Complexity Measures

- In traditional algorithm design, the primary complexity measures used to determine the efficiency of an algorithm are running time and memory space used
- In network algorithms, *inputs are spread across the computers* of the network and *computations must be carried out across many computers* too
- We need to consider various complexity measures in order to characterise the *performance of network algorithms*

Computational Rounds

- Several network algorithms proceed via a *series* of global computational rounds so as to eventually converge on a solution
- The *number of rounds* needed for convergence can be used as a crude approximation of time
- In *synchronous algorithms*, these rounds are determined by clock ticks
- In *asynchronous algorithms*, these rounds are often determined by propagating waves of events across the network

Space Measure

- The amount of space needed by a computation can be used for network algorithms, but it must be qualified as to whether it is:
  - a global bound on the total space used by all computers in the algorithms, or
  - a local bound on how much space is needed per computer involved

Local Running Time

- For asynchronous algorithms, we can focus on analysis of local computing time needed for a particular computer to participate in a network algorithm
- If *all computers* are essentially performing the *same type of function*, then a single local running time bound can suffice for all
- If the *computers* and their *functions differ*, we need to characterise the local running time for *each class of computers*

Message Complexity

- This parameter measures the total number of messages (of fixed or unlimited size) that are *sent between all pairs of computers* during the computation
- **For example**, if message M is routed via p edges to get from one computer to another, we would say that the message complexity of this communication task is $p|M|$, where $|M|$ denotes the size of M

Complexity Measures

- *Complexity measures* for network algorithms are often thought of as being functions of some intuitive notions of the ''size'' of the problem:
  - #of words used in description of the input
  - the number of processors deployed
  - the number of communication connections between processors

Basic Probability

- When we analyse algorithms that use randomisation or if we wish to analyse the average-case performance of an algorithm
- Then we need to use some basic facts from probability theory

Sample space

- Sample space is defined as the set of all possible outcomes from some experiment
- E.g., flipping a coin until it comes up heads. The sample space is *infinite*, with the i^{th} *outcome* being a sequence of i tails followed by a single head, for $i = 0,1,2,3,4,\ldots$.

Probability Space, Events

- Probability space is a sample space S together with a probability function P, that maps subsets of S to real numbers in interval $[0,1]$.
- Formally each subset A of S is called an event

Probability Function

- Probability function P is assumed to posses the following properties
  - $P(\emptyset) = 0$
  - $P(S) = 1$
  - $0 \le P(A) \le 1$, for any $A \subseteq S$
  - *If* $A, B \subseteq S$ and $A \cap B = \emptyset$,

    then $P(A \cup B) = P(A) + P(B)$

Independent Events

- Two events are *independent* if

$$P(A \cap B) = P(A) \cdot P(B)$$

- A collection of events $\{A_1, A_2, \ldots, A_n\}$ is *mutually independent* if

$$P(A_{i1} \cap \ldots \cap A_{ik}) = P(A_{i1}) \cdot \ldots \cdot P(A_{ik})$$

  for any subset $\{A_{i1}, \ldots, A_{ik}\}$

Example

- E.g., Let A be the event that the roll of a die is a 6, and let B be the event the roll of a second die is a 3, and let C be the event that sum of these two dice is a 10.
- Then A and B are independent events, but C is not independent with either A or B

Sub-linear algorithms

- In what follows we will consider algorithms which the running time is *sub-linear* in the size of the input.
- In particular, what it means is that only part of the input can be read and processed.
- There are many examples in which this setting is interesting, mainly when dealing with large data sets.
- Some specific examples include: large data streams, scientific databases, the world wide web, data from the Genome Project, and high-resolution images.

Sub-linear algorithms

- The *exact solution* to a *decision problem* is characterized by always correct answer of the form either yes/no or accept/reject
- In case of *approximate solution* to a decision problem the answer is positive (with high probability):
  - on all positively recognized inputs by an algorithm that provides the exact solutions, and
  - on some other inputs which are relatively "close" to those positively recognized by the exact algorithm

Sub-linear algorithms

- More formally, we split the set of all possible inputs into YES-instances (on which an exact algorithm should accept) and NO-instances (on which an exact algorithm should reject).
- We will require that on YES-instance, the algorithm will accept with probability at least 2/3, and all NO-instances will be split into two groups.
- One group contains all inputs that are "far" from all YES-instances, and one group that contains inputs that are "close" to some YES-instance.
- We require that the algorithm rejects inputs of the first kind with probability at least 2/3, and in real terms we do not care how the algorithm handles inputs of the second kind.

Example: 0^*1^* Strings

- The *0^*1^* (00..011..1) string problem*
- ***Input:*** $x = x[0..n-1]$, where $x[i] \in \{0, 1\}\ \forall\ i \in \{0,..,n-1\}$
- ***Output:*** positive if $x \in 0^*1^*$ and negative otherwise
- To get an exact answer all the time, it is necessary to read the entire input. This is because there may be only 1 bit that is "wrong", and we must see that bit in order to detect that the string is actually a NO-instance.

