Lecture 21: Hardness of Approximation

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

November 21, 2024

Overview

- Background and Motivation
  - Why Hardness of Approximation?
  - How do we prove Hardness of Approximation?
  - Hardness of Approximation - Example
- Proofs & Hardness of Approximation
- Conclusion
- Acknowledgements

Why Study Hardness of Approximation?

- Since the 50s and 60s (before we "formally knew" about NP) researchers from many areas noticed that certain combinatorial problems were much harder to solve than others

Why Study Hardness of Approximation?

- Since the 50s and 60s (before we "formally knew" about NP) researchers from many areas noticed that certain combinatorial problems were much harder to solve than others
- What do we do when we see such a hard problem?

Why Study Hardness of Approximation?

- Since the 50s and 60s (before we "formally knew" about NP) researchers from many areas noticed that certain combinatorial problems were much harder to solve than others
- What do we do when we see such a hard problem?
  - design algorithm which is efficient on "most" instances and always gives us the exact/best answer

Why Study Hardness of Approximation?

- Since the 50s and 60s (before we "formally knew" about NP) researchers from many areas noticed that certain combinatorial problems were much harder to solve than others
- What do we do when we see such a hard problem?
  - design algorithm which is efficient on "most" instances and always gives us the exact/best answer
  - design (always) efficient algorithm, but finds sub-optimal solutions

Approximation Algorithms

Why Study Hardness of Approximation?

- Since the 50s and 60s (before we "formally knew" about NP) researchers from many areas noticed that certain combinatorial problems were much harder to solve than others
- What do we do when we see such a hard problem?
  - design algorithm which is efficient on "most" instances and always gives us the exact/best answer
  - design (always) efficient algorithm, but finds sub-optimal solutions

    Approximation Algorithms
  - For $\alpha \geq 1$, an algorithm is *α-approximate* for a minimization (maximization) problem if on every input instance the algorithm finds a solution with cost $\leq \alpha \cdot OPT$ ($\geq \frac{1}{\alpha} \cdot OPT$).

Why Study Hardness of Approximation?

- Since the 50s and 60s (before we "formally knew" about NP) researchers from many areas noticed that certain combinatorial problems were much harder to solve than others
- What do we do when we see such a hard problem?
  - design algorithm which is efficient on "most" instances and always gives us the exact/best answer
  - design (always) efficient algorithm, but finds sub-optimal solutions

    Approximation Algorithms

  - For $\alpha \geq 1$, an algorithm is *α-approximate* for a minimization (maximization) problem if on every input instance the algorithm finds a solution with cost $\leq \alpha \cdot OPT$ ($\geq \frac{1}{\alpha} \cdot OPT$).
- For some problems, it is possible to prove that even the design of approximation algorithms for certain values of α is impossible, unless P = NP (in which case we would have an exact algorithm).

  Hardness of Approximation

Why Study Hardness of Approximation?

- Since the 50s and 60s (before we "formally knew" about NP) researchers from many areas noticed that certain combinatorial problems were much harder to solve than others
- What do we do when we see such a hard problem?
  - design algorithm which is efficient on "most" instances and always gives us the exact/best answer
  - design (always) efficient algorithm, but finds sub-optimal solutions

    Approximation Algorithms

  - For $\alpha \geq 1$, an algorithm is *α-approximate* for a minimization (maximization) problem if on every input instance the algorithm finds a solution with cost $\leq \alpha \cdot OPT$ ($\geq \frac{1}{\alpha} \cdot OPT$).
- For some problems, it is possible to prove that even the design of approximation algorithms for certain values of α is impossible, unless P = NP (in which case we would have an exact algorithm).

  Hardness of Approximation

- Important to know the limits of efficient algorithms!

How do we Prove Hardness of Approximation?

- When we prove that a combinatorial problem $\mathcal{C}$ is NP-hard, we usually pick our favorite NP-complete combinatorial problem L and we show a *reduction* that

How do we Prove Hardness of Approximation?

- When we prove that a combinatorial problem $\mathcal{C}$ is NP-hard, we usually pick our favorite NP-complete combinatorial problem L and we show a *reduction* that
  - maps every YES instance of L to a YES instance of $\mathcal{C}$
  - maps every NO instance of L to a NO instance of $\mathcal{C}$

How do we Prove Hardness of Approximation?

