Design and Analysis of Algorithms

Mohammad GANJTABESH

mgtabesh@ut.ac.ir

School of Mathematics, Statistics and Computer Science,
University of Tehran,
Tehran, Iran.

Solving Hard Problems

Different Approaches

- Try to design algorithms for solving hard problems and accept their (worst case) exponential complexity if they are efficient and fast enough for most of the problem instances (partition the set of all instances into two subsets: easy instances hard instances).
- Try to design algorithms with slowly increasing worst case exponential time complexity ($O(1.2^n)$).
- Try to design algorithms that provide solutions with costs close to the cost of the optimal solutions (Approximation Algorithms).

Pseudo-Polynomial-Time Algorithms

- Integer-Valued Problems: problems whose inputs can be viewed as a collection of integers.
- The coding of input is a word over $\{0, 1, \#\}$, where

$$x = x_1 \# x_2 \# \cdots \# x_n.$$

- The interpretation is

$$Int(x) = (Number(x_1), Number(x_2), \cdots, Number(x_n)).$$

- We define

$$Max - Int(x) = \max\{Number(x_i) | i = 1, 2, \cdots, n\}.$$

Pseudo-Polynomial-Time Algorithms

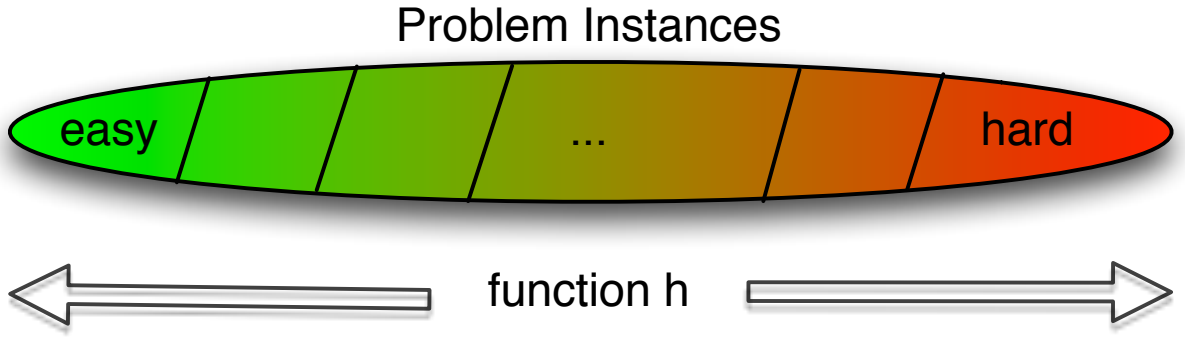
Definition

Let U be an integer-valued problem, and let A be an algorithm that solves U. We say that A is a pseudo-polynomial-time algorithm for U if there exists a polynomial p of two variables such that for every instance x of U we have:

$$Time_A(x) = O(p(|x|, Max - Int(x))).$$

Definition

Let U be an integer-valued problem, and let h be a nondecreasing function from $\mathbb{N}$ to $\mathbb{N}$. The h-value-bounded subproblem of U, $Value(h) - U$, is the problem obtained from U by restricting the set of all input instances of U to the set of input instances x with $Max - Int(x) < h(x)$.



Pseudo-Polynomial-Time Algorithms

Theorem

Let U be an integer-valued problem, and let A be a pseudo-polynomial-time algorithm for U. Then, for every polynomial h, there exists a polynomial-time algorithm for $\mathit{Value}(h) - U$ (i.e. if U is a decision problem then $\mathit{Value}(h) - U \in P$, and if U is an optimization problem then $\mathit{Value}(h) - U \in PO$).

Example

Dynamic Programming for $0/1$-Knapsack Problem...

Limits of Applicability

Question: Is there exist a pseudo-polynomial-time algorithm for any integer-valued problem?

Limits of Applicability

Question: Is there exist a pseudo-polynomial-time algorithm for any integer-valued problem?
Answer: No!

Definition

An integer-valued problem U is called strongly NP-hard if there exists a polynomial p such that the problem $Value(p) - U$ is NP-hard.

Limits of Applicability

Question: Is there exist a pseudo-polynomial-time algorithm for any integer-valued problem?
Answer: No!

Definition

An integer-valued problem U is called strongly NP-hard if there exists a polynomial p such that the problem $Value(p) - U$ is NP-hard.

Theorem

Let $P \neq NP$, and let U be a strongly NP-hard integer- valued problem. Then there does not exist any pseudo-polynomial-time algorithm solving U.

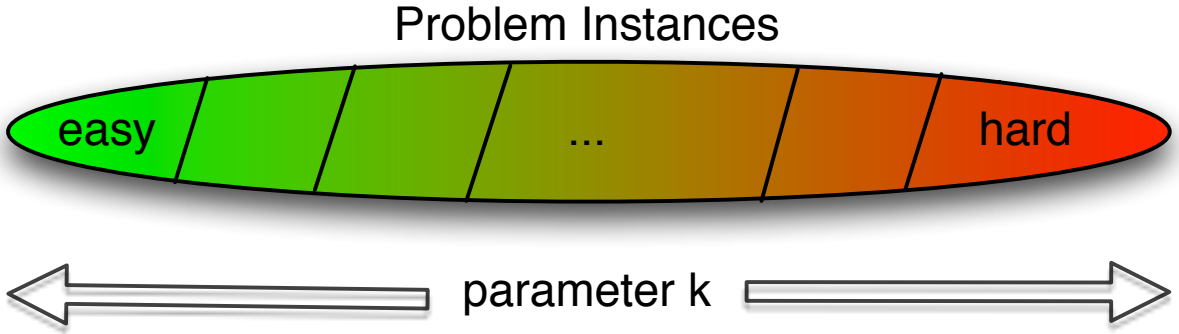
To prove the nonexistence of any pseudo-polynomial-time algorithm for an integer-valued problem U, it is sufficient to show that $Value(h) - U$ is NP-hard for a polynomial h.

Lemma

TSP is strongly NP-hard.

Parameterized Complexity

- Analyzing a given hard problem more precisely than by taking care of its worst case complexity.
- Searching for a parameter that partitions the set of all input instances into possibly infinite many subsets.
- The idea is to design an algorithm that is polynomial in the input size but possibly not in the value of the chosen parameter (example: $O(2^k n^2)$).
- Our efforts result in a partition of the set of all input instances into a spectrum of subclasses according to their hardness.



- Thinking about tractability by classifying the input instances of a particular problem according to their computational difficulty.

