CSCE-658 Randomized Algorithms

Lecture #4, February 2, 2016

Lecturer: Professor Jianer Chen

4 Does randomness help solving NP-hard problems?

We have seen that randomness can help improving computational efficiency. A natural question is "can randomness help solving NP-complete problems?"

The answer to this question depends on how you interpret the word "help." If you meant whether randomness helps developing faster algorithms for NP-complete problems, then the answer is "sometimes yes." On the other hand, if you meant whether randomized algorithms can solve NP-complete problems in polynomial time, then the answer is "probably not."

In this section, we study the answers to the question for both versions.

4.1 Faster algorithms for NP-complete problems via randomness

In previous sections, we have seen how randomized algorithms solve the Min-Cut problem, where we look for a partition of the vertices in a graph into two parts that minimizes the number of edges that cross the two parts. We can formulate a maximization version of this problem, i.e., we try to find a partition of the graph vertices into two parts such that the number of edges that cross the two parts is maximized. In the literature, this problem is named the Max-Cut problem, which is one of the most famous NP-complete problems that plays an important role in the study of computational optimization and complexity theory, and has received extensively study.

The Max-Cut problem can be re-formulated as a set problem, which can be called the 2-Set Splitting problem and is given as follows: Given a base set U, and a collection $\mathcal{C}$ of 2-subsets of U (a 2-subset of U is a subset of two elements in U), construct a partition of the base set U that splits the maximum number of 2-subsets in $\mathcal{C}$. Here we say that a subset S is *split* by the partition if the subset S contains elements in both parts of the partition. Note that the 2-Set Splitting problem is identical to the Max-Cut problem if we take the base set U as the graph vertex set, and take each 2-subset $\{u, w\}$ in the collection $\mathcal{C}$ as an edge $[u, w]$ in the graph.

A more general version of this set problem, where we allow the collection $\mathcal{C}$ to contain *arbitrary* subsets of the base set U, is called the Set Splitting problem, which is also a well-known NP-complete problem. In this section, we study how randomized algorithms solve the *parameterized version* of this problem. Formally, the problem we are focused on is defined as follows:

> Set Splitting. Given a base set U, a collection $\mathcal{C}$ of subsets of U, and a parameter k, either construct a partition of U that splits at least k subsets in $\mathcal{C}$, or report that no such a partition exists.

Note that the Set Splitting problem is identical to the Max-Cut problem on *hypergraphs* where we treat each element in U as a vertex and each subset in $\mathcal{C}$ as a hyperedge.

Since the Set Splitting problem is NP-hard, we do not expect a polynomial-time algorithm for the problem. On the other hand, it has become increasingly interesting in the current algorithmic research to study *parameterized algorithms* for NP-hard problems, where we measure the complexity of the algorithms in terms of both the input length n and a parameter k. In particular, for the Set Splitting problem as defined above, we look for algorithms solving the problem whose running time is bounded by $O(f(k)n^c)$, where $f(k)$ is an arbitrary function of the parameter k that is independent of the input size n, and c is a small constant. If we believe that P $\neq$ NP, then we cannot expect that $f(k)$ be a polynomial of k. However, in many applications, the value of the parameter k is small, which will make this kind of algorithms useful, even $f(k)$ is not a polynomial of k. In particular, there has been extensive research along this direction, focusing on such parameterized algorithms for which the function $f(k)$ in the complexity is as small as possible. For the Set Splitting problem, previous research has generated a sequence of improved parameterized algorithms, with running times $O(72^k n^{O(1)})$, $O(8^k n^{O(1)})$, and $O(2.68^k n^{O(1)})$, respectively (see the introductory section and the references in [4] for a quick survey on this line of research), These algorithms are deterministic algorithms.

