CS3063 Theory of Computing

Semester 4 (21 Intake), Jan – May 2024

Lecture 14

Decidability (Solvability) – 2

Sanath Jayasena

Announcements

- The last lecture (L14)
- Please complete online **student feedback** on Moodle

- **Final Exam** (physical exam at campus), worth 70%
  – On 17th May (Fri) at 1.00pm, 2 hours long
  – Closed book / closed notes
  – Will evaluate all topics covered in the semester
  – Past final exam papers on Moodle

Outline:

Lecture 14

Decidability - 2

- Decidability – contd…
  - Unsolvable problems
  - Reductions, examples
- Intractable Problems
  - Overview
  - NP-Completeness

PART 1

Outline:

Lecture 14

Decidability - 2

- Decidability – contd…
  - Unsolvable problems
  - Reductions, examples
- Intractable Problems
  - Overview
  - NP-Completeness

Problems: Solvable, Unsolvable

- A class of problems with two outputs (yes/no), ***decision problems***, is said to be
  - **Solvable** (**decidable**): if there exists some definite algorithm which always terminates (halts) with output either yes or no
  - **Unsolvable** (**undecidable**): otherwise

Reducing Decision Problems

- If P1 and P2 are decision problems, P1 is ***reducible*** to P2 (denoted P1≤ P2) if there is an algorithmic procedure to, given instance I of P1, find an instance F(I) of P2 so that for every I, answers for I and F(I) are the same
  - This can be stated in terms of ***languages***
- If P1 ≤ P2, we can conclude
  - If P2 is solvable, then P1 is solvable
    - Solving P1 cannot be harder than solving P2
    - If an algorithm exists to solve P2 efficiently, it can solve P1 efficiently
  - If P1 is unsolvable, then P2 is unsolvable

Example Unsolvable Problems

- ***Self-accepting***
  - Given a TM *T*, does *T* accept the string **e**(*T*)?

- **e**(*T*) is the encoding of *T* as a string
- Recall our discussion on *universal TMs*
  - To a universal TM T_u, we give as input a specific TM T_1 encoded as a string **e**(T_1) followed by (the input string **z** to T_1) encoded as **e**(**z**)

Example Unsolvable Problems

- ***Accepts***
  - Given a TM *T* and a string ***w***, is ***w*** in L(*T*)?
- Obvious approach?
  - Give the string ***w*** to *T* and see what happens
  - Works only if *T* halts but not if loops forever
- Can prove unsolvable by reducing ***Self-accepting*** to ***Accepts***
  - (Theorem 11.5, p. 413)

Example Unsolvable Problems

- ***Halts***
  - Given a TM *T* and a string ***w***, does *T* halt on input ***w*** ?
- Known as ***The halting problem*** (***HP***)
- The most well-known unsolvable problem
- Consider a computer program you wrote
  - There cannot be any general method to test it and decide whether it will terminate for a given input

Discussion: Halting Problem

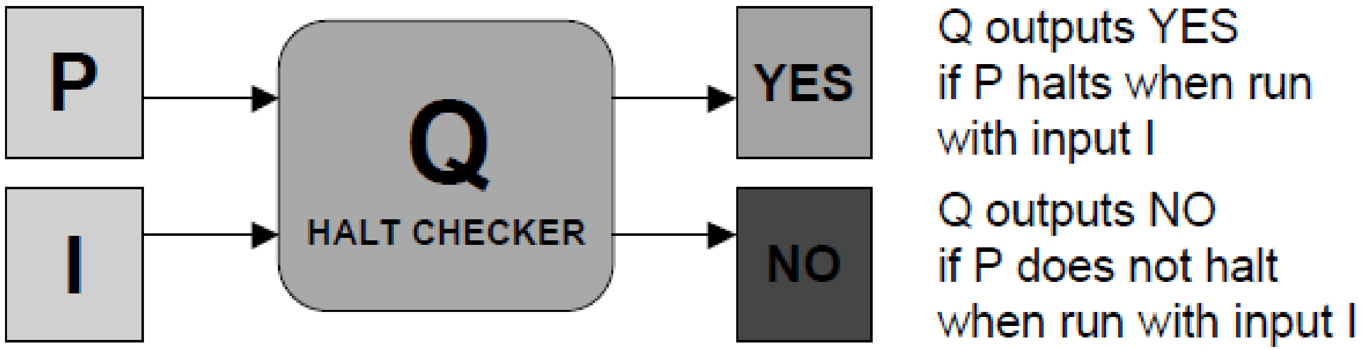
- Can we write a general program **Q** that takes as its input any program **P** and an input **I** and determines if program **P** will terminate (halt) when run with input **I**?
  - **Q** will output: **YES** if **P** terminates successfully on input **I**, **NO** if **P** never terminates on input **I**

