSP solutions: 2-opt

- Once a cycle has been found (for example using the Nearest Neighbor method), we can improve the solution by:

- Removing two edges and adding two new edges connecting the same vertices.

Iterative Improvement: 2-Opt

- Once a cycle has been found
  (for example using the Nearest Neighbor method):
  - Repeatedly replace two edges with edges of lower cost, connecting the same vertices.
  - This can be generalized to *k*-Opt.



Undecidable Problems

- Are there problems that are impossible to solve?

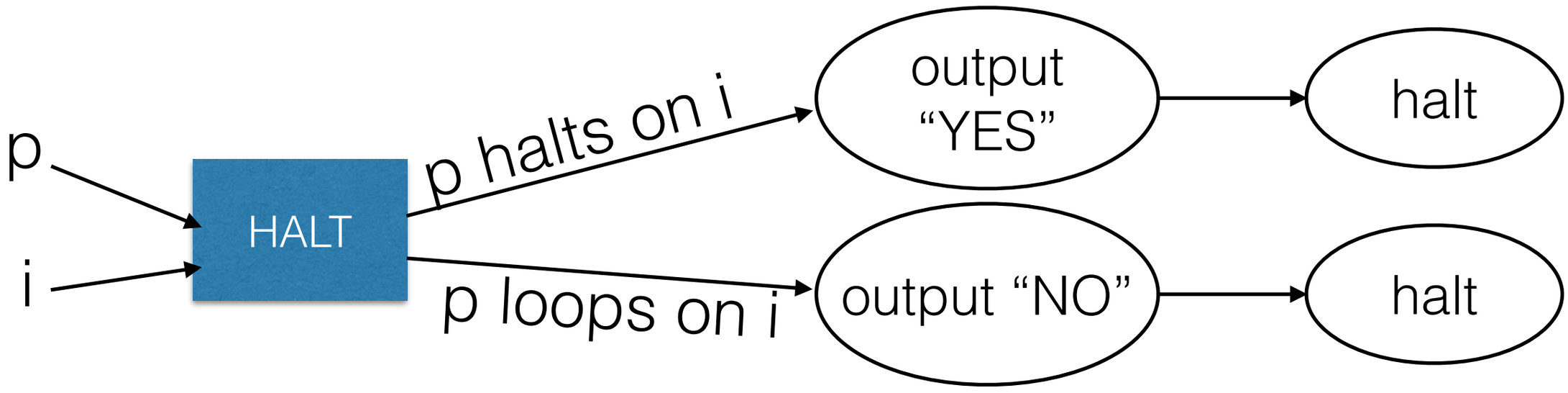
- Halting Problem:
  - Given a program description and some input, determine if the program will terminate (halt) or run forever (loop).

    This problem is **recursively undecidable.**

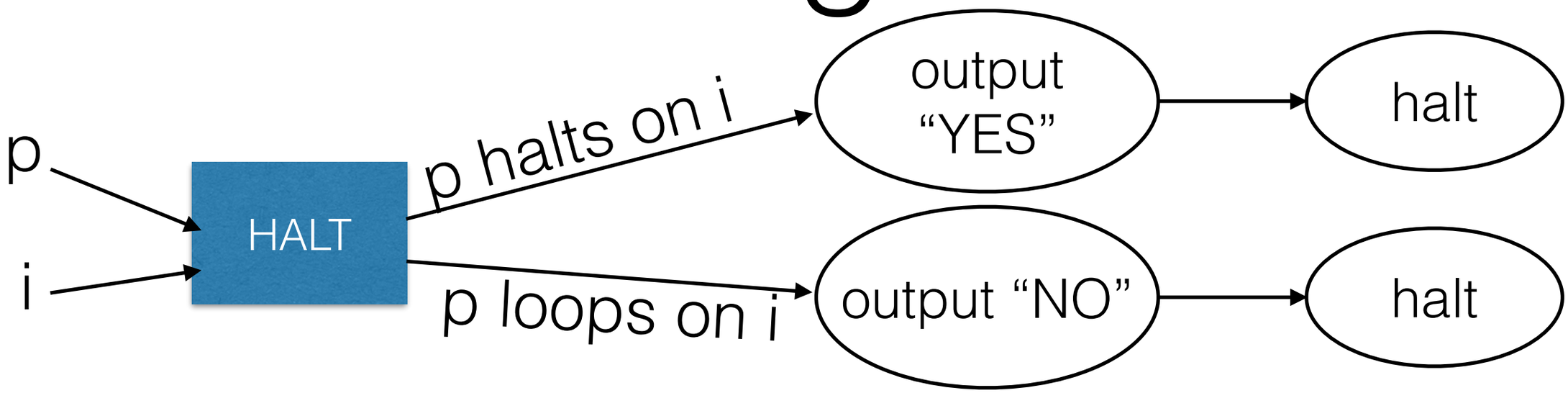
    Turing 1936

The Halting Problem

- Assume we wrote a program called HALT(p,i)
  - HALT outputs “YES” and halts if *p* halts on *i.*
  - HALT outputs “NO” and halts if *p* loops on *i*.