Parameterized Complexity

Definition

Let U be a problem, and let L be the language of all instances of U. A parameterization of U is any function $Par : L \mapsto \mathbb{N}$ such that

(i) Par is polynomial-time computable.

(ii) For infinitely many $k \in \mathbb{N}$, the k-fixed-parameter set $Set_U(k) = \{x \in L \mid Par(x) = k\}$ is an infinite set.

We say that A is a Par-parameterized polynomial-time algorithm for U if

(i) A solves U.

(ii) There exists a polynomial p and a function $f : \mathbb{N} \mapsto \mathbb{N}$ such that, for every $x \in L$,

$$Time_A(x) \leq f(Par(x)) \cdot p(|x|).$$

If there exists a Par-parameterized polynomial-time algorithm for U, then we say that U is fixed-parameter-tractable according to Par.

Parameterized Complexity

Theorem

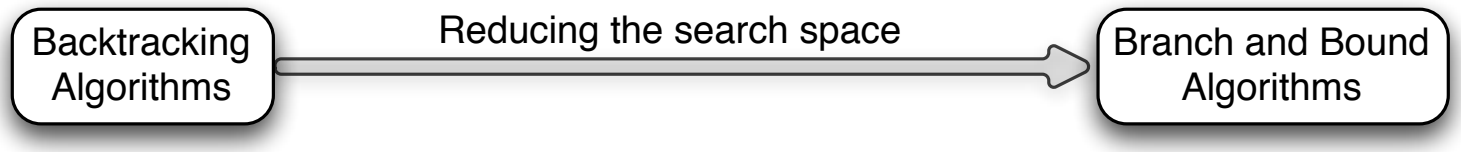
Let U be an integer-valued problem. Then, every pseudo-polynomial- time algorithm for U is a Par-parameterized polynomial-time algorithm for U.

Example

Parameterization of Vertex Cover Problem Consider (G, k) as input, one has to decide whether G possesses a vertex cover of size at most k. We define $Par(G, k) = k$ for all inputs (G, k). It can be shown that Par is a parameterization of the vertex cover problem.

Branch-and-Bound

- Branch-and-Bound is based on backtracking, which is an exhaustive searching technique in the space of all feasible solutions.



- The cardinality of the sets of feasible solutions are typically as large as 2^n, $n!$, or even n^n for inputs of size n.
- The idea of the branch-and-bound technique is to speed up backtracking by omitting the search in some parts of the space of feasible solutions, because one is already able to recognize that these parts do not contain any optimal solution in the moment when the exhaustive search would start to search in these parts.
- The branch-and-bound is based on some pre-computation of a bound on the cost of an optimal solution (a lower bound for maximization problems and an upper bound for minimization problems).

Lowering Worst Case Complexity

- Similar to the idea of branch-and-bound (accept an exponential complexity).
- But in contrast to branch-and-bound, the worst case complexity of the designed algorithm is substantially smaller than the complexity of any naive approach.
- For instance, consider the following table for an optimization problem with a set of feasible solutions of cardinality 2^n:

Complexity	$n = 10$	$n = 50$	$n = 100$	$n = 300$
2^n	1024	(16 digits)	(31 digits)	(91 digits)
$2^{n/2}$	32	$\sim 33 \cdot 10^6$	(16 digits)	(46 digits)
$(1.2)^n$	7	9100	$\sim 29 \cdot 10^6$	(24 digits)
$10 \cdot 2^{\sqrt{n}}$	89	1350	10240	$\sim 1.64 \cdot 10^6$
$n^2 \cdot 2^{\sqrt{n}}$	894	~ 336000	$\sim 10.24 \cdot 10^6$	$\sim 14.8 \cdot 10^9$

Solving $3SAT$ in Less than 2^n Complexity

- Consider the decision problem $3SAT$, i.e., to decide whether a given formula F in $3CNF$ is satisfiable.
- If F is over n variables, the naive approach leads to the (worst case) complexity $O(|F| \cdot 2^n)$.
- Using the divide-and-conquer method, we show that $3SAT$ can be decided in $O(|F| \cdot 1.84^n)$ time.

Solving 3*SAT* in Less than 2^n Complexity

- Let ℓ be a literal that occurs in F. Then $F(\ell = 1)$ denotes a formula that is obtained from F by consecutively applying the following rules:
  - (i) All clauses containing the literal ℓ are removed from F.
  - (ii) If a clause of F contains the literal $\overline{\ell}$ and still at least one different literal from $\overline{\ell}$, then $\overline{\ell}$ is removed from the clause.
  - (iii) If a clause of F consists of the literal $\overline{\ell}$ only, then $F(\ell = 1) = 0$ (i.e., an unsatisfiable formula).

Solving 3*SAT* in Less than 2^n Complexity

- Let ℓ be a literal that occurs in F. Then $F(\ell = 1)$ denotes a formula that is obtained from F by consecutively applying the following rules:
  - (i) All clauses containing the literal ℓ are removed from F.
  - (ii) If a clause of F contains the literal $\overline{\ell}$ and still at least one different literal from $\overline{\ell}$, then $\overline{\ell}$ is removed from the clause.
  - (iii) If a clause of F consists of the literal $\overline{\ell}$ only, then $F(\ell = 1) = 0$ (i.e., an unsatisfiable formula).
- $F(\ell = 0)$ can be constructed similarly.
- In general, for literals $\ell_1, \ell_2, \cdots, \ell_c, h_1, h_2, \cdots, h_d$, the formula

$$F(\ell_1 = 1, \ell_2 = 1, \cdots, \ell_c = 1, h_1 = 0, h_2 = 0, \cdots, h_d = 0)$$

  is obtained from F by constructing $F(\ell_1 = 1)$, then by constructing $F(\ell_1 = 1)(\ell_2 = 1)$, etc.