- This computational problem is undecidable!
  - No such general program **Q** can exist!

Discussion: Halting Problem

[Ref: http://www.cs.cmu.edu/~tcortina/15-105sp09/lectures.html]

- Proof by contradiction
  - Assume a program **Q** exists that requires a program **P** and an input **I**
  - **Q** determines if program **P** will halt when **P** is executed using input **I**

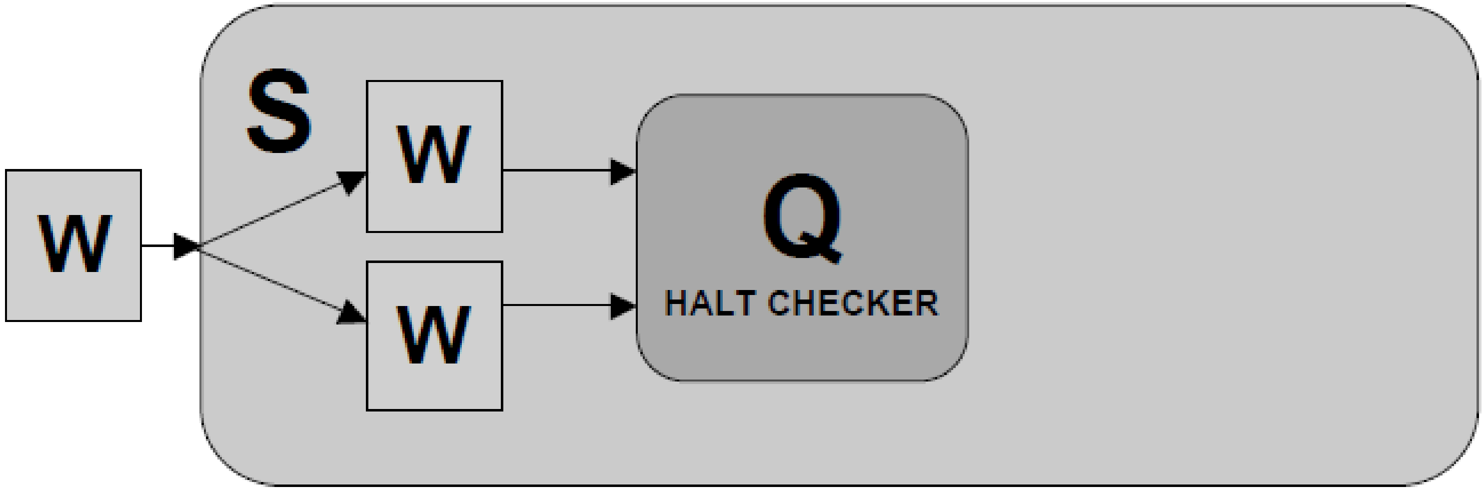


Discussion: Halting Problem

- Show that **Q** can never exist through contradiction
- Define a new program **S** that takes a program **W** as its input