Consider the following randomized algorithm for the Set Splitting problem:

Algorithm 7 RandomPartition
Input: A base set U, a collection $\mathcal{C}$ of subsets of U, and an integer k
Output: Either return a partition of U that splits at least k subsets in $\mathcal{C}$,
    or report no such a partition exists.

```
1. randomly partition $U$ into $(L, R)$;
2. **if** $(L, R)$ splits at least $k$ subsets in $\mathcal{C}$
   **then** output all split subsets in $\mathcal{C}$;
   **else** report no such a partition exists.
```

This is a trivial linear-time algorithm, where "randomly partition U into (L, R)" means that each element in U is placed in L or R with an equal probability $1/2$. Now what is the probability that this algorithm is correct?

If there is no partition of U that can splits k or more subsets in $\mathcal{C}$, then certainly the algorithm RandomPartition will correctly report this fact. Thus, we only need to consider the case in which there is a partition that splits at least k subsets in $\mathcal{C}$. Pick any such partition (L, R), and fix k subsets $S_1, \ldots, S_k$ that are split by (L, R). Moreover, for each subset S_i, fix two elements a_i and b_i such that $a_i \in L$ and $b_i \in R$. Note that for certain i and j, we may have $a_i = a_j$ or $b_i = b_j$, but the sets $\{a_1, \ldots, a_k\}$ and $\{b_1, \ldots, b_k\}$ are disjoint.

Since each element in U has a probability $1/2$ to be placed in L or in R, the probability that all elements in $\{a_1, \ldots, a_k\}$ are placed in L and all elements in $\{b_1, \ldots, b_k\}$ are placed in R is $1/2^h \geq 1/2^{2k} = 1/4^k$, where $h \leq 2k$ is the total number of different elements in $\{a_1, \ldots, a_k\} \cup \{b_1, \ldots, b_k\}$. Such a partition certainly splits at least k subsets in $\mathcal{C}$. In conclusion, if partitions of U that split at least k subsets of $\mathcal{C}$ exist, then the algorithm RandomPartition returns such a partition with a probability at least $1/4^k$.

In fact, the algorithm RandomPartition does better than this, which can be derived by the following more precise analysis. For this, we construct a bipartite graph H, which has the h different elements in $\{a_1, \ldots, a_k\} \cup \{b_1, \ldots, b_k\}$ as its vertices, and k edges $[a_i, b_i]$ for $i = 1, \ldots, k$ (thus, each edge represents a subset that is split by the partition (L, R)). Note that since for certain i and j we may have $a_i = a_j$ or $b_i = b_j$, the graph H may have fewer than $2k$ vertices. Moreover, certain pair of vertices may be connected by multiple edges (this happens if for some i and j we have both $a_i = a_j$ and $b_i = b_j$).

The graph H may not be connected. Let $H_1, \ldots, H_r$ be the connected components of H. For each H_i, let h_i and k_i be the number of vertices and the number of edges in H_i, respectively. Since H_i is connected, $h_i \leq k_i + 1$. A vertex in H_i that comes from the set $\{a_1, \ldots, a_k\}$ (resp. from the set $\{b_1, \ldots, b_k\}$) will be called an *a-vertex* (resp. a *b-vertex*). Similar to the above analysis, the probability that a random partition (L, R) places all a-vertices of H_i in L and all b-vertices of H_i in R is equal to $1/2^{h_i}$. Moreover, note that if a partition (L', R') places all a-vertices of H_i in R' and all b-vertices of H_i in L', then the k_i subsets corresponding to the k_i edges of H_i also get split. Since the event that all a-vertices of H_i are placed in L and all b-vertices of H_i are placed in R and the event that all a-vertices of H_i are placed in R and all b-vertices of H_i are placed in L are disjoint, we conclude that a random partition splits the k_i subsets corresponding to the k_i edges of H_i with a probability at least

$$\frac{1}{2^{h_i}} + \frac{1}{2^{h_i}} = \frac{2}{2^{h_i}} = \frac{1}{2^{h_i - 1}} \geq \frac{1}{2^{k_i}},$$