- When we prove that a combinatorial problem $\mathcal{C}$ is NP-hard, we usually pick our favorite NP-complete combinatorial problem L and we show a *reduction* that
  - maps every YES instance of L to a YES instance of $\mathcal{C}$
  - maps every NO instance of L to a NO instance of $\mathcal{C}$
- For hardness of approximation what we would like is a (more robust) reduction of the form:
  - maps every YES instance of L to a YES instance of $\mathcal{C}$
  - maps every NO instance of L to a VERY-MUCH-NO instance of $\mathcal{C}$

Traveling Salesman Problem

- **Input:** set of points X and a symmetric distance function

$$d : X \times X \to \mathbb{R}_{\geq 0}$$

Traveling Salesman Problem

- **Input:** set of points X and a symmetric distance function

$$d : X \times X \to \mathbb{R}_{\geq 0}$$

- For any path $p_0 \to p_1 \to \cdots \to p_t$ in X, *length* of the path is sum of distances traveled

$$\sum_{i=0}^{t-1} d(p_i, p_{i+1})$$

Traveling Salesman Problem

- **Input:** set of points X and a symmetric distance function

$$d : X \times X \to \mathbb{R}_{\geq 0}$$

- For any path $p_0 \to p_1 \to \cdots \to p_t$ in X, *length* of the path is sum of distances traveled

$$\sum_{i=0}^{t-1} d(p_i, p_{i+1})$$

- **Output:** find a cycle that reaches all points in X of shortest length.

Traveling Salesman Problem

- **Input:** set of points X and a symmetric distance function

$$d : X \times X \to \mathbb{R}_{\geq 0}$$

- For any path $p_0 \to p_1 \to \cdots \to p_t$ in X, *length* of the path is sum of distances traveled

$$\sum_{i=0}^{t-1} d(p_i, p_{i+1})$$

- **Output:** find a cycle that reaches all points in X of shortest length.
- Definitely a problem we would like to solve
  - Efficient route planning (mail system, shuttle bus pick up and drop off...)

Traveling Salesman Problem

- **Input:** set of points X and a symmetric distance function

$$d : X \times X \to \mathbb{R}_{\geq 0}$$

- For any path $p_0 \to p_1 \to \cdots \to p_t$ in X, *length* of the path is sum of distances traveled

$$\sum_{i=0}^{t-1} d(p_i, p_{i+1})$$

- **Output:** find a cycle that reaches all points in X of shortest length.
- Definitely a problem we would like to solve
  - Efficient route planning (mail system, shuttle bus pick up and drop off...)
- One of the famous NP-complete problems

Hardness of Approximation - TSP

1. *General* TSP *without* repetitions (General TSP-NR)

Hardness of Approximation - TSP

1. *General* TSP *without* repetitions (General TSP-NR)
    - if $P \neq NP$ then there is no poly-time constant-approximation algorithm for General TSP-NR.

Hardness of Approximation - TSP

1. *General* TSP *without* repetitions (General TSP-NR)
    - if $P \neq NP$ then there is no poly-time constant-approximation algorithm for General TSP-NR.
    - More generally, if there is any function $r : \mathbb{N} \rightarrow \mathbb{N}$ such that $r(n)$ computable in polynomial time, then it is hard to $r(n)$-approximate General TSP-NR if we assume that $P \neq NP$

Hardness of Approximation - TSP

1. *General* TSP *without* repetitions (General TSP-NR)
   - if $P \neq NP$ then there is no poly-time constant-approximation algorithm for General TSP-NR.
   - More generally, if there is any function $r : \mathbb{N} \to \mathbb{N}$ such that $r(n)$ computable in polynomial time, then it is hard to $r(n)$-approximate General TSP-NR if we assume that $P \neq NP$
2. How does one prove any such hardness of approximation?

   By *reduction* to another NP-hard problem.

Hardness of Approximation - TSP

1. *General* TSP *without* repetitions (General TSP-NR)
   - if $P \neq NP$ then there is no poly-time constant-approximation algorithm for General TSP-NR.
   - More generally, if there is any function $r : \mathbb{N} \rightarrow \mathbb{N}$ such that $r(n)$ computable in polynomial time, then it is hard to $r(n)$-approximate General TSP-NR if we assume that $P \neq NP$
2. How does one prove any such hardness of approximation?

