Randomized Algorithms - Monte Carlo and Las Vegas Algorithms

An algorithm as we know always produces the right result in finite number of steps. Usually, one strives to design efficient algorithms. The effort in algorithm design is always ensuring that it always produces the correct result in an efficient way, or in other words, an optimal solution.

There are cases where we are interested in designing algorithms that may not be efficient, or may not produce the same result every time, for the same inputs. Consider a very simple case of using linear search to look for a key in the list of integers:

10, 20, 30, 10, 50, 60, 10, 70, 80, 90, 100

If you are looking for value 10, a randomized algorithm may return any one of the positions where it is found in the array (shown in color, below)

10, 20, 30, **10**, 50, 60, **10**, 70, 80, 90, 100

Such randomized results are good especially for security purposes where you may not want the adversary to guess the amount of work involved in producing a result. For instance, in the linear search problem, a malicious adversary may give inputs that *always* compel your (non-randomized) algorithm to search through *all* elements. If, on the other hand, your algorithm is randomized, it will search using random indices, and therefore, will have a chance of arriving at the result more quickly, despite your adversary's malicious intent.

In other cases, when we do not know of efficient algorithms for a problem, randomized algorithms have a chance of producing correct results. This is more attractive than always running an inefficient program! Thus, the advantages of a randomized algorithm are:

- The algorithm is usually simple and easy to implement,
- The algorithm is fast with very high probability, and/or
- It has a potential to produce optimum output with very high probability

What are the disadvantages and difficulties?

- There is a finite probability of producing incorrect answer. However, the probability of getting a wrong answer can be made arbitrarily small by the repeated employment of randomness.

- Analysis of running time or probability of getting a correct answer is usually difficult.

- Because generating true randomness is impossible, one needs to depend on pseudo randomness. So, the result highly depends on the quality of the random numbers that are conceptually used to drive the algorithm.

A randomized algorithm which runs in a fixed amount of time (or steps) but only gives the right answer some of the time, is called a **Monte Carlo** algorithm. This technique may produce wrong results. But the probability of producing wrong result may be made appropriately small and very negligible.

A **Las Vegas** algorithm always produces the correct answer its running time is a random variable whose expectation is bounded, say, by a polynomial. Thus, a Las Vegas algorithm is a randomized algorithm that always produces a correct result, ***or simply doesn't find one***, yet it *cannot not guarantee a time constraint*; the time complexity varies on the input. It does, however, guarantee an upper bound in the worst-case scenario.

Looking up a "good book on drawing" in a huge book shop is a good example for Las Vegas algorithm. You know what you are looking for, and yet, you don't know how long it will take for you to choose a "good book on drawing". In the worst case, you may get tired after skimming through tens of books and abandon the effort completely. Thus, when you end up buying, you produce the right result; in the worst case, you may not produce any result. In any case, you will never produce a wrong result.

Las Vegas Algorithm for Searching a List Containing Repeated Integers

Assume that we have to search for a key in an array (or list) of integers. Also assume that the elements in the list repeat. So, there may be more than one index (location in the array) where a particular key value may be present.

```
LASVEGAS_SEARCH(L, key)
# let N be the size of the list L.
# or let N be the number of elements in the list L.
   1. do
   2.     i ← RANDOM(N)
   3.     if (L[i] == key) return i
   4. while (true)
```

Note that the worst case running time is not bound. The do-while loop may go on forever. However, since numbers in the array are repeating and the distribution of that repetition is random, it is possible that the random index chosen in step 2 (in the algorithm above) actually chooses the key in a couple of comparisons.

This is a Las Vegas algorithm because it does not give up correctness, but gamples with time. There is no worst-case upper bound here in this example. In the best case, the key may be found in O(1) time.

Monte Carlo Algorithm for Finding MinCut of a Graph

You may recall that the MaxFlow problem is also a different way of stating the mincut problem.

