# ADVANCED ALGORITHMS THEORY SWISS KNIFE

GABRIEL ROVESTI

# 1 Table of Contents

**Disclaimer**

This file contains basically a refined version of full notes (and of my full theory file) to summarize the theory content and make it clearly visible if possible in a proper way, concise but understandable. This respects and follows, both chronologically and logically, the topics seen in 2023/2024.

# 2 Graph – General Definitions

- $G = (V, E)$ as the graph itself
  - $V$ = set of vertices (aka nodes)
  - $E \subseteq V\ x\ V$ (cartesian product = all) is a collection of edges
    - an edge is a pair of vertices $(u, v)$
      - it indicates the connection between two nodes
      - a connection of vertices allows for repetition
- directed graphs, which happens if $(u, v) \neq (v, u)$
- undirected graphs, which happens if $(u, v) = (v, u)$
- arc = edge inside directed graphs (also called *directed edges*)
- given an edge $e = (u, v)$
  - $e$ is incident on $u$ and $v$ (happens if vertex if one of endpoints in that edge)
  - $u$ and $v$ are adjacent (there is an edge between the two vertices)
- neighbors of a vertex: all vertices $v$ s.t. $(u, v) \in E$
  - all vertices directly connected to a given vertex by an edge
- degree of a vertex $v$, denoted as $d(v)$ or $degree(v)$
  - the number of edges incident on $v$
- path: $u_1, u_2 \dots u_k$ and $(u_i, u_{i+1}) \in E, \forall\ 1 \leq i \leq k$
  - finite/infinite sequence of nodes which joins a sequence of vertices via edges
- simple path: $u_i$ (all vertices) are all distinct
  - same definition as above and vertices/nodes are all distinct/so are the edges
  - e.g., 5,1,8,7,6,1,4 has 1 repeated twice so it's not simple
- cycle: simple path s.t. $u_1 = u_k$ (starts from a given vertex/ends at same node)
- subgraph: $G' = (V', E')\ s.t.$
  - $V' \subseteq V$
  - $E' \subseteq E$
  - the edges of $E'$ are incident only on vertices of $V'$
  - in words: it is a subset of the larger original graph
- spanning subgraph: a subgraph with $V' = V$
  - a subgraph which "spans" the original graph (so there are all the vertices)
  - following other definitions
    - subgraph obtained by edge deletions only but retaining all vertices
    - so it's a subgraph of $G$ with same vertex set as $G$
- connected graph: if $\forall u, v \in V, \exists$ a path from $u$ to $v$
- connected components: a partition of $G$ in subgraphs $G_i = (V_i, E_i), \forall\ 1 \leq i \leq k\ s.t.$
  - $G_i$ is connected $\forall i$
  - $V = V_1 \cup V_2 \cup \dots \cup V_k$
  - $E = E_1 \cup E_2 \cup \dots \cup E_k$
  - $\forall i \neq j$ there is no edge between $V_i$ and $V_j$
- tree: connected graph without cycles
  - any two vertices are connected by *exactly* one path
- forest: set of trees (disjoint)
  - also = undirected graph in which any two vertices are connected by *at most* one path

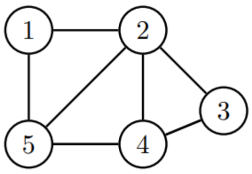
- spanning tree: a spanning subgraph connected and without cycles
- spanning forest: a spanning subgraph without cycles

Generally, remember:

- $n = |V|$ (number of nodes)
- $m = |E|$ (number of edges)
- the size of a graph is $n + m$

There are also multiple ways of representing:

- an adjacency list
  - an array $A$ of $n$ lists, one $\forall$ vertex $v \in V$ (consider the example below)
  - each containing all the vertices adjacent to $v$ (represented by table below)



| | |
|---|---|
| 1 | 2,5 |
| 2 | 1,3,4,5 |
| 3 | 2,4 |
| 4 | 2,5,3 |
| 5 | 4,1,2 |

What if directed? Only vertices pointed for that vertex.

- Pro: space usage $\theta(n + m)$ i.e. linear
- Con: no quick way to determine if a given edge is in the graph

- an adjacency matrix
  - a $n \; x \; n$ matrix $A$ s.t. $A[i, j] = 1$ if $edge(i, j) \in E$, 0 otherwise

$$A = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \cancel{0} & 1 & 0 & 0 & 1 \\ 2 & 1 & \cancel{0} & 1 & 1 & 1 \\ 3 & 0 & 1 & \cancel{0} & 1 & 0 \\ 4 & 0 & 1 & 1 & \cancel{0} & 1 \\ 5 & 1 & 1 & 0 & 1 & \cancel{0} \end{array} \rightarrow \text{symmetric}$$

- If graph is directed → the matrix is *asymmetric*
- If graph is undirected → the matrix is *symmetric*

In case of a *weighted graph*, each cell of the matrix has either the value of the edge weight (as number) $w$ or $-/null$ to represent null costs. This kind of graph represents costs, capacities, etc.

- Pro: Quick to determine if a given edge is present
- Con: Space required is $\theta(n^2)$ → can be superlinear in the input size
  - if number of vertices increases, the space required by matrix grows quadratically

# 3 Depth First Search - DFS

## 3.1 Description

The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It may find:

- new edges (discovery edge)
- non-tree edges, linked to ancestors (back edges)

Visit a vertex, then a neighbor of the vertex, then a neighbor of the neighbor – these are all neighbours, classified with adjacency lists.

## 3.2 Algorithm

```
procedure $DFS(G,V)$
    $visit\ v$
    $L_V[v].ID = 1$
    for all $e \in G.incidentEdges(v)$: do
        if $L_E(e).label = null$ then
            $w = G.opposite(v,e)$
            if $L_V[w].ID = 0$ then
                $L_E[e].label = DISCOVERY\ EDGE$
                $DFS(G,w)$
        else
            $L_E[e].label = BACK\ EDGE$
```

## 3.3 Complexity

Given:

- $n_s$: number of vertices of $C_s$ (one invocation $\forall v \in C_s$)
- $m_s$: number of edges of $C_s$ (costs related to node, excluding recursive invocations inside)

The complexity overall is:

$$\theta\left(\sum_{v \in C_s} d(v)\right) = \theta(m_s)$$

More in general: $O(n+m)$ – $n$ vertices and $m$ edges

## 3.4 APPLICATIONS

There are several:

- $s-t$ path (between two generic vertices)
    - done adding a $parent$ field
- finding cycles
    - use $parent$ field on vertices and $ancestor$ on edges
- find connected components
    - run the algorithm $n$ times
    - consider all untouched vertices
    - see which have back edges, meaning they "close" the cycle
    - otherwise, return
- find a spanning tree

# 4 Breadth First Search - BFS

## 4.1 Description

The algorithm is iterative, starts from a source vertex and visits all vertices connected to a specific component, partitioning them in levels according to their distance. It still has discovery edges:

- but adds cross edges – which connect vertices at different levels

## 4.2 Algorithm

```
procedure $BFS(G, s)$
    $visit(s)$
    $L_V[s].ID = 1$
    $Create\ a\ set\ L_0\ containing\ s$
    $i = 0$
    while $(!L_i.isEmpty)$ do:
        $Create\ a\ set\ of vertices\ L_{i+1}$
        for each $v \in L_i$ do:
            for each $e \in G.incidentEdges(v)$ do:
            if $L_E[e].label = null$ then
                $w = G.opposite(v, e)$
                if $L_v[w].ID = 0$ then
                    $L_E[e].label = DISCOVERY\ EDGE$
                    $visit\ w$
                    $L_V[w].ID = 1$
                    $add\ w\ in\ L_{i+1}$
                else
                    $L_E[e].label = CROSS\ EDGE$
    $i = i + 1$
```

## 4.3 Complexity

$$O(n + m)$$

## 4.4 APPLICATIONS

- Same as for DFS in $\theta(n + m)$ time

So, again:

- $s - t$ path (between two generic vertices)
    - done adding a $parent$ field
- finding cycles
    - use $parent$ field on vertices and $ancestor$ on edges
- find connected components
    - run the algorithm $n$ times
    - consider all untouched vertices
    - see which have back edges, meaning they “close” the cycle
    - otherwise, return
- find a spanning tree

# 5 MINIMUM SPANNING TREE – MST

- Input: a graph $G = (V, E)$ undirected, connected and *weighted*
  - A weight $w: E \to \mathbb{R}$
  - defines $w(u, v)$ = cost of edge $(u, v)$
- Output: a spanning tree $T \subseteq E$ of $G$ s.t. $w(t) = \sum_{u,v \in T} w(u, v)$ is minimized
  - Goal is minimizing the sum for all weights of every edge of the tree

## 5.1 GENERIC GREEDY ALGORITHM

```
procedure $Generic - MST(G)$
    $A = \emptyset$
    while $A\ does\ not\ form\ a\ spanning\ tree$ do:
        $find\ an\ edge\ (u, v)\ that\ is\ safe\ for\ A$   $//\ crucial\ step$
        $A = A \cup \{(u, v)\}$          $//\ add\ vertices\ to\ A$
    return $A$          $//\ A\ is\ an\ MST$
```

## 5.2 DEFINITIONS

- A cut of graph $G = (V, E)$ is a partition of $V \to (S, V \setminus S)$
  - in words, a partition of vertices into two disjoint subsets
  - it can be done on one or more edges
- An edge $(u, v) \in E$ crosses a cut $(S, V \setminus S)$ if $v \in S$ and $v \in V \setminus S$ (or viceversa)
  - so, if its endpoints lie in different subsets of the partition defined by the cut
- A cut respects a set of edges $A$ if no edge of $A$ *crosses* the cut
- Given a cut, an edge that crosses the cut and is of minimum weight is called light edge (for that cut) → they are useful, because when included in MSTs, they have minimum weight

There is also the *minimum cut*, for which we have $d(v) \geq t\ \forall v \in V$, where $t$ is a generic size of graph. Summing up all $n$ vertices, we obtain $\sum_{v \in V} d(v) \geq tn$, concluding it's $\sum_{v \in V} d(v) = 2m$.

# 6 PRIM'S ALGORITHM

## 6.1 DESCRIPTION

The algorithm is iterative and selects light edges at every step, growing a spanning tree from there. Consider this gif to see the running. We have to preserve "safe edge" property – take only minimal edges not already inside of the tree.

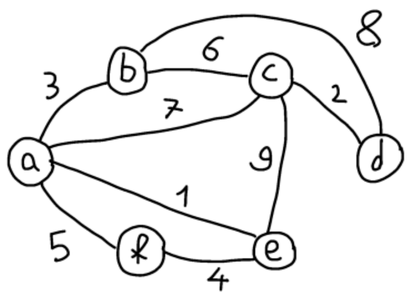
## 6.2 ALGORITHM

```
procedure $Prim(G, S)$
    $X = \{S\}$
    $A = \emptyset$
    while $there\ is\ an\ edge\ (u, v)\ with\ u \in X\ and\ v \notin X$ do:
        $(u^*, v^*) = a\ minimum\ weight\ such\ edge\ (aka\ light\ edge)$
        $add\ vertex\ v^*\ to\ X$
        $add\ edge\ (u^*, v^*)\ to\ A$
    return $A$
```

## 6.3 COMPLEXITY

$O(m * n)$

## 6.4 EXAMPLE OF EXECUTION FOR EXAM



Traversal: $(a, e), (a, b), (e, f), (b, c), (c, d)$

# 7 EFFICIENT PRIM – HEAP IMPLEMENTATION

## 7.1 DESCRIPTION

The previous is not so efficient in large structures. The right kind of data structure to improve the algorithm is a *priority queue*, implemented with a heap.