   By *reduction* to another NP-hard problem.
3. In our case, let's reduce it to the *Hamiltonian Cycle Problem*

Theorem

If there is an algorithm M which solves TSP without repetitions with α-approximation, then $P = NP$.

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.
2. Proof:

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.
2. Proof:
3. If we had an algorithm M which solved the α-approximate TSP without repetition problem, then
    - from graph $G(V, E)$, construct weighted graph $H(V, F, w)$ such that

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.
2. Proof:
3. If we had an algorithm M which solved the α-approximate TSP without repetition problem, then
   - from graph $G(V, E)$, construct weighted graph $H(V, F, w)$ such that
   - All edges $\{u, v\} \in F$ (that is, H is the complete graph on V)

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.
2. Proof:
3. If we had an algorithm M which solved the α-approximate TSP without repetition problem, then
   - from graph $G(V, E)$, construct weighted graph $H(V, F, w)$ such that
   - All edges $\{u, v\} \in F$ (that is, H is the complete graph on V)
   - $w(u, v) = \begin{cases} 1, & \text{if } \{u, v\} \in E \\ (1 + \alpha) \cdot |V|, & \text{if } \{u, v\} \notin E \end{cases}$

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.
2. Proof:
3. If we had an algorithm M which solved the α-approximate TSP without repetition problem, then
   - from graph $G(V, E)$, construct weighted graph $H(V, F, w)$ such that
   - All edges $\{u, v\} \in F$ (that is, H is the complete graph on V)
   - $w(u, v) = \begin{cases} 1, & \text{if } \{u, v\} \in E \\ (1+\alpha) \cdot |V|, & \text{if } \{u, v\} \notin E \end{cases}$
4. If G has a Hamiltonian Cycle, then OPT for the TSP is of value $\leq |V|$

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.
2. Proof:
3. If we had an algorithm M which solved the α-approximate TSP without repetition problem, then
   - from graph $G(V, E)$, construct weighted graph $H(V, F, w)$ such that
   - All edges $\{u, v\} \in F$ (that is, H is the complete graph on V)
   - $w(u, v) = \begin{cases} 1, & \text{if } \{u, v\} \in E \\ (1 + \alpha) \cdot |V|, & \text{if } \{u, v\} \notin E \end{cases}$
4. If G has a Hamiltonian Cycle, then OPT for the TSP is of value $\leq |V|$
5. If G has no Hamiltonian Cycle, then OPT for TSP must use an edge not in V, thus value is $\geq (1 + \alpha) \cdot |V|$

Hardness of Approximation

1. **Hamiltonian Cycle Problem:** given a graph $G(V, E)$, decide whether there exists a cycle $\mathcal{C}$ which passes through every vertex at most once.
2. Proof:
3. If we had an algorithm M which solved the α-approximate TSP without repetition problem, then
   - from graph $G(V, E)$, construct weighted graph $H(V, F, w)$ such that
   - All edges $\{u, v\} \in F$ (that is, H is the complete graph on V)
   - $w(u, v) = \begin{cases} 1, & \text{if } \{u, v\} \in E \\ (1 + \alpha) \cdot |V|, & \text{if } \{u, v\} \notin E \end{cases}$
4. If G has a Hamiltonian Cycle, then OPT for the TSP is of value $\leq |V|$
5. If G has no Hamiltonian Cycle, then OPT for TSP must use an edge not in V, thus value is $\geq (1 + \alpha) \cdot |V|$
6. Thus, M on input H will output a Hamiltonian Cycle of G, if G has one, or it will output a solution with value $\geq (1 + \alpha) \cdot |V|$

Complexity Classes

- **NP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine V, such that:

$$x \in L \Leftrightarrow \exists w \in \{0,1\}^{\text{poly}(|x|)} \text{ s.t. } V(x,y) = 1$$

Complexity Classes

- **NP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine V, such that:

$$x \in L \Leftrightarrow \exists w \in \{0,1\}^{\mathrm{poly}(|x|)} \text{ s.t. } V(x,y) = 1$$

