Randomised Algorithms II

1 Randomised algorithms

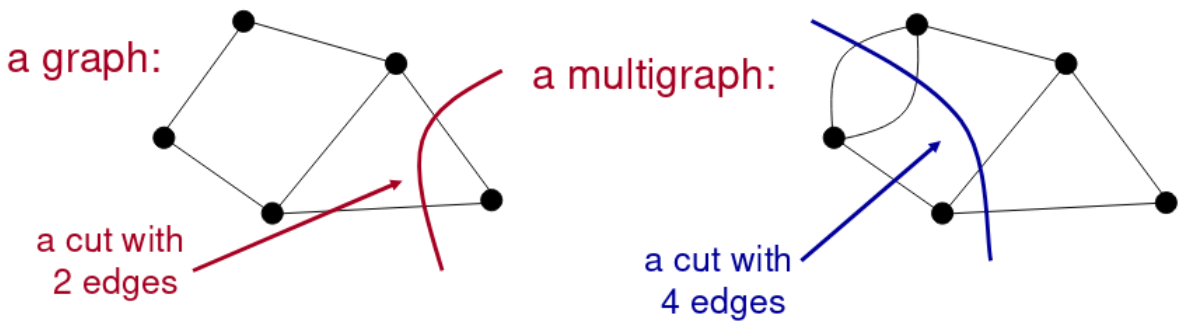
- The behaviour of a randomised algorithm is influenced by random decisions
- If it runs many times with the same input it has different outputs/running times
- Why randomised algorithms?
  - they are usually simpler to devise
  - and run faster than deterministic ones
- Las Vegas algorithms:
  - Always find the correct solution
  - The running time varies for every execution
- Monte Carlo algorithms:
  - Not always the correct solution
  - But we can bound the probability of an incorrect solution
- Both very useful

2 Monte Carlo algorithms

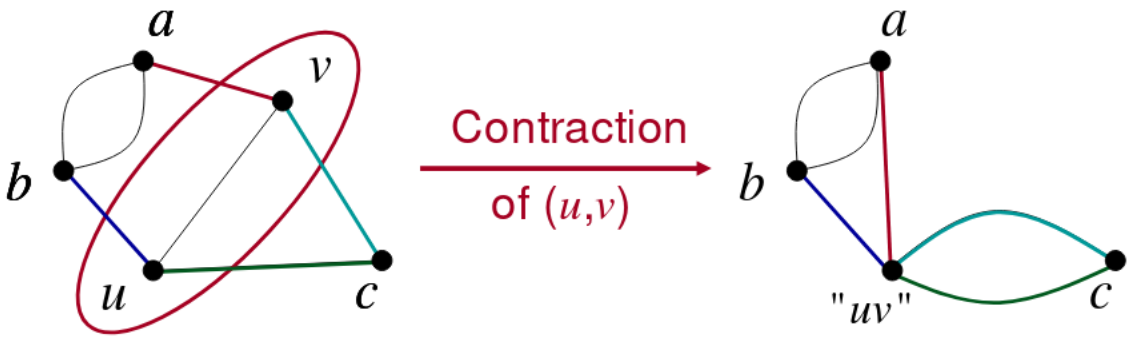
- For decision problems (answer is YES/NO)
  - algorithms with two sided error - any answer can be wrong
- Algorithms with one sided error
  - a YES answer is always correct
  - a NO answer may be false
- Examples:
  - ”check Bob’s claim that two coins are different”
  - ”graph non-isomorphism problem”
- What about optimisation problems?
  - here the answer is a number
  - not a YES/NO number

3 The min-cut problem

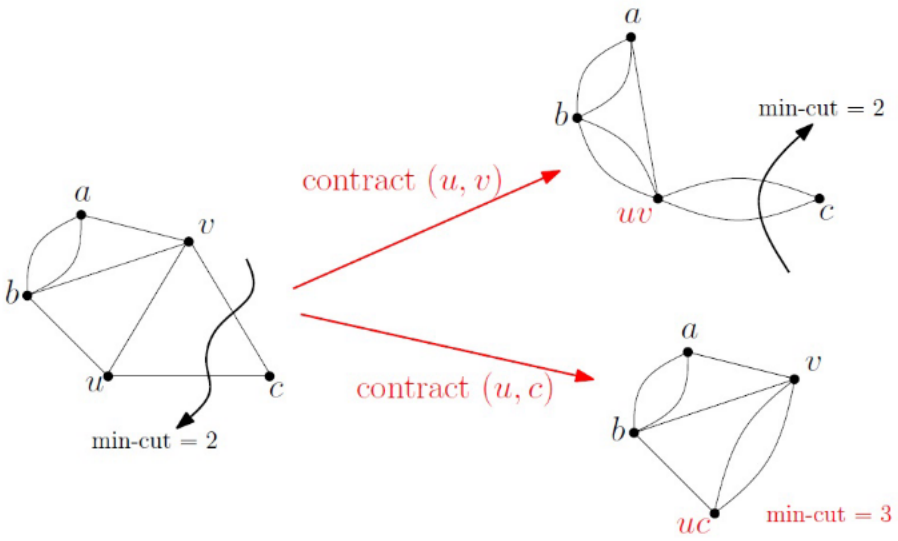
- In a graph two vertices can have 0 or 1 edge between them
- In a multigraph two vertices can be connected by more edges