- Recap about this data structure
  - $insert$ → add an object to the heap (possibly fast)
  - $extractMin$ → remove an object with the smallest key (highest priority)
  - $delete$ → given a pointer to an object, remove it
- In a heap with $n$ objects, the complexity of these operations is $O(log(n))$

We can redefine the algorithm exploiting this efficient data structure basically with the same principle:

- consider a min heap starting from whatever vertex, which is the root
- from there, always extract the minimum value (means checking if it is min heap),
  - then update the path

## 7.2 ALGORITHM

```
procedure $Prim\ (G, s)$
    for each $v \in V$: do
        $key[u] = +\infty$
        $\pi(v) = NULL\ \ //\pi\ = parent\ of\ v\ in\ the\ tree\ being\ built$
$Key[s] = 0$
$H = V$
while $H \neq 0$ do:
    $v^* = extractMin(H)$
    for each $v\ adjacent\ to\ v^*$: do
      if $v \in H\ and\ w(v^*, v) < Key(v)$ then
        $\pi(v) = v^*$
        $delete\ v\ from\ H$
        $Key(v) = w(v^*, v)$
        $insert\ v\ into\ H$
```

## 7.3 COMPLEXITY

- $init$ → $O(n)$
- $while$ → $n$ iterations
- $extractMin$ → $O(\log(n))$

Total cost of only $extractMin$ operations: $O(n\log(n))$

- *for loop*: executed $O(m)$ times in total (every vertex is explored)
    - $v \in H \rightarrow O(1)$
        - this here is a simple check
    - $Key(v) = w(v^*, v) \rightarrow delete + insert$: $O(2\log(n)) = O(\log(n))$
        - two operations

Total cost of *for loop*: $O(m\log(n))$ (iterating for all adjacent nodes, quantity equal to node degree)

This way, the total complexity of the algorithm is $O(n\log(n) + m\log(n)) = O(m\log(n))$ (since $G$ is connected, we recall) → near-linear time.

# 8 KRUSKAL'S ALGORITHM

## 8.1 DESCRIPTION

It picks the minimum weighted edge at first and the maximum weighted edge at last. It sorts edges by weight and then adds them continuously, preserving the "safe edge" property – take only the unexplored. It does so preventing the adding of cycles.

## 8.2 ALGORITHM

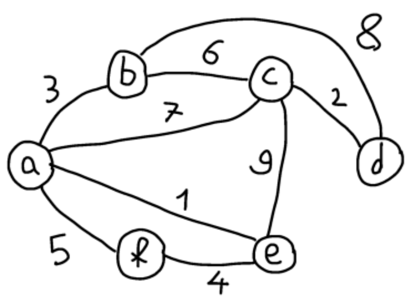
```
procedure $Kruskal(G)$
    $A = \emptyset$
    $Sort\ safe\ edges\ of\ G\ by\ weight$
    for each $edge\ e\ in\ non-decreasing\ order\ of\ weight$: do
        if $A \cup \{e\}\ is\ acyclic$ then:
            $A = A \cup \{e\}$
    return $A$
```

## 8.3 COMPLEXITY

- sorting: $O(m \log(n))$
- for loop: check whether $e = (u, v)$ closes a cycle is equivalent to check whether $A$ contains an $u - v$ path → DFS on $G = (V, A)$ → complexity: $O(n)$

Total: $O(m * n)$ → $O(m \log(n)) + O(m * n) = O(m * n)$

## 8.4 EXAMPLE OF EXECUTION FOR EXAM



Traversal: $(a, e), (c, d), (a, b), (e, f), (b, c)$

# 9 Efficient Kruskal – Union-Find

## 9.1 Description

It can be implemented as fast as Prim's, considering the most frequent operation here is cycle check (equivalently, path check), which happens when an edge is added to $A$.

We create a new data structure supporting this operation fast and to do that, we use a structure called Union-Find (also called *disjoint set*). This is a structure to merge *disjoint sets* (also non-overlapping in their elements) of objects and supports at least three operations:

- *Init*: given an array $X$ of objects
  - it creates a Union-Find data structure with each object $x \in X$ in its own set
- *Find*: given an object $x$, return the name of the set that contains $x$
  - depth: number of edges traversed by *Find*
- *Union*: given two objects $x, y$ merge the sets that contain $x$ and $y$ into a single set
  - done whenever the sets are distinct
  - if $x, y$ are already in the same set, this operation does nothing

## 9.2 Algorithm

```
procedure $Kruskal(G)$
    $A = \emptyset$
    $U = init(V)$
    $sort\ edges\ of\ E\ by\ weight$
    for each $edge\ e = (v, w)\ in\ non-decreasing\ order\ of\ weight$: do
    if $Find(v) \neq Find(w)$ then:
        $A = A \cup \{(v, w)\}$
        $Union(v, w)$
return $A$
```

## 9.3 Complexity

- Init: $O(n)$
- Sorting: $O(m \log(n))$
- $2m$ Find: $O(m \log(n))$
- $n - 1$ Union: $O(n \log(n))$ → only when I go inside an "if" and when the edge is added
- $A$ updating: $O(n)$

In total: $O(m \log(n))$

# 10 Shortest Path

- Given a weighted graph, the <u>length</u> of a path $p = v_1, v_2, \dots v_k$ is defined as $len(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$
- A <u>shortest path</u> from a vertex $u$ to a vertex $v$ is a path with minimum length among all $u - v$ paths
- The <u>distance</u> between $2$ vertices $s$ and $t$, denoted as $dist(s, t)$ is the length of a shortest path from $s$ to $t$; if there is no path at all from $s$ to $t$ then $dist(s, t) = +\infty$

The problem itself is the following:

- Given a directed, weighted graph and a source vertex $s \in V$ and a destination $t \in V$, compute the shortest path from $s$ to $v$

# 11 Single-Source Shortest Path (SSSP)

- input: a directed, weighted graph $G$ with edge weights $w: E \to \mathbb{R}$ and a source vertex $s \in V$
- output: $dist(s, v), \forall v \in V$
    - shortest path to all destinations

There are two major cases to solve: a special one and a more general one.

# 12 NON-NEGATIVE WEIGHTS – DIJKSTRA

## 12.1 DESCRIPTION

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices. It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.

- input: directed $G, s \in V, w: E \to \mathbb{R}_{\geq 0}$
- output: $dist(s, v) = len(v), \forall v \in V$
  - with $len(v)$ coming as shorthand form of the previous one

## 12.2 ALGORITHM

```
procedure $Dijkstra(G, s)$
    $X = \{s\}$
    $len(s) = 0$
    $len(v) = \infty$
    while $there\ is\ an\ edge\ (v, w)\ with\ v \in X\ and\ w \notin X$: do
        $(v^*, w^*) = such\ an\ edge\ minimizing\ len(v) + w(v, w)$
        $add\ w^*\ to\ X$
        $len(w^*) = len(v^*) + w(v^*, w^*)$
```

## 12.3 COMPLEXITY

$$O(m * n)$$

# 13 Efficient Dijkstra – Heap

## 13.1 Description

Normal implementation uses adjacency list. This implementation improves efficiency by using a priority queue (usually implemented as a binary heap) to select the vertex with the smallest tentative distance efficiently. The implementation is almost identical to Prim with heaps.

## 13.2 Algorithm

```
procedure $Dijkstra(G, s)$        (almost identical to Prim's implementation with heaps)
    $X = \{s\}$
    $H = \emptyset$
    $key(s) = 0$
    for each $v \neq s$: do
        $key(v) = \infty$
    for each $v \in V$: do
        $insert\ v\ into\ H$
    while $H\ is\ non-empty$: do
        $w^* = extractMin(H)$
        $add\ w^*\ to\ X$
        $len(w^*) = key(w^*)$
        for each $edge\ (w^*, y)\ s.t.\ y \notin X$: do
            $delete\ y\ from\ H$
            $key(y) = \min\{key(y), len(w^*) + w(v^*, w^*)\}$
            $insert\ y\ into\ H$
```

## 13.3 Complexity

- considering graph as adjacency list, $n$ vertices and $m$ edges
- $\log(n)$ iterations because of heap usage

Total number of operations: $O((n+m)\log(n)$ (there are $O(n+m)$ operations on heaps)

# 14 General Case: SSSP Problem

We reformulate the previous problem a bit:

- Input: a directed weighted graph $G = (V, E)$ and a source vertex $s \in V$
- Output: one of the following
  - $dist(s, v)$ ∀ vertex $v \in V$
  - a declaration that $G$ contains a negative cycle

Need to forbid negative cycles in shortest paths, they lead to infinitely small paths, which is an NP-Hard problem.

The main Dijkstra problems are two:

- It never <u>revisits/updates</u> its decisions, but it should for all vertices!
  - Once a vertex is marked as “closed”, we will never develop this node again
  - If we have a vertex in open such that its cost is minimal - by adding any positive number to any vertex - the minimality will never change
  - Without the constraint on positive numbers - the above assumption is not true
  - It assumes them to be positive to make the algorithm run faster and does this to avoid considering paths which can’t be shorter
- $len(v)$ should be an *estimated distance*, which needs to be updated for every vertex
  - how many times? $\leq n - 1$ edges $\Rightarrow n - 1$ times should be enough
  - maximum number of edges in a simple path between any two vertices

# 15 BELLMAN-FORD'S ALGORITHM

## 15.1 DESCRIPTION

- Input: A directed graph $G$ with edge weights $w: E \to \mathbb{R}$ and a source vertex $s \in V$
- Output: Either $dist(s, v)\ \forall v \in V$ or a declaration that $G$ contains a negative cycle

The algorithms is used when the graph might possess negative weights and can even detect negative cycles. If the graph contains one, there is no cheapest path, instead one can make it cheaper by one more walk around said negative cycle (in $n - 1$ iterations it reaches a fix-point, if it doesn't it means a negative cycle exists). Still, it's slower compared to Dijkstra.

## 15.2 ALGORITHM

```
procedure $Bellman - Ford\ (G, s)$
    $len(s) = 0$
    $len(v) = \infty\ \forall v \neq s$
    for $n - 1\ iterations$ do
        for each $edge\ (u, v) \in E$: do
            $len(v) = \min\{len(v), len(u) + w(u, v)\}$
        for each $edge\ (u, v) \in E$: do
            if $len(v) > len(u) + w(u, v)$ then
                return "$G\ contains\ a\ negative\ cycle$"
```

## 15.3 COMPLEXITY

$$O(m * n)$$

# 16 All-Pairs Shortest Paths (APSP)

- Input: A directed, weighted graph $G = (V, E)$
- Output: One of the following:
  - $dist(u, v)$ ∀ ordered vertex pair
  - a declaration that $G$ contains a negative cycle
    - this can be problematic in finding a shortest path
    - now we would have to output $n^2$ shortest paths

Consider:

- If we use Bellman-Ford - very high complexity → $O(m * n^2)$
  - Using dynamic programming, the complexity can be reduced to $O(n^3 \log(n))$
  - This holds rewriting B-F recurrence controlling the allowable size of the input

# 17 Floyd-Warshall's Algorithm

## 17.1 Description

It's used to find the shortest paths between all pairs of nodes in a weighted graph, with positive or negative edges.

- instead of restricting the number of edges allowed in a solution, restrict the identities of the vertices that are allowed in a solution
    - in other words, now paths can pass through only certain vertices
- Basically, it compares many possible paths through the graph between each pair of vertices

It iterates on 3 vertices: $u, v, k$ in 3 nested loops, testing whether using $k$ in the path is better.