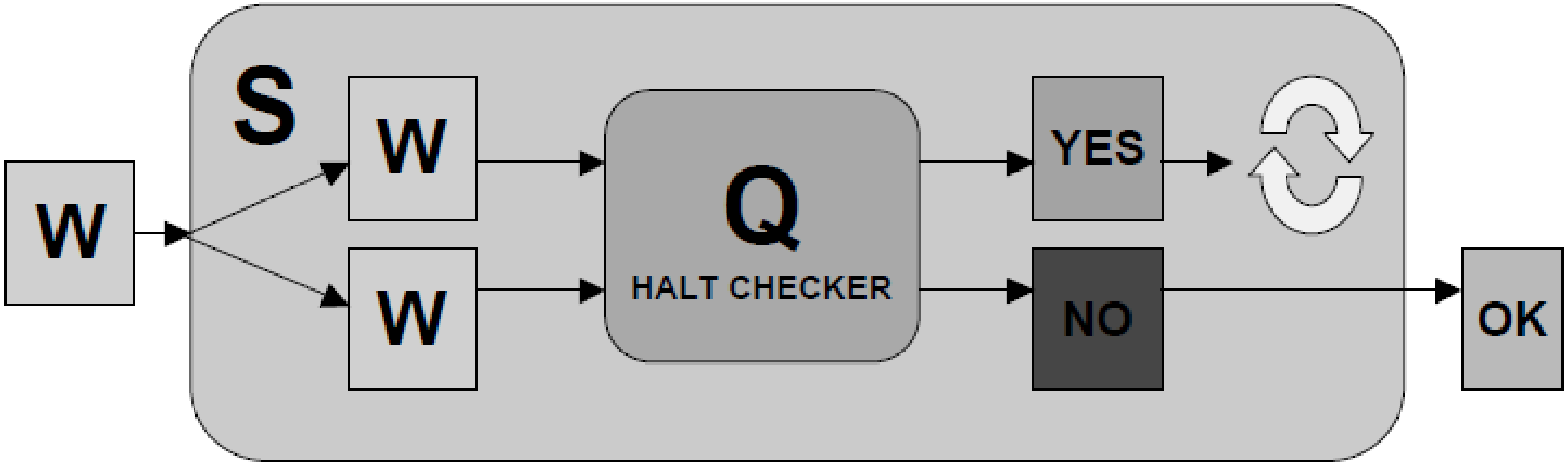
Discussion: Halting Problem

- **S** feeds **W** as the inputs for **Q** as the program and the program's input



