Computationally Hard Problems

Randomized Algorithms, Complexity Classes and the Class $\mathcal{NP}$

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

 Randomized Algorithms

Deterministic Algorithms

- Most of the algorithms you have seen in your studies are deterministic.
- A deterministic algorithm A will always produce the same output when the same input is given repeatedly.
- A deterministic algorithm A will always have the same running time when the same input is given repeatedly.
- This is a desired and nice behavior.

Randomized Algorithms

- A randomized algorithm A might produce **different** outputs when the same input is given repeatedly.
- A randomized algorithm A might have different running times when the same input is given repeatedly.
- Sometimes it might not stop at all.
- This behavior is normally undesired and even dangerous.
- Nevertheless randomized algorithms are very useful if the undesired behavior can be somewhat “statistically” controlled.

 Randomized Algorithms

Definition

- A randomized algorithm has access to a *fair coin*.
- This is a device that on demand outputs a 0 or 1.
- This happens **independently according to uniform distribution** on $\{0, 1\}$.
- The probability of seeing a 0 is 0.5 as is that for a 1.
- A coin flip counts one computational step.

Definition

The coin can be used in form of *randomized flow-control statements.*

if $(\text{coin} = 1)$ **then**
  do something
else
  do something else
end if

It cannot be determined beforehand which of the two statements will be executed.

Random Number Generators

As another syntactical tool we introduce a *random number generator*:

This is a subroutine $\mathrm{rand}(a, b)$ which receives two integers a, b, $a < b$ and returns a random number.

The number is drawn **independently according to uniform distribution** from the set $\{a, a+1, \ldots, b-1, b\}$.

Generating a random number counts as one computational step.

However, it is not necessarily “cheap” and “easy” to obtain real random numbers in practice.

Running Time

Consider the following algorithm:

while $(\text{coin} = 0)$ **do**
  do something
end while

How long does it run (how often is "do something" executed)?

Let us determine the possible values for this.

The while-loop is never executed if $\text{coin} = 1$ the first time.

The while-loop is executed exactly once if $\text{coin} = 0$ the first time and $\text{coin} = 1$ the second time.

The while-loop is executed exactly twice if $\text{coin} = 0$ the first and second time and $\text{coin} = 1$ the third time.

Best-Case and Worst-Case Running Time

The *best-case running time* $T_{\mathsf{b}}^{R}(n)$ of randomized algorithm R for input size n is the shortest running time on inputs of size n, assuming a best possible outcome of the random numbers and the input:

$$T_{\mathsf{b}}^{R}(n) := \min\{t \mid \boldsymbol{P}\left[T^{R}(\boldsymbol{X}) = t\right] > 0, \|\boldsymbol{X}\| = n\}$$

Similarly, *worst-case running time* applies to the worst possible:

$$T_{\mathsf{w}}^{R}(n) := \max\{t \mid \boldsymbol{P}\left[T^{R}(\boldsymbol{X}) = t\right] > 0, \|\boldsymbol{X}\| = n\}$$

Both these measures are often not very helpful. In the example:

- $T_{\mathsf{b}}^{R}(n) = 0$.
- $T_{\mathsf{w}}^{R}(n) = \infty$.

Towards an Expected Running Time

while $(\text{coin} = 0)$ **do**
  do something
end while

The statement "do something" can be executed $0, 1, 2, 3, \ldots$ times.

Are all values equally likely? NO!

Intuitively, it is unlikely that the coin shows 0 many times in a row.

Low values are more likely.

Tools from Probability Theory

- A **random variable** returns a value depending on the outcome of a (random) experiment. Example: roll a die $\to$ random variable X denotes number of "points" (from $1, 2, \ldots, 6$)
- The **expectation/expected value** $\boldsymbol{E}[X]$ of a r.v. X is the theoretical average of the possible outcomes:

$$\boldsymbol{E}[X] = \sum_{i=1}^{n} \boldsymbol{P}[X = s_i] \cdot s_i \ .$$

Example: Let X be the fair die. Then the expectation is:

$$\boldsymbol{E}[X] = \sum_{i=1}^{6} \boldsymbol{P}[X = s_i]\, s_i = \sum_{i=1}^{6} \frac{1}{6} i = \frac{1}{6} \sum_{i=1}^{6} i = \frac{1}{6} 21 = 3.5 \ .$$