## 17.2 Algorithm

```
procedure $Floyd - Warshall(G)$
    $label\ the\ vertices\ V = \{1,2,\dots,n\}\ arbitrarily$
    $A = n\ x\ n\ x\ (n+1)\ array$
    for $u = 1$ to $n$: do
        for $v = 1$ to $n$: do
            if $u = v$ then $A[u, v, 0] = 0$
            else if $(u, v) \in E$ then $A[u, v, 0] = w(u, v)$
            else $A[u, v, 0] = +\infty$
    for $k = 1$ to $n$: do
        for $u = 1$ to $n$: do
            for $v = 1$ to $n$: do
                $A[u, v, k] = \min\{A[u, v, k-1], A[u, k, k-1] + A[k, v, k-1]\}$
    for $u = 1$ to $n$: do
        if $A[u, u, n] < 0$ then return "$G\ contains\ a\ negative\ cycle$"
```

## 17.3 Complexity

$$O(n^3)$$

# 18 Maximum Flows

- a flow network is a directed graph $G = (V, E)$ where each edge has a capacity $c(e) \in \mathbb{R}^+$, along with a designated source $s \in V$ and sink $t \in V$
    - for convenience, write $c(e) = 0$ if $e \notin E$, no edges enter $s$ and no edges leave $t$

- a flow is a function $f: E \to \mathbb{R}^+$ satisfying the following constraints (how much stuff I send through the edges in general)
    - (*capacity*) $\forall e \in E, f(e) \leq c(e)$ – value of the flow at most capacity of that edge
    - (*conservation*) $\forall u \in V \setminus \{s, t\}$ we have
$$\sum_{u \in V \ s.t. (v,u) \in E} f(v, u) = \sum_{v \in V \ s.t. (u,v) \in E} f(u, v)$$
    - the amount of flow going in nodes must be equal to the flow going out from those (conservation of flows)
        - initially, such flow is 0, which is "how much we can pass on the edge"

- the value of a flow is
$$|f| = \sum_{v \in V \ s.t. (s,v) \in E} f(s, v)$$
    - basically, the sum of all flows going in and out vertices thanks to edges
    - as a matter of fact, the amount of stuff traveling from source to sink
    - such flow has to be less than or equal to the capacity

As for the problem itself:

- given a flow network, find a flow $f$ of maximum value. Such flow is measured on *the maximum value received in a sink node*

# 19 Ford-Fulkerson's Algorithm

## 19.1 Description

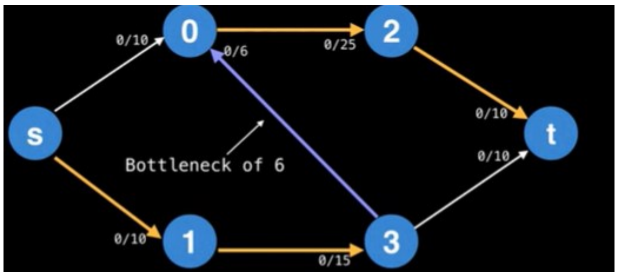
Given a flow network $G$ a flow $f$, the residual network of $G$ w.r.t (with respect to) flow $f$, $G_f$, is a network with vertex set $V$ and with edge set $E_r$ as follows:

- for every edge $e = (u, v)$ in $G$
  - if $f(e) < c(e)$, add $e$ to $G_f$ with capacity $C_f(e) = c(e) - f(e)$
  - if $f(e) > 0$, add another edge $(v, u)$ to $G_f$ with capacity $C_f(e) = f(e)$

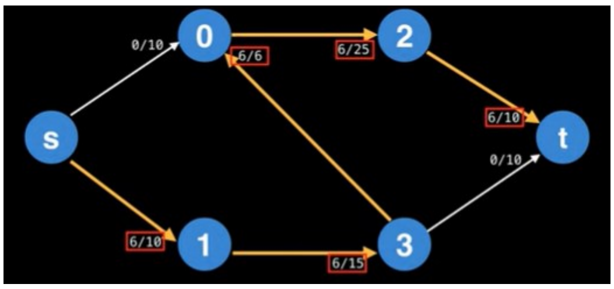
The Ford-Fulkerson (F-F) algorithm repeatedly finds an $s - t$ path $P$ in $G_f$ (e.g., using BFS) and uses $P$ to increase the current flow.

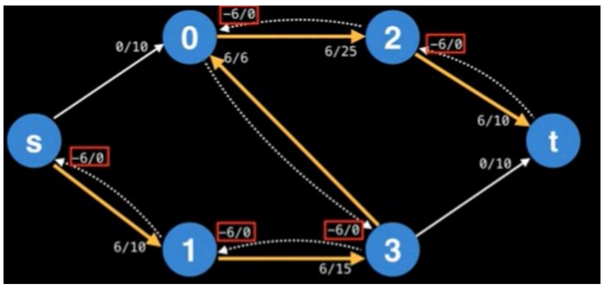
- $P$ is called augmenting path
  - This is a path of edges in the residual graph with unused capacity greater than 0 from the source $s$ to the sink $t$
  - This can only flow on edges not fully saturated yet
- In an augmenting path, the *bottleneck* is the smallest edge on the path. We can use this one to augment the flow along the path

In figure below, in orange the augmenting path, in light-blue as written the bottleneck:



- Augmenting the flow means updating the flow values along the augmenting path (left)
  - For forward edges, this means increasing the flow by the bottleneck value
- When augmenting the flow along the augmenting path
  - you also need to decrease the flow along each residual edge (backward edges) by the bottleneck value (right)
  - residual edges exist to "undo" bad augmenting paths which do not lead to a maximum flow





The residual graph, so, contains also residual edges. This algorithm continues to find augmenting paths and augments the flow until no more augmenting paths exist.

- The algorithm simply takes in every iteration the bottleneck
- Then considers the bottleneck and keeps incrementing selecting every possible $s - t$ path until max flow is reached
- Each iteration is then reported into the residual graph, accounting for the bottleneck
  - e.g. if we chose $7/11$ in an iteration, in the next iteration
    - 4 forward (remaining)
    - 7 backward (spent)

## 19.2 ALGORITHM

```
procedure $Floyd - Warshall(G)$
    $label\ the\ vertices\ V = \{1,2,\dots,n\}\ arbitrarily$
    $A = n\ x\ n\ x\ (n+1)\ array$
    for $u = 1$ to $n$: do
        for $v = 1$ to $n$: do
            if $u = v$ then $A[u,v,0] = 0$
            else if $(u,v) \in E$ then $A[u,v,0] = w(u,v)$
            else $A[u,v,0] = +\infty$
    for $k = 1$ to $n$: do
        for $u = 1$ to $n$: do
            for $v = 1$ to $n$: do
                $A[u,v,k] = \min\{A[u,v,k-1], A[u,k,k-1] + A[k,v,k-1]\}$
    for $u = 1$ to $n$: do
        if $A[u,u,n] < 0$ then return "$G\ contains\ a\ negative\ cycle$"
```

## 19.3 COMPLEXITY

- Assume capacities are integers; then
  - the flow value increases by $\geq 1$ is each iteration
  - the complexity of each iteration is $O(m)$

Total complexity is $O(m * |f^*|)$, where $f^*$ is a max flow

# 20 NP-HARDNESS

There are problems which can be solved in linear time:

- e.g. Eulerian circuit – a graph where edges are traversed all at once

There are problems which can be solved in polynomial time:

- e.g. Minimum Spanning Tree (MST), minimizing the weights inside all tree

There are also problems where no polynomial algorithms are present to solve the problem:

- e.g. Traveling Salesperson Problem (TSP), Hamiltonian Circuit

We define the following *complexity classes*:

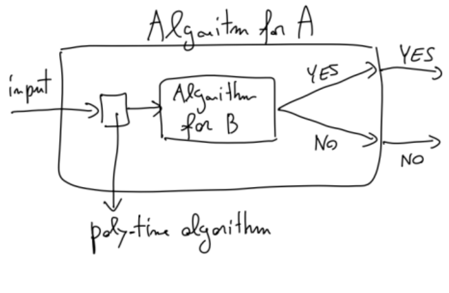
1) $P$ is the set of decision problems that can be solved in polynomial time
2) $NP$ is the set of decision problems with the following property:
    a. if the answer is YES, then there is a proof of this fact (called "certificate") that can be checked in polynomial time
3) $co - NP$, which is essentially the opposite of $NP$:
    b. property: if the answer is NO, then there is a proof of this fact that can be checked in polynomial time

Other features of problems:

- a problem is said to be <u>NP-Hard</u> if a polynomial time algorithm for this one would imply the existence of a polynomial time algorithm for every problem in NP
- More formally, a problem is <u>NP-Hard</u> if every problem in NP reduces in polynomial time to it
    a. unless $P = NP$, which is not yet solved
    b. if a problem is NP-Hard, it provides evidence the problem may not be in $P$
- a problem is <u>NP-Complete</u> if it's both in NP and NP-Hard
    a. e.g. the Cook-Levin Theorem for Boolean Satisfiability problem (SAT)
        i. made up of clauses with conjunction/disjunction, usually 3 (3-SAT)

We use a <u>reduction</u> given it's a very powerful tool:

- a reduction is an algorithm for transforming one problem into one another
- a problem $A$ reduces to $B$ if there is an algorithm able to solve $B$ can be translated into one which solves $A$
- remember reductions works from $Y$ (problem I know to be hard) to $X$ (new problem)
- the reduction is FROM $Y$
    a. I already know it's NP-hard
- to $X$
    a. the "new" problem



## 20.1 NP-Hard Problems

- Independent Set
    a. given a graph $G = (V, E)$ an independent set in $G$ is a subset $I \subseteq V$ with no edges between them
- (Maximum) Independent Set (this one will be referred to as simply "Independent Set" meaning the latter)
    a. compute an independent set of maximum size
- SAT/3-SAT (thanks to Cook-Levin Theorem)
    a. SAT - Boolean satisfiability of a formula (has to be equal to TRUE)
    b. 3-SAT - Boolean satisfiability of a formula made by 3-clauses
- Hamiltonian Circuit
    a. a cycle that traverses all the vertices only once
- (Maximum) Clique
    a. largest complete subgraph
- (Minimum) Vertex cover
    a. minimum number of vertices that "touches" all edges

Examples of reductions:

- Using Hamiltonian circuit to solve TSP → $Ham \leq_p TSP$
    a. If we had a fast algorithm for TSP, we would also solve Hamiltonian problem
- Using 3SAT to solve Independent Set → $3SAT \leq_p IndependentSet$
    a. If we had a fast algorithm for Independent Set, we would also solve 3SAT
- Using Independent Set to solve Clique → $Clique \leq_p IndependentSet$
- Using Independent Set to solve Vertex Cover → $Vertex\ Cover \leq_p IndependentSet$

## 20.2 NP-Hard Proofs

- *Theorem*: TSP (Traveling Salesperson Problem) is NP-Hard

- *Proof*: Reduction from Hamiltonian circuit to TSP ($Ham \leq_p TSP$)

Wait a minute: TSP is not a decision problem!

No worries. Define $TSP$ as:

- input: $G = (V, E)$ complete, undirected, weighted graph $k \in \mathbb{R}$
- output: $\exists$ in $G$ a Hamiltonian circuit of cost $\leq k$?

We could try to use all possible $k$ values, but $k$ is not guaranteed to be polynomial; using only the cycles will not work either.

*What we actually do*: Pick an arbitrary input instance for $Ham.$ and create the following input for TSP:

- $G' = (V, E')$ complete, undirected, weighted graph with:

$$w(e \in E') = \begin{cases} 1, & if\ e \in E \\ +\infty, & otherwise \end{cases}$$

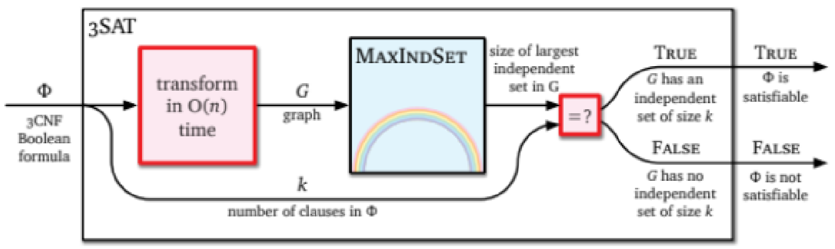
If we use $k = m$ this reduction takes poly-time $((O(n^2))$. Then:

- if $G$ has an Hamiltonian circuit, then the TSP algorithm run on $G'$ returns an Hamiltonian circuit with cost $n$
- if $G$ doesn't have a Hamiltonian circuit, then any Hamiltonian circuit in $G'$ must have $\geq 1$ edge not in $G$, hence of weight $\infty$. Hence, in this case, a TSP algorithm run on $G'$ returns a Hamiltonian circuit of cost $> n$

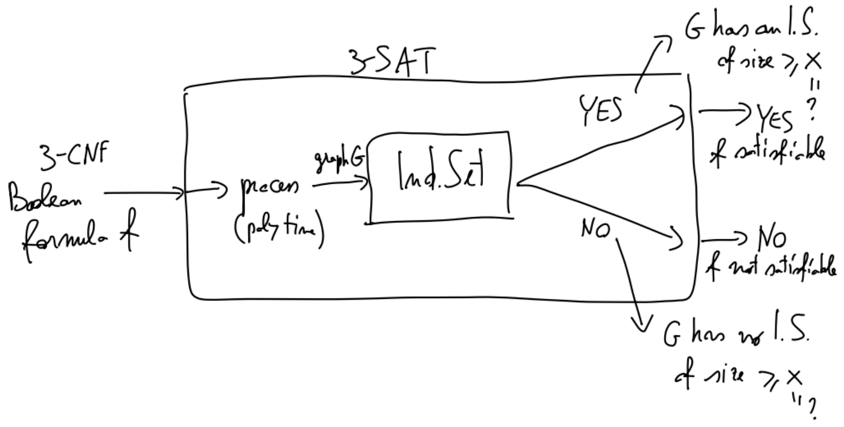
If we had a fast algorithm for TSP we would also solve the Hamiltonian circuit problem.

