ALGORITHM & COMPLEXITY

Handouts

Author: Pietro Miotto

Please note that these handouts may contain conceptual errors, typos and misleading information. Always [illegible] something that does not "click" in your mind.
I also strongly advice to integrate these notes with David Mousse's and lesson's slides.
The main idea of this handout is to help you understand the concept of the subject quickly, making them easier to grasp, understand and study.

Stable Matching Problem

This is a fast summary intended for studying for the final exam, no proofs are written down and some concepts are skipped.

Stable Marriage

The problem. We may have two groups of entities where we wish to make an assignment from one to the other and where each side has some **notion of preference**. The goal is to produce a pairing that is in some sense "**stable**" in the sense that matched pairs should not have an obvious incentive to split up in order to form a different partnership. The concept of stable is that there should be no man who can say to another woman, "We each prefer each other to our assigned partners—let's elope!" If no such instability exists, the pairing is said to be stable.

Definition 1: Given a pair of sets X and Y, a *matching*, is a collection of pairs (x, y), where $x \in X$ and $y \in Y$, and each element of X appears in at most one pair, and each element of Y appears in at most one pair. A matching is *perfect* if every element of X and Y occurs in some pair. (**Beware:** Perfectness in a matching has nothing to do with optimality or stability. It simply means that everyone has a mate.)

Definition 2: Given sets X and Y of equal size and a preference ordering for each element of each set, a perfect matching is *stable* if there is no pair (x, y) that is *not* in the matching and x prefers y to its current match and y prefers x to its current match.

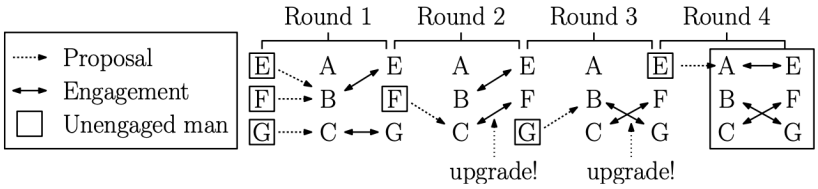
NW. A matching is said to be:

- **stable:** if two element in a matched pair can both find a better matching
- **perfect:** if every element is matched. Perfectness in a matching has nothing to do with optim ility. It s tha veryone ha a ma

G e Sh oley Algor m

The G Sh Algorith

```
// Inp  . zn pre ence l ts, each consisti of  names.
// Out t: A match g tha pairs each man wit ea woman.
Initia y all men d all omen are unengaged
while here is an henga asn't yet oposed to very
    Let m be any such man
    Let w be the highest woman on his list to whom he has not yet proposed
    if (w is unengaged) then she accepts ((m, w) are now engaged)
    else {
        Let m' be the man w is engaged to currently
        if (w prefers m to m') {
            Break off the engagement (m', w)
            Create the new engagement (m, w) (upgrade)
            Man m' is now unengaged
        }
    }
}
```



Correctness of the GS Algorithm. I skip the proof since are not asked in the exam.

Lemma1. Once a woman becomes engaged, she remains engaged for the remainder of the algorithm (although her mate may change), and her mate can only get better over time in terms of her preference list. *Lemma 1 follows from the fact that a woman only breaks off an engagement to form a new one to a man of higher preference.*

Lemma2. The mates assigned to each man decrease over time in terms of his preference list. Lemma 2 follows from the fact that each man makes offers in decreasing preference order.

Lemma3. The GS Algorithm terminates after at most (i.e. no more than) n2 iterations of the while loop.

Proof: Consider the pairs (m, w) in which man m has not yet proposed to woman w. Initially there are n^2 such pairs, but with each iteration of the while loop, at least one man proposes to one woman. Once a man proposes to a woman, he will never propose to her again (by Lemma 2). Thus, after n^2 iterations, no one is left to propose.

Lemma4. On termination of the GS algorithm, the set of engagements form a perfect matching. Recall that perfect means that every item is matched.

Proof: Every time we create a new engagement we break an old one. Thus, at any time, each woman is engaged to exactly one man, and vice versa. The only thing that could go wrong is that, at the end of the algorithm, some man m is unengaged after exhausting his list. Since there is a 1-to-1 correspondence between engaged men and engaged women, this would imply that some woman w is also unengaged. From Lemma 1 we know that once a woman is asked, she will become engaged and will remain engaged henceforth (although possibly to different mates). This implies that w has never been asked. But she appears on m's list, and therefore she must have been asked, a contradiction.