Solving 3*SAT* in Less than 2^n Complexity

- Let ℓ be a literal that occurs in F. Then $F(\ell = 1)$ denotes a formula that is obtained from F by consecutively applying the following rules:
  - (i) All clauses containing the literal ℓ are removed from F.
  - (ii) If a clause of F contains the literal $\overline{\ell}$ and still at least one different literal from $\overline{\ell}$, then $\overline{\ell}$ is removed from the clause.
  - (iii) If a clause of F consists of the literal $\overline{\ell}$ only, then $F(\ell = 1) = 0$ (i.e., an unsatisfiable formula).
- $F(\ell = 0)$ can be constructed similarly.
- In general, for literals $\ell_1, \ell_2, \cdots, \ell_c, h_1, h_2, \cdots, h_d$, the formula

$$F(\ell_1 = 1, \ell_2 = 1, \cdots, \ell_c = 1, h_1 = 0, h_2 = 0, \cdots, h_d = 0)$$

  is obtained from F by constructing $F(\ell_1 = 1)$, then by constructing $F(\ell_1 = 1)(\ell_2 = 1)$, etc.
- Obviously, $\ell_1 = 1, \ell_2 = 1, \cdots, \ell_c = 1, h_1 = 0, h_2 = 0, \cdots, h_d = 0$ determine a partial assignment to the variables of F. Thus, the question whether $F(\ell_1 = 1, \ell_2 = 1, \cdots, \ell_c = 1, h_1 = 0, h_2 = 0, \cdots, h_d = 0)$ is satisfiable is equivalent to the question whether there exists an assignment satisfying $\ell_1 = 1, \ell_2 = 1, \cdots, \ell_c = 1, h_1 = 0, h_2 = 0, \cdots, h_d = 0$, and F.

Solving 3*SAT* in Less than 2^n Complexity

- The important fact is that, for every literal ℓ of F and $a \in \{0,1\}$, $F(\ell = a)$ contains fewer variables than F.
- Let for all positive integers n and r,

$$3CNF(n,r) = \{\varphi | \varphi \text{ is a formula over at most } n \text{ variables in } 3CNF \text{ and } \varphi \text{ contains at most } r \text{ clauses}\}.$$

- Let $F \in 3CNF(n,r)$ and let $(\ell_1 \vee \ell_2 \vee \ell_3)$ be some clause of F. Then

$$F \text{ is satisfiable} \iff \text{at least one of the formulae } F(\ell_1 = 1), F(\ell_1 = 0, \ell_2 = 1), \text{ or } F(\ell_1 = 0, \ell_2 = 0, \ell_3 = 1) \text{ is satisfiable.}$$

- Following the previous rules, it is obvious that

$$\begin{aligned} F(\ell_1 = 1) &\in 3CNF(n-1, r-1) \\ F(\ell_1 = 0, \ell_2 = 1) &\in 3CNF(n-2, r-1) \\ F(\ell_1 = 0, \ell_2 = 1, \ell_3 = 1) &\in 3CNF(n-3, r-1). \end{aligned}$$

- In this way, we have reduced the F from $3CNF(n,r)$ to three sub-instances of F from $3CNF(n-1, r-1)$, $3CNF(n-2, r-1)$, and $3CNF(n-3, r-1)$.

Solving 3*SAT* in Less than 2^n Complexity

Algorithm 3.5.2.1 (D&C-3SAT(F)).

Input: A formula F in 3CNF.

Step 1: **if** $F \in 3\text{CNF}(3,k)$ or $F \in 3\text{CNF}(m,2)$ for some $m,k \in \mathbb{N}-\{0\}$,
then decide whether $F \in 3\text{SAT}$ or not by testing all assignments to the variables of F;
if $F \in 3\text{SAT}$ **output**(1) **else** **output**(0).

Step 2: Let H be one of the shortest clauses of F.
if $H=(l)$ **then** **output**(D&C-3SAT($F(l=1)$));
if $H=(l_1 \vee l_2)$
then output(D&C-3SAT($F(l_1=1)$)
$\vee$D&C-3SAT($F(l_1=0,l_2=1)$));
if $H=(l_1 \vee l_2 \vee l_3)$
then **output**(D&C-3SAT($F(l_1=1)$)
$\vee$D&C-3SAT($F(l_1=0,l_2=1)$)
$\vee$D&C-3SAT($F(l_1=0,l_2=0,l_3=1)$)).

Local Search

Definition

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ be an optimization problem. For every $x \in L_I$, a neighborhood on $M(x)$ is any mapping $f_x : M(x) \mapsto Pot(M(x))$ such that

(i) $\alpha \in f_x(\alpha)$ for every $\alpha \in M(x)$,

(ii) if $\beta \in f_x(\alpha)$ for some $\alpha \in M(x)$, then $\alpha \in f_x(\beta)$, and

(iii) for all $\alpha, \beta \in M(x)$ there exists a positive integer k and $\gamma_1, \gamma_2, \cdots, \gamma_k \in M(x)$ such that $\gamma_1 \in f_x(\alpha)$, $\gamma_{i+1} \in f_x(\gamma_i)$ for $i = 1, 2, \cdots, k-1$, and $\beta \in f_x(\gamma_k)$.

If $\alpha \in f_x(\beta)$ for some $\alpha, \beta \in M(x)$, we say that α and β are neighbors in M(x). The set $f_x(\alpha)$ is called the neighborhood of the feasible solution α in $M(x)$. The (undirected) graph

$$G_{M(x),f_x} = (M(x), \{(\alpha, \beta) \mid \alpha \in f_x(\beta),\ \alpha \neq \beta,\ \alpha, \beta \in M(x)\})$$

is the neighborhood graph of $M(x)$ according to the neighborhood f_x. Let, for every $x \in L_I$, f_x be a neighborhood on $M(x)$. The function $f : \cup_{x \in L_I}(\{x\}, M(x)) \mapsto \cup_{x \in L_I} Pot(M(x))$ with the property $f(x, \alpha) = f_x(\alpha)$ for every $x \in L_I$ and every $\alpha \in M(x)$ is called a neighborhood for U.

Local Search

- Local search in $M(x)$ is an iterative movement in M from a feasible solution to a neighboring feasible solution.
- Condition (ii) of the previous definition assures us that the neighborhood graph $G_{M(x),f}$ is connected, i.e., every feasible solution $\beta \in M(x)$ is reachable from any solution $\alpha \in M(x)$ by iteratively moving from a neighbor to a neighbor.
- Instead of a neighborhood on $M(x)$, one can use so-called local transformations on $M(x)$. A local transformation transforms a feasible solution α to a feasible solution β by some local changes of the specification of α.

Local Search

- Local search in $M(x)$ is an iterative movement in M from a feasible solution to a neighboring feasible solution.
- Condition (ii) of the previous definition assures us that the neighborhood graph $G_{M(x),f}$ is connected, i.e., every feasible solution $\beta \in M(x)$ is reachable from any solution $\alpha \in M(x)$ by iteratively moving from a neighbor to a neighbor.
- Instead of a neighborhood on $M(x)$, one can use so-called local transformations on $M(x)$. A local transformation transforms a feasible solution α to a feasible solution β by some local changes of the specification of α.