- **BPP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine M, such that for every $x \in \{0,1\}^*$, we have

$$\Pr_{R \in \{0,1\}^{\mathrm{poly}(|x|)}} [M(x,R) = L(x)] \geq 2/3$$

Complexity Classes

- **NP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine V, such that:

$$x \in L \Leftrightarrow \exists w \in \{0,1\}^{\text{poly}(|x|)} \text{ s.t. } V(x,y) = 1$$

- **BPP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine M, such that for every $x \in \{0,1\}^*$, we have

$$\Pr_{R \in \{0,1\}^{\text{poly}(|x|)}}[M(x,R) = L(x)] \geq 2/3$$

- **RP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine M, such that:

$$x \in L \Rightarrow \Pr_{R \in \{0,1\}^{\text{poly}(|x|)}}[M(x,R) = 1] \geq 2/3$$

$$x \notin L \Rightarrow \Pr_{R \in \{0,1\}^{\text{poly}(|x|)}}[M(x,R) = 1] = 0$$

Complexity Classes

- **NP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine V, such that:

$$x \in L \Leftrightarrow \exists w \in \{0,1\}^{\mathsf{poly}(|x|)} \text{ s.t. } V(x,y) = 1$$

- **BPP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine M, such that for every $x \in \{0,1\}^*$, we have

$$\Pr_{R \in \{0,1\}^{\mathsf{poly}(|x|)}} [M(x,R) = L(x)] \geq 2/3$$

- **RP:** Set of languages $L \subseteq \{0,1\}^*$ such that there exists a poly-time Turing Machine M, such that:

$$x \in L \Rightarrow \Pr_{R \in \{0,1\}^{\mathsf{poly}(|x|)}} [M(x,R) = 1] \geq 2/3$$

$$x \notin L \Rightarrow \Pr_{R \in \{0,1\}^{\mathsf{poly}(|x|)}} [M(x,R) = 1] = 0$$

- **co-RP:** languages $L \subseteq \{0,1\}^*$ s.t. $\overline{L} \in RP$

Proof Systems

A proof system looks like this:

Proof Systems

A proof system looks like this:

1. A prover and a verifier agree on the following:
   - The prover must provide proofs in a certain format
   - The verifier can use algorithms from a certain complexity class for verification

Proof Systems

A proof system looks like this:

1. A prover and a verifier agree on the following:
    - The prover must provide proofs in a certain format
    - The verifier can use algorithms from a certain complexity class for verification
2. A statement is given to both prover and verifier (for instance "Graph $G(V, E)$ has a Hamiltonian Cycle")

Proof Systems

A proof system looks like this:

1. A prover and a verifier agree on the following:
   - The prover must provide proofs in a certain format
   - The verifier can use algorithms from a certain complexity class for verification
2. A statement is given to both prover and verifier (for instance "Graph $G(V, E)$ has a Hamiltonian Cycle")
3. A prover writes down a proof of the statement

Proof Systems

A proof system looks like this:

1. A prover and a verifier agree on the following:
   - The prover must provide proofs in a certain format
   - The verifier can use algorithms from a certain complexity class for verification
2. A statement is given to both prover and verifier (for instance "Graph $G(V, E)$ has a Hamiltonian Cycle")
3. A prover writes down a proof of the statement
4. The verifier uses an algorithm of their choice to check the statement and proof, and accepts or rejects accordingly.

Proof Systems

A proof system looks like this:

1. A prover and a verifier agree on the following:
   - The prover must provide proofs in a certain format
   - The verifier can use algorithms from a certain complexity class for verification
2. A statement is given to both prover and verifier (for instance "Graph $G(V, E)$ has a Hamiltonian Cycle")
3. A prover writes down a proof of the statement
4. The verifier uses an algorithm of their choice to check the statement and proof, and accepts or rejects accordingly.
5. NP as a proof system:
   - $L \subseteq \{0, 1\}^n$ is the language, verifier can use any deterministic, poly-time Turing Machine

Proof Systems

A proof system looks like this:

1. A prover and a verifier agree on the following:
   - The prover must provide proofs in a certain format
   - The verifier can use algorithms from a certain complexity class for verification
2. A statement is given to both prover and verifier (for instance "Graph $G(V, E)$ has a Hamiltonian Cycle")
3. A prover writes down a proof of the statement
4. The verifier uses an algorithm of their choice to check the statement and proof, and accepts or rejects accordingly.
5. NP as a proof system:
   - $L \subseteq \{0,1\}^n$ is the language, verifier can use any deterministic, poly-time Turing Machine
   - Given an element x, the prover gives a proof (also known as witness) $w \in \{0,1\}^{\text{poly}(|x|)}$

Proof Systems

A proof system looks like this:

1. A prover and a verifier agree on the following:
   - The prover must provide proofs in a certain format
   - The verifier can use algorithms from a certain complexity class for verification
2. A statement is given to both prover and verifier (for instance "Graph $G(V, E)$ has a Hamiltonian Cycle")
3. A prover writes down a proof of the statement
4. The verifier uses an algorithm of their choice to check the statement and proof, and accepts or rejects accordingly.
5. NP as a proof system:
   - $L \subseteq \{0,1\}^n$ is the language, verifier can use any deterministic, poly-time Turing Machine
   - Given an element x, the prover gives a proof (also known as witness) $w \in \{0,1\}^{\text{poly}(|x|)}$
   - Verifier picks a poly-time Turing Machine V and outputs
     $$\begin{cases} TRUE, & \text{if } V(x,w) = 1 \\ FALSE, & \text{otherwise} \end{cases}$$

Proof Systems - Completeness and Soundness

How good is a proof system?

1. Two parameters (aside from efficiency):
   - **Completeness:** *correct* statements *have a proof* in the system
   - **Soundness:** *false* statements *do not have a proof* in the system

Proof Systems - Completeness and Soundness

How good is a proof system?

1. Two parameters (aside from efficiency):
   - **Completeness:** *correct* statements *have a proof* in the system
   - **Soundness:** *false* statements *do not have a proof* in the system
2. NP as a proof system:
   - $L \subseteq \{0,1\}^n$ is the language, verifier can use any poly-time Turing Machine
   - Given an element x, the prover gives a proof (also known as witness) $w \in \{0,1\}^{\text{poly}(|x|)}$
   - Verifier picks a deterministic, poly-time Turing Machine V and outputs
     $$\begin{cases} TRUE, & \text{if } V(x,w) = 1 \\ FALSE, & \text{otherwise} \end{cases}$$

Proof Systems - Completeness and Soundness

How good is a proof system?

1. Two parameters (aside from efficiency):
   - **Completeness:** *correct* statements *have a proof* in the system
   - **Soundness:** *false* statements *do not have a proof* in the system
2. NP as a proof system:
   - $L \subseteq \{0,1\}^n$ is the language, verifier can use any poly-time Turing Machine
   - Given an element x, the prover gives a proof (also known as witness) $w \in \{0,1\}^{\text{poly}(|x|)}$
   - Verifier picks a deterministic, poly-time Turing Machine V and outputs $\begin{cases} TRUE, & \text{if } V(x,w)=1 \\ FALSE, & \text{otherwise} \end{cases}$
   - **Completeness:** $x \in L \Rightarrow \exists w \in \{0,1\}^{\text{poly}(|x|)}$ such that $V(x,w)=1$

Proof Systems - Completeness and Soundness

How good is a proof system?

1. Two parameters (aside from efficiency):
   - **Completeness:** *correct* statements *have a proof* in the system
   - **Soundness:** *false* statements *do not have a proof* in the system
2. NP as a proof system:
   - $L \subseteq \{0,1\}^n$ is the language, verifier can use any poly-time Turing Machine
   - Given an element x, the prover gives a proof (also known as witness) $w \in \{0,1\}^{\text{poly}(|x|)}$
   - Verifier picks a deterministic, poly-time Turing Machine V and outputs $\begin{cases} TRUE, & \text{if } V(x,w) = 1 \\ FALSE, & \text{otherwise} \end{cases}$
   - **Completeness:** $x \in L \Rightarrow \exists w \in \{0,1\}^{\text{poly}(|x|)}$ such that $V(x,w) = 1$
   - **Soundness:** $x \notin L \Rightarrow \forall w \in \{0,1\}^{\text{poly}(|x|)}$ we have $V(x,w) = 0$

Probabilistic Proof Systems

What if we allow our verifier to run a randomized algorithm?