Lemma5. The matching output by the GS algorithm is a stable matching. (proof skipped)

Efficiency of Stable Matching Algorithm. As observed in Lemma3, the algorithm performs at maximum n^2 matches, i.e. runs in $O(n^2)$ time. But note that when we express running time, we do so in terms of the input size. In this case, the input for n men and n women consists of 2n preference lists, each consisting of n elements. Thus the input size is $N = 2n^2$. Since the algorithm runs in $O(n^2)$ time, this is really a linear-time algorithm!

To better understand: Since the time complexity of the algorithm is $O(n^2)$ and the input size is also proportional to n^2 (i.e. $N = n^2$ is proportional to n^2), the algorithm's time complexity can be described as linear in terms of the input size N — in other words, the algorithm runs in O(N) time, where N is the size of the input. This means that the algorithm's running time grows linearly with respect to the size of the input.

Time Complexity and Recursive Formulation: Algorithm Analysis' Mathematical Background

Big-O Notation
The purpose of the notation is to allow us to ignore less important elements, such as constant factors, and focus on important issues, such as the growth rate for large values of n. The final **goal** is to **compare functions**, focusing on their growth rate and ignoring constant factors.

Notation	Relational Form	Limit Definition
$f(n)$ is $o(g(n))$	$f(n) \prec g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n)$ is $O(g(n))$	$f(n) \preceq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ or 0
$f(n)$ is $\Theta(g(n))$	$f(n) \approx g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$
$f(n)$ is $\Omega(g(n))$	$f(n) \succeq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ or ∞
$f(n)$ is $\omega(g(n))$	$f(n) \succ g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

The $\preceq$ (and the other symbols) expresses a sort of fuzzy "≤" relation between functions, where by fuzzy, we mean that constant factors are ignored and we are only interested in what happens as n tends to infinity.
For example, $f(n)$ $g(n)$ is summarising the main definition of the Big-O notation, i.e.

$f(n)$ is $O(g(n))$ if $\exists$ constants $c > 0$ and $n_0 \geq 0$ such that
$f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$

Recall the following limits facts to evaluate the above fractions

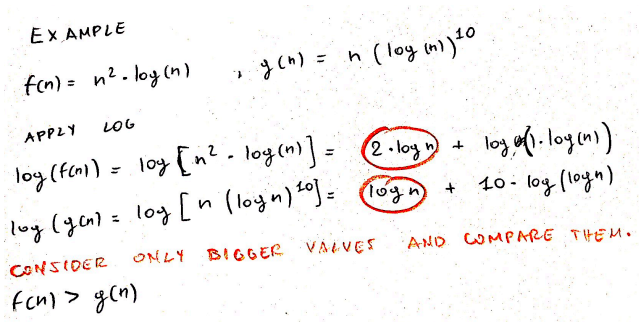
- For $a, b > 0$, $\lim_{n\to\infty} \frac{(\log n)^a}{n^b} = 0$ (polynomials grow faster than polylogs).
- For $a > 0$ and $b > 1$, $\lim_{n\to\infty} \frac{n^a}{b^n} = 0$ (exponentials grow faster than polynomials).
- For $a, b > 1$, $\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = c \neq 0$ (logarithm bases do not matter).
- For $1 < a < b$, $\lim_{n\to\infty} \frac{a^n}{b^n} = 0$ (exponent bases do matter).

Comparing Functions' complexities

Recall always that: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

Also, recall that **log bases do not matter, while exponent base do matter.**

The quickest way to compare the complexities of two functions (when we are unable to spot it at a first glance), is to apply logarithms to both functions. On the right, the fundamental rule of logarithms to keep in mind.



- $\log(ab) = \log(a) + \log(b)$
- $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
- $\log(a^b) = b \cdot \log(a)$
- $a^{\log_c(b)} = b^{\log_c(a)}$
- $a^b = n \implies b = \log_a(n)$

Recurrences

A recurrence is a **function defined** recursively **in terms of itself**. So is when a function calls itself or a "part" of itself (where by part we mean the same function is called on a part of the input). Here I include an example (useful to refresh the concept of recurrences, but in this course we are not asked to compute them in this way, so do not rely to much on this example).

```
function factorial(n)
    if n == 0 then
        return 1
    else
        return n× factorial(n − 1)
    end if
end function
```

Time Complexity Computation

Let $T(n)$ be the time complexity of the `factorial` function when called with input n.