A Probabilistic Analysis (1/2)

- We want to determine the probability of the algorithm running a certain number of steps. Let X be the random variable "number of times *do something* is executed".
- We shall determine the probability $\boldsymbol{P}[X = n]$ that X assumes value n, for $n = 0, 1, 2, \ldots$.
- The loop is never executed if $\mathrm{coin} = 1$, which happens with probability 0.5.
- Thus $\boldsymbol{P}[X = 0] = 0.5$.
- The loop is executed once if $\mathrm{coin} = 0$ and then $\mathrm{coin} = 1$, each happens with probability 0.5.
- By independence of the two events, both happen with probability $0.5 \cdot 0.5 = 0.25$.
- Thus $\boldsymbol{P}[X = 1] = 0.25$.

A Probabilistic Analysis (2/2)

- The loop is executed n times if $\text{coin} = 0$ the first n tosses and then $\text{coin} = 1$ in the $(n+1)$-st toss.
- Each happens with probability 0.5.
- By independence of the events, all happen with probability $0.5^n \cdot 0.5 = 0.5^{n+1}$.
- Thus $\boldsymbol{P}\left[\, X = n \,\right] = 0.5^{n+1}$.

Expected Running Time

The *expected running time* of a randomized algorithm is computed with respect to the algorithms **internal** randomization. It is the expected value of the (random) running time.

$$\boldsymbol{E}[X] = \sum_{n=0}^{\infty} \boldsymbol{P}[X = n] \cdot n \; .$$

Recall that for **deterministic** algorithms, the *average* running time depends on the probability distribution of the external inputs. The average running time is also an expected value, but here the inputs are random.

Expected Running Time

while $(\text{coin} = 0)$ **do**
do something
end while

In our case, the values are $0, 1, 2, \ldots$ and the probabilities are $0.5, 0.25, 0.125, \ldots$

$$\boldsymbol{E}\,[\,X\,] = \sum_{n=0}^{\infty} n \cdot \boldsymbol{P}\,[\,X = n\,] = \sum_{n=0}^{\infty} n \cdot \left(\frac{1}{2}\right)^{n+1}$$

This will evaluate to 1: either use the series $\sum_{n=0}^{\infty} n \cdot q^n = q/(1-q)^2$ for $0 \leq q < 1$ (Appendix B.3 in lecture notes) or do it more manually using $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 1$.

Expected Running Time, Using Formula

Using the formula $\sum_{n=0}^{\infty} n \cdot q^n = \frac{q}{(1-q)^2}$ with $q = 1/2$ we have:

$$\begin{aligned}\sum_{n=0}^{\infty} n \cdot \left(\frac{1}{2}\right)^{n+1} &= \frac{1}{2}\sum_{n=0}^{\infty} n \cdot \left(\frac{1}{2}\right)^{n} \\ &= \frac{1}{2}\frac{1/2}{(1-1/2)^2} = \frac{1}{2}\frac{1/2}{1/4} = 1\end{aligned}$$

Expected Running Time, Manually

$$\sum_{n=0}^{\infty} n \cdot \left(\tfrac{1}{2}\right)^{n+1}$$

$$\begin{array}{lcccccccccl}
= 0 \cdot \frac{1}{2} + 1 \cdot \left(\frac{1}{2}\right)^2 + 2 \cdot \left(\frac{1}{2}\right)^3 + 3 \cdot \left(\frac{1}{2}\right)^4 + \cdots \\
= & 0 & + & \left(\frac{1}{2}\right)^2 & + & \left(\frac{1}{2}\right)^3 & + & \left(\frac{1}{2}\right)^4 & + \cdots & & \\
 & & & & + & \left(\frac{1}{2}\right)^3 & + & \left(\frac{1}{2}\right)^4 & + \cdots & & \\
 & & & & & & + & \left(\frac{1}{2}\right)^4 & + \cdots & & \\
 & & & & & & & & + \cdots & & \\
= & 0 & + & \frac{1}{4} & + & \frac{1}{8} & + & \frac{1}{16} & + \cdots & = & \frac{1}{2} \\
 & & & & + & \frac{1}{8} & + & \frac{1}{16} & + \cdots & = & \frac{1}{4} \\
 & & & & & & + & \frac{1}{16} & + \cdots & = & \frac{1}{8} \\
 & & & & & & & & + \cdots & = & \frac{1}{16} \\
 & & & & & & & & & & \vdots \\
\hline
 & & & & & & & & & & 1
\end{array}$$