Definition

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ be an optimization problem, and let, for every $x \in L_I$, the function f_x be neighborhood on $M(x)$. A feasible solution $\alpha \in M(x)$ is a local optimum for the input instance x of U according to f_x, if

$$cost(\alpha) = goal\{cost(\beta) \mid \beta \in f_x(\alpha)\}.$$

We denote the set of all local optima for x according to the neighborhood f_x by $LocOPT_U(x, f_x)$.

Local Search

Roughly speaking, a local search algorithm starts off with an initial solution and then continually tries to find a better solution by searching neighborhoods. If there is no better solution in the neighborhood, then it stops. Having a structure on $M(x)$ determined by a neighborhood $Neigh_x$ for every $x \in L_I$, one can describe a general scheme of local search as follows.

LSS($Neigh$)-Local Search Scheme according to a neighborhood $Neigh$

Input: An input instance x of an optimization problem U.
Step 1: Find a feasible solution $\alpha \in \mathcal{M}(x)$.
Step 2: **while** $\alpha \notin LocOPT_U(x, Neigh_x)$ **do**
begin find a $\beta \in Neigh_x(\alpha)$ such that
$cost(\beta) < cost(\alpha)$ if U is a minimization problem and
$cost(\beta) > cost(\alpha)$ if U is a maximization problem; $\alpha := \beta$
end
Output: **output**(α).

Local Search

Besides the choice of the neighborhood, the following two free parameters of *LSS*(*Neigh*) may influence the success of the local search:

- An initial feasible solution can be randomly chosen or it can be precomputed by any other algorithmic method. The choice of an initial solution can essentially influence the quality of the resulting local optimum. This is the reason why one sometimes performs *LSS*(*Neigh*) several times starting with different initial feasible solutions (multi-start local search).
- There are several ways to choose the cost-improving feasible solution in Step 2 as follows:
  - The first improvement strategy means that the current feasible solution is replaced by the first cost-improving feasible solution found by the neighborhood search.
  - The best improvement strategy means that the current feasible solution is replaced by the best feasible solution in its neighborhood.

  Obviously, the first improvement strategy can make one single run of the while cycle faster than the best improvement strategy, but the best improvement strategy may decrease the number of executed runs of the while cycle.

Local Search

The time complexity of any local search algorithm can be roughly bounded by

$$\text{(time of the search neighborhood)} \times \text{(the number of improvements)}.$$

Question

For which NP-hard optimization problems can one find a neighborhood *Neigh* of polynomial size such that *LSS(Neigh)* always outputs an optimal solution?

Local Search

The time complexity of any local search algorithm can be roughly bounded by

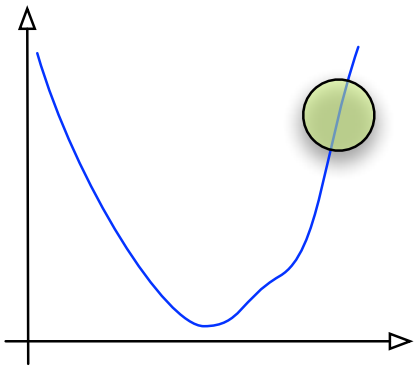
(time of the search neighborhood) $\times$ (the number of improvements).

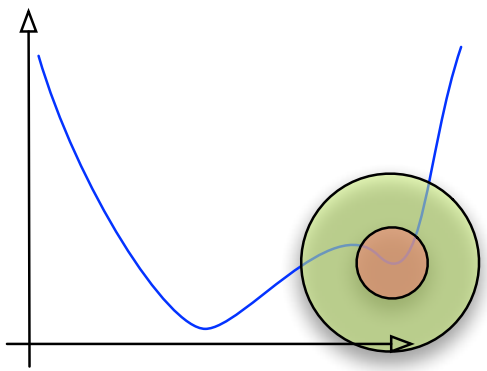
Question

For which NP-hard optimization problems can one find a neighborhood *Neigh* of polynomial size such that *LSS(Neigh)* always outputs an optimal solution?

Definition

Let $U = (\sum_I, \sum_O, L, L_I, M, cost, goal)$ be an optimization problem, and let f be a neighborhood for U. f is called an exact neighborhood, if, for every $x \in L_I$, every local optimum for x according to f_x is an optimal solution to x. A neighborhood f is called polynomial-time searchable if there is a polynomial-time algorithm that, for every $x \in L_I$ and every $\alpha \in M(x)$, finds one of the best feasible solutions in $f_x(\alpha)$.





Hardness from Local Search Point of View

How we can prove that an optimization problem is hard for local search in the above-mentioned sense?

Theorem

Let $U \in NPO$ be a cost-bounded integer-valued optimization problem such that there is a polynomial-time algorithm that, for every instance x of U, computes a feasible solution for x. If $P \neq NP$ and U is strongly NP-hard, then U does not possess an exact, polynomial-time searchable neighborhood.

Corollary

If $P \neq NP$, then there exists no exact polynomial-time searchable neighborhood for TSP, $\Delta - TSP$, and $Weight - VCP$.

Definition

Let $U = (\sum_I, \sum_O, L, L_I, M, cost, goal)$ be an optimization problem from *NPO*. We define the suboptimality decision problem to U as the decision problem $(SUBOPT_U, \sum_I \cup \sum_O)$, where $SUBOPT_U = \{(x, \alpha) \in L_I \times \sum_O^* \mid \alpha \in M(x) \text{ and } \alpha \text{ is not optimal}\}$.

Theorem

Let $U \in NPO$. If $P \neq NP$, and $SUBOPT_U$ is NP-hard, then U does not possess any exact, polynomial-time searchable neighborhood.

Hardness from Local Search Point of View

Restricted Hamiltonian Cycle Problem (RHC)

The restricted Hamiltonian cycle problem (RHC) is to decide, for a given graph $G = (V, E)$ and a Hamiltonian path P in G, whether there exists a Hamiltonian cycle in G.

Lemma

RHC is NP-Complete.

Lemma

$SUBOPT_{TSP}$ is NP-Hard (i.e. RHC $\leq_p$ $SUBOPT_{TSP}$).

Corollary

$SUBOPT_{TSP}$ is NP-hard , and so, if P $\neq$ NP, TSP does not possess any exact, polynomial-time searchable neighborhood.

Approximation Algorithms

Main Question

If an optimization problem does not admit any efficient algorithm computing an optimal solution, is there a possibility to efficiently compute at least an approximation of the optimal solution?