- **Base Case**:
$$T(0) = O(1)$$
When $n = 0$, the function immediately returns 1, which is a constant-time operation.
- **Recursive Case**:
$$T(n) = T(n-1) + O(1)$$
When $n > 0$, the function performs a multiplication (which is $O(1)$) and then makes a recursive call to compute `factorial`(n-1).
- **Solving the Recurrence**:

$$T(n) = T(n-1) + O(1)$$
$$T(n-1) = T(n-2) + O(1)$$
$$\vdots$$
$$T(1) = T(0) + O(1)$$

Adding up these equations, we get

$$T(n) = T(0) + n \cdot O(1)$$

Since $T(0) = O(1)$, we have:

$$T(n) = O(1) + n \cdot O(1) = O(n)$$

Conclusion: The time complexity $T(n)$ of the `factorial` function is $O(n)$. This means that the function has a linear time complexity with respect to the input n.

There are several ways to compute a recursive function complexity, the mains are:
- by substitution
- by recursive tree
- by **master theorem**
- by akra-bazzi method

In this course we see only a simplification of the master theorem, which can be used to compute most of the cases (not all of them, in that case you should rely on the original master theorem).

Theorem: (Simplified Master Theorem) Let $a \geq 1$, $b > 1$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT(n/b) + cn^k,$$

defined for $n \geq 0$.

Case 1: $a > b^k$ then $T(n)$ is $\Theta(n^{\log_b a})$.

Case 2: $a = b^k$ then $T(n)$ is $\Theta(n^k \log n)$.

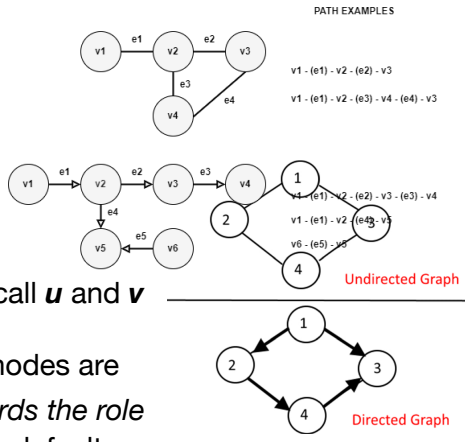
Case 3: $a < b^k$ then $T(n)$ is $\Theta(n^k)$.

Example

$$\begin{aligned} T(n) &= 1 && \text{if } n = 1, \\ T(n) &= 2T(n/2) + n && \text{if } n > 1. \end{aligned}$$

Using this version of the Master Theorem we can see that in our recurrence $a = 2$, $b = 2$, and $k = 1$, so $a = b^k$ and Case 2 applies. Thus $T(n)$ is $\Theta(n \log n)$.

Chapter 3 - Graphs



A **graph** is a way of encoding pairwise relationships among a set of objects. A graph consists of:

- a collection **V** of nodes
- a collection **E** of edges, each of which *joins* two of the nodes. An edge ***e*** ∈ E , therefore, can be represented as a *two-elements subset (sottoinsieme)* of V -> ***e* = {*u*, *v*}** for some ***u*,*v*∈*V***, where we call ***u*** and ***v*** the *ends* of *e* (**u** and **v** are nodes).

A **directed graph** (or **digraph**) is a graph where the edges between nodes are **directed edges**, i.e. each e' ∈ E' is an ordered pair (u, v). *In other words the role of u and v are not interchangeable, thus u is called tail and v head.* By default, however, the term "graph" will mean an undirected graph

degree in Undirected Graphs: is how many edges are connected to the vertex without considering their direction.
degree in Directed Graphs: is divided in **in-degree** (number of edges "entering" the vertex) and **out-degree** (number of edges "outing" the vertex)

Examples of "applied graphs":

- **Transportation networks:** nodes are the stations/airports, edge the flight/travel
- **Communication networks:** a node for each computer and an edge joining *u* and *v* if there is a direct physical link or define a node to be the set of all machines controlled by a single Internet servic ith an *v* a *v* if there is dire *peer* *p* tween hem
- **nfor tion netw ks:** des correspond Web pa es and here an edge from *u* *v* if *u* as a perlink *v*. The of the raph i crucial re
- **Socia Network** n this ase is useful to de e th notion o *ffiliat n*: given a set *X* o pe le and a et *Y* of org izati s, we could define a e e betwe *u* ∈ *X* nd *v* ∈ *Y* if perso belon to organization *v*

In a graph:

Number of edges: $0 \leq m \leq \binom{n}{2} = n(n-1)/2 \in O(n^2)$.

Sum of degrees: $\sum_{v \in V} \deg(v) = 2m$.

In a digraph:

Number of edges: $0 \leq m \leq n^2$.