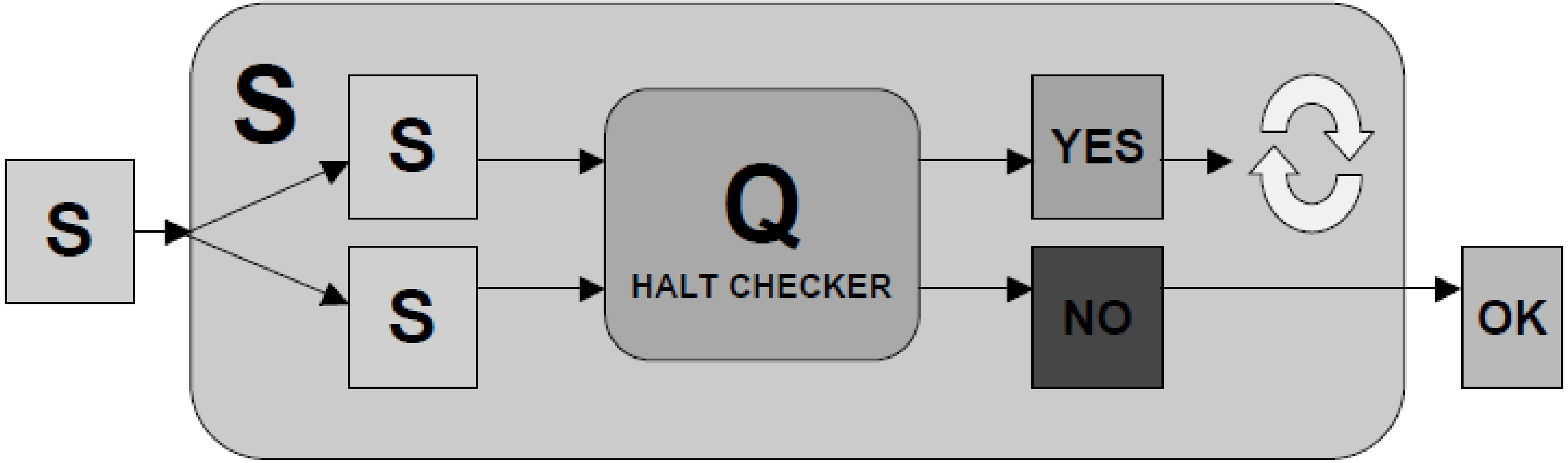
Discussion: Halting Problem

- Then **S** looks at the answer **Q** gives
  - – If **Q** answers YES, **S** purposely forces itself into an infinite loop
  - – If **Q** answers NO, **S** halts with an output of OK



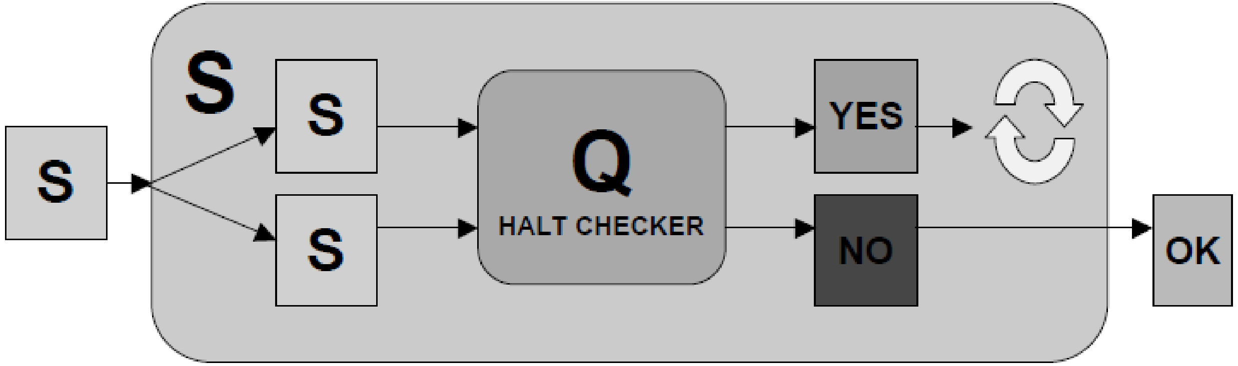
Discussion: Halting Problem

- Since **S** requires as its input a program, and **S** is a program, what happens if the input to **S** is itself?



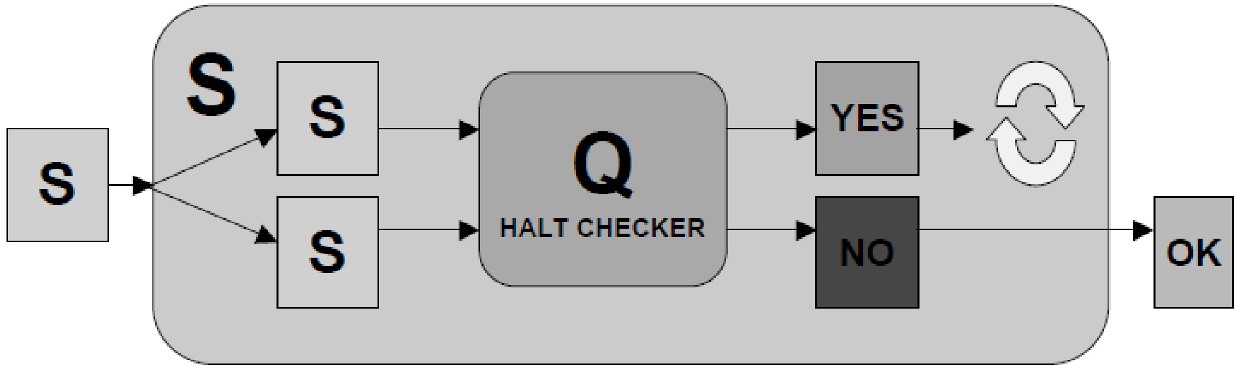
Discussion: Halting Problem

- In other words, **S** asks **Q**:
  - "What do I do if I execute using myself as input?"
- If **Q** outputs YES, it computes that **S** will halt if it uses itself as input
  - But if **Q** outputs YES, **S** purposely goes into an infinite loop when it uses itself as input



