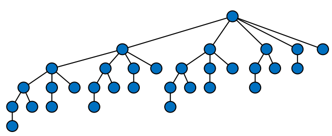
# **Lecture 19.** Approximation Algorithms

**CpSc 8400: Algorithms and Data Structures**

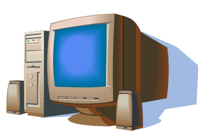
**Brian C. Dean**



**School of Computing**
**Clemson University**
**Spring, 2016**

## A Super-Brief Introduction to the Theory of Computation

What is a computer?



What is a computational problem?

What is an algorithm?

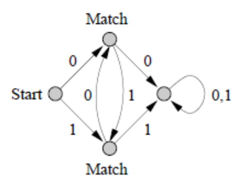
2

# What is a Problem?

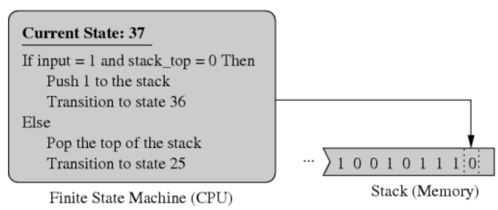
- A **decision problem** is a problem with a yes/no answer.
  - E.g., “Is this graph connected?”
- We can encode a decision problem by the set of binary strings corresponding to ‘yes’ inputs (binary encodings of connected graphs, in our example).
- An algorithm or machine that “solves” a decision problem is one that can “recognize” precisely all of binary strings corresponding to ‘yes’ inputs.



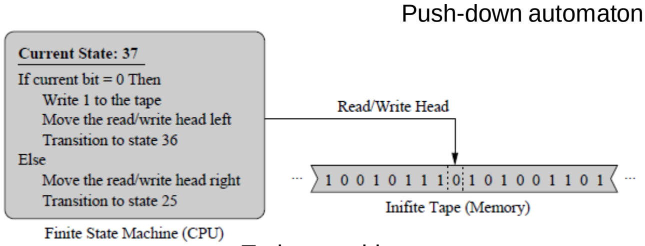
# What is an Algorithm / Computer?




Finite-state automaton
*(recognizes binary strings of the form (0(10)*)|(1(01)*))*




Push-down automaton




Turing machine

## Different Models of Algorithms / Computing Machines

- Simple models of computation:
  - Finite state automata
  - Pushdown automata
  - Turing machines
- "Non-regular" problems cannot be solved by FSAs, but these might be solvable with PDAs.
- "Non-context-free" problems cannot be solved by PDAs, but these might be solvable with a Turing machine.
- "Undecidable" problems cannot be solved by Turing machines, and according to Alan Turing, these cannot be solved on *any* computing machine!
  - **Alan Turing (1930s):** no computing machine is more powerful than a Turing machine in terms of the set of problems it can solve.
  - The **halting problem** is a famous undecidable problem.

5

## From Computability to Complexity Theory

- Turing's work helped define what problems can and cannot be solved by algorithms.
- The next question is what problems can be solved **efficiently** by algorithms in various models of computation.
  - Edmonds (1960s): "Efficiently" = "in polynomial time".
  - We define **P** to be the **complexity class** of all decision problems solvable in polytime on a Turing machine.
  - We tend to analyze running time in the RAM model of computation, but one can also show that an algorithm running in polytime on a RAM can be transformed into a polytime Turing machine algorithm, and vice-versa.

6

## The Classes P and NP

- One of the fundamental questions we would like to answer as algorithm designers: given a new problem, is this problem in **P**?
  - I.e., can we solve the problem in polytime?
  - For most of the problems we've seen in this course, the answer is yes.
- Unfortunately, for many problems in reality, it's not easy at all to tell if they belong to **P**.
- However, it is often much easier to show that these problems belong to a larger class called **NP**, containing problems that can be solved in non-deterministic polytime on a Turing machine.
  - **NP** = "non-deterministic polynomial", not "non-polynomial"!
  - Informally, **NP** contains decision problems for which the correct answer to a "yes" instance can be verified in polytime.