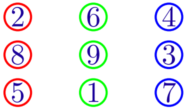
Why Randomized Algorithms

- A better running time than deterministic algorithms.
- Less memory requirements than deterministic algorithms.
- Easier implementation than deterministic algorithms.
- The opportunity to foil an adversary.

Foiling an Adversary

The problem

- We are given $n = 3k$ balls.



- k are red, k are green, k are blue.
- The balls are numbered $1, \dots, n$.
- Numbers are chosen by an adversary: we cannot see the numbers.
- We want to get three balls with equal color (assuming $k \geq 3$).
- We are only allowed questions of this type: "Do balls i, j, and m have the same color?"
- We would like to minimize the number of questions.

Foiling an Adversary

A deterministic approach

- Ask for triples in a certain fixed order.
- For example: $(1, 2, 3)$, $(1, 2, 4)$, . . . , $(1, 2, n)$, $(1, 3, 4)$, ...
- The adversary will number the balls in such a way that "as many as possible" of the first triples are mixed-colored.
- By giving number 1 to a red ball, $2, 3, \ldots k$ to green balls he can force at least

  how many? $(k-1)(n-k) \approx \frac{2n^2}{9}$ questions.

Foiling an Adversary

A randomized approach

- Randomly select three numbers i, j, m between 1 and n.
- Ask whether the balls with those numbers have the same color.
- If so, stop.
- Otherwise randomly select three new numbers and iterate.
- The algorithm might run very long (forever) ...
- ...but that is highly unlikely.
- The adversary has "no good strategy" to number the balls because he does not know the questions.

Foiling an Adversary

A randomized approach, pseudo-code

```
success ← false
while (not success) do
    i ← rand(1, n)
    j ← rand(1, n)
    m ← rand(1, n)
    if i, j, m are pairwise different then
        if i, j, m have the same color then
            success ← true
        end if
    end if
end while
```

Foiling an Adversary

A randomized approach, analysis (1/2)

We use combinatorial arguments.

- We pick the three numbers i, j, m at random.
- i, j, m must be *different*. This is called an *intact* triple.

  After i has been picked (n different choices), there are $n-1$ good choices for j and then $n-2$ good choices for m.

  The total number of choices for the triple (i, j, m) is n^3.

  Hence, the probability of an intact triple is $\frac{n(n-1)(n-2)}{n^3} = 1 - \frac{3}{n} + \frac{2}{n^2}$, approaches 1 for growing n.
- Next: assuming an intact triple, what is the probability of getting a monochromatic triple (three times the same color)?

Foiling an Adversary

A randomized approach, analysis (2/2)

Given pairwise distinct i, j, m

- Ball i is drawn and has some color, say red.
- Then there are $n-1$ balls left, $k-1$ of them red.
- Hence, ball j is red with prob. $\frac{k-1}{n-1}$.
- Same argumentation: ball m is red with prob. $\frac{k-2}{n-2}$.
- Altogether: intact, monochromatic triple with prob. $p := \left(1 - \frac{3}{n} + \frac{2}{n^2}\right) \cdot \frac{(k-1)(k-2)}{(n-1)(n-2)}$, approaches $\frac{1}{9}$ for growing n.
- Running time is t if first $t-1$ failures happen, and then success. This has probability $(1-p)^{t-1}p$.
- Expected running time $\boldsymbol{E}[X] = \sum_{t=1}^{\infty} t\boldsymbol{P}[X = t] = \sum_{t=1}^{\infty} t \cdot (1-p)^{t-1} \cdot p$.
- Result approaches 9 from above for growing n, e. g., $\boldsymbol{E}[X] = 9.092\ldots$ for $n = 300$.