where we have used the inequality $h_i \leq k_i + 1$. Now since for two connected components H_i and H_j of H, the vertex set of H_i and the vertex set of H_j are disjoint, the event that a random partition splits the k_i subsets that correspond to the k_i edges of H_i and the event that it splits the k_j subsets that correspond to the k_j edges of H_j are independent for all $i \neq j$. Therefore, the probability that a random partition splits the k_i subsets that correspond to the k_i edges of H_i for all $i = 1, \ldots, r$, i.e., the probability that a random partition splits all the k subsets $S_1, \ldots, S_k$ in $\mathcal{C}$, is at least

$$\frac{1}{2^{k_1}} \cdot \frac{1}{2^{k_2}} \cdots \frac{1}{2^{k_r}} = \frac{1}{2^{k_1+k_2+\cdots+k_r}} = \frac{1}{2^k}.$$

This proves the following lemma.

Lemma 4.1 *If there is a partition of the base set U that splits at least k subsets in $\mathcal{C}$, then the algorithm* RandomPartiton *returns such a partition with a probability at least $1/2^k$. If there is no such a partition, then the algorithm correctly reports this fact (with a probability 1).*

Now we can simply use the standard trick to amplify the success probability.

Algorithm 8 SplittingSet
Input: A base set U, a collection $\mathcal{C}$ of subsets of U, and an integer k
Output: Either return a partition of U that splits at least k subsets in $\mathcal{C}$,
or report no such a partition exists.

```
1.   repeat $2^k t$ times
1.1      call the algorithm RandomPartition on $U$, $\mathcal{C}$, and $k$;
1.2      if step 1.1 returns a partition $(L, R)$ that splits at least $k$ subsets in $\mathcal{C}$
1.3      then return the partition $(L, R)$; stop.
2.   return("no such a partition").
```

Theorem 4.2 *If there are partitions of the set U that splits at least k subsets in $\mathcal{C}$, then the algorithm* SplittingSet *returns such a partition with a probability at least $1-1/e^t$. If there is no such a partition, then the algorithm correctly reports this fact (with a probability 1). The algorithm* SplittingSet *runs in time $O(2^k m)$, where m is the size of the input instance.*

proof. The algorithm SplittingSet fails in finding an existing partition that splits at least k subsets in $\mathcal{C}$ only if all calls to RandomPartition fail in finding such a partition. By Lemma 4.1, RandomPartition fails in finding the partition with a probability bounded by $1 - 1/2^k$. Thus, the probability that all $2^k t$ calls in step 1.1 fail is bounded by

$$\left(1 - \frac{1}{2^k}\right)^{2^k t} \leq \frac{1}{e^t}.$$

where we have used Lemma 1.6. This shows that the algorithm SplittingSet has a success probability at least $1 - 1/e^t$. Moreover, since RandomPartition runs in time $O(m)$, the running time of SplittingSet is bounded by $O(2^k m)$. This proves the theorem. □

Note that the algorithm SplittingSet is faster than all those deterministic algorithms we mentioned at the beginning of this subsection.

The algorithm SplittingSet actually does more than just finding a partition of the base set U that splits k subsets in $\mathcal{C}$. In fact, if we look for a partition that splits k subsets in $\mathcal{C}$ where the subsets are required to satisfy *any* specified properties, then, by the analysis given above, our algorithm will find such a partition with a high probability. For example, the algorithm can be used to solve the weighted version of the Set Splitting problem. We leave the detailed examination of the algorithm for this version to the reader.

4.2 Randomized algorithms would not overcome NP-hardness

Consider the question ''can randomized algorithms help solving NP-complete problems in polynomial time." Let Q be an NP-complete problem. Recall that by definition, Q is a decision problem whose instances require a yes/no answer. Suppose that Q is solved by a randomized algorithm A. First we need to clarify what do we mean by "a randomized algorithm solving a decision problem." In fact, there are many subtle details that we must address when we study the complexity of randomized algorithms and complexity classes defined based on randomized algorithms.