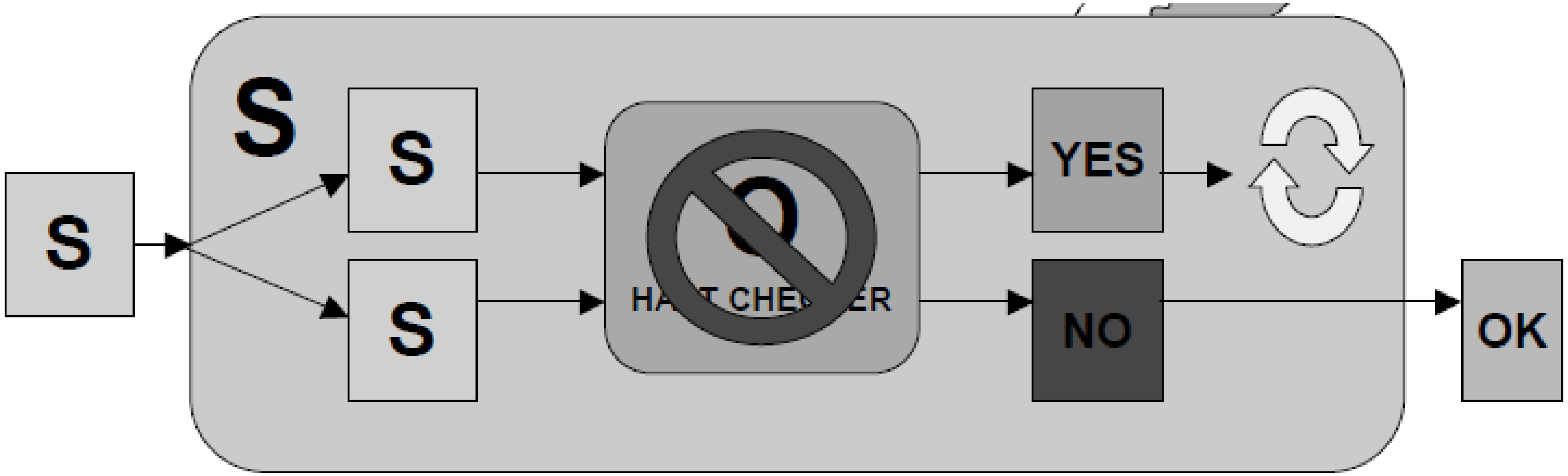
Discussion: Halting Problem

- **S** asks **Q**:
  - “What do I do if I execute using myself as input?”
- If **Q** outputs NO, it computes that **S** will not halt if it uses itself as input -will run forever
  - But if **Q** outputs NO, **S** purposely halts with the output “OK” when it uses itself as input



Discussion: Halting Problem

- We get contradictions no matter what **Q** outputs
- Our initial assumption must have been false! **Q** cannot exist !



Discussion: Halting Problem

- The Halting Problem is unsolvable
- We can **never** write a computer program that determines if ANY program halts with ANY input
  - It doesn’t matter how powerful the computer is
  - It doesn’t matter how much time we devote to the computation
  - It’s undecidable!

Other Unsolvable Problems

- **The Post Correspondence Problem (PCP)**
  - Type of a puzzle, by Emile Post in 1946
  - Consider a collection of dominos, each containing two strings, one on each side
  - An individual domino looks like → $\left[\frac{a}{ab}\right]$
  - A collection of dominos looks like $\left\{\left[\frac{b}{ca}\right],\left[\frac{a}{ab}\right],\left[\frac{ca}{a}\right],\left[\frac{abc}{c}\right]\right\}$