Easy Implementation

MaxCut problem

Problem [MaxCut]

Input: An undirected graph $G = (V, E)$ and a constant $k \in \mathbb{N}$

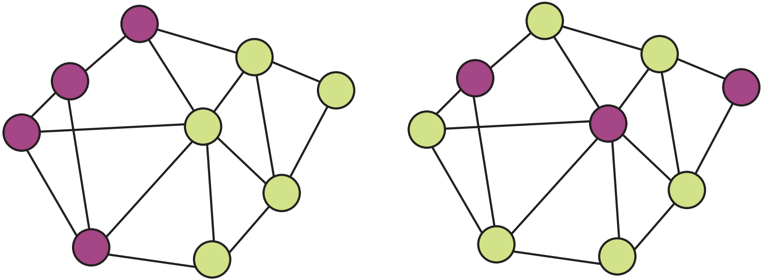
Output: YES if there is a partition of V into two sets V_1, V_2 such that there are at least k edges between V_1 and V_2 and NO otherwise.

Output for the optimizing version: A partition of V into two sets V_1, V_2 such that there is a maximum number of edges between V_1 and V_2

This problem is “hard” to solve.

Easy Implementation

MaxCut problem, example



A graph with a cut of size 5 (left) and one of size 11 (right). The sets V_1 and V_2 are shown in different colors.

Easy Implementation

A randomized solution

Randomly partition the nodes.

for $i = 1, 2, \ldots, |V|$ **do**
  if $(\text{coin} = 0)$ **then**
    put v_i into V_1
  else
    put v_i into V_2
  end if
end for
(find edges between V_1 and V_2)

The running time is $O(|V|)$ plus $O(|E|)$ for finding the cut.

Easy Implementation

Analysis

Let X be the random variable denoting the number of edges in the cut. Edge $\{a, b\} \in E$ is in the cut iff

$$[(a \in V_1) \wedge (b \in V_2)] \vee [(b \in V_1) \wedge (a \in V_2)]$$

$$\boldsymbol{P}[\, a \in V_1 \,] = 0.5, \quad \boldsymbol{P}[\, a \in V_2 \,] = 0.5, \quad \boldsymbol{P}[\, b \in V_1 \,] = 0.5, \quad \boldsymbol{P}[\, b \in V_2 \,] = 0.5$$

$$\boldsymbol{P}[\, \{a, b\} \text{ is in cut} \,] = 0.5 \cdot 0.5 + 0.5 \cdot 0.5 = 0.5$$

Hence “every second” edge is in the cut

$$\boldsymbol{E}[\, X \,] = \sum_{\{a,b\} \in E} 0.5 = 0.5 \cdot |E|$$

(Lecture Notes, App. A2, “Size of Random Selection”)

Easy Implementation

How to use

- Given a graph $G = (V, E)$.
- Run the randomized algorithm, an let C be the resulting cut, i.e., the set of edges between V_1 and V_2
- If $|C| \geq (1/2)\,|E|$, be happy and stop.
- If $|C| < (1/2)\,|E|$, run the algorithm again.
- Reason: Because the **expected** size of a cut is $(1/2)\,|E|$ **some** executions of the algorithm have to yield cuts of that size or larger (otherwise the expected size would be smaller).
- We repeat until we are lucky.
- As the running time is fast, we can do this many times.

However, we implicitly exploit that we have a decent probability of finding a large cut.

Stopping Randomized Algorithms

A randomized algorithm A can run very long.

We create A' which stops after $f(n)$ steps (n the input size and f is some function) as follows:

- Initialize a counter c with 0 and compute $f(n)$
- After every instruction of the original program do
  - Increment the counter c by one.
  - Check if c exceeds the running time bound ($c > f(n)$).
  - If this is the case then A' is terminated. If an output is expected, A' will deliver a default value DON'T KNOW.
  - If A' stops earlier, it made the same computations as A and returns the output which A would have returned.

Time-bounded Algorithms

Definition

A randomized algorithm whose running time is bounded by a function $f : \mathbb{N} \mapsto \mathbb{N}$ is called f-*bounded*.
That is, the algorithm stops after at most $f(\|\boldsymbol{X}\|)$ steps on input $\boldsymbol{X}$. The output might be the default output (DON'T KNOW).