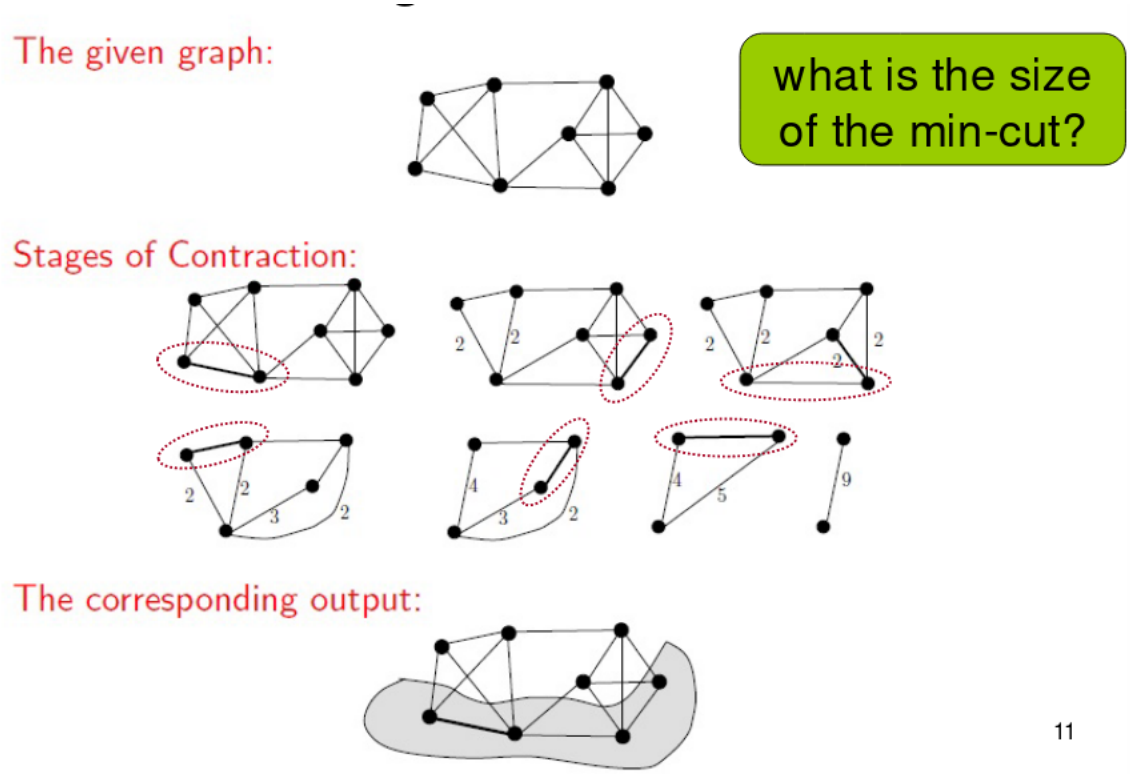
- A cut in a (multi)graph is a set of edges whose removal disconnects it
- The min-cut problem: given a multigraph, find a cut with the fewest edges
- There exist efficient deterministic algorithms using ”network flows”
- Lets try randomisation - a really simple algorithm
- First, an easy observation:
  - Let S be a min-cut of a multigraph
  - If S has an edge between vertices a and b then S has all edges between a and b
  - Why? Since otherwise there exists a smaller cut
- Contraction of an edge $e = (u, v)$:
  - merge u and v into a single new vertex ”uv”
  - all edges between u and v disappear
  - any other edge (w, u) or (w, v) of the old vertices u, v becomes an edge $(w, "uv)$ of the new vertex ”uv”



- A simple Monte Carol algorithm:
  - pick randomly an edge and contract it
  - iterate until only two vertices remain
  - return the number of edges between these two vertices
- At every step of the algorithm
  - decrease the number of vertices by one
  - $\Rightarrow$ n-2 iterations for a graph with n vertices
- Does this algorithm **always** find a min-cut? Not always
- An edge contraction
  - does not reduce the size of a min cut
  - sometimes it may increase it
- Why?
  - every cut of the graph after the contraction was also a cut before the contraction, not the inverse