Other Unsolvable Problems

- **The Post Correspondence Problem (PCP)**
  - Task: make a list of dominos so that the string we get by concatenating the symbols on the top is the same as that on the bottom
    - This is called a **match;** E.g., $\left\{\left[\frac{a}{ab}\right],\left[\frac{b}{ca}\right],\left[\frac{ca}{a}\right],\left[\frac{a}{ab}\right],\left[\frac{abc}{c}\right]\right\}$
    - For some collections, there is no match
  - **PCP**: ***determine whether a given collection of dominos has a match***
    - This is unsolvable

Other Unsolvable Problems

- For any programming language, to determine whether or not a given program:
  - can loop for ever for some input
  - ever produces an output
  - eventually halts on the given input

Other Unsolvable Problems

- Fermat's Last Theorem
  - To determine whether or not a program -- that searches through all positive integers x, y, z and integer n > 2 for a solution to the equation $x^n + y^n = z^n$ -- will halt if and when a solution is found

Other Unsolvable Problems

- For formal languages to determine whether or not:
  a) Two context-free grammars are equivalent
  b) The language generated by a context-sensitive language is empty
  c) A given string belongs to a type 0 (RE) language

PART 2

Outline:

Lecture 14

Decidability - 2

- Decidability – contd…
  - Unsolvable problems
  - Reductions, examples
- Intractable Problems
  - Overview
  - NP-Completeness

Intractable Problems: Outline

- Introduction
- Example problems
- Class of $\mathcal{P}$ and $\mathcal{NP}$
- $\mathcal{NP}$-Completeness
- Reductions
- Proving $\mathcal{NP}$-Completeness

Introduction

- Problems that have algorithms whose complexities are polynomial in *n* where *n* is a suitably defined input size
  - "***tractable problems***"- not so hard
- Let us consider problems that have algorithms with exponential complexity
  - Even best known algorithms may take years or centuries on fastest computers
  - "***intractable problems***" - hard

Introduction

- In this discussion we generally consider the ***time*** complexity
  - Time required to solve a problem
- ***Space*** complexity is also of importance in some problems or situations
  - Space complexity refers to the space required (memory/storage) for a computation

Decision Problems

- We generally face optimization problems
  - Shortest path, knapsack, matrix-chain etc.,...
- $\mathcal{NP}$-Completeness restricts attention to *decision problems*
  - They have *either **yes** or **no** as the solution*
  - But have to provide an additional input to specify a problem instance

Example

- A *clique* is a complete subgraph (each pair of vertices is connected by an edge)
  - Size of a clique is the number of its vertices
- *The clique problem*
  - **Optimization problem**: Given a graph G=(V,E), find the clique of maximum size
  - **Decision problem**: Given a graph G=(V,E) and an integer *k*, is there a clique of size *k*?

Example ...contd

- Consider the decision problem
  - Naïve approach: list all *k*-subsets of V and check each to see if it forms a clique
  - Complexity proportional to n^k, where n=|V|, but k is not a constant
- An efficient algorithm is *unlikely* to exist
  - No one has found one, but…
  - No one has proved no such algorithm exists
- The clique problem is **NP**-complete !

More Example Problems

- Graph Coloring
- Subset Sum
- Satisfiability
- Hamiltonian Cycle
- Traveling Salesperson

Graph Coloring Problem

- Given a graph, how to color vertices so that adjacent ones have different colors
  - Chromatic number is the smallest number of colors needed to color a graph
- *The graph coloring problem*
  - **Optimization problem**: Given a graph G=(V,E), find the chromatic number
  - **Decision problem**: *Given a graph G=(V,E) and an integer k, is G k-colorable?*

Subset Sum Problem

- Given a set *S* of positive integers and an integer *k*
- *Is there a subset R of S such that the sum of the elements in R is equal to k?*
- Example
  - *S*={1,16,64,256,1040,1041,1093,1284,1344} and *k*=3754
  - *R*={1,16,64,256,1040,1093,1284} is a solution

Satisfiability Problem

- Given a Boolean formula is it satisfiable?
  - Is there an assignment of values 0 or 1 to variables so that the formula evaluates to 1?
- *Conjunctive Normal Form* (CNF)
  - A *clause* is the OR of some *literals*
  - A Boolean formula consisting of several clauses separated by AND is a CNF formula
  - Example $$(a+b+c)(\bar{b}+d+\bar{e}+f)(\bar{a}+e)$$