Sum of degrees: $\sum_{v \in V} \text{in-deg}(v) = \sum_{v \in V} \text{out-deg}(v) = m$

n= number of nodes/vertex
m = number of edges
So, in a undirected graph:
m is included between **0** and **n(n-1)/2.**

Sparsity/Density of a graph: A graph is sparse if m is O(n). Else it is dense

Paths

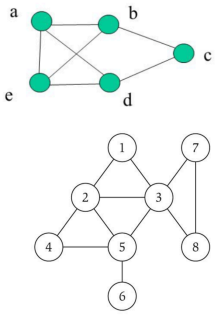
Path for undirected graphs: We define a *path* in an undirected graph $G=(V,E)$ to be a sequence P of nodes $v_1,v_2,\ldots,v_{k-1},v_k$ with the property that each consecutive pair v_i , v_{i+1} is joined by an edge in G. We say that an undirected graph is ***connected*** if, for every pair of nodes u and v, there is a path from u to v.

Path for directed graphs: We say that a directed graph is *strongly connected* if, for every two nodes u and v, there is a path from u to v and a path from v to u.

Distance: we define the *distance* between two nodes u and v to be the minimum number of edges in a u-v path

Simple path: is a path were all nodes are **distinct** (i.e. you don't encounter the same note twice).

Cycle: a closed Path
– Example: (a, d, c, b, e, a) is a cycle



cycle C = 1-2-4-5-3-1

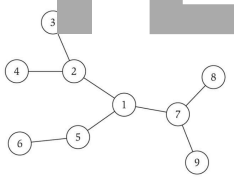
NW. Loops and Cycles are not the same thing:

- **(self-)loop:** loop in a graph refers to an edge that connects a vertex to itself, i.e. is an **edge** (not a path, or a path on length 1) that starts in vertex v and ends in the same vertex v. They are mainly found in directed graphs.*Self-loop edges are allowed for directed graphs (not allowed in undirected graphs)*
- **cycle:** cycle is a **path** that starts and ends in the same vertex. A cycle is a path where all nodes are **distinct** except for the **first** and the **last**.

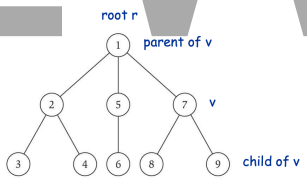
Trees

We say that an **undirected graph** is a *tree* if it is **connected** and does **not contain a cycle**
In other word a tree is an **acyclic** (connected) graph.
An acyclic undirected graph is **disconnected** (i.e. there is a path between every pair of vertices) is a **(free) tree** (where tree means "unrooted"), if it is **disconnected** is called **forest**, since it is collection of individual (free) trees, not connected between each other.
- An acyclic directed graph (connected) is called **DAG**.

For thinking about the structure of a tree T, it is useful to root it at a particular node r . Rooted trees are fundamental objects in computer science, because they encode the notion of a hierarchy. Rooting a tree T can make certain questions about T conceptually easy to answer. For example, given a tree T on n nodes, how many edges does it have? Every n-node tree has exactly n – 1 edges. Thus, a tree with 10 nodes has 10-1 edges (9).



a tree



the same tree, rooted at 1

(3.2) *Let G be an undirected graph on n nodes. Any two of the following statements implies the third.*

(i) G is connected.
(ii) G does not contain a cycle.
(iii) G has n – 1 edges.

Basically is saying that, if a graph is connected and has no cycle (i.e. is a tree), it also has n-1 edges. Or, if a graph has n-1 edges and does not contain cycle, is connected (so it's a tree).

In a rooted tree:
- The **depth** of a node: number of edges from the root to the node.
- The **height** of a node: number of edges from node to deepest leaf.
- The **height** of a tree is a height of the root.

Shortest Path: Dijkstra O(m log n) or Θ((n+m)logn)

BFS can be used for **single-source** shortest path (i.e. it can compiute the shortest path from a source vertex to all other vertices), assuming that the graph is **not weighted**. But it does so in **O(V+E)** time.

NW.

- **Length of a path:** In unweighted graphs, length of a path is **the number of edges** it crosses. In weighted graph length of a path is the **sum of weights** along the path.
- **Distance** between two vertices: is the minimum length of any path between the two vertices (i.e. is the length of the shortest path between the two). This is denoted by **δ(u,v)**, which is the **real distance** between two vertices (the estimate of the real distance is noted as **d[v]**).

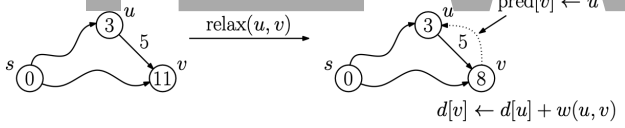
Single-Source Shortest Path: given a digraph G = (V,E) with numeric edge weights and a distinguished source vertex, s ∈ V, the objective is to determine the distance δ(s,v) from s to every vertex v in the graph.