4 The algorithm in action



5 How good is the solution

- The input graph has n vertices
- suppose that the min-cut S has k edges

Lemma: the graph has at least $kn/2$ edges
Proof:

- If there is a vertex v with less than k neighbours then there is a cut with less than k edges
- This is a contradiction, since the min-cut has k edges and all the vertices of the graph have at least k neighbours
- Now count the neighbours of all vertices and the graph has at least $kn/2$ edges

Thus:

- If we pick randomly an edge in the input graph, this edge will belong to the min cut S with probability at most $\frac{k}{nk/2} = \frac{2}{n}$
- This random edge will not belong to the min cut S with probability at least $1 - \frac{2}{n}$

After i iterations (i.e. i contractions)

- the current graph has n-i vertices
- suppose that we did not contract yet any edge of the min cut S

Then, similarly:

- If we pick randomly an edge in the current graph, this edge will belong to the min cut S with probability at most $\frac{k}{(n-i+1)k/2} = \frac{2}{(n-i+1)}$
- This random edge will not belong with the min cut S with probability at least $1 - \frac{2}{(n-i+1)}$

Summarising:

- During all (n-2) iterations of the algorithm, we did not contract any edge of the min cut S with probability at least

$$\left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n-2}\right) \cdot \left(1 - \frac{2}{n-3}\right) \cdots \left(1 - \frac{2}{4}\right) \cdot \left(1 - \frac{2}{3}\right)$$
$$= \left(\frac{n-2}{n}\right) \cdot \left(\frac{m-3}{n-1}\right) \cdot \left(\frac{n-4}{n-2}\right) \cdot \left(\frac{n-5}{n-3}\right) \cdots \left(\frac{2}{4}\right) \cdot \left(\frac{1}{3}\right)$$
$$= \frac{2}{n(n-1)} \geq \frac{2}{n^2}$$

That is:

- At the end of the algorithm, the min-cut S is found with probability at least $\frac{2}{n^2}$
- The min cut S is not found with probability at most:

$$1 - \frac{2}{n^2}$$

- We run this algorithm x times
- keep the smallest cut that we found
- The probability that after x times we do not detect the min cut is at most $\left(1 - \frac{2}{n^2}\right)^x$ (too small)

6 The sorting problem

The sorting problem:

- Given a set of n numbers, sort them in increasing order
- Many algorithms, the most widely used: quick-sort
  - Choose a pivot $y \in S$
  - Partition the elements of $S - y$ into two subsets S_1, S_2
    - every element of S_1 is smaller than y
    - every element of S_2 is larger than y
  - recursively sort the sets S_1 and S_2
  - output:

    * Sorted S_1, followed by y, followed by sorted S_2

The main idea of quick-sort: divide and conquer

- Try to find an element y that splits $S - y$ into two sets S_1 and S_2 of the same size
- We need to solve two instances of the half size
- If we choose the worst pivot
  - i.e.always too large/too small pivot
  - We need $O(n^2)$ time for a set S with n numbers (all possible comparisons)
- If we always choose the best pivot:
  - i.e. always a pivot in the "middle"
  - We need $O(n \log n)$ time
- We get the same running time $O(n \log n)$
  - even if S_1 and S_2 have almost the same size
- This gives us hope
  - In every set S of n elements, there exist $n/2$ elements y that split $S - y$ into two almost equal parts
- In other words:
  - If we randomly pick one element y, it will be a "good pivot" with probability $\frac{1}{2}$
- Randomised quick sort
  - exactly as the standard quick sort
  - but randomly choose the pivot at every step

7 Randomised Quick-Sort

- Always the correct answer (a sorted sequence)
  - no matter which pivot we chose at every step
  - our choices influence only the running time
  - a Las Vegas algorithm
- The running time is a random variable: the number of comparisons
- For every $1 \leqslant i, j \leqslant n$ define the random variable:
  - $X_{i,j}$ equals 1 if the ith and jth largest elements of S are being compared at some iteration
  - $X_{i,j}$ equals 0, otherwise
- The expected running time is $E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}\right]$
- By linearity of expectation, this equals:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} E\left[X_{i,j}\right]$$

- We can prove that $E\left[X_{i,j}\right] = \frac{2}{j-i+1}$
- Using that, we can prove that the expected running time is $O(n \log n)$
- Random Quick-Sort is expected to perform as a Quick Sort that always chooses a "good" pivot