- Jump from exponential complexity to polynomial complexity.
- Small change in the requirements - instead of an exact optimal solution one demands a solution whose cost differs from the cost of an optimal solution by at most $\varepsilon\%$ of the cost of an optimal solution for some $\varepsilon > 0$.

Approximation Algorithms

Definition

Let $U = (\sum_I, \sum_O, L, L_I, M, cost, goal)$ be an optimization problem, and let A be a consistent algorithm for U. For every $x \in L_I$, the relative error $\varepsilon_A(x)$ of A on x is defined as

$$\varepsilon_A(x) = \frac{\mid Cost(A(x)) - Opt_U(x) \mid}{Opt_U(x)}.$$

For any $n \in \mathbb{N}$, we define the relative error of A as

$$\varepsilon_A(n) = \max\{\varepsilon_A(x) \mid x \in L_I \ \& \ \mid x \mid = n\}.$$

Definition

For every $x \in L_I$, the approximation ratio $R_A(x)$ of A on x is defined as

$$R_A(x) = \max\{\frac{Cost(A(x))}{Opt_U(x)}, \frac{Opt_U(x)}{Cost(A(x))}\} = 1 + \varepsilon_A(x).$$

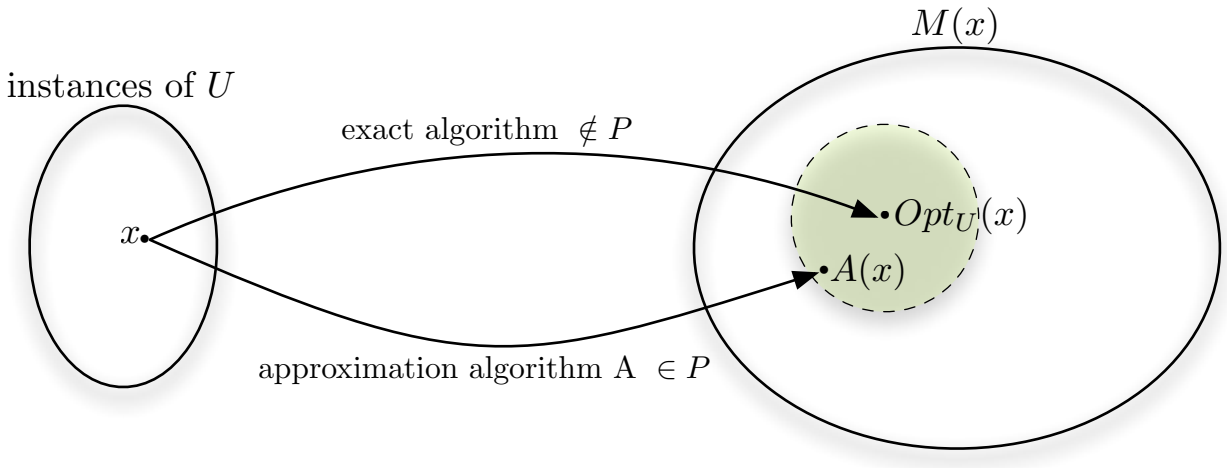
For any $n \in \mathbb{N}$, we define the approximation ratio of A as

$$R_A(n) = \max\{R_A(x) \mid x \in I_U \ \& \mid x \mid = n\}.$$

Approximation Algorithms

Definition

For any positive real $\delta > 1$, we say that A is a δ-approximation algorithm for U if $R_A(x) \leq \delta$ for every $x \in L_I$.



Definition

For every function $f : \mathbb{N} \mapsto \mathbb{R}^+$, we say that A is an $f(n)$-approximation algorithm for U if $R_n(x) \leq f(n)$ for every $n \in \mathbb{N}$.

- If U is a minimization problem, then $R_A(x) = \frac{Cost(A(x))}{Opt_U(x)}$.
- If U is a maximization problem, then $R_A(x) = \frac{Opt_U(x)}{Cost(A(x))}$.

Approximation Algorithms: Classification

NPO(*I*): Contains every optimization problem from *NPO* for which there exists a *FPTAS* (knapsack problem).

NPO(*II*): Contains every optimization problem from *NPO* that has a *PTAS* (makespan scheduling problem).

NPO(*III*): Contains every optimization problem $U \in NPO$ such that

- there is a polynomial-time δ-approximation algorithm for some $\delta > 1$.
- there is no polynomial-time d-approximation algorithm for U for some $d < \delta$, i.e., there is no *PTAS* for U.

(minimum vertex cover problem, Max-Sat, and Δ-TSP)

NPO(*IV*): Contains every $U \in NPO$ such that

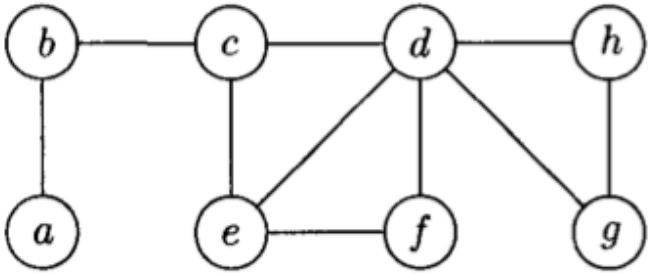
- there is a polynomial-time $f(n)$-approximation algorithm for U for some $f : \mathbb{N} \mapsto \mathbb{R}^+$, where f is bounded by a polylogarithmic function.
- there does not exist any polynomial-time δ-approximation algorithm for U for any $\delta \in \mathbb{R}^+$.

(set cover problem)

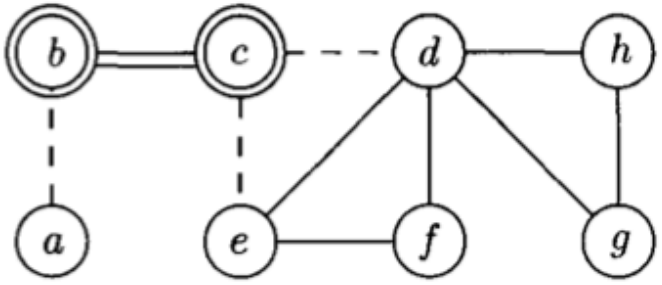
NPO(*V*): Contains every $U \in NPO$ such that if there exists a polynomial-time $f(n)$-approximation algorithm for U, then $f(n)$ is not bounded by any polylogarithmic function (TSP and maximum clique).