Reduction from $3SAT$ (problem in logic) to $Independent\ Set$ (problem in graphs) → $3SAT \leq_p IndependentSet$

They seem totally unrelated problems, but let's see what we have to do (figure here is from "Algorithms" book of Jeff Erickson, suggested in particular for the whole NP-Hardness chapter):



What we are conjecturing is the following:



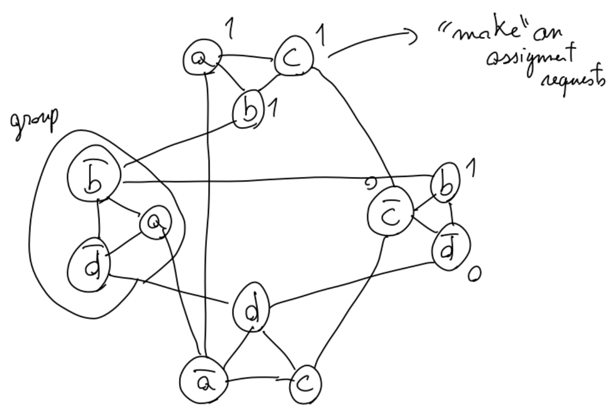
Basically, the presence of an independent set in the constructed graph corresponds to a satisfying truth assignment for the 3SAT instance.

Let's see the main ideas (figure representing the scenario):

- pick an arbitrary $3CNF$ Boolean formula $f$ with $k$ clauses

$$(a \lor b \lor c) \land (b \lor \neg c \lor \neg d) \land (\neg a \lor c \lor d) \land (a \lor \neg b \lor \neg d)$$

- vertices: each vertex represents one literal in $f$
    a. a *group* of 3 vertices represents a clause (one of the $k$ clauses)
        i. assignment request = choose vertices and make a request



- edges:
    a. We add an edge between a literal and its inverse, for all the literals
    b. We add an edge between every pair of vertices that are in the same group

There are two ways to think about 3SAT: (this reasoning coming from here)

- 1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true
- 2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick $x_i$ and $\neg x_i$

The reduction works this way:

- the graph will have one vertex for each literal in a clause
- connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

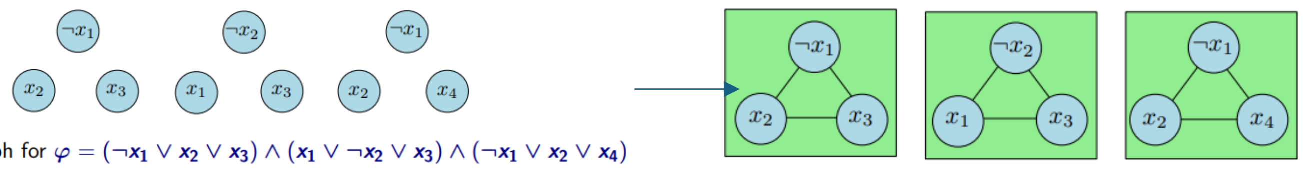
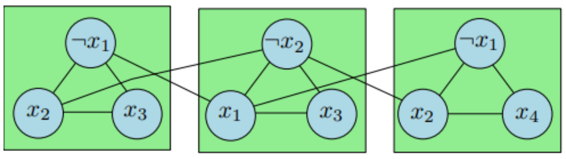


Figure: Graph for $\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

- connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict



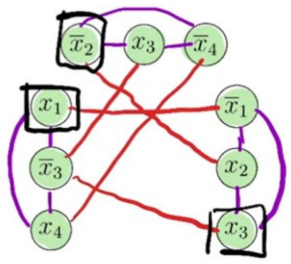
- Take $k$ to be the number of clauses, ensuring they are all "covered"

Remember what *satisfiable* means:

- it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE

$$\varphi = (x_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee x_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee x_2 \vee x_3)$$

1. Create a vertex for each literal.
2. Connect each literal to the other two literals in the same clause.
3. Connect each literal $x_i$ to $\overline{x}_i$.



1) idea: independent set represents <u>conflicts</u> ⇒ add an edge between every pair of vertices that are inconsistent (*asking* for opposite assignments to the same variable)
    a. in words: if you choose one vertex, it means it's part of a clause
    b. you have to choose other two vertices which are sure to be different because they are a different independent set
    c. if you choose one vertex, you <u>have</u> to choose the complement
        i. in order to realize the AND inside the formula

*Observation*: an independent set with $\geq 1$ vertex in each group gives a satisfying truth assignment → should look for indipendent sets of size $\geq k$ to say "YES, $f$ it's satisfiable".

*Issue:* an independent set now is free to recruit <u>multiple</u> vertices from a group, so I might output "YES, $f$ is satisfiable" even if this not true! ⇒ idea: force the recruitment of <u>one</u> vertex per group.

2) add one edge between every pair of vertices that one in the same group

*Claim*: $G$ contains an independent set of size exactly $k$ ⇔ the formula $f$ is satisfiable

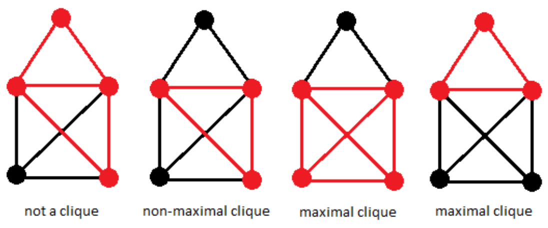
*Proof*:

1) suppose $f$ is satisfiable. Pick any satisfying assignment. Each clause in $f$ has $\geq 1$ $TRUE$ literal. Thus, we can choose a subset $S$ of $k$ vertices in $G$ that contains exactly one vertex per group such that the corresponding $k$ literals are all $TRUE$. The set $S$ is an independent set because it does not contain both endpoints of any edge of a group, nor of any edge that connects inconsistent literals (as it is derived from a consistent truth assignment)

2) suppose $G$ contains an independent set of size $k$. Each vertex in $S$ must be in a different group. Assign $TRUE$ to each literal of $S$. Since inconsistent literals are connected by an edge, this assignment is consistent. Since $S$ contains 1 verftex per group, each clause in $f$ contains (at least) one $TRUE$ literal ⇒ $f$ is satisfiable

- <u>(Maximum) Clique</u>: compute the longest complete subgraph
    a. other name for a complete graph (from now on, the problem will be called Clique)
    b. below, a useful figure to clearly see the problem

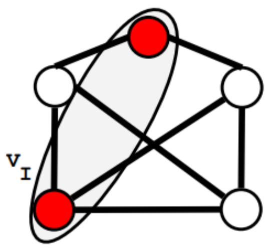
*Show that Clique is NP-Hard*.



<u>Solution</u> (a nice graphical explanation <u>here</u>)

Decision version:

- Input: $\langle G = (V, E), k \rangle$
- Output: $\exists$ in $G$ a clique of size $k$?



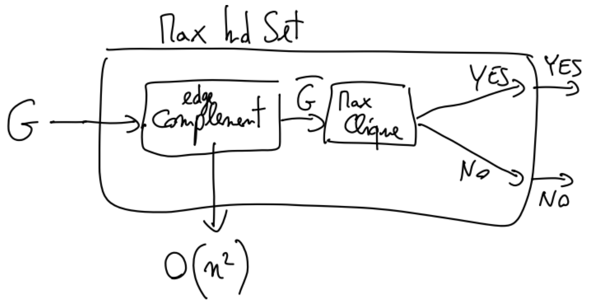
We operate a reduction from Maximum Independent Set (Ham. circuit is not really related to it; as you can see <u>here</u>, one can use 3SAT in order to show Clique is NP-complete). Figure here shows Independent Set.

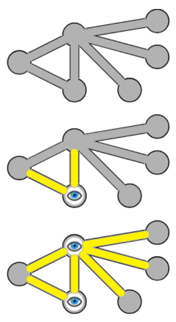
- *Intuition*
    a. clique: vertices with <u>all</u> edges between them
    b. maximum independent set: vertices with <u>no</u> edges between them

- *Definition*
    a. given a graph $G = (V, E)$, its <u>edge-complement</u> $\overline{G} = (V, \overline{E})$ has the same vertex $V$ and an edge set $\overline{E}$ such that $(u, v) \in \overline{E} \Leftrightarrow (u, v) \notin E$ (so, no common edges)

- *Observation*
    a. a set of vertices $S$ is independent in $G \Leftrightarrow S$ is a clique in $\overline{G} \Rightarrow$ the largest independent set in $G$ has the same size as the largest clique in $\overline{G}$

To make it super complete, let's draw the schema of what we are doing – takes $O(n^2)$ time, givemn the constant work needed to traverse all edges *and* vertices:





*Definition*: a <u>vertex cover</u> of a graph is a set of vertices that includes that at least one endpoint of every edge of the graph

b. Side figure represents such, to be clearer to you

Another problem is:

- <u>(Minimum) Vertex Cover</u>: compute the smallest vertex in a given graph
    a. From now on, only called Vertex Cover

*Show that Vertex Cover is NP-Hard*.

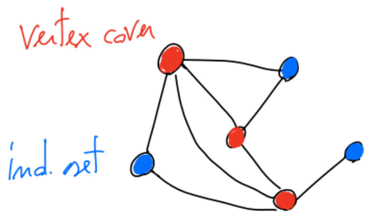
<u>Solution</u> (once again, a nice graphical explanation of this one here)

Decision version:

- Input: $< G = (V, E), k >$
- Output: $\exists$ in $G$ a vertex cover of size $k$?

We operate a reduction from Maximum Independent Set (once again, this is the most similar problem to the one we are proving)

- *Observation*
    a. a set of vertices $S$ is independent in $G \Leftrightarrow V \setminus S$ is a vertex cover of $G$
        i. in blue there is an independent set (actually the biggest one)
        ii. the other ones are the vertex cover

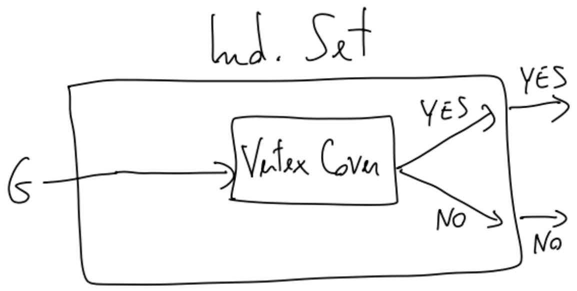


$\Rightarrow$ the longest independent set in $G$ has size $n - k$, where $k$ is the size of the smallest vertex cover of $G$

Independent set:

- Input: $\langle G = (V, E), n - k\rangle$
- Output: $\exists$ in $G$ an independent set of size $n - k$?

Once again, let's represent this in a complete way:



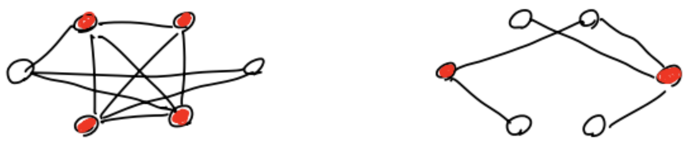
Exercise

- *Show that*:
    a. $Vertex\ Cover \leq_p Independent\ Set$
    b. $Clique \leq_p Vertex\ Cover$

  ⇒ *these 3 problems are equivalent.*

Solution (official = shorter)

- "same" as $Independent\ Set \leq_p\ Vertex\ Cover$
- we can consider the following figure for this one
    a. consider a clique of size 4 in the middle (left)
    b. if you take the complement of this one (right)
- $G$ has a clique of size $k \Leftrightarrow\ \overline{G}$ has a vertex over of size $n - k$
    a. proof: see the book (§ - p. 1106 of 4th edition – theorem 34.12)



Solution (longer and better explained)

a. Suppose that we have an efficient algorithm for solving Independent Set, it can simply be used to decide whether $G$ has a vertex cover of size at most $k$, by asking it to determine whether G has an independent set of size at least $n -\ k$

Given an instance of the Vertex Cover problem, consisting of a graph $G = (V, E)$ and an integer $k$ representing the size, we construct an instance of the Independent Set problem as follows:

1. Let $G' = G$ (i.e., the graph for the Independent Set instance is the same as the original graph G).

2. Let $k' = |V| - k$ (i.e., the target size of the independent set is the number of vertices in $G$ minus the size of the vertex cover $k$).