Example: 0^*1^* Strings

- But suppose we only want an approximate answer.
- First we define a distance between two strings as the fraction of bits on which the differ. This is called the *relative Hamming distance*:

$$\delta(x,y)=|\{i: x[i] \neq y[i]\}|/n.$$

- We will examine the distance of a string to a good string - that is, how many bits, if any, must be changed in order for the string to be of the form 0^*1^*.
- For example, the string *1010..10* is at distance *1/2* from being good.

Example: 0^*1^* Strings

- So now what we require from our algorithm is that if $x \in 0^*1^*$, then it accepts. If x is of distance at least ε from a good string, then our algorithm should reject with probability at least *2/3*.
- Consider the following algorithm:
  - Pick $s = 4/\varepsilon$ bits *uniformly at random*.
  - Check if they are in the correct order, and if not, *reject*.
  - Otherwise, *accept*.

Example: 0^*1^* Strings

- We will now analyze this algorithm.
- First note that it always accepts good strings, as any bits we choose will be in the correct order.
- Now suppose that x needs at least $\varepsilon \cdot n$ modifications in order to become a good string.
- We will show that we reject such x with probability *2/3*.

Example: 0^*1^* Strings

- Consider the first $\varepsilon \cdot n/2$ *1*s of the string x. Note that there must be at least $\varepsilon \cdot n/2$ *1*s, otherwise we could flip all 1s to *0*s, and our string will be the all *0* string. But this string is good, and x is at distance only $\varepsilon/2$ from it, contradicting the assumption that x is ε-far from good. We call these *1*s as *early 1s*.
- Now, x also contains at least $\varepsilon \cdot n/2$ 0s after the early *1*s, for the same reason. We call these as late *0*s.
- Now, while testing random bits, if we pick both an early *1* and a late *0*, we reject the string, otherwise we accept it.
- In what follows we analyze the probability that we do not reject a distant string, in which case the algorithm fails.

Example: 0^*1^* Strings

- P[*x[i]* is an early *1*] = $\varepsilon/2$, P[*x[i]* is a late *0*] = $\varepsilon/2$
  --- we also known that $(1-\alpha/n)^n \le e^{-\alpha}$, thus
- P_1= P[no early *1* found in *s* samples] = $(1- \varepsilon/2)^s \le e^{-\varepsilon \cdot s/2}$
- P_0= P[no late *0* found in *s* samples] $\le (1- \varepsilon/2)^s \le e^{-\varepsilon \cdot s/2}$
- P_{10}=P [no early 1 nor late 0 in *s* samples] $\le P_1 + P_0 \le 2e^{-\varepsilon \cdot s/2}$
- And since $s = 4/\varepsilon$ we get $P_{10} \le 1/3$
- It is interesting to note that the query complexity (the number of sampled bits) is O($1/\varepsilon$) which is independent of *n*.

Property Testing

- Recall that a language is a class of finite (e.g., strings, graphs) objects.
- A language is can be sometimes called a (class of objects possessing some) *property*.
- The research area that studies the notion of approximation for decision problems (such as language membership) is called *Property Testing*.

Notion of the distance

- As we saw in the *0*1** example, our notion of approximation for decision problems calls for a distance function on the inputs $\delta(x,y) \in [0,1]$. The distance function depends on the problem at hand.
- Examples of distance functions that have been considered in the literature include:
- ***Relative Hamming distance*** - the fraction of bits/characters/matrix entries on which *x* and *y* differ.
- ***Relative Edit distance*** - the minimum number of character substitutions, inserts and deletes needed to transform *x* into *y* divided by the length of *x*

Notion of the distance

- Most of the applications studied in the field require use of the Hamming distance.
- A distance from an input *x* to the property *P* is defined as the distance between *x* and the input in *P* closest to *x*:

$$\delta(x,P) = \min_{y \in P} \delta(x,y)$$

- We say an input *x* is ε-far from *P* if $\delta(x,P) \ge \varepsilon$

Input representation

- An important issue in defining the distance between inputs is the input representation.
- For example, a graph can be represented by an adjacency matrix or adjacency lists for all its vertices.
- The distance, as we defined it, depends on the representation.
- Also, representation defines what an algorithm can access in one step. We will usually work in the random access model, where in a single step the algorithm can access one bit/character/matrix entry or an entry in an adjacency list.

The concept of ε-tester

Definition: (ε -tester)

- An algorithm A is an ε-tester for property P if it
  1) accepts all $x \in P$ with probability at least $2/3$, and
  2) rejects all x that are ε-far from P with prob. at least $2/3$

  where the probability is taken over the internal coin tosses of the algorithm A.
- We can reduce the error to any constant Δ by repeating the algorithm $O(\log 1/\Delta)$ and taking the majority/maximal answer.

Testing sorted list

- How can we perform the ε-test of an input list to check whether its element are sorted, i.e., where
- ***Input:*** a list $X=(x_1,.., x_n)$ of arbitrary numbers.
- ***Solution:*** an ε-tester for the list X with the running time $O(\log n / \varepsilon)$

The ε-tester for almost sorted lists

1) Pick $s = \theta(1/\varepsilon)$ random numbers x_i (i.e., pick the indices i uniformly at random).
2) Do a binary search for each x_i, and reject it if you find any numbers out of order.
3) Accept it if for all the binary searches no numbers were out of order.

- ***Theorem:*** The ε-tester recognizes the ε-far (unsorted) lists with probability $\geq 2/3$.