Probabilistic Proof Systems

What if we allow our verifier to run a randomized algorithm?

Definition (Probabilistic Proof System)

In a probabilistic proof system, the verifier has a randomized algorithm V for which:

1. Given language L (the language of correct statements)

Probabilistic Proof Systems

What if we allow our verifier to run a randomized algorithm?

Definition (Probabilistic Proof System)

In a probabilistic proof system, the verifier has a randomized algorithm V for which:

1. Given language L (the language of correct statements)
2. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$

Probabilistic Proof Systems

What if we allow our verifier to run a randomized algorithm?

Definition (Probabilistic Proof System)

In a probabilistic proof system, the verifier has a randomized algorithm V for which:

1. Given language L (the language of correct statements)
2. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$
3. $x \notin L \Rightarrow$ for any "proof" w, we have $\Pr[V^w(x) = 1] \leq 1/2$

Probabilistic Proof Systems

What if we allow our verifier to run a randomized algorithm?

Definition (Probabilistic Proof System)

In a probabilistic proof system, the verifier has a randomized algorithm V for which:

1. Given language L (the language of correct statements)
2. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$
3. $x \notin L \Rightarrow$ for any "proof" w, we have $\Pr[V^w(x) = 1] \leq 1/2$

Definition (Probabilistic Checkable Proofs (PCPs))

The class of *Probabilistic Checkable Proofs* consists of languages L that have a *randomized poly-time* verifier V such that

Probabilistic Proof Systems

What if we allow our verifier to run a randomized algorithm?

Definition (Probabilistic Proof System)

In a probabilistic proof system, the verifier has a randomized algorithm V for which:

1. Given language L (the language of correct statements)
2. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$
3. $x \notin L \Rightarrow$ for any "proof" w, we have $\Pr[V^w(x) = 1] \leq 1/2$

Definition (Probabilistic Checkable Proofs (PCPs))

The class of *Probabilistic Checkable Proofs* consists of languages L that have a *randomized poly-time* verifier V such that

1. $x \in L \Rightarrow \exists$ proof w such that $\Pr[V^w(x) = 1] = 1$
2. $x \notin L \Rightarrow \forall$ "proof" w, we have $\Pr[V^w(x) = 1] \leq 1/2$

Quantifying Probabilistic Proof Systems

Definition (Probabilistic Checkable Proofs (PCPs))

The class of *Probabilistic Checkable Proofs* (PCP) consists of languages L that have a randomized poly-time verifier V such that

1. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$
2. $x \notin L \Rightarrow$ for any proof w, we have $\Pr[V^w(x) = 1] \leq 1/2$

Quantifying Probabilistic Proof Systems

Definition (Probabilistic Checkable Proofs (PCPs))

The class of *Probabilistic Checkable Proofs* (PCP) consists of languages L that have a randomized poly-time verifier V such that

1. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$
2. $x \notin L \Rightarrow$ for any proof w, we have $\Pr[V^w(x) = 1] \leq 1/2$

- $PCP[r(n), q(n)]$ consists of all languages $L \in PCP$ such that, on inputs x of length n

Quantifying Probabilistic Proof Systems

Definition (Probabilistic Checkable Proofs (PCPs))

The class of *Probabilistic Checkable Proofs* (PCP) consists of languages L that have a randomized poly-time verifier V such that

1. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$
2. $x \notin L \Rightarrow$ for any proof w, we have $\Pr[V^w(x) = 1] \leq 1/2$

- $PCP[r(n), q(n)]$ consists of all languages $L \in PCP$ such that, on inputs x of length n
  1. Uses $O(r(n))$ random bits
  2. Examines $O(q(n))$ bits of a proof w

Note that n *does not* depend on w, only on x.

Quantifying Probabilistic Proof Systems

Definition (Probabilistic Checkable Proofs (PCPs))

The class of *Probabilistic Checkable Proofs* (PCP) consists of languages L that have a randomized poly-time verifier V such that

1. $x \in L \Rightarrow$ there exists proof w such that $\Pr[V^w(x) = 1] = 1$
2. $x \notin L \Rightarrow$ for any proof w, we have $\Pr[V^w(x) = 1] \leq 1/2$

- $PCP[r(n), q(n)]$ consists of all languages $L \in PCP$ such that, on inputs x of length n
  1. Uses $O(r(n))$ random bits
  2. Examines $O(q(n))$ bits of a proof w

Note that n *does not* depend on w, only on x.