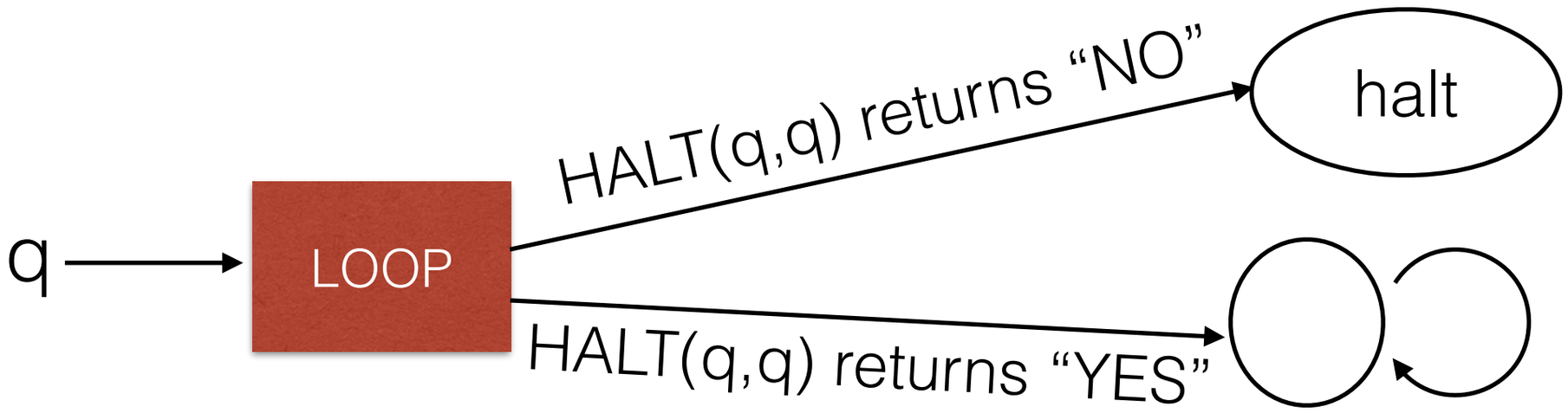


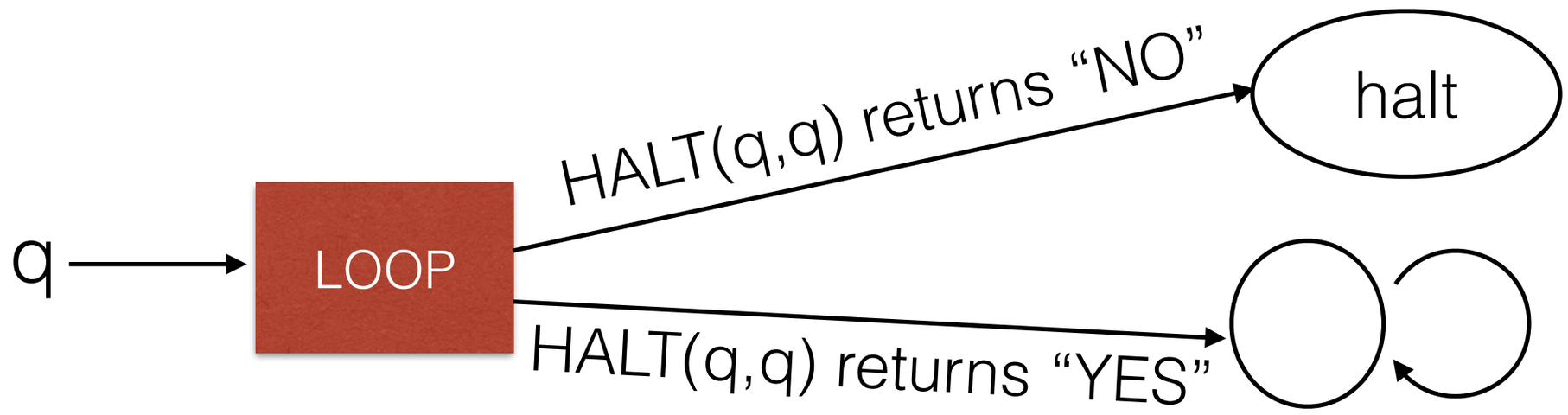
The Halting Problem



Write another program called LOOP(*q*)

- LOOP halts if HALT(q,q) returns "NO"
- LOOP loops if HALT(q,q) returns "YES"





What happens if we run LOOP(LOOP)?

- Assume LOOP(LOOP) halts
  - Then HALT(LOOP,LOOP) must have returned “NO”.

- Assume LOOP(LOOP) loops.
  - Then HALT(LOOP,LOOP) must have returned “YES”.