## Examples of Problems in NP

- **SAT:** Is a boolean formula satisfiable.
- **Traveling Salesman Problem (TSP)**: Does a given graph contain a "Hamiltonian cycle" (visiting every node exactly once) of cost ≤ k?
  - More generally, the TSP asks for the minimum-cost Hamiltonian cycle, but that is roughly equivalent in complexity to its "decision variant" above.
- Is this in **P**? Nobody knows (we suspect not).
- Is this in **NP**? Yes, we can verify in polytime whether a prospective solution is correct.

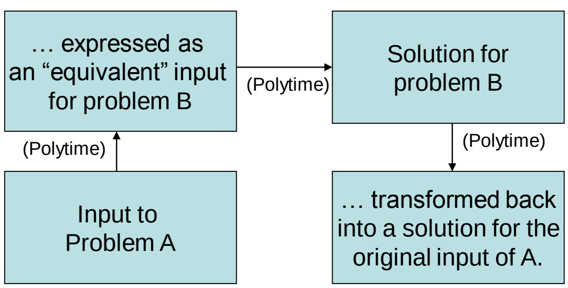
## The NP-Hard Problems

- A problem is said to be **NP-hard** if a polytime solution to this problem would imply a polytime solution to every problem in **NP**.
  - If the problem is also a decision problem in NP, then we say it is **NP-Complete**.
- **Cook-Levin Theorem** (1970s): The SAT problem (determining whether a boolean formula can be satisfied) is NP-Complete.
- Now that we have one NP-Complete problem, other problems can be shown to be NP-Complete via polytime **reductions**.



## Proving NP-Completeness with Polynomial-Time Reductions

- Problem A: Known to be NP-Hard
- Problem B: Want to prove this is NP-Hard
- Reduce **problem A to problem B** (note direction!)




- Example: reducing partition (NP-Hard) to knapsack.

## The NP-Hard Problems

- We now know of thousands of NP-hard problems. Why are these so important?
- A polytime solution to just one of these problems would give us a polytime solution to all of them!
- And since nobody has managed to solve an NP-hard problem in polytime yet, we strongly suspect that **P ≠ NP**.
- However, proving this is probably the biggest open problem in theoretical computer science today!
- So if you can show a problem is NP-hard, that's good evidence that it probably cannot be solved in polytime.

11

## How to Handle Hard Problems

- Many of the problems we encounter in practice happen to be NP-hard or worse.
- These problems need solving – we can't just give up on account of their being NP-hard!
- Basically, we can't have all three of these:
  1. An optimal solution
  2. To every possible input
  3. In polynomial time
- So we have several possible approaches:
  - (1+2). Study **heuristics** that tend to give us fast algorithms that work well "in practice" but don't have any theoretical performance guarantees.
  - (1+3). Study **special cases** that can be solved in polytime.
  - (2+3). Devise **approximation algorithms** that run fast, but might produce suboptimal solutions (hopefully within some provably close distance of an optimal solutions).

12

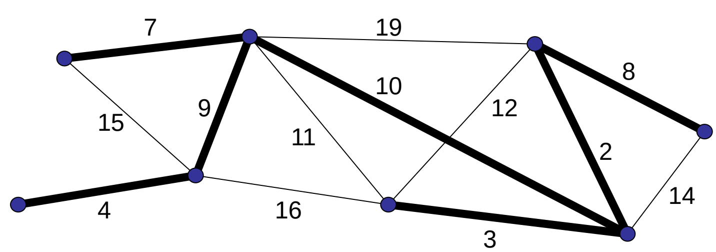
## Approximation Algorithms

- An **α-approximation algorithm** for a problem is an algorithm (typically that runs in polytime) whose solution never differs from an optimal solution by more than a factor of α.
  - The factor α is called the **performance guarantee** of the algorithm. The smaller, the better.
- The difficulty with approximation algorithms, of course, is proving that your solution is not far away from an optimal solution you don't know!
  - So a key step is to find useful ways to bound an optimal solution. For example, for TSP:

    cost(minimum spanning tree) ≤ cost(optimal TSP tour)

13

## Recall: The Minimum Spanning Tree Problem




- **Goal:** Find a minimum-cost subset of the edges in a graph that forms a tree, and that connects together all nodes.
- Very well-studied problem, and can be solved very efficiently.