Theorem (PCP theorem [AS'98, ALMSS'98])

$$PCP[\log n, 1] = NP$$

PCP and Approximability of Max 3SAT

Definition (Max 3SAT)

- **Input:** a 3CNF formula φ on boolean variables $x_1, \ldots, x_n$ and m clauses
- **Output:** the maximum number of clauses of φ which can be simultaneously satisfied.

Theorem

1. *The PCP theorem implies that there is an $\varepsilon > 0$ such that there is no polynomial time $(1+\varepsilon)$-approximation algorithm for Max 3SAT, unless $P = NP$.*
2. *Moreover, if Max 3SAT is hard to approximate within a factor of $(1+\varepsilon)$, then the PCP theorem holds.*

- In other words, the PCP theorem and the hardness of approximation of Max 3SAT are equivalent.

PCP and Approximability of Max 3SAT

1. Let us assume the PCP theorem holds.
   - Let $L \in PCP[\log n, 1]$ be an NP-complete problem.
   - Let V be the $(O(\log n), q)$ verifier for L, where q is a constant

PCP and Approximability of Max 3SAT

1. Let us assume the PCP theorem holds.
   - Let $L \in PCP[\log n, 1]$ be an NP-complete problem.
   - Let V be the $(O(\log n), q)$ verifier for L, where q is a constant
2. We now describe a reduction from L to Max 3SAT which has a gap.

PCP and Approximability of Max 3SAT

1. Let us assume the PCP theorem holds.
   - Let $L \in PCP[\log n, 1]$ be an NP-complete problem.
   - Let V be the $(O(\log n), q)$ verifier for L, where q is a constant
2. We now describe a reduction from L to Max 3SAT which has a gap.
3. Given an instance x of problem L, we construct 3CNF formula φ_x with m clauses such that, for some ε we have
   - $x \in L \Rightarrow \varphi_x$ is satisfiable
   - $x \notin L \Rightarrow$ no assignment satisfies more than $(1 - \varepsilon) \cdot m$ clauses of φ_x

PCP and Approximability of Max 3SAT

1. Let us assume the PCP theorem holds.
   - Let $L \in PCP[\log n, 1]$ be an NP-complete problem.
   - Let V be the $(O(\log n), q)$ verifier for L, where q is a constant
2. We now describe a reduction from L to Max 3SAT which has a gap.
3. Given an instance x of problem L, we construct 3CNF formula φ_x with m clauses such that, for some ε we have
   - $x \in L \Rightarrow \varphi_x$ is satisfiable
   - $x \notin L \Rightarrow$ no assignment satisfies more than $(1 - \varepsilon) \cdot m$ clauses of φ_x
4. Enumerate all random inputs R for the verifier V.
   - Length of each random string is $O(\log n)$, by definition. So number of such random inputs is poly(n).
   - For each R, V chooses q positions $i_1^R, \ldots, i_q^R$ and a boolean function $f_R : \{0,1\}^q \to \{0,1\}$ and accepts iff $f_R(w_{i_1^R}, \ldots, w_{i_q^R}) = 1$.

PCP and Approximability of Max 3SAT

1. Enumerate all random inputs R for the verifier V.
   - Length of each random string is $O(\log n)$, by definition. So number of such random inputs is poly(n).
   - For each R, V chooses q positions $i_1^R, \ldots, i_q^R$ and a boolean function $f_R : \{0,1\}^q \rightarrow \{0,1\}$ and accepts iff $f_R(w_{i_1^R}, \ldots, w_{i_q^R}) = 1$.