Fact

Let $f : \mathbb{N} \mapsto \mathbb{N}$ be a function. On input $\boldsymbol{X}$ an f-bounded randomized algorithm makes at most $f(\|\boldsymbol{X}\|)$ calls to the random number generator.

We shall use the fact to "outsource" randomization.

Outsourcing the Randomization

- Deterministic algorithms are "easier" to analyze than randomized ones.
- We know that an f-bounded randomized algorithm will at most use $f(\|\boldsymbol{X}\|)$ random numbers on a given input $\boldsymbol{X}$.
- We compute these numbers beforehand and put them in a string $R = r_1 r_2 r_3 \ldots r_{f(\|\boldsymbol{X}\|)}$.
- The algorithm receives as input the "real" input $\boldsymbol{X}$ and R.
- The algorithm is changed as follows: Instead of making a call to the random number generator, it takes the next number r_i from R.
- We write $A(\boldsymbol{X}, R)$.

Outsourcing the Randomization

In pseudocode of algorithms, we shall still used the old notation

$$v \leftarrow \mathrm{rand}(a, b)$$

instead of

$\mathrm{next} \leftarrow 1$ (* initialize counter *)

$\vdots$

$v \leftarrow R[\mathrm{next}]$ (* call to random generator *)

$\mathrm{next} \leftarrow \mathrm{next} + 1$

$\vdots$

Outsourcing the Randomization

- The resulting algorithm $A(\boldsymbol{X}, R)$ is **deterministic**.
- It is randomized by varying the "helper information" R.
- That is, $A(\boldsymbol{X}, R)$ and $A(\boldsymbol{X}, R')$ might give different results or have different running times if $R \neq R'$.
- If the algorithm uses random number generator $\mathrm{rand}(a, b)$ then the string $R = r_1 r_2 r_3 \ldots r_{f(\|\boldsymbol{X}\|)}$ is generated according to uniform distribution on $\{a, \ldots, b\}^{f(\|\boldsymbol{X}\|)}$.
- Depending on the problem under consideration, the additional information R might "somewhat" help to solve the problem.

Complexity Classes

- A *complexity class* is a collection of problems which are "equally difficult" to solve.
- "Difficulty" can be measured as use of resources.
- "Difficulty" can also be measured by how much additional information helps to solve the problem.
- Here, we measure how much (how often) randomization helps.

The Class $\mathcal{P}$

Definition

A yes-no–problem is in $\mathcal{P}$ if there is a polynomial p and a **deterministic** p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:
True answer for $\boldsymbol{X}$ is YES then $A(\boldsymbol{X}) = \mathrm{YES}$
True answer for $\boldsymbol{X}$ is NO then $A(\boldsymbol{X}) = \mathrm{NO}$

These are the "good old" efficiently deterministically solvable problems.

The Class $\mathcal{NP}$

Definition

A yes-no–problem is in $\mathcal{NP}$ ("nondeterministic polynomial") if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:
True answer for $\boldsymbol{X}$ is YES then $\exists R, \|R\| \leq p(\|\boldsymbol{X}\|)\ :\ A(\boldsymbol{X}, R) = \text{YES}$
True answer for $\boldsymbol{X}$ is NO then $\forall R\ :\ A(\boldsymbol{X}, R) = \text{NO}$
Here R is a sequence of random numbers of the type required by the algorithm.

An algorithm with these properties is called an $\mathcal{NP}$-*algorithm*.

The Class $\mathcal{NP}$

Alternative definition

Definition

A yes-no–problem is in $\mathcal{NP}$ if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:
True answer for $\boldsymbol{X}$ is YES then $\boldsymbol{P}_R\,[\,A(\boldsymbol{X}, R) = \mathrm{YES}\,] > 0$
True answer for $\boldsymbol{X}$ is NO then $\boldsymbol{P}_R\,[\,A(\boldsymbol{X}, R) = \mathrm{NO}\,] = 1$
where $\boldsymbol{P}_R\,[\,Z\,]$ denotes the probability of event Z over uniform distribution of R, $\|R\| \leq p(\|\boldsymbol{X}\|)$.