To show that this reduction is correct, we need to prove the following:

1. If $G$ has a vertex cover of size $\leq k$, then $G'$ has an independent set of size $\geq k'$.

2. If $G'$ has an independent set of size $\geq k'$, then G has a vertex cover of size $\leq k$.

Let's prove both (1) and (2):

- Suppose $C$ is a vertex cover of size $\leq k$ in $G$. Then, the set $V \setminus C$ is an independent set in $G'$ (since $C$ covers all the edges, no two vertices in $V \setminus C$ can be adjacent). Furthermore, $|V \setminus C|\ \geq |V| - k = k'$

- Suppose $S$ is an independent set of size $\geq k'$ in $G'$. Then, the set $V \setminus S$ is a vertex cover in $G$ (since $S$ is independent, every edge must have at least one endpoint in $V \setminus S$). Furthermore, $|V \setminus S| \leq\ |V| - k' = k.$

b. To show that $Clique \leq_p Vertex\ Cover$, we need to provide a polynomial-time reduction from the Clique problem to the Vertex Cover problem. Here's one way to construct the reduction:

Given an instance of the Clique problem, consisting of a graph $G = (V, E)$ and an integer $k$, we construct an instance of the Vertex Cover problem as follows:

1. Let $G' = G$ (i.e., the graph for the Vertex Cover instance is the same as the original graph G).
2. Let $k' = |V| - k$ (i.e., the target size of the vertex cover is the number of vertices in $G$ minus the size of the clique $k$).

To show that this reduction is correct, we need to prove the following:

1. If $G$ has a clique of size $\geq k$, then $G'$ has a vertex cover of size $\leq k'$.
2. If $G'$ has a vertex cover of size $\leq k'$, then $G$ has a clique of size $\geq k$.

Proof of (1): Suppose $C$ is a clique of size $\geq k$ in $G$. Then, the set $V \setminus C$ is a vertex cover in $G'$ (since $C$ is a clique, every edge must have at least one endpoint in $V \setminus C$). Furthermore, $|V \setminus C| \leq |V| - k = k'$.

Proof of (2): Suppose $S$ is a vertex cover of size $\leq k'$ in $G'$. Then, the set $V \setminus S$ is a clique in $G$ (since $S$ is a vertex cover, every edge must have both endpoints in $V \setminus S$, which means $V \setminus S$ is a clique). Furthermore, $|V \setminus S| \geq |V| - k' = k$.

# 21 Approximation Algorithms

These kinds of algorithms are are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems) with *provable* guarantees on the distance of the returned solution to the optimal one. They solve problems not solvable in polynomial time using approximation.

An *optimization problem* can be described as follows:

$$\Pi: I \; x \; S$$

where $\Pi =$ approximation problem, $I$ = set of inputs and $S$ = set of solutions.

$$c: S \to \mathbb{R}^+$$

Above, the *cost function* $c$ maps each solution to a positive real number.

$$\forall i \in I, S(i) = \{s \in S: i \; \Pi_s\}$$

Above, the the *set of feasible solutions*, and our goal follows.

$$s^* \in S(i) \; and \; c(s^*) = \min / \max \; \{c(S(i))\}$$

Here, we want to find the best solution $s^*$ for a minimization/maximization problem. Specifically, we want to find it for the specific instance of that problem $(i\Pi s)$.

*Definition*: Let $\Pi$ be an optimization problem and let $A_\Pi$ be an algorithm for $\Pi$ that returns, $\forall i \in I, A_\Pi \in S_i$. We say that $A_\Pi$ has an <u>approximation factor</u> of $\rho(n)$ if $\forall i \; in \; I$ such that $|i| = n$ we have:

- minimization problem (basically, an explicit lower-bound of the optimal solution)

$$\frac{c(A_\Pi(i))}{c(s^*(i))} \leq \rho(n)$$

- maximization problem (basically, an explicit upper-bound of the optimal solution)

$$\frac{c(s^*(i))}{c(A_\Pi(i))} \leq \rho(n)$$

Here, we assume that $c$ maps each feasible solution to a real number $\geq 1$.

*Goal*: $\rho(n) = 1 + \epsilon$, with $\epsilon$ as small as possible.

*Definition*: An <u>approximation scheme</u> for $\Pi$ is an algorithm with 2 inputs $A_\Pi(i, \epsilon)$ that $\forall \epsilon$ is a $(1 + \epsilon)$-approximation.

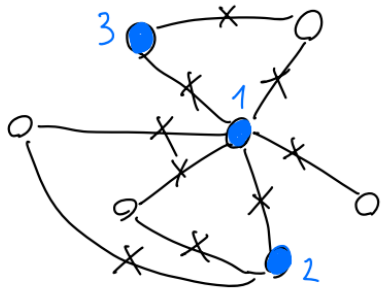
- In this case we just have to choose *how much approximation* we want by tuning the value of $\epsilon$
- In other words: fixed an instance $i$ of size $n$, the quality is $\epsilon$ (whatever $\epsilon$ is)

## 21.1 Examples of Approximations

Very first algorithm you can think of? Use a *greedy approach*:

- select the vertex for the highest degree
- "remove" the touched edges
- repeat

Consider the following figure – take 3 as the highest, then 2 and 1 and remove touched edges as said:



Unfortunately, for this algorithm $\rho(n) = \Omega(\log(n))$.

How to prove a LB (Lower Bound)? It's enough to show one "bad" input instance.

Another algorithm (greedy approach):

- choose *any* edge
- add its endpoints to the solution
- "remove" the covered edges
- repeat

We'll show that this is a 2-approximation algorithm.

```
procedure $Approx\_Vertex\_Cover(G)$
    $V' = \emptyset$
    $E' = E$
    while $E' \neq \emptyset$: do
        $Let\ (u, v)\ be\ an\ arbitrary\ edge\ of\ E'$
        $V' = V' \cup \{u, v\}$
        $E' = E' \setminus \{(u, z), (v, w)\}$
        $//\ remove\ edges\ that\ have\ u\ and\ v\ as\ endpoints$
    return $V'$
```

*Complexity*: $O(n + m)$

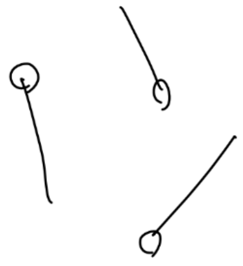
We'll show $\frac{|V'|}{|V^*|} \leq 2$

Given $A$ = set of selected edges:

- $A$ is a matching: $\forall e, e' \in A \Rightarrow e \cap e' = \emptyset$
    a. i.e. set of edges with no vertices in common
    b. every edge is disjoint, so there is no couple of edges sharing a common node
- $Approx\_Vertex\_Cover$ selects a maximal matching: $\forall$ edge $y, A \cup y$ is not a matching
    a. this is a matching which cannot be increased
        i. not possible to select an edge which touches other vertices

*Proof*:

1. lower bound to the optimal solution $V^*$



What can one say about $|V^*|$ $vs$ $|A|$?

$A$ is a matching $\Rightarrow$ in $V^*$ there must be $\geq 1$ vertex $\forall$ edge of $A$ (right figure)

In whatever vertex cover, particularly $V^*$, we have to cover all graph edges and, in particular, all $A$ edges. But $A$ is a matching (so, every edge of $A$ is disjoint), so:

$$|V^*| \geq |A|$$

2. upper bound to the optimal solution $V'$

What can one say about $|V'|$ $vs$ $|A|$?

- $|V'| = 2|A|$ by construction and so:

$$(1. + 2.) \Rightarrow |V'| \leq \cancel{2|A|} \leq 2|V^*| \Rightarrow \frac{|V'|}{|V^*|} \leq 2$$

# 22 TSP & METRIC TSP

## 22.1 TRAVELLING SALESPERSON PROBLEM (TSP)

*Definition*: Given a complete, undirected graph and a function $w\colon E \to \mathbb{R}^+$, output a tour $T \subseteq E$ (i.e. a cycle that passes through every vertex exactly once) minimizing $\sum_{e \in T} w(e)$. Collectively:

$$T \subseteq E \ minimizing \sum_{e \in T} w(e)$$

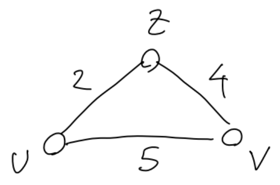
- $w\colon E \to \mathbb{R}^+$ : we will work only on positive weights).
  a. We can do this without loss of generality (wlog) because every TSP tour has the same number of edges ⇒ we can add a large weight to each edge, such that edges have non-negative weights

## 22.2 METRIC TSP

Metric TSP is a special case of TSP where the weight function $w$ satisfies the triangle inequality:

$\forall\, u, v, z \in V,$ it holds that $w(u, v) \le w(u, z) + w(z, v)$

The following is an example of that:



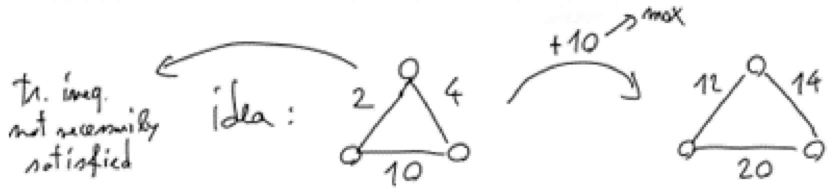
The problem can be shown to be NP-Hard, using an instance of TSP to build this one and using an Hamiltonian circuit to show we can assign a weight to each edge being "balanced" overall in the choice, always ensuring the best choice.

### 22.2.1 Metric TSP is NP-Hard

*Theorem:* $Metric\ TSP$ is NP-Hard

*Proof*: $TSP \le_p Metric\ TSP$

The idea is the following (where inequality is not strictly satisfied):



Given an instance of the TSP problem $< G = (V, E), w, k >$, we build an instance of $Metric\ TSP < G' = (V, E), w', k' >$ such that the triangle inequality is satisfied in $G'$ . In order to to this, we can define the weight function $w\ '$ as follows:

$$w'(u, v) = w(u, v) + W$$

giving $W = \max_{u,v \in V}\{w(u, v)\}$

$$\begin{tikzcd} \langle G=(V,E), w, k\rangle \arrow[d, squiggly] \\ \langle G'=(V,E), w', k'\rangle \end{tikzcd}$$

$$w'(u,v) = w(u,v) + \underset{\max_{u,v\in V}\{w(u,v)\}}{\underset{\parallel}{W}}$$

Think of a value $k'$ in such a way there if there exist an Hamiltonian circuit, there will be one in $G'$ in such a way the cost of the tour will work for every edge, so:

$$k' = k + nW$$

To be shown yet:

1) $w'$ satisfies triangle inequality
2) $\exists$ an Hamiltonian circuit of cost $k$ in $G \Leftrightarrow \exists$ Hamiltonian circuit of cost $k'$ in $G'$

Let's see how to solve them formally:

1) $w'(u, v) \leq^? w'(u, w) + w'(w, v)$ (is it at most the weight of the others)?
   $w(u, v) + W \leq^? w(u, w) + w(w, v) + 2W$ (does this hold adding a general weight)?
   $w(u, v) \leq^? w(u, w) + w(w, v) + W$ (simply adding $W$ both members)
   $\underbrace{w(u, w)}_{\geq 0} + \underbrace{w(w, v)}_{\geq 0} + \underbrace{W - w(u.v)}_{\geq 0} \geq^? 0$ (is it true this is at most 0)?