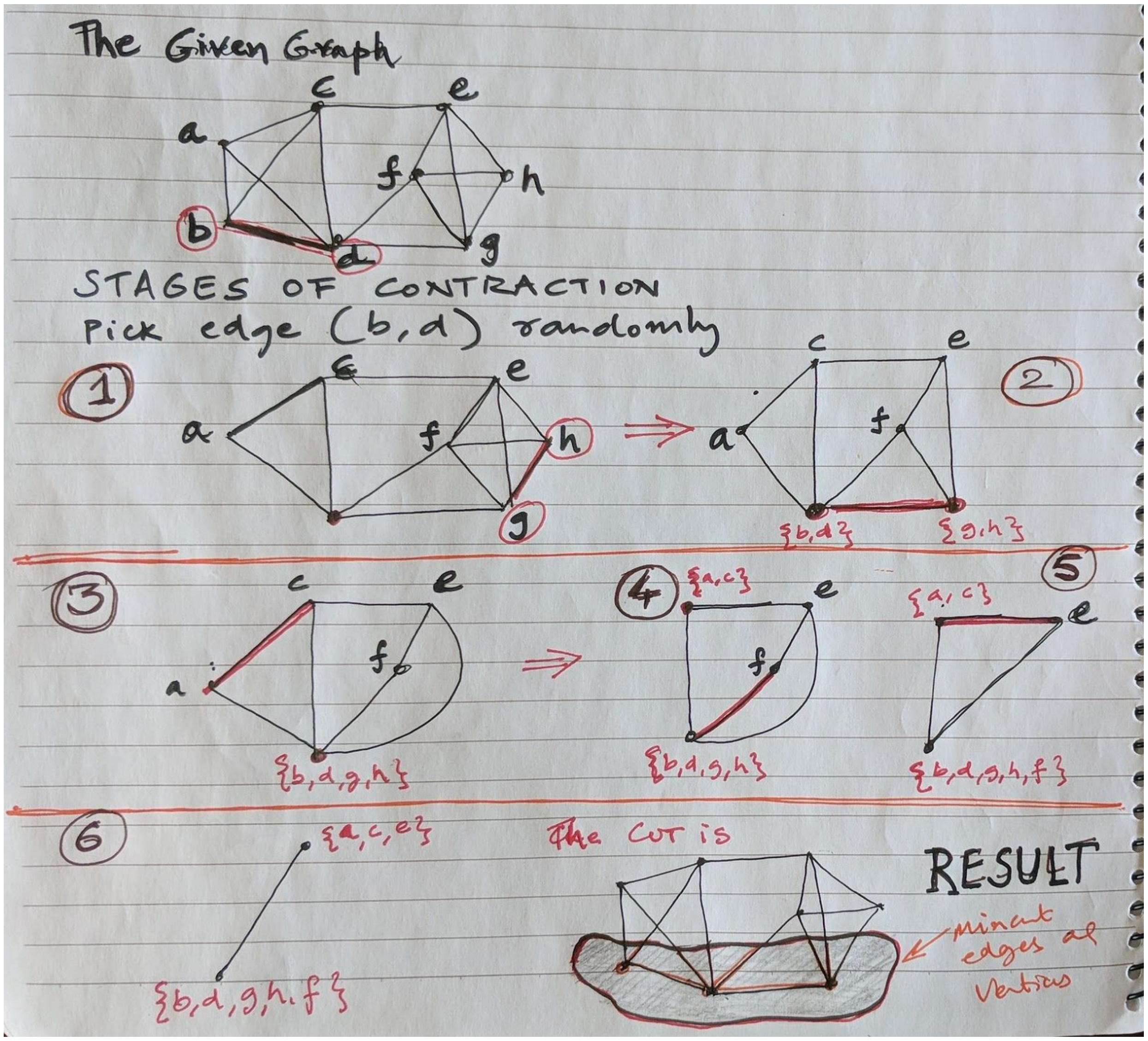
In graph theory, a minimum cut of a graph is a cut (or a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge) that is minimal in some measure. For our discussion, a mincut is a cut with the minimum number of edges among all cuts in an undirected graph, determining the edge connectivity of the graph.