Satisfiability Problem ...contd

- 3-CNF: a CNF formula in which each clause has 3 literals
  - E.g., $(a+b+\overline{c})(d+\overline{e}+f)(\overline{a}+\overline{f}+g)$

- *Given a 3-CNF formula, is it satisfiable?*
  - That is: is there an assignment (to variables) that evaluates the formula to 1?
  - This is also called the 3-CNF-SAT problem

Hamiltonian Path Problem

- A Hamiltonian path of a graph
  - A simple path that passes through every vertex exactly once
- *Does a given undirected graph has a Hamiltonian path?*

- Can also specify for directed graphs

Traveling Salesperson Problem

- Known as **TSP** or minimum tour problem
- A salesperson wants to minimize total traveling cost (distance or time) required to visit a set of cities and return to the starting point
- *Given a weighted, complete graph and an integer **k**, is there a Hamiltonian cycle with total weight at most **k**?*

Class of $\mathbf{P}$

- An *algorithm* is *polynomially bounded* if its worst-case complexity is bounded by a polynomial of the input size
- *Polynomially bounded problem*: one that has a polynomially bounded algorithm
- $\mathbf{P}$ is the class of decision problems that are polynomially bounded

Class of NP

- For any of the example decision problems discussed, one may have a "proposed solution" that can be checked
- **NP** is the class of decision problems for which a given proposed solution for a given input can be checked in polynomial time to see if it really is a solution
  - (a loose definition)

Encoding of a Problem

- Inputs for a problem and proposed solutions described by strings of symbols
- Need conventions to describe graphs, sets, functions etc., using the symbols i
- The set of conventions for a particular problem is the *encoding of the problem*
  - An input and a proposed solution can be any string from the character set
  - Checking a proposed solution means checking that the string makes sense

Nondeterministic Algorithms

- Useful to classify problems
- Such an algorithm has *three phases*
  - *Nondeterministic “guessing” phase*: arbitrary string, ***s***, is written starting at some place in memory (***s*** may differ for each time it is run)
  - *Deterministic “verifying” phase*: a normal algorithm will consider the input to the decision problem and ***s***; may return true/false
  - *Output phase*: if verifying phase outputs true, then outputs yes; else no output

Class of NP, Again

- **NP** is the class of decision problems for which there is a polynomially bounded nondeterministic algorithm

- Examples
  - Clique, graph coloring, Hamiltonian path, subset sum, satisfiability, TSP, …

Relationship Between **NP** and **P**

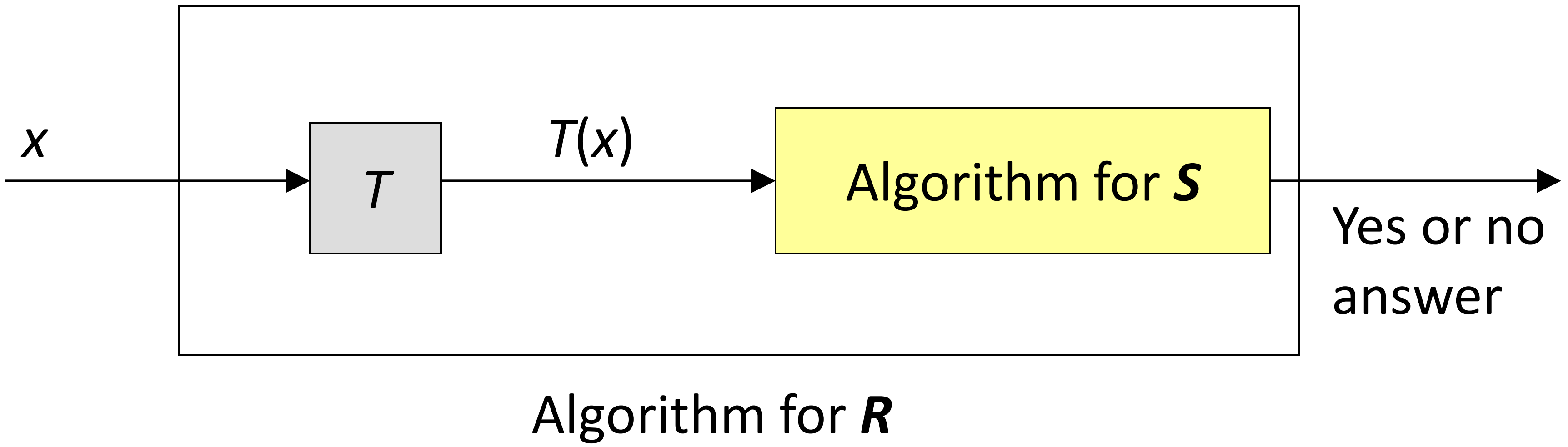
- It is not known whether **P=NP** or whether **P** is a proper subset of **NP**
- It is believed **NP** is much larger than **P**
  - But no problem in **NP** has been proved as not in **P**
  - No known deterministic algorithms that are polynomially bounded for many problems in **NP**
  - So, "does **P** = **NP**?" is still an open question!