First, how do we implement a random step in an algorithm? In previous sections, we simply assume that we can pick an object "randomly" in a pre-specified pool of certain size, or more often, that we can generate a random integer in a specified range. However, on low-level machine operations, our computers can only handle binary bits. Thus, we will assume that our algorithms can only randomly generate a random bit from $\{0, 1\}$ with an equal probability (i.e., $1/2$). Note that picking a random integer from an interval of size m can be implemented by generating $\lceil \log m \rceil$ binary random bits. This implementation is not entirely satisfactory. For example, how do you randomly generate "one out of three" using binary random bits? Nevertheless, we will take this model, and refer the readers to reference [7] for deeper studies towards the model of randomized computation.

Under this assumption, the execution of a randomized algorithm A on an input x can be modeled by a binary tree $T_A(x)$ in which the root represents the beginning of the algorithm, each internal node v with two children corresponds to generating a random binary bit $0/1$, with an equal probability going to the two children, and each leaf, labeled yes/no, corresponds to the outcome of a particular execution of the randomized algorithm A on the input x. With different outcomes of the generated random bits, the corresponding executions of the randomized algorithm A on x are different. Each execution of the randomized algorithm A on the input x corresponds to a path P from the root to a leaf in the tree $T_A(x)$, which at each internal node v randomly generated a binary bit whose outcome $(0/1)$ instructed the algorithm A to move to the child of v that is on the path P. Thus, the depth of the tree $T_A(x)$ corresponds to the maximum number of random bits that are generated by the algorithm A on the input x, which is bounded by the running time of the algorithm A on x (in the worst case). Note that the tree $T_A(x)$ only models the random steps of the algorithm A, and ignores all other details in the computation. There can be fairly non-trivial (deterministic) computations between two random-bit generating steps (which corresponds to the edge between an internal node and one of its children).

If we let the root of the tree $T_A(x)$ be at level 0, then the probability of reaching a node at level h in the tree $T_A(x)$ is $1/2^h$. The probability that the algorithm A on the input x returns 'yes' (resp. 'no') is equal to the sum of all the probabilities of the leaves that are labeled 'yes' (resp. 'no').

For different instances x_1 and x_2, the corresponding trees $T_A(x_1)$ and $T_A(x_2)$ may differ: in particular, the algorithm A on input x_1 may terminates at a node v with a definite decision (thus, v is a leaf in $T_A(x_1)$), while on input x_2, the algorithm A at node v may need to continue its process and generating further random bits (thus v is an internal node in $T_A(x_2)$. To make the tree $T_A(x)$ "oblivious" to all inputs x of length n, we modify the algorithm A as follows:

(1) pick any upper bound $p(n)$ for the number of random bits generated by the algorithm A on any input of length n (e.g., $p(n)$ is an upper bound of the running time of A on inputs of length n);

(2) during the execution of the algorithm A, record the number of generated random bits;

(3) if the algorithm A decides to terminate with a decision D (= yes/no) before generating $p(n)$ random bits, modify the algorithm so that it continues by repeatedly generating random bits until it generates exactly $p(n)$ random bits, then the algorithm terminates with the same decision D.

The above modification of the algorithm A on input x corresponds to replacing each leaf v at level h with $h < p(n)$ in the tree $T_A(x)$ with a complete binary tree rooted at v that has depth $p(n) - h$ and has all its leaves labeled by the decision that is the label of the node v in the original tree $T_A(x)$. In particular, the binary tree for the modified algorithm is a complete binary tree of depth $p(n)$, which is independent of the content of the input x (as long as the length of x is n). It is easy to verify that the modified algorithm is "equivalent" to the original algorithm, in the sense that on the same input x, they have the same probability to return 'yes'. Moreover, the modified algorithm does change the asymptotic bound of the running time.

Finally, for a decision problem Q, we assume a fixed encoding for the instances of Q over an alphabet Σ of c symbols, where c is a fixed constant (at the machine level encoding, we have $\Sigma = \{0, 1\}$ and $c = 2$). In particular, the total number of instances of length n under the encoding is bounded by c^n.