Karger's algorithm provides an efficient randomized method for finding this cut. This can also be viewed as a Monte Carlo algorithm, as described below.

Karger's algorithm repeatedly selects an edge at random, and "contracts" the graph by merging the two vertices at the end of the selected edge. Here is an informal definition of the operation, followed by a diagram that shows how contraction and random edge selection are carried out, and how the progression of the algorithm.

Contraction of an Edge

Give a graph G = (V, E), **Contraction** of an edge e = (**x**, **y**) implies merging two vertices **x** and **y**, into a single vertex and removing the self loops, if any. The node obtained by merging two vertices is sometimes called the *supernode*. The resulting contracted graph is denoted by **G/xy**



The diagram (shown above) illustrates (as an example) how contractions of the edges in successive steps yields a mincut of the graph. It is important to note that edges are chosen at random, and the chosen edge is removed from consideration in the next step. Therefore, there is always an upper-bound in this algorithm, which is the number of edges in the graph. The result of the algorithm is not guaranteed to be minimal. It will be a correct solution, though the number of edges in the final mincut is actually not optimal.

The description of the algorithm follows:

```
MINCUT(G)
   1. i = 0
   2. $G_0 \leftarrow G_i$
   3. while $G_i$ has more than two vertices
   4. do
   5.    pick randomly an edge ei from the set of edges E of $G_i$
   6.    $G_{i+1} \leftarrow G_i/e_i$
   7.    i ← i + 1
   8. done
   9. supernodes at the end of the remaining edge represent the mincut.
```

Determining Polynomial Identity

Consider the problem of verifying the following identity

`(x + 1) (x - 2) (x + 3) (x - 4) (x + 5) (x - 6) == x^6 - 7x^3 + 25`

In simple cases such as these we may manually evaluate the expressions on
both sides of the equation, and conclude that they are not identical.

We can see, however, that in a general setting, it may be arbitrarily hard to decide the validity of an identity. More generally, given two polynomials `F(x)` and `G(x)`, we can verify the identity ***F(x) == G(x)*** by converting them to a standard form (also called the canonical form)

$$\Sigma(i = 0 \text{ to } d)\ (c_i * x^i)$$

(That is, two polynomials are equal if all coefficients in their normal forms are equal)

However, verting an arbitrary polynomial to a canonical form like the one shown in the first example in the beginning, is computationally complex. We could instead utilize randomness to obtain a faster method to verify polynomial identity.

Monte Carlo Method for Computing Polynomial Identity

Assume that the largest exponent of x (or the largest degree), in F(x) and G(x) is d. The algorithm chooses an integer r uniformly at random in the range {1,..., 100d}. By "uniformly at random", it is meant that all integers are equally likely to be chosen. The algorithm then computes the value `F(r)` and `G(r)`. If `F(r) != G(r)` the algorithm decides that the two polynomials are not equivalent. And of `F(r) == G(r)`, then the algorithm would have proved their equivalence.

Suppose that it takes one computational step to generate an integer chosen uniformly at random in the range `{1,...,100d}`. Assume also that multiplication and addition can be performed in one computation step. Then the multiplication of d terms in `F(x)` and `G(x)` can be performed in O(d) time, which is way faster than computing the canonical forms of `F(x)` and `G(x)`.

You can see that this is a Monte Carlo method because it runs for a fixed amount of time and may produce correct result. It may also produce wrong result, when a number r in the range {1,...100d} happens to be a root of both F(x) and G(x) in which case F(r) and G(r) evaluate to zero and appear to be equal.

What is the chance of producing the wrong result?
We all know that a polynomial of degree **`d`** has not more than **`d`** roots. Then, there are no more than `d` values is `{1,..., 100d}` for which `F(r) == G(r)`. Since there are `100d` values in the range `{1,... 100d}` the chance that the algorithm will randomly pick such a value is no more than `1/100`.