NP-Completeness

- "*NP-complete problems*": the hardest problems in **NP**
- Interesting property
  - If any *one* **NP**-complete problem can be solved in polynomial time, then *every* problem in **NP** can also be solved similarly
  - (This is why many believe **P≠NP**)
- E.g.,: satisfiability, clique, graph coloring, Hamiltonian path, subset sum, TSP, …

Polynomial-time Reductions

- We use *reductions* (or *transformations*) to prove that a problem is **NP**-complete



- x is an input for ***R***; $T(x)$ is an input for ***S***

Polynomial-time Reductions ...contd

- We want to solve a problem ***R***; we already have an algorithm for ***S***
- We have a transformation function *T*
  - Correct answer for ***R*** on *x* is "yes", iff the correct answer for ***S*** on *T*(*x*) is "yes"
- Problem ***R*** is *polynomially reducible* to ***S*** if such a transformation *T* can be computed in polynomial time
- The point of reducibility: ***S*** is at least as hard to solve as ***R***

NP-Hard, **NP**-Complete Problems

- If ***R*** is polynomially reducible to ***S*** and ***S*** is in **P**, then ***R*** is also in **P**
- ***S*** is **NP**-hard if every problem ***R*** in **NP** is polynomially reducible to ***S***
  - **NP**-hard does not mean “in **NP** and hard” but “at least as hard as any problem in **NP**”
- ***S*** is **NP**-complete if ***S*** is in **NP** and ***S*** is **NP**-hard

Important Historical Results

- *(Stephen) Cook's Theorem*
  - The satisfiability problem is **NP**-complete
- Work of *Richard Karp*
  - Decision versions of several optimizations problems shown to be **NP**-complete
- With Karp's work, many problems for which polynomially bounded algorithms were being sought unsuccessfully were shown to be **NP**-complete by others

How to Prove a Problem *S* is **NP**-Complete?

1. Show ***S*** is in **NP**
2. Select a known **NP**-complete problem ***R***
   – Since ***R*** is **NP**-complete, all problems in **NP** are reducible to ***R***
3. Show how ***R*** can be polynomially reducible to ***S***
   – Then all problems in **NP** can be polynomially reducible to ***S*** (because polynomial reduction is transitive)
4. Therefore ***S*** is **NP**-complete

Importance of NP-Completeness

- NP-complete problems are “intractable”
- Important for algorithm designers, engineers
- Suppose you have a problem to solve
  - Your colleagues have spent a lot of time to solve it exactly but in vain
  - See if you can prove that it is NP-complete
  - If yes, then spend your time developing an *approximation (heuristic) algorithm*
- Many natural problems can be NP-complete

Conclusion

- We discussed
  - Unsolvable problems
  - Intractable problems and NP-completeness
- CA (Assignments, quizzes,…)
- Final exam (see course outline also)
  - Worth 70%
- Please fill the online feedback form