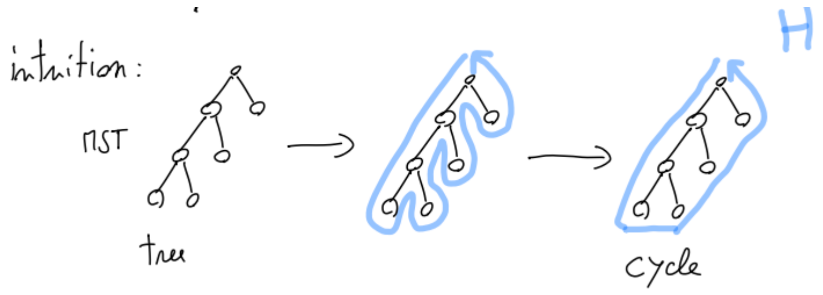
   We only ask if the definition of triangle inequality is satisfied correctly.
   Note it's important that the weights of edges are non-negative (otherwise, last part does not hold)

2)
   a. $\Rightarrow$: $\exists$ Ham. circuit of cost $k$ in $G$. Note that an optimal solution contains exactly $n$ edges and the same circuit in $G'$ introduces a weight for every edge (so, $+W\ \forall$ edge). Thus, the cost of said tour in $G'$ is $k + nW$
   b. just remove the $+W\ \forall$ edge to obtain a Ham. circuit of cost $k$ in $G$

## 22.3 2-Approximation Algorithm for Metric TSP

What is the most similar problem to $Metric\ TSP$? MST (Minimum Spanning Tree). We give the following *intuition*:

- we give an MST
- we want to build a cycle: what to do on a tree to achieve it?
- basically, there is a DFS traversing all the nodes
- the cycle forms having all nodes touched exactly once



We can simply solve this problem by adding the edge $(e, a)$ to the $Preorder$ list and make it an Hamiltonian circuit. We are free to add every edge we want because the graph is complete by definition. To do so, we define the following:

```
procedure $Approx - Metric - TSP(G)$:
    $V = \{v_1, v_2, \dots v_n\}$
    $r = v_1$ $//root\ from\ which\ Prim\ is\ run$
    $T^* = Prim(G, r)$
    $\langle v_{i_1}, v_{i_2}, \dots v_{i_n} \rangle H' = PREORDER(T^*, r)$
    $//\ lists\ all\ the\ vertices\ in\ the\ tree\ in\ an\ ordered\ fashion\ following\ a\ preorder\ walk$
    return $\langle H', v_{i_1} \rangle \geq H$ $//\ basically, close\ the\ cycle$
```

This algorithm uses Prim as a subroutine to compute the MST. As such, this is super-fast and can be characterized as a near-linear algorithm.

So, to fully summarize:

1. Given a complete weighted graph $G$, pick any vertex $v$ as the root, and find a minimum spanning tree $T$, using Prim's algorithm
2. Compile a list $L$ of vertices encountered in a preorder traversal of $T$
3. Return $L$ as a tour

## 22.4 3/2 (OR 1.5) APPROXIMATION ALGORITHM FOR METRIC TSP

Christofides algorithm (or Christofides–Serdyukov algorithm) was born in 1976.

Reason for 2-approximation factor was the fact the preorder traversal of $T^*$ used every edge of $T^*$ exactly twice. We'll try to improve on this by constructing a tour that traverses MST edges only once.

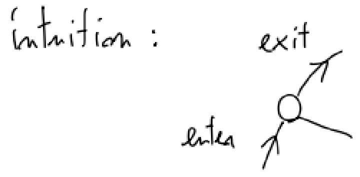
We give a couple of definitions useful for this context:

- A path (or cycle) is Eulerian if it crosses every edge of the graph exactly once
- A connected graph is Eulerian if there exists an Eulerian cycle

If the MST was Eulerian (cannot be) then we would have a 1-approx algorithm (which would be optimal, given one would cross every edge exactly once). $Approx\_Metric\_TSP$ is finding a "cheap" Eulerian cycle in the MST, but effectively needs to double its edges.

*Question*: is there a cheaper Eulerian cycle?

*Theorem*: A connected graph is Eulerian ⇔ every vertex has even degree. The intuition is the following: enter a vertex, then exiting from it using a new edge, doing that without using edges more than once.



We want to focus on the *odd* degree vertices, given I have to cross again vertices (the even ones are fine, given we don't pass on them again). So, let's handle the odd-degree vertices of the MST explicitly.

*Property*: in any (finite) graph, the number of vertices of odd degree is even.

*Proof*: We use the following equality:

$$\sum_{v\in V} \deg(v) = 2m$$

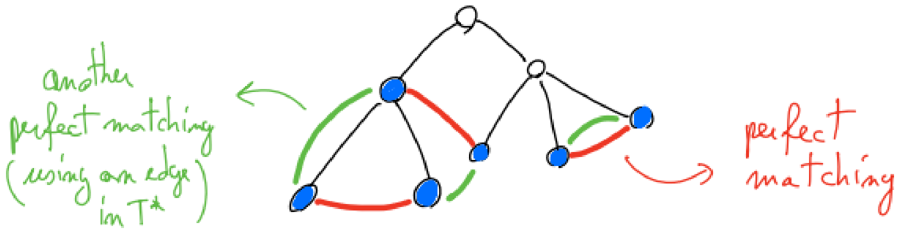
Basically, the sum of odd vertices with even ones, will get us an even result, that's the main intuition. So, we can split such summation into two parts:

$$\underbrace{\sum_{u\in even} \deg(u)}_{even} + \sum_{w\in odd} \deg(w) = \underbrace{2m}_{even}$$

Since the result must be even, the sum of degrees must be even too. But this happens only if the number of odd degree vertices is even.

*Idea*: augment the initial MST $T^*$ with (the cheapest basically) a minimum-weight perfect matching (perfect means that it includes all the vertices) between the vertices that have odd degree in the MST.

For instance, let's consider the following MST, coloring in blue the odd-degree vertices. Imagine we add a perfect matching colored in red.



⇒ the resulting graph has only even-degree vertices, i.e. is an Eulerian graph.

Let's write the algorithm, which does exactly for things:

$Christofides(G)$

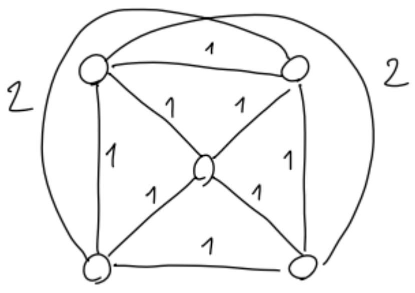
1) $T^* \leftarrow Prim(G,r)$ // $T^* = (V, E^*)$

2) Let $D$ be the set of vertices of $T^*$ with odd-degree. Compute a min-weight perfect matching $M^*$ on the graph induced by $D$ // this can be done in polynomial time (Edmonds, 1965)

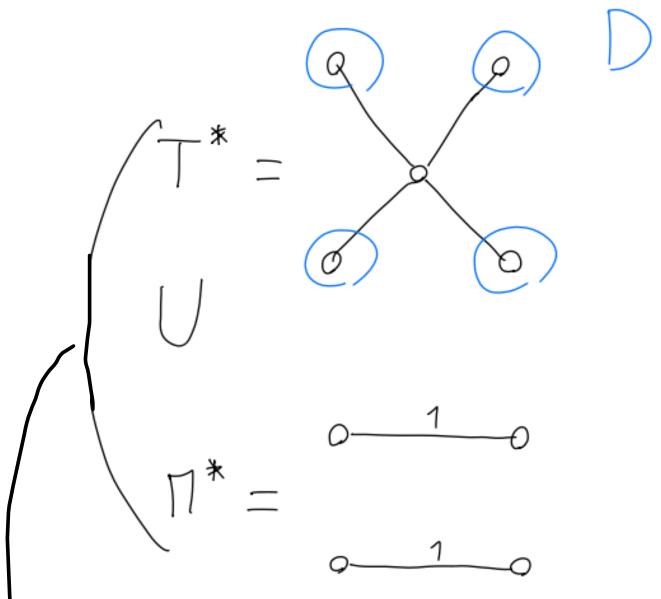
3) The graph $(V, E^* \cup M^*)$ is Eulerian // any edge in both $E^*$ and $M^*$ appears twice in this (multi)graph.

4) Return the cycle that visits all the vertices of $G$ in the order of their first appearance in the Eulerian cycle (basically, skipping all repeated vertices – shortcutting)

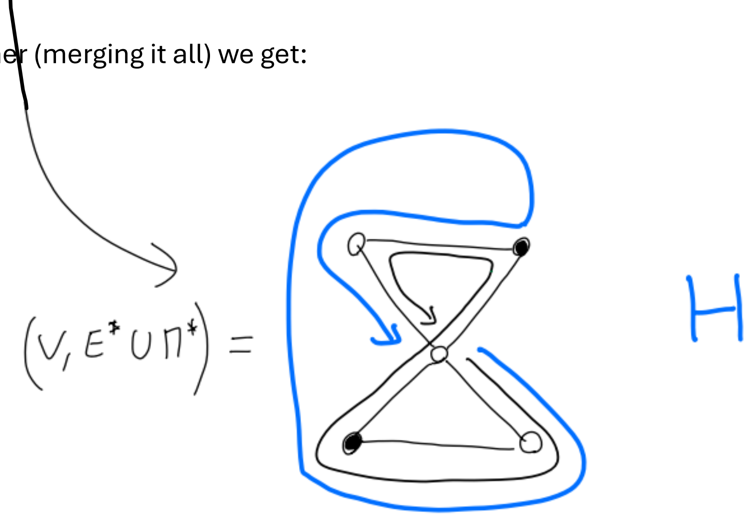
Consider the following example, connecting all vertices:



Now take the odd-degree vertices $T^*$ and compute the minimum-weight perfect matching $M^*$.



Putting all of this together (merging it all) we get:



Analysing the algorithm:

- $w(H) \le w(T^*) + w(M^*)$
    a. by triangle inequality

- $w(T^*) \le w(H^*)$

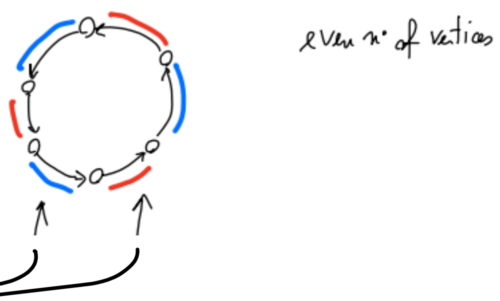
The goal to reach is $w(H) \le \frac{3}{2} w(H^*)$. We would need to prove:

- $w(M^*) \le^? \frac{1}{2} w(H^*)$ (by triangle inequality)

We will do the following clever step:

$w$(optimal tour of the odd-degree vertices of $T^*$) $\leq w(H^*)$

partition this in 2 perfect matchings:



One of these 2 has cost $\leq \frac{w(H^*)}{2}$

Putting all pieces together we get:

$$w(H) \leq w(H^*) + \frac{w(H^*)}{2} = \frac{3}{2} w(H^*)$$

# 23 SET COVER

Set cover is an optimization problem that models many problems requiring resources to be allocated. It aims to find the least number of subsets that cover some universal set.

Its inputs are:

- $I = (X, F)$ = instance of the set covering problem
- $X =$ set of elements of any kind, called "universe"
- $F \subseteq \{S : S \subseteq X\} = B(X)$
    a. $B$ stands for "Boolean": set of all subsets of $X$

There is a constraint that needs to be always respected: $\forall x \in X, \exists\, S \in F : x \in S$ i.e., "$F$ covers $X$"

Optimization problem: (smallest subset of $F$ having its members covering all $X$) → find $F' \subseteq F$ s.t.

1) $F'$ covers $X$
2) $\min\ |F'|$

*Example*:

$X = \{1,2,3,4,5\}$

$F = \{\{1,2,3\}, \{2,4\}, \{3,4\}, \{4,5\}\}$

$\Rightarrow F^* = \{\{1,2,3\}, \{4,5\}\}$

## 23.1 SET COVER IS NP-HARD

*Assertion*: Set Cover (in its decision version $\langle (X, F), k \rangle$) is NP-hard.

*Proof*: $Vertex\ Cover \leq_p Set\ Cover$

- Given an instance of Vertex Cover Problem $\langle G = (V, E), k \rangle$
- we create an instance of Set Cover problem $\langle (X, F), k \rangle$

Basically $\langle G = (V, E), k \rangle \rightarrow^f \langle (X, F), k \rangle$

where:

- $X = E$
- $F = \{S_1, S_2, \dots S_n\}$ one $\forall$ vertex $\in V$, $1,2, \dots n$
- $S_i = \{e = (u, v)$ such that $u = i$ or $v = i\}$, which is the set of covered edges by node $e$

Basically, there are $|V| = n$ subsets $S_i$, and each subset is the set of edges incident to vertex $i$. Now show that finding a Set Cover of size $k \Leftrightarrow$ finding a Vertex Cover of size $k$.