PCP and Approximability of Max 3SAT

1. Enumerate all random inputs R for the verifier V.
   - Length of each random string is $O(\log n)$, by definition. So number of such random inputs is poly(n).
   - For each R, V chooses q positions $i_1^R, \ldots, i_q^R$ and a boolean function $f_R : \{0,1\}^q \to \{0,1\}$ and accepts iff $f_R(w_{i_1^R}, \ldots, w_{i_q^R}) = 1$.
2. Simulate the computation f_R of the verifier for different random inputs R and witnesses w as a Boolean formula.
   - Can be done with a CNF of size 2^q
   - Converting to 3CNF we get a formula of size $q \cdot 2^q$

PCP and Approximability of Max 3SAT

1. Enumerate all random inputs R for the verifier V.
   - Length of each random string is $O(\log n)$, by definition. So number of such random inputs is poly(n).
   - For each R, V chooses q positions $i_1^R, \ldots, i_q^R$ and a boolean function $f_R : \{0,1\}^q \to \{0,1\}$ and accepts iff $f_R(w_{i_1^R}, \ldots, w_{i_q^R}) = 1$.
2. Simulate the computation f_R of the verifier for different random inputs R and witnesses w as a Boolean formula.
   - Can be done with a CNF of size 2^q
   - Converting to 3CNF we get a formula of size $q \cdot 2^q$
3. Let φ_x be the 3CNF we get by putting together all the 3CNFs constructed above

PCP and Approximability of Max 3SAT

1. Enumerate all random inputs R for the verifier V.
   - Length of each random string is $O(\log n)$, by definition. So number of such random inputs is poly(n).
   - For each R, V chooses q positions $i_1^R, \ldots, i_q^R$ and a boolean function $f_R : \{0,1\}^q \to \{0,1\}$ and accepts iff $f_R(w_{i_1^R}, \ldots, w_{i_q^R}) = 1$.
2. Simulate the computation f_R of the verifier for different random inputs R and witnesses w as a Boolean formula.
   - Can be done with a CNF of size 2^q
   - Converting to 3CNF we get a formula of size $q \cdot 2^q$
3. Let φ_x be the 3CNF we get by putting together all the 3CNFs constructed above
4. If $x \in L$ then there is a witness w such that $V(x, w)$ accepts for every random string R. In this case, φ_x is satisfiable!

PCP and Approximability of Max 3SAT

1. Enumerate all random inputs R for the verifier V.
   - Length of each random string is $O(\log n)$, by definition. So number of such random inputs is $\text{poly}(n)$.
   - For each R, V chooses q positions $i_1^R, \ldots, i_q^R$ and a boolean function $f_R : \{0,1\}^q \to \{0,1\}$ and accepts iff $f_R(w_{i_1^R}, \ldots, w_{i_q^R}) = 1$.
2. Simulate the computation f_R of the verifier for different random inputs R and witnesses w as a Boolean formula.
   - Can be done with a CNF of size 2^q
   - Converting to 3CNF we get a formula of size $q \cdot 2^q$
3. Let φ_x be the 3CNF we get by putting together all the 3CNFs constructed above
4. If $x \in L$ then there is a witness w such that $V(x, w)$ accepts for every random string R. In this case, φ_x is satisfiable!
5. If $x \notin L$ then the verifier says NO for half of the random strings R.
   - For each such random string, at least one of its clauses fails
   - Thus at least $\varepsilon = \dfrac{1}{2 \cdot q \cdot 2^q}$ of the clauses of φ_x fails.

Conclusion

- Important to study hardness of approximation for NP-hard problems
- Different hard problems have different approximation parameters
- For hardness of approximation, need more *robust reductions* between combinatorial problems
- Proof systems, in particular *Probabilistic Checkable Proofs*, allows us to get such strong reductions
- Many more applications in computer science and industry!
  - Program Checking (for software engineering)
  - Zero-knowledge proofs in cryptocurrencies
  - many more...

Acknowledgement

- Lecture based largely on:
  - Section's 1-3 of Luca's survey [Trevisan 2004]
  - [Motwani & Raghavan 2007, Chapter 7]
- See Luca's survey `https://arxiv.org/pdf/cs/0409043`

References I

Trevisan, Luca (2004)
Inapproximability of combinatorial optimization problems.
arXiv preprint cs/0409043 (2004).

Motwani, Rajeev and Raghavan, Prabhakar (2007)
Randomized Algorithms

Arora, Sanjeev, and Shmuel Safra (1998)
Probabilistic checking of proofs: A new characterization of NP.
Journal of the ACM (JACM) 45, no. 1 (1998): 70-122.

Arora, Sanjeev, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy (1998)
Proof verification and the hardness of approximation problems.
Journal of the ACM (JACM) 45, no. 3 (1998): 501-555.