We will assume the above formulations in the following discussion. In particular, for a randomized algorithm A that solves a decision problem Q in time $t(n)$, the corresponding binary tree T_A for instances of length n is a complete binary tree of depth bounded by $t(n)$, where the instances are encoded by a fixed encoding scheme such that the total number of instances of length n is bounded by c^n for a constant $c \geq 2$.

Now we can define the following complexity classes based on randomized algorithms.

Definition 4.1 (One-Side Error Class RP) A decision problem Q is in the class *RP* if there is a polynomial-time randomized algorithm A such that (1) for each yes-instance x of Q, the algorithm A returns a 'yes' with a probability at least 1/2; and (2) for each no-instance x of Q, the algorithm A returns a 'no' with probability 1. Such an algorithm A will be called an *RP-algorithm*.

Consider the MIN-CUT problem in Section 1. We can formulate a decision version of the problem:

MIN-CUT(D). Given graph G and integer k, is the size of a min-cut of G bounded by k?

We make simple modifications on Algorithm 3 in Section 1, as follows: (1) let $t = 1$; (2) if any call to CONTRACTION in step 1 returns a cut of size bounded by k then return 'yes'; (3) step 2 of the algorithm is changed to "return 'no'." Now for a yes-instance (G, k) of MIN-CUT(D) (i.e., the size of a min-cut of G is bounded by k), Theorem 1.3 in Section 1 says that the algorithm will return a 'yes' with a probability at least $1 - 1/e > 1/2$. On the other hand, for a no-instance (G, k), the algorithm will always return a 'no' in step 2 (thus returns a 'no' with probability 1). This modified algorithm is thus an RP-algorithm and shows that the problem MIN-CUT(D) is in the class RP.

A more general randomized model is defined as follows.

Definition 4.2 (Two-Side Error Class BPP) A decision problem Q is in the class *BPP* if there is a plynomial-time randomized algorithm A such that (1) for each yes-instance x of Q, the algorithm A returns a 'yes' with a probability at least 3/4; and (2) for each no-instance x of Q, the algorithm A returns a 'no' with probability at least 3/4. Such an algorithm A will be called a *BPP-algorithm*.

Note that if a problem Q is in RP, then it is in BPP. This can be seen as follows: let A be an RP-algorithm for Q that on a yes-instance returns a 'yes' with a probability at least 1/2, and on a no-instance, returns a 'no' with probability 1. Now let A' be the randomized algorithm that runs A twice, and returns a 'yes' if and only if any of the two executions of A returns a 'yes.' Note that for a no-instance, A' still returns a 'no' with probability 1 (thus $\geq 3/4$). On the other hand, for a yes-instance, since the algorithm A returns a 'no' with a probability $\leq 1/2$, and the two executions of the algorithm A in A' are independent, the probability that A' returns a 'no' (i.e., the probability that both executions of A returns 'no') is bounded by $(1/2)^2 = 1/4$. Thus, on a yes-instance, the algorithm A' returns a 'yes' with a probability at least 3/4. The algorithm A' for Q shows that the problem Q is in BPP. Thus, RP $\subseteq$ BPP.

Remark 1. In the definition of the class RP, for a yes-instance, the condition that the RP-algorithm A returns a 'yes' with a probability $\geq 1/2$ can be replaced by the condition that the algorithm A returns a 'yes' with a probability $\geq c$ for any constant $c > 0$. In fact, if $c < 1/2$, then by repeating the algorithm A $\lceil (-1)/\log(1-c) \rceil$ times (note that $\log(1-c) < 0$), we will get an RP-algorithm that on a yes-instance returns a 'yes' with a probability $\geq 1/2$.

Remark 2. In the definition of the class BPP, the condition that the success probability of the BPP-algorithm A is at least 3/4 can be replaced by the condition that the success probability is at least $1/2 + \epsilon$ for a fixed constant $\epsilon > 0$. This can be shown via Chernoff bound, which will be discussed later.