- $\Rightarrow$ Suppose $\{S_1, S_2, \dots, S_k\}$ is a set cover for $X$. Then, every edge in $E$ must be incident to at least one vertex $u_1, \dots, u_k$. This happens because every element if one node of the adjacency list and so we find the minimal number of nodes touching all edges of graph, guaranteeing it will be minimal (for all sizes, given, even if less than $k$). Therefore, it forms a vertex cover of size $k$ in $G$.
- $\Leftarrow$ Suppose $u_1, \dots, u_k$ is a vertex cover in $G$. Then, $S_i$ covers all the edges incident to vertex $u_i$. Therefore, $\{S_1, \dots, S_k\}$ is a set cover of size $k$ for $X$

*Written by Gabriel R.*

## 23.2 Greedy Approximation Algorithm

The greedy method works by picking at each stage, the set $S$ that covers the greatest number of remaining elements that are uncovered:

- choose the subset that contains the largest number of uncovered elements
- remove from $X$ those covered elements
- repeat until $X = \emptyset$

```
$Approx\_Set\_Cover(X, F)$
    $U = X$
    $F' = \emptyset$ // solution
    while $U \neq \emptyset$: do
        // take the set of F covering as many elements as possible
        let $S \in F = |S \cap U| = \max_{S' \in F}\{|S' \cap U|\}$
        $U \leftarrow U \setminus S$    // update the list of available elements, removing those from S
        $F \leftarrow F \setminus \{S\}$    // remove the sets already considered inside of F
        $F' \leftarrow F' \cup \{S\}$
    return $F'$
```

*Correctness*: (it does the job – it covers all the elements) At every iteration $|U|$ decreases by at least one.

*Complexity*:

- n. of iterations $\leq |X|$ (every $S_i \in F$ contains at least an element)
- n. of iterations $\leq |F|$ (every $S_i \in F$ contains at least two elements)
- $\Rightarrow$ n. of iterations $\leq \min\{|X|, |F|\}$
- $\forall$ iterations the complexity is $\leq |X| * |F|$ (scanning all elements and decreasing elements in both sets)
- $\Rightarrow O(|X| * |F| * min\{|X|, |F|\})$

# 24 Randomized Algorithms

Randomized algorithms are algorithms that may do *random* choices, basically using a source of randomness in its logic. We give some basic examples:

- Example 1: Randomized quicksort (RQS)

Quicksort but chooses the pivot at random so to break the unlucky element choice and get on average a good probability on result.

- Example 2: Verifying polynomial identity

Checks if polynomials are equivalent and there are different approaches:

- check all the terms (slow)
- choose a random integer, compute the polynomials and check if they are equal
    a. it may be wrong, outputting YES even If they are different

## 24.1 Classification of Randomized Algorithms

We divide these into two main categories:

1) randomized algorithms that never fail, which are called "LAS VEGAS" algorithms
    a. (e.g., randomized quicksort)

$$\forall i \in I, A_R(i) = s \quad s.t. (i, s) \in \Pi$$

where $\Pi \subseteq I \; x \; S$ is the decisional problem, $i$ is an input instance, $A_R$ is the random algorithm which applied to the input instance produces a solution $s$ s.t. the couple $(i, s)$ belongs to $\Pi$

Randomness comes into play in the analysis of the complexity – because it depends from the randomness of the choices. $\forall n, T(n)$ is a random variable of which we usually study its expectation $E[T(n)]$ or $\Pr\big(T(n) > c * f(n)\big) \rightarrow \leq \frac{1}{n^k}$ (so, for some constants $c$ and $k$, we say that $T(n) = O(f(n))$ with high probability (here, $T(n)$ is called complexity function) – this second one is more powerful than the first, so $Pr$ more powerful than $E$.

2) randomized algorithms that may fail are called "MONTE CARLO" algorithms
    a. e.g., verifying polynomial identities

$$\forall i \in I, A_R(i) = s \quad s.t. (i, s) \notin \Pi$$

We study $\Pr\,((i, s) \notin \Pi)$ as a function of $n = |i|$ → family of random variables (binary)

Moreover, even $T(n)$ may be a random variable. For decision problems, these algorithms can be divided into:

- one-sided: they may fail only on one answer
    a. e.g., can make right all YES instances but may be wrong on all NO instances
- two-sided: they may fail in both answers
    a. e.g., it can make wrong all YES instances but can make wrong all NO instances

## 24.2 Karger's Algorithm for Minimum Cut

A quite simple MONTE CARLO and elegant algorithm created in 1993. Let's start from the problem itself it wants to solve: the minimum cut revolves finding a cut of minimum size, that is, the minimum number of edges whose removal disconnects the graph. A couple of useful definitions to see the problem:

*Definition*: A multiset is a collection of objects with repetitions allowed. It's usually denoted between a couple of brackets, as you can see here.

$$S = \{\{objects\}\}$$

$$\forall\ object\ o \in S, m(o) \in \mathbb{N} \setminus \{0]$$

where $m$ = multiplicity, so how many copies of "o" are in $S$.

*Definition*:

- a multigraph $G = (V, \mathcal{E})$ s.t. $\forall\ V \subseteq \mathbb{N}$, $V$ finite and $\mathcal{E}$ is a multiset of elements $(u, v)\ s.t. u \neq v$

Note: A simple graph $G = (V, E)$ is also a multigraph.

*Definition*:

- given $G = (V, \mathcal{E})$ connected, a cut $C \subseteq \mathcal{E}$ is a multiset of edges s.t. $G' = (V, \mathcal{E} \setminus C)$ is not connected.

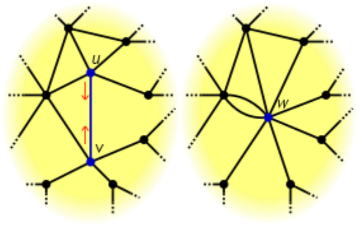
Let's give Karger's idea here:

- choose an edge at random
- "contract" the two vertices of that edge, removing all the edges incident both vertices

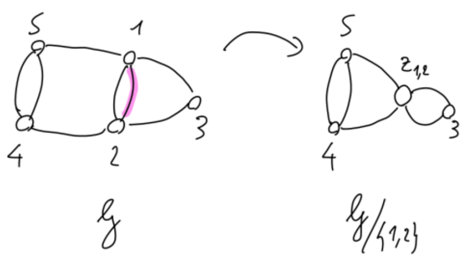
This works with very low probability, but let's use the trick we saw already: repeating this a good enough number of times, can actually refine the analysis and obtain a good level of probability.

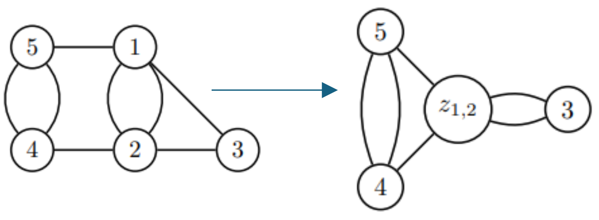
We see below two examples of the same contraction in Karger and on the right a generic contraction.

- Basically, it makes the two vertices to collapse in just one vertex connected with all the previous adjacent vertices
- If as a result there are several edges between some pairs of (newly formed) vertices, retain them all.



example :





*Definition*: given $G = (V, \mathcal{E})$ and $e = (u, v) \in \mathcal{E}$, the contraction of $G$ with respect to $e$, $\frac{G}{e} = (V', \mathcal{E}')$ is the multigraph with $V' = V \setminus \{u, v\} \cup \{z_{u,v}\}$ with $z_{u,v} \notin V$ coming from the fusion of $u$ and $v$:

$$\mathcal{E}' = \mathcal{E} \setminus \{\{(x, y)\ s.t.\ (x = u)\ or\ (x = v)\}\}$$

$$\cup \{\{(z_{u,v}, y)\ s.t.\ (u, y) \in \mathcal{E}\ or\ (u, y) \in \mathcal{E}, y \neq u\ and\ y \neq v$$

We describe the algorithm here:

```
$FULL\_CONTRACTION\ (G = (V, \mathcal{E}))$
    for $i = 1$ to $n - 2$: do
        $e \leftarrow RANDOM(\mathcal{E})$             $//\ choose\ an\ edge\ at\ random\ in\ multiset$
        $G' = (V', \mathcal{E}') \leftarrow \frac{G}{e}$          $//\ contraction\ on\ graph\ with\ random\ edge\ selection$
        $V \leftarrow V'$                      $//\ construct\ the\ new\ graph\ with\ vertices\ and\ edges$
        $\mathcal{E} \leftarrow \mathcal{E}'$
    return $|\mathcal{E}|$                     $//\ return\ the\ graph\ and\ its\ cardinality$
```

Consider $k$ → how many times to repeat $FULL\_CONTRACTION$, which depends on the probability of making a mistake – hence, it depends on the analysis of the algorithm)

```
$KARGER(G = (V, \mathcal{E}), k)$
    $min - cut = \mathcal{E}$
    for $i = 1\ to\ k$: do
        $t = FULL\_CONTRACTION(G)$
        if $|t| < |min - cut|$ then
            $min - cut = t$
    return $min - cut$
```

→ repeats $FULL - CONTRACTION$ $k$ times to reduce the probability of error

→ to be determined by the analysis

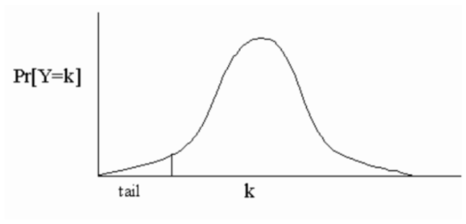
The analysis suggests us the following:

- in order to obtain $\Pr(\text{Karger succeds}) > 1 - \frac{1}{n^d}$ we need to repeat "Full contraction" $k = \frac{dn^2 \ln n}{2}$ times

# 25 Chernoff Bounds

Chernoff bounds are tools from modern probability theory that are frequently used in the analysis of randomized algorithms. They're a more powerful version of the Markov's lemma. This mainly uses *indicator* random variables (that is, variables which can have either value 0 or 1).

Generally, Chernoff bounds are a tool which allow to study the concentration of an event around its mean (specifically, in the "tail" – see figure – so, far from the mean) and to overcome the previous fact we use them.



- The markup is a little loose, not very significant
- Better augmentation allows me to move from analysis to the average case to the more desirable high probability analysis

The idea between Chernoff bounds is to transform the original random variable into a new one, such that the distance between the mean and the bound we will get is significantly stretched. It answers the question about how tight the bound we can get when having more information about the distribution of the random variables.

We give Chernoff's lemma here: let $X_1, X_2, \dots X_n$ independent indicator random variables where $E[X_i] = p_i, 0 < p_i < 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then $\forall\ \delta > 0$:

$$\Pr(X > (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

In words: the outcome concentrates around the min is very high – to the contrary, the probability of deviating from the min should be very low.

## 25.1 Chernoff Bound Variants

Consider the following *variants of Chernoff bounds* (weaker but easier to state and to use):

1) $\Pr(X < (1-\delta)\mu) < e^{-\frac{\mu\delta^2}{2}}, 0 < \delta \leq 1$

2) $\Pr(X > (1+\delta)\mu) < e^{-\frac{\mu\delta^2}{2}}, 0 < \delta \leq 2e - 1$

## 25.2 Analysis in High Probability of Randomized Quicksort

As an example of application of Chernoff bounds, we do the analysis of Randomized Quicksort, in which we remember the pivot is chosen at random (possibly, not very far from the median). This is a LAS VEGAS algorithm, since it always sorts.

```
$RandQuickSort(S)$                    $|S| = n$, all distinct
    if $|S| \leq 1$ then return $\langle S \rangle$
    $p = RANDOM(S)$        $// pick\ a\ “pivot”\ element\ uniformly\ at\ random\ from\ S$
```

```
$S_1 = \{x \in S \ s.t. \ x < p\}$          $// \ O(n) \ time$
$S_2 = \{x \in S \ s.t. \ x > p\}$          $// \ O(n) \ time$
$Z_1 = RandQuickSort(S_1)$
$Z_2 = RandQuickSort(S_2)$
return $\langle Z_1, p, Z_2 \rangle$
```

We want to approximate as close as possible to the actual median. The generic event $E$ can be characterized as the "good choice" of the pivot between all the statistically possible choices.