14

# A 2-Approximation Algorithm for Metric TSP

- Let c(i, j) be the cost of movement from node i to node j in a graph.
- Suppose the c's are symmetric and satisfy the triangle inequality (a fairly common occurrence):

  $c(i, j) + c(j, k) \geq c(i, k)$ for all (i, j, k).

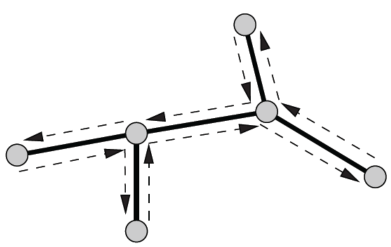
  (note that this implies that all edges are present in the graph)

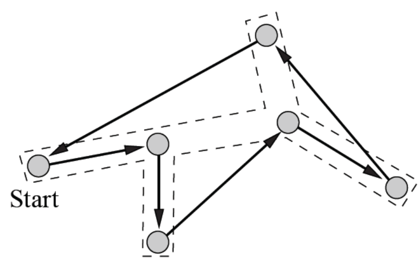
- Let T* be an optimal TSP tour. Recall that cost(MST) ≤ cost(T*).
- We'll show how to construct a tour T of cost at most 2cost(MST) ≤ 2cost(T*)…



# Walking Around a Tree and Introducing Shortcut Edges

- Start by walking "around" the tree this isn't a proper tour though; it visits nodes multiple times.
- Starting cost: 2cost(MST) ≤ 2cost(T*).
- Now add "shortcut" edges to convert into a proper tour; note this doesn't increase cost, due to the triangle inequality!





## A “Greedy” 2-Approximation for 0/1 Knapsack

- **Input:** N items, each with a size and value, and a capacity C.
- **Goal:** Find a maximum-value collection of items fitting inside a knapsack of capacity C.
- Suppose we sort the items so that

  val(1) / size(1) ≥ val(2) / size(2) ≥ …

  and greedily fill the knapsack in this order until we reach an overflowing item (which we allow to overflow). Our solution so far has value ≥ OPT…

17

## A “Greedy” 2-Approximation for 0/1 Knapsack

- **Input:** N items, each with a size and value, and a capacity C.
- **Goal:** Find a maximum-value collection of items fitting inside a knapsack of capacity C.
- Suppose we sort the items so that

  val(1) / size(1) ≥ val(2) / size(2) ≥ …

  and greedily fill the knapsack in this order until we reach an overflowing item (which we allow to overflow). Our solution so far has value ≥ OPT…
- Now take the better of two solutions:
  - All the items except the overflowing one.
  - Just the single overflowing item.
  - At least one of these must have value ≥ OPT/2

18

# A Polynomial-Time Approximation Scheme (PTAS) for 0/1 Knapsack

- Let $\varepsilon > 0$ be any constant.
- Use 2-approximation algorithm to compute B such that $OPT/2 \leq B \leq OPT$.
- Round down every item's value to the next-lowest multiple of $\varepsilon B / n$.
- Total value lost by an optimal solution in the process: at most $\varepsilon B \leq \varepsilon OPT$.
- Now optimally solve the discretized problem with dynamic programming! This will give us a solution of value $\geq (1 - \varepsilon)OPT$ which, when regarded in terms of the original item values, will still have value $\geq (1 - \varepsilon)OPT$.

19