Approximation Algorithms: Vertex Cover Problem

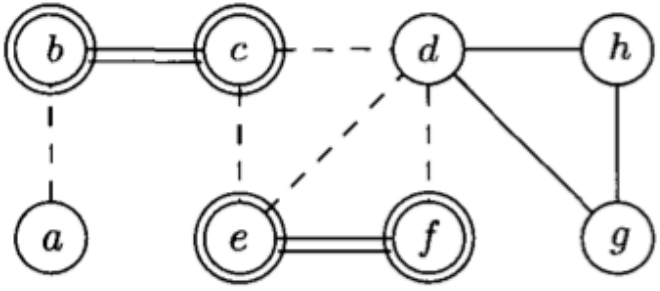
One quickly finds a maximal matching of the given graph and considers all vertices adjacent to the edges of the matching as a vertex cover.



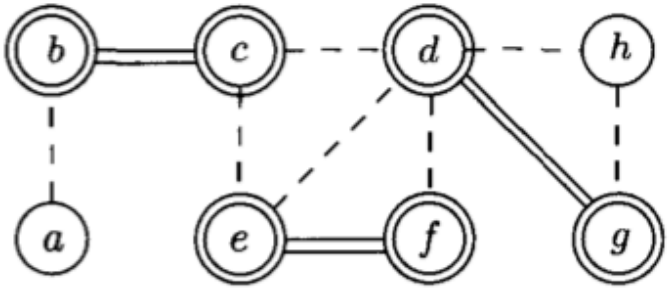
(a)



(b)



(c)



(d)

Approximation Algorithms: Vertex Cover Problem

Algorithm 4.3.2.1. Input: A graph $G = (V, E)$.

Step 1: $C := \emptyset$ {during the computation $C \subseteq V$, and at the end C should contain a vertex cover};
$A := \emptyset$ {during the computation $A \subseteq E$ is a matching, and at the end A is a maximal matching};
$E' := E$ {during the computation $E' \subseteq E$, E' contains exactly the edges that are not covered by the actual C, and at the end $E' = \emptyset$}.

Step 2: **while** $E' \neq \emptyset$
do begin choose an arbitrary edge $\{u, v\}$ from E';
$C := C \cup \{u, v\}$;
$A := A \cup \{\{u, v\}\}$;
$E' := E' - \{$all edges incident to u or $v\}$
end

Output: C.

Approximation Algorithms: Vertex Cover Problem

Lemma

Algorithm 4.3.2.1 *works in time* $O(|E|)$.

Lemma

Algorithm 4.3.2.1 *always computes a vertex cover with an approximation ratio at most* 2.

Proof.

- Note that $|C| = 2 \times |A|$, where A is a matching of the input graph $G = (V, E)$.
- To cover the $|A|$ edges of the matching A, one needs at least $|A|$ vertices.
- Since $A \subseteq E$, the cardinality of every vertex cover is at least $|A|$, i.e., $Opt_{MIN-VCP}(G) \geq |A|$.
- Thus, $|C| = 2 \times |A| \leq 2 \times Opt_{MIN-VCP}(G)$.

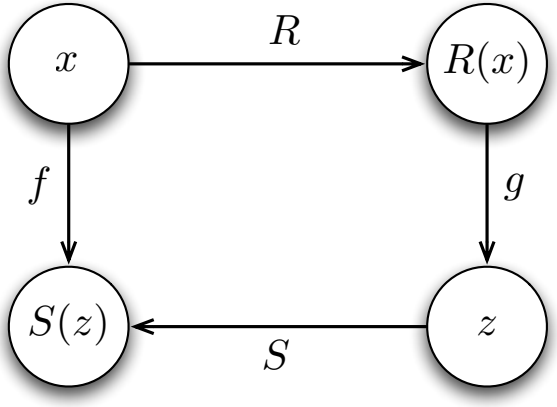
□

Approximation Algorithms and Complexity

Definition

Suppose that f and g are two functions over positive real numbers. The function-reduction is defined as follows:

$$f \preccurlyeq g \iff \exists(R,S),\ R,S \in \mathit{LSpace} \text{ such that}$$
$$1)\ \text{if } x \in \mathit{Instance}(f) \text{ then } R(x) \in \mathit{Instance}(g),$$
$$2)\ \text{if } z = \mathit{Solution}(R(x)) \text{ then } S(z) = \mathit{Solution}(x).$$



Definition

Suppose that FC is a family of functions. The class of $FC-$Complete functions is defined as follows:

$$f \in FC - \mathit{Complete} \iff \begin{cases} f \in FC, \\ \forall g \in FC,\ g \preccurlyeq f. \end{cases}$$

Approximation Algorithms and Complexity

Definition

Suppose that A and B are two optimization problems. The $L-$Reduction is defined as follows:

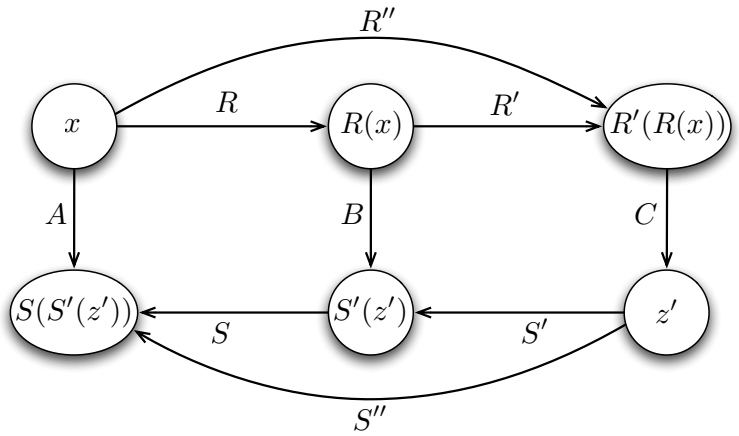
$$A \preccurlyeq B \iff \exists \alpha, \beta > 0, \exists (R, S),\ R, S \in LSpace \text{ such that}$$

1) if $x \in Instance(A)$ then $R(x) \in Instance(g)$,
   such that: $Opt(R(x)) \leq \alpha Opt(x)$
2) if $z \in Solution(R(x))$ then $S(z) \in Solution(x)$,
   such that: $|Opt(x) - Cost(S(z))| \leq \beta |Opt(R(x)) - Cost(z)|$

, where α and β are constant numbers.

Lemma

The L−Reduction has transitive property.