The Class $\mathcal{RP}$

Definition

A yes-no–problem is in $\mathcal{RP}$ (*random polynomial time*) if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:
True answer for $\boldsymbol{X}$ is YES then $\boldsymbol{P}_R\,[\,A(\boldsymbol{X}, R) = \mathrm{YES}\,] \geq \frac{1}{2}$
True answer for $\boldsymbol{X}$ is NO then $\boldsymbol{P}_R\,[\,A(\boldsymbol{X}, R) = \mathrm{NO}\,] = 1$
An algorithm with these properties is called an $\mathcal{RP}$-*algorithm*.

$\mathcal{RP}$-algorithms are also called *Monte Carlo* algorithms. They have one-sided error. In contrast to $\mathcal{NP}$-algorithms, there is a good chance of getting the correct result for YES-inputs.

The Class $\mathcal{BPP}$

Definition

A yes-no–problem is in $\mathcal{BPP}$ (*bounded error probabilistic polynomial*) if there is a polynomial p and an $\varepsilon > 0$ (independent of $\|\boldsymbol{X}\|$) and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

True answer for $\boldsymbol{X}$ is YES : $\boldsymbol{P}_R\,[\,A(\boldsymbol{X}, R) = \mathrm{YES}\,] \geq \frac{1}{2} + \varepsilon$

True answer for $\boldsymbol{X}$ is NO : $\boldsymbol{P}_R\,[\,A(\boldsymbol{X}, R) = \mathrm{NO}\,] \geq \frac{1}{2} + \varepsilon$

An algorithm with these properties is called a $\mathcal{BPP}$-*algorithm*.

It has two-sided error.

ε is a positive constant. Think of $\frac{1}{2} + \epsilon = \frac{2}{3}$.

The Class $\mathcal{ZPP}$

Definition

A yes-no–problem is in $\mathcal{ZPP}$ (*zero error probabilistic polynomial*) if there is a polynomial p and a randomized p-bounded algorithm A such that for every input $\boldsymbol{X}$ the following holds:

True answer for $\boldsymbol{X}$ is YES : $\boldsymbol{P}_R\,[\,A(\boldsymbol{X},R) = \mathrm{YES}\,] \geq \frac{1}{2}$

True answer for $\boldsymbol{X}$ is YES : $\boldsymbol{P}_R\,[\,A(\boldsymbol{X},R) = \mathrm{NO}\,] = 0$

True answer for $\boldsymbol{X}$ is NO : $\boldsymbol{P}_R\,[\,A(\boldsymbol{X},R) = \mathrm{NO}\,] \geq \frac{1}{2}$

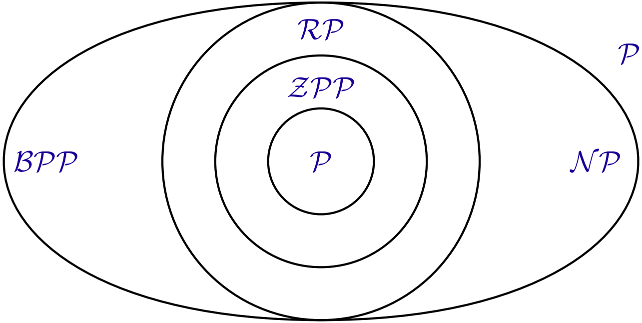
True answer for $\boldsymbol{X}$ is NO : $\boldsymbol{P}_R\,[\,A(\boldsymbol{X},R) = \mathrm{YES}\,] = 0$

An algorithm with these properties is called a $\mathcal{ZPP}$-*algorithm*. It never answers wrong, but might refuse to answer (DON'T KNOW).

$\mathcal{ZPP}$-algorithms are also called *Las Vegas* algorithms.

Relation Between the Classes

$\mathcal{RP} \subseteq \mathcal{BPP}$



$\mathcal{P} \subseteq \mathcal{ZPP} \subseteq \mathcal{RP} \subseteq \mathcal{NP}$

Proof: Almost all inclusions follow immediately from the definition (for $\mathcal{ZPP} \subseteq \mathcal{RP}$ replace DON'T KNOW answer by NO).

Only the inclusion $\mathcal{RP} \subseteq \mathcal{BPP}$ requires a non-trivial additional argument discussed in the exercises.