So, calculating the cost of the algorithm:

- total work at each level is mostly linear, so $\leq c * n$
- depth of the recursion tree $= \min\left\{integer\ i\ s.t. \left(\frac{3}{4}\right)^i n \leq 1\right\} = \lceil \log_{\frac{4}{3}}(n) \rceil = O(\log(n))$

  So, continuing: $\left(\frac{3}{4}\right)^i n \leq 1 \Leftrightarrow \left(\frac{3}{4}\right)^i \leq \frac{1}{n} \Leftrightarrow \left(\frac{4}{3}\right)^i \geq n \Leftrightarrow \log_{\frac{4}{3}}\left(\frac{4}{3}\right)^i \geq i \geq \log_{\frac{4}{3}}(n)$

  $\Rightarrow T_{RQS}(n) = O(nlog(n))$

Fix <u>one</u> root-leaf path $P$ and the following lemma says, "with h.p. the path chosen in short"; specifically shorter than log $(n)$".

*Lemma*: $\Pr\left(|P| > \underbrace{a}_{\hookrightarrow \text{constant}} * \log_{\frac{4}{3}}(n)\right) < \frac{1}{n^3}$

We are trying to find the probability there is at least one big path around the mid value.

If this is true, we're done, applying the very frequent/very famous following lemma.

*Lemma* (Union bound): for any random events $E_1, \dots E_K$:

$$\Pr(E_1 \cup E_2 \cup \dots \cup E_k) \leq \Pr(E_1) + \Pr(E_2) + \dots + \Pr(E_k)$$

If the lemma is true, it follows that:

- Given the event $E_i =$ the path $p_i$ has length $> a * \log_{\frac{4}{3}}(n)$:

$$\Pr(\exists\ path > a * \log_{\frac{4}{3}}(n)) = \Pr\left(\bigcup_{i=1}^{n} E_i\right) \leq^{union\ bound}$$

$$\leq \sum_{i=1}^{n} \Pr(E_i) <_{lemma} n * \frac{1}{n^3} = \frac{1}{n^2}$$

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

$\Rightarrow T_{RQS}(n) = O(nlog(n))$ w.h.p.

Suppose $p$ is always the median of $S$; then

$$T_{RQS}(n) = \begin{cases} 2T_{RQS}\left(\frac{n}{2}\right) + O(n), & n > 1 \\ 0, & n \leq 1 \end{cases}$$

So, basically:

- there are $n$ nodes (excluding leaves associated to $\emptyset$) $\Rightarrow \leq n$ paths root-leaf
- we will show these paths are not so long; using any method (Master theorem or whatever), we get the same height of the tree

$$T_{RQS}(n) =^{Master\ Theorem} O(n\log(n))$$

However, $p$ is the median with probability $\frac{1}{n}$, which is very low.

(The following part is useful for the exam)

The event $E$ can be characterized as "in the first $l = a * \log_{\frac{4}{3}}(n)$ nodes of $P$ there have been $< \log_{\frac{4}{3}}(n)$ lucky choices". I'm studying this last event:

- $X_i, 1 \leq i \leq l = a * \log_{\frac{4}{3}}(n)$
- $X_i = 1$ if at the $i^{th}$ vertex of $P$ there is a lucky choice of the pivot
- $\Pr(X_i = 1) = \frac{1}{2}\ \forall i$
- $X_i$ are independent

We want the probability of $P(\sum_{i=1}^{l} X_i) < \log_{\frac{4}{3}}(n)$ to be bound (and to be very low).

Given $X = \sum_{i=1}^{l} X_i$, its expected value is as follows:

$$\mu = E[X] = E[\sum_{i=1}^{l} X_i] = \sum_{i=1}^{l} E[X_i] = \sum_{i=1}^{l} \frac{1}{2} = \frac{l}{2} = \frac{a}{2}\log_{\frac{4}{3}}(n)$$

Now, let's apply the following Chernoff bound:

$$\Pr(X < (1-\delta)\mu) < e^{\frac{-\mu\delta^2}{2}}, 0 < \delta \leq 1$$

$$\downarrow$$

$$(1-\delta)\mu = \log_{\frac{4}{3}}(n)$$

$$(1-\delta)\frac{a}{2}\log_{\frac{4}{3}}(n) = \log_{\frac{4}{3}}(n)$$

One possible choice is $a = 8, \delta = \frac{3}{4}$.

$$\Pr\left(X < \log_{\frac{4}{3}}(n)\right) < e^{-\frac{8}{4} * log_{\frac{4}{3}}(n) * \frac{9}{16}}$$

$$= e^{-\frac{8}{4} * log_{\frac{4}{3}}(n) * \frac{9}{8}}$$

$$< e^{-log_{\frac{4}{3}}(n)}$$

$$= e^{\frac{-ln(n)}{ln\left(\frac{4}{3}\right)}}$$

$$= \left(e^{-ln(n)}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)}}$$

$$= \left(\frac{1}{n}\right)^{\frac{1}{\ln\left(\frac{4}{3}\right)} \simeq 3{,}47}$$

$$< \frac{1}{n^3}$$

## 25.3 Applications

### 25.3.1 Exit polls

The first to be analyzed here is exit polls: approximate the percentage (%) of voters that in an election voted for one of the available options, without counting all votes.

How does it work?

- Some votes are drawn at random, which will go to represent the approximate solution.
- The idea is that *enough* votes should be drawn
  - which assures me the goodness of the solution w.h.p.

One urn $U$ (container) with $n$ balls (which represent parties), both white and black.

*Goal*: approximate the true value of white balls $\alpha * n$.

*Assumption*: we know that there are $\alpha_{min} * n$ white balls

To determine $\alpha$:

- to do this with a *deterministic* algorithm (exact), the complexity is $\Omega(n)$
- to do this with a *randomized approximated* algorithm, the complexity if $O(\log(n))$
  - and that's why we can do exit polls!

The algorithm will output a quantity $\beta$ such that $\Pr\left(\frac{|\beta-\alpha|}{\alpha} > \epsilon\right)$ is very low.

- with $\frac{|\beta-\alpha|}{\alpha}$ being the relative error
- while $\epsilon$ being the confidence threshold
  - an example of very low value is e.g., $< \frac{1}{n^2}$.

```
$APPROXIMATE\_\alpha\ (U, \epsilon, \alpha_{min})$
    $n = |U|$          $//\ balls\ present$
    $k = f(n, \epsilon, \alpha_{min})$        $//\ n°\ of\ extractions\ to\ be\ determined\ in\ the\ analysis$
    $x = 0$
    repeat $k$ $times$
        $p = RANDOM(U)$
        $if\ color(p) = white\ then\ x{+}{+}$
    return $\frac{x}{k}$
        $\downarrow$
        $\beta$ (value which approximates $\alpha$)
```

Complexity is (very fast dependent on the number of samples) → $O(k)$.

Main question: "What's the value of $k$ that guarantees the high probability"?

- $k$ indicator random variables
- $X_i = 1$ if the extracted ball is white (of course, 0 otherwise)
- $\Pr(X_i = 1) = \alpha$ (the parameter is not known, fraction voting basically)
- $X = \sum_{i=1}^{k} X_i$, which is the n° of extracted white balls (estimate)
- $\mu = E[X] = E[\sum_{i=1}^{k} X_i] = \sum_{i=1}^{k} E[X_i] = k\alpha$

*Event*: (written in the form of approximate Chernoff bound - represents the case where our estimate $\beta$ (which is $\frac{x}{k}$) differs from the true value $\alpha$ by more than a relative error $\epsilon$.

$$"\frac{|\beta - \alpha|}{\alpha} > \epsilon" = "\frac{\left|\frac{X}{k} - \alpha\right|}{\alpha} > \epsilon"$$

$$= "\underbrace{\frac{|x - \alpha k|}{\alpha k}} > \epsilon"$$

After isolating the expected value, we'll use this Chernoff bound:

$$\Pr(|X - \mu| > \epsilon\mu) < 2\underbrace{e^{\frac{-\mu\epsilon}{2}}}_{\text{want: } \sim \frac{1}{n^2}}, 0 < \epsilon \leq 1$$

*Issue*: $\alpha$ is unknown ⇒ use $\alpha_{min}$ instead (so, it's a lower bound for $\alpha$ and so indeed $a_{min} \leq \alpha$).

$$2e^{\frac{-k\alpha\epsilon^2}{2}} \leq 2e^{\frac{-k\alpha_{min}\epsilon^2}{2}} \to k = \frac{2}{n^2}$$

$$-\frac{k\alpha_{min}\epsilon^2}{2} = -\ln(n^2) \to e^{-\ln(n^2)} = \frac{1}{n^2} \Rightarrow k = \frac{2\ln(n^2)}{\alpha_{min}\epsilon^2} = O\left(\frac{\log(n)}{\epsilon^2}\right)$$

So, we exponentially decreased the complexity of the deterministic algorithm maintaining a relative margin of error which w.h.p. is very small.

### 25.3.2 Load balancing

The problem is also called "balls-and-bins" and it can be described as follows:

- $n$ servers
- $n$ jobs/requests, that arrive one by one

There are some *issues* however:

- distributed environment, so no central control
- limited information, where we don't know the servers' loads (latency)

*Goal*: minimize max load over the $n$ servers

*Simple algorithm*: assign each job to a server chosen uniformly at random.

- This is definitely more efficient than maintaining some state and/or statistics, which might cause slight delays – the policy is simple and lightweight

*Theorem (famous result)*: if $n$ inputs are assigned uniformly at random to $n$ servers, then with probability $\geq 1 - \frac{1}{n}$ every server has $\leq \frac{3\ln(n)}{\ln(ln(n))}$ requests, assuming sufficiently high $n$.

*Proof*:

- Consider a fixed server
  - $X_i = 1$ if the $i^{th}$ job/request gets assigned to that server (one for every request)
  - $\Pr(X_i = 1) = \frac{1}{n}$
  - $X_i$'s are independent
  - $X = \sum_{i=1}^{n} X_i =$ load of that server (quality to be analyzed)
  - $\mu = E[X] = \sum_{i=1}^{n} E[X_i] = n * \frac{1}{n} = 1$
    - in words, $\mu =$ n. of requests on average a server gets

Now, we will study $X$ in high probability. We'll use the following (Chernoff lemma in original version, so the strongest version. If we apply the weaker ones, we obtain only a single $ln$ factor, this does a tight approximation):

$$\Pr(X > (1+\delta)\mu) < \left(\frac{(e^{\delta})}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

$$\downarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} \Rightarrow \delta = \frac{3\ln(n)}{\ln(\ln(n))} - 1$$

What we want to claim exactly is that the probability of a server exceeding the specified load is at most $\frac{1}{n^2}$:

$$\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}} \leq^{?} \frac{1}{n^2}$$

$$\Updownarrow \quad \text{(take logs of both sides)}$$

$$\delta - (1+\delta)\ln(1+\delta) \le -2\ln(n)$$

$$\Updownarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} - 1 - \frac{3\ln(n)}{\ln(\ln(n))}\ln\left(\frac{3\ln(n)}{\ln(\ln(n))}\right) \le -2\ln(n)$$

$$\Updownarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} - \frac{1}{\ln(n)} - \frac{3}{\ln(ln(n))}(\ln(3) + \ln(\ln(n)) - \ln(ln(ln(n)))) \le -2\ln(n)$$

$$\Updownarrow$$

$$\frac{3\ln(n)}{\ln(\ln(n))} - \frac{1}{\ln(n)} - \frac{3}{\ln(ln(n))} - 3 + \frac{3\ln(\ln(\ln(n)))}{\ln(ln(n))} \le -2$$

$$\Updownarrow \quad n \text{ sufficiently high}$$

$$O(1) + O(1) + O(1) - 3 + O(1) \le -2 \checkmark$$

Now, let's apply the union bound (which allows to sum up for all the probabilities for every server) to see that the same is true for every server <u>simultaneously</u>:

$E_i$ = the $i^{th}$ server gets more than $\frac{3\ln(n)}{\ln(\ln(n))}$ requests

$$\Pr\left(\exists\ server\ that\ gets\ more\ that\ \frac{3\ln(n)}{\ln(\ln(n))}\ requests\right) = \Pr\left(\bigcup_{i=1}^{n} E_i\right) \le_{union\ bound} \sum_{i=1}^{n}\Pr(E_i) = n * \frac{1}{n^2} = \frac{1}{n}$$

In other words, the probability that no server gets more than $\frac{3\ln(n)}{\ln(\ln(n))}$ jobs is $> 1 - \frac{1}{n}$.