Since BPP-algorithms are more powerful than RP-algorithms, we will study the consequence that a BPP-algorithm solves an NP-complete problem. We first note that the success probability 3/4 in the definition of a BPP-algorithm can be replaced with a constant that is strictly larger than 3/4. To see this, let A be a BPP-algorithm that solves a decision problem Q. Let A' be the randomized algorithm that runs A three times and takes the majority outcomes as its output (i.e., A' returns 'yes' if and only if at least two of the three executions of A return 'yes'). Since the algorithm A has a failure probability $\leq 1/4$, the failure probability of the algorithm A' is equal to

$$\begin{aligned} & \Pr[A \text{ fails twice and succeeds once}] + \Pr[A \text{ fails all three times}] \\ = \; & 3(\Pr[A \text{ fails}])^2 \Pr[A \text{ succeeds}] + (\Pr[A \text{ fails}])^3 \\ \leq \; & 3 \cdot (1/4)^2 + (1/4)^3 \\ = \; & 13/64 \end{aligned}$$

The factor 3 in the second line is because the successful execution of A can appear in three different places during the execution sequence. Moreover, we have used the fact $\Pr[A \text{ succeeds}] \leq 1$. Thus, the success probability of the algorithm A' is at least $1 - 13/64 = 51/64 > 3/4$.

Now we have the following theorem that is important when we want to amplify the success probability of a BPP-algorithm.

Theorem 4.3 *Let Q be a problem in BPP. Then for any polynomial $q(n)$, there is a BPP-algorithm that solves the problem Q with a probability at least $1 - 1/2^{q(n)}$ on instances of length n.*

PROOF. By the assumption, there is a BPP-algorithm A_1 that runs in time $O(p(n))$ for a polynomial p and solves the problem Q. By the remark above, we can assume that the success probability of the algorithm A_1 is at least 51/64. Thus its failure probability is bounded by 13/64.

We develop a new randomized algorithm A_{2t+1} for Q that on an instance x of Q, runs the algorithm A_1 $2t+1$ times and takes the majority outcomes as its solution.

For any integer $i \leq t$, let E_i be the event that among the $2t+1$ executions of the algorithm A_1, in exactly i times the algorithm A_1 succeeds. Then

$$\begin{aligned} \Pr[E_i] &= \Pr[A_1 \text{ succeeds exactly } i \text{ times}] \\ &= \binom{2t+1}{i} (\Pr[A_1 \text{ succeeds}])^i (\Pr[A_1 \text{ fails}])^{2t+1-i} \\ &\leq \binom{2t+1}{i} (13/64)^{2t+1-i} \\ &\leq \binom{2t+1}{i} (13/64)^{t+1}. \end{aligned}$$

Note that in the last inequality, we have used fact $t \geq i$.

Since the algorithm A_{2t+1} fails only if there are fewer than $t+1$ successes among the $2t+1$ executions of the algorithm A_1, we derive that the probability that the algorithm A_{2t+1} fails is bounded by

$$\begin{aligned} \Pr\left[\bigcup_{i=0}^{t} E_i\right] &= \sum_{i=0}^{t} \Pr[E_i] \leq \sum_{i=0}^{t} \binom{2t+1}{i} (13/64)^{t+1} = (13/64)^{t+1} \sum_{i=0}^{t} \binom{2t+1}{i} \\ &\leq (13/64)^{t+1} 2^{2t+1}/2 = (13/64)(13/16)^t < (13/16)^t. \end{aligned} \tag{10}$$

To bound the failure probability by $1/2^{q(n)}$, i.e., to make $(13/16)^t \leq 1/2^{q(n)}$, let $t_0 = \lceil q(n)/\log(16/13) \rceil$, and run the algorithm A_{2t_0+1}, whose error probability, by (10), is bounded by $(13/16)^{t_0} \leq 1/2^{q(n)}$. Since the algorithm A_1 runs in time $O(p(n))$ for a polynomial p of n on instances of length n, the algorithm A_{2t_0+1} runs in time $O(q(n)p(n))$, which is still a polynomial of n. Thus, A_{2t_0+1} is a BPP-algorithm for the problem Q with a success probability at least $1 - 1/2^{q(n)}$. $\Box$