Approximation Algorithms and Complexity

Theorem

If there is an $L-$reduction (R,S) from A to B with constants α and β, and there is a polynomial-time $\varepsilon-$approximation algorithm for B, then there is a polynomial-time $\frac{\alpha\beta\varepsilon}{1-\varepsilon}-$approximation algorithm for A.

Corollary

If there is an $L-$reduction from A to B and there is a PTAS for B, then there is a PTAS for A.

Theorem

If $P \neq NP$ then there is no polynomial-time $d-$approximation algorithm for TSP.

Theorem

If $P \neq NP$ then there is no polynomial-time $f(n)-$approximation algorithm for TSP, where $f(n)$ is a polynomial function.

Randomized Algorithms

A randomized algorithm can be viewed as a nondeterministic algorithm that has a probability distribution for every nondeterministic choice.

- One usually considers only the random choices from two possibilities, each with the probability 1/2.
- Another possibility is to consider a randomized algorithm as a deterministic algorithm with an additional input that consists of a sequence of random bits.

Randomized Algorithms

For a fixed input instance x of the problem considered, the runs (computations) of a randomized algorithm on x may differ in the dependence on the actual sequence of random bits. Thus, running time or output can be considered as random variables.

- Randomized algorithms whose outputs can be considered as random variables are called Monte Carlo algorithms.
- A randomized algorithm that always (independent of the random choice) computes the correct output and only the complexity is considered as a random variable is called a Las Vegas algorithm.

Formalize the Randomized Algorithms,

1) Take deterministic Turing machines A with an infinite additional tape (read-only, binary, can be read from left to right).
2) Take a nondeterministic Turing machine with nondeterministic guesses over at most two possibilities and to assign the probability $1/2$ to every such possibility.

For us, it is sufficient to view randomized algorithms as algorithms that sometimes may ask for some random bits and may proceed in different ways depending on the values of these random bits.

Randomized Algorithms: new complexity measure

Definition

Let $Random_A(x)$ be the maximum number of random bits used over all random runs (computations) of A on x. Then, for every $n \in \mathbb{N}$,

$$Random_A(n) = \max\{Random_A(x) \mid x \text{ is an input of size } n\}.$$

This complexity measure is important because of the following two reasons:

- Producing random sequence is very expensive, specially for longer sequences.
- If $Random_A(x)$ is bounded by a logarithmic function, then the number of distinct runs on any fixed input of size n is bounded by $2^{Random_A(n)} \leq p(n)$ for a polynomial p (one can deterministically simulate A by following all runs of A). If A computes in polynomial time in every run, then this deterministic simulation runs in polynomial time, too (derandomization).

Randomized Algorithms

Definition

For every run (random computation) C of a randomized algorithm A on an input x, $Prob_{A,x}(C)$ is the probability of execution of C on x. This probability is the multiplication over the probabilities of all random choices done in run C of A on x, i.e., the probability of the corresponding random sequence (it is $1/2$ to the power of the number of random bits asked in C).

Definition

Because a randomized algorithm A may produce different results on a fixed input x, the output of a result y is considered an (random) event.The probability that A outputs y for an input x, $Prob(A(x) = y)$, is the sum of all $Prob_{A,x}(C)$, where C outputs y.

Randomized Algorithms

Definition

Let $Time(C)$ be the time complexity of the run C of A on x. Then the expected time complexity of A on x is

$$Exp - Time_A(x) = E[Time] = \sum_C Prob_{A,x}(C) \cdot Time(C),$$

where the sum is taken over all runs C of A on x. Also, the expected time complexity of A for every $n \in \mathbb{N}$ is

$$Exp - Time_A(n) = \max\{Exp - Time_A(x) \mid x \text{ is an input of size } n\}.$$

Since analyzing the $Exp - Time_A(n)$ for randomized algorithm A is usually difficult, the worst case approach from the beginning is used:

$$Time_A(x) = \max\{Time(C) \mid C \text{ is a run of } A \text{ on } x\}.$$

Then, the (worst case) time complexity of A is

$$Time_A(n) = \max\{Time_A(x) \mid x \text{ is an input of size } n\}.$$

Randomized Algorithms: Classification

1) Las Vegas algorithms: never give a wrong output!

2) Monte Carlo algorithms: may produce wrong answer.

   - one-sided-error
   - two-sided-error
   - unbounded-error

Randomized Algorithms: Las Vegas

Definition 1

A randomized algorithm A is a Las Vegas algorithm computing a problem F if for any input instance x of F,

$$Prob(A(x) = F(x)) = 1.$$

(appropriate for computing a function and $Exp - Time_A(n)$ is the complexity measure)

Randomized Algorithms: Las Vegas

Definition 1

A randomized algorithm A is a Las Vegas algorithm computing a problem F if for any input instance x of F,

$$Prob(A(x) = F(x)) = 1.$$

(appropriate for computing a function and $Exp - Time_A(n)$ is the complexity measure)

Definition 2

A randomized algorithm A is a Las Vegas algorithm computing a problem F if for every input instance x of F,

$$\begin{aligned} Prob(A(x) = F(x)) &\geq \frac{1}{2}, \\ Prob(A(x) = \text{"?"}) = 1 - Prob(A(x) = F(x)) &\leq \frac{1}{2}. \end{aligned}$$

(appropriate for decision problems and $Time_A(n)$ is the complexity measure)

Example 1: Randomized Quick Sort

Algorithm 5.2.2.2 RQS (RANDOMIZED QUICKSORT)

Input: $a_1, \ldots, a_n$, $a_i \in S$ for $i = 1, \ldots, n$, $n \in \mathbb{N}$.

Step 1: Choose an $i \in \{1, \ldots, n\}$ uniformly at random.
{Every $i \in \{1, \ldots, n\}$ has equal probability to be chosen.}

Step 2: Let A be the multiset $\{a_1, \ldots, a_n\}$.
if $n = 1$ **output**(S)
else the multisets $S_<, S_=, S_>$ are created.
$S_< := \{b \in A \mid b < a_i\}$;
$S_= := \{b \in A \mid b = a_i\}$;
$S_> := \{b \in A \mid b > a_i\}$.

Step 3: Recursively sort $S_<$ and $S_>$.

Output: RQS$(S_<)$, $S_=$, RQS$(S_>)$.

Theorem

RQS is a Las Vegas algorithm.

Example 2: Randomized Select

Algorithm 5.2.2.4. RANDOM-SELECT(S, k)

Input: $S = \{a_1, a_2, \ldots, a_n\}$, $n \in \mathbb{N}$, and a positive integer $k \leq n$.

Step 1: **if** $n = 1$ **then return** a_1
else choose an $i \in \{1, 2, \ldots, n\}$ randomly.

Step 2: $S_< = \{b \in S \mid b < a_i\}$;
$S_> = \{c \in S \mid c > a_i\}$.

Step 3: **if** $|S_<| > k$ **then** RANDOM-SELECT$(S_<, k)$
else if $|S_<| = k - 1$ **then return** a_i
else RANDOM-SELECT$(S_>, k - |S_<| - 1)$.

Output: the kth smallest element of S
(i.e., an a_l such that $|\{b \in S \mid b < a_l\}| = k - 1$).

Theorem

Random-Select is a Las Vegas algorithm.

One-Sided-Error Monte Carlo algorithm

Definition

Let L be a language, and let A be a randomized algorithm. We say that A is a one-sided-error Monte Carlo algorithm recognizing L if

(i) for every $x \in L$, $Prob(A(x) = 1) \geq 1/2$, and

(ii) for every $x \notin L$, $Prob(A(x) = 0) = 1$.

- It never says "yes" if the input does not have the required property.
- It is very practical because the probability of getting the right answer grows exponentially with the number of repetitions. (How?)

One-Sided-Error Monte Carlo algorithm

- Let $a_1, a_2, \cdots, a_k$ be k answers of k independent runs of a one-sided-error Monte Carlo algorithm A on the same input instance x.
- If there exists $i \in \{1, 2, \cdots, k\}$ such that $a_i = 1$, then we know with certainty that $x \in L$.
- If $a_1 = a_2 = \cdots = a_k = 0$, the probability that $x \in L$ is $(1/2)^k$.
- So, we decide to consider $x \notin L$, and this is true with probability $1 - 1/2^k$.

Two-Sided-Error Monte Carlo algorithm

Definition

Let F be a computing problem. We say that a randomized algorithm A is a two-sided-error Monte Carlo algorithm computing F if there exists a real number ε, $0 < \varepsilon \leq 1/2$, such that for every input x of F

$$Prob(A(x) = F(x)) \geq \frac{1}{2} + \varepsilon$$

.

- The strategy is to let the algorithm run t times on the given input, and
- To take as the output the result which appears at least $\lceil t/2 \rceil$ times, if any.

Two-Sided-Error Monte Carlo algorithm

- Let $p = p(x) \geq 1/2 + \varepsilon$ be the probability that A computes the correct result on a given input x in one run.
- The probability that A gives the correct answer on the input x exactly $i \leq \lfloor t/2 \rfloor$ times in t runs is

$$pr_i(x) = \binom{t}{i} p^i (1-p)^{t-i} \leq \binom{t}{i} \left(\frac{1}{4} - \varepsilon^2\right)^{\frac{t}{2}}.$$

Two-Sided-Error Monte Carlo algorithm

- Let $p = p(x) \geq 1/2 + \varepsilon$ be the probability that A computes the correct result on a given input x in one run.
- The probability that A gives the correct answer on the input x exactly $i \leq \lfloor t/2 \rfloor$ times in t runs is

$$pr_i(x) = \binom{t}{i} p^i (1-p)^{t-i} \leq \binom{t}{i} \left(\frac{1}{4} - \varepsilon^2\right)^{\frac{t}{2}}.$$

- Now, consider the following algorithm A_t.

Algorithm A_t

Input:	x
Step 1:	Run the algorithm A on x t times independently and save the t outputs $y_1, y_2, \ldots, y_t$.
Step 2:	$y :=$ an output from the multiset $\{y_1, \ldots, y_t\}$ with the property $y = y_i$ for at least $\lceil t/2 \rceil$ different is from $\{1, \ldots, t\}$, if any. $\{y =$? if there is no output with at least $\lceil t/2 \rceil$ occurrences in the sequence $y_1, \ldots, y_t.\}$
Output:	y

Two-Sided-Error Monte Carlo algorithm

- Since A_t computes $F(x)$ if and only if at least $\lceil t/2 \rceil$ runs of A finish with the output $F(x)$, one obtains

$$Prob(A_t(x) = F(x)) \geq 1 - \sum_{i=0}^{\lfloor t/2 \rfloor} pr_i(x) > 1 - \frac{1}{2}(1 - 4\varepsilon^2)^{t/2}.$$

- Thus, if one looks for a k such that $Prob(A_k(x) = F(x)) \geq 1 - \delta$ for a chosen constant δ and any input x, then it is sufficient to take

$$k \geq \frac{2\ln(2\delta)}{\ln(1 - 4\varepsilon^2)}.$$

- Obviously, if δ and ε are assumed to be constants then k is a constant, and $Time_{A_k}(n) \in O(Time_A(n))$.

Corollary

If A be a One-Sided-Error Monte Carlo algorithm, then A_2 is a Two-Sided-Error Monte Carlo algorithm.

Unbounded-Error Monte Carlo algorithm

Definition

Let F be a computing problem. We say that a randomized algorithm A is a unbounded-error Monte Carlo algorithm computing F if for every input x of F,

$$Prob(A(x) = F(x)) > \frac{1}{2}$$

.

- What is the essential difference between two-sided-error and unbounded-error Monte Carlo algorithms?
- We have to analyze the number of necessary repetitions k of an unbounded-error Monte Carlo algorithm A in order to get a two-sided error Monte Carlo algorithm A_k with

$$Prob(A_k(x) = F(x)) \geq 1 - \delta$$

for some constant δ, $0 \leq \delta \leq 1/2$.

Unbounded-Error Monte Carlo algorithm

- For an input x, A can have $2^{Random_A(|x|)}$ different computations, each with probability $2^{-Random_A(|x|)}$.
- It may happen that

$$Prob(A(x) = F(x)) = \frac{1}{2} + 2^{-Random_A(|x|)} > \frac{1}{2}$$

- Now we have

$$k = k(|x|) \geq (-\ln(2\delta)) \cdot 2^{2Random_A(|x|)-1}.$$

- Since $Random_A(|x|) \leq Time_A(|x|)$ and one considers $Time_A(n)$ to be bounded by a polynomial, k may be exponential in $|x|$.

In order to get a randomized algorithm $A_{k(n)}$ with

$$Prob(A_{k(|x|)}(x) = F(x)) \geq 1 - \delta$$

from an unbounded-error Monte Carlo algorithm A, one is forced to accept

$$Time_{A_{k(n)}}(n) = O(2^{2Random_A(n)} \cdot Time_A(n)).$$

And many other approaches...