Now suppose that an NP-complete problem Q is solved by a BPP-algorithm A. As we discussed, we assume that the instances of Q are encoded by a fixed encoding scheme such that the total number of (yes- and no-) instances of length n is bounded by c^n, where $c \geq 2$ is an integer. By Theorem 4.3, we can assume that the failure probability of the algorithm A is bounded by $1/(2c^n)$ on instances of length n. Assume that the instances of length n for Q are $x_1, \ldots, x_m$, where $m \leq c^n$. Let E_i be the event that the algorithm A fails on the instance x_i. Then $\Pr[E_i] \leq 1/(2c^n)$.

Consider the binary tree T_A corresponding to the algorithm A on instances of length n, which is a complete binary tree of depth $O(p(n))$, where $p(n)$ is a polynomial of n and $O(p(n))$ is the running time of the algorithm A on inputs of length n. Suppose that the tree T_A has $N = 2^h$ leaves, all at level h, where $h = O(p(n))$. From $\Pr[E_1] \leq 1/(2c^n)$, we know that on the input x_1, there are at most $N/(2c^n)$ of the N leaves of T_A that are labeled with an incorrect decision. Mark all leaves that have the incorrect label on input x_1. Similarly we do this for all other instances x_2, ..., x_m of length n (note that the tree T_A is the same for all instances of length n), and mark the leaves that are ever labeled with an incorrect decision for some instances (note that a leaf may be labeled with an incorrect decision for more than one instances). The total number of leaves that are ever marked in this process (over all instances of length n) is bounded by

$$m \cdot N/(2c^n) \leq c^n \cdot N/(2c^n) = N/2.$$

Therefore, there are some leaves in the tree T_A (in fact at least half of the leaves) whose labels are always correct for *all* instances of length n! Let l be any such a leaf and let P_l be the path from the root to l in the tree T_A. The path P_l can be encoded as a binary string s_n of length $h = O(p(n))$, where the i-th bit of s_n is the outcome of the i-th random bit generation along the path P_l. Now with the string s_n, we can convert the randomized algorithm A into a deterministic algorithm A_d, which simulates the randomized algorithm A but at each step of generating a random bit, it reads directly the corresponding bit in the string s_n instead. The algorithm A_d has the same running time as that of A, thus runs in polynomial time. Moreover, by the assumption on the path P_l and the string s_n, the algorithm A_d gives correct solutions to all instances of Q.

Therefore, a BPP-algorithm for the NP-complete problem Q implies the existence of an (infinite) set of binary strings $\mathit{Magic} = \{s_1, \ldots, s_n, \ldots\}$, where each string s_n has its length bounded by a polynomial $p(n)$ of n (where the polynomial p is fixed for all n), and a deterministic polynomial-time algorithm A_d such that on an instance x of length n for Q, the algorithm A_d, with the help of the string s_n, solves x. Since every problem in NP can be reduced to the NP-complete problem Q in polynomial time, the set *Magic* is in fact universal for *all* problems in NP: with the help of the set *Magic*, every problem in NP can be solved by a deterministic algorithm in polynomial time!

This does not quite imply P = NP, yet, because we do not know if the string s_n can be constructed in polynomial time. It has been a long-time interesting problem whether such a "sparse" set *Magic* exists. On the other hand, some very interesting results have been obtained on the consequence of the existence of such a set. One of them is that the existence of such a set would collapse the *polynomial-time hierarchy*, which seems very unlikely, although is a little bit weaker than claiming P = NP [16].

Theorem 4.4 (Karp-Lipton [13]) *Unless the polynomial-time hierarchy collapses, there is no such a set Magic with which an NP-complete problem can be solved by a deterministic polynomial-time algorithm.*

Combining Theorem 4.4 and our discussion above, we conclude that it is very unlikely that any NP-complete problem has a BPP-algorithm.