Dijkstra main assumptions: graph is directed and weights are positive.

The basic structure of Dijkstra's algorithm is to maintain an estimate of the shortest path for each vertex, call this d[v]. Intuitively d[v] stores the length of the shortest path from s to v that the algorithm currently knows of. Indeed, there will always exist a path of length d[v], but it might not be the ultimate shortest path. Initially, we know of no paths, so d[v] = ∞, and d[s] = 0. As the algorithm proceeds and sees more and more vertices, it updates d[v] for each vertex in the graph, until all [illegible] "converge" [illegible] true shortest distances.

Relaxation: The process by which an estimate is updated sometimes called [illegible]. In Dijkstra, every time you "expose" a new node **u**, we "relax" all vertices in **u**'s adjacency list —> if you can see that your solution is not yet matched an optimum value, then push it a little closer to the optimum. This means that, if you discover a path from s to v shorter than d[v], then you need to update d[v]. It is not hard to see that if we perform relax(u, v) repeatedly over all edges of the graph, the d[v] values will eventually converge to the final true distance value from s. The cleverness of any shortest path algorithm is to perform the updates in a judicious manner, so the convergence is as fast as possible. In particular, the best possible would be to order relaxation operations in such a way that each edge is relaxed exactly once. Assuming that the edge weights are nonnegative, Dijkstra's algorithm achieves this objective.

```
re   (u, v)
if (d[u]   (u, v)   d[v]) {    // is the   th throu   u s   ter?
    d[v]   [u] +                ves, the   ake i
    pred   = u                  // record ti   we   throug
}
```



Dijkstra's Algorithm

```
dijkstra(G,w,s) {
    for each (u in V) {                 // initialization
        d[u] = +infinity
        mark[u] = undiscovered
        pred[u] = null
    }
    d[s] = 0                            // distance to source is 0
    Q = a priority queue of all vertices u sorted by d[u]
    while (Q is nonEmpty) {             // until all vertices processed
        u = extract vertex with minimum d[u] from Q
        for each (v in Adj[u]) {
            if (d[u] + w(u,v) < d[v]) { // relax(u,v)
                d[v] = d[u] + w(u,v)
                decrease v's key in Q to d[v]
                pred[v] = u
            }
        }
        mark[u] = finished
    }
    [The pred pointers define an ''inverted'' shortest path tree]
}
```

How Does Dijkstra works?

Dijkstra's algorithm operates by maintaining a subset of vertices, S ⊆ V , for which we claim we "know" the true distance, that is d[v] = δ(s, v). Initially S = ∅, the empty set, and we set d[s] = 0 and all others to +∞. One by one, we select vertices from V \ S to add to S. How do we select which vertex among the vertices of V \ S to add next to S? The best way in which to perform relaxations is by increasing order of distance from the source —> we take the vertex of V \ S for which d[u] is minimum. That is, we take the unprocessed vertex that is closest (by our estimate) to s. Relaxing the not-yet-in-S vertex with the minimum d[] is proven to yield the final distance value δ. In order to perform this selection efficiently, we store the vertices of V \ S in a priority queue (e.g. a heap), where the key value of each vertex u is d[u].

Dijkstra Correctness (Lemma) ——>

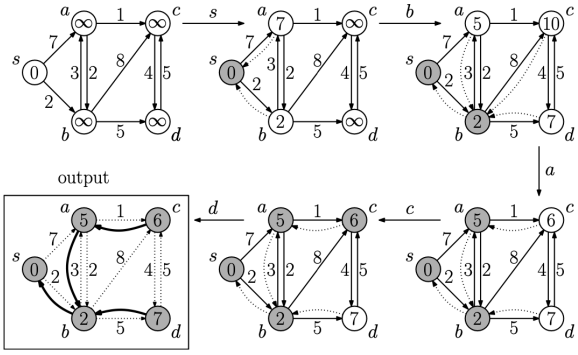


Fig. 26: Dijkstra's Algorithm example.

Lemma: When a vertex u is added to S, $d[u] = \delta(s,u)$.

Proof: Suppose to the contrary that at some point Dijkstra's algorithm *first* attempts to add a vertex u to S for which $d[u] \neq \delta(s,u)$. By our observations about relaxation, $d[u]$ is never less than $\delta(s,u)$, thus we have $d[u] > \delta(s,u)$. Consider the situation just prior to the insertion of u, and consider the true shortest path from s to u. Because $s \in S$ and $u \in V \setminus S$, at some point this path must first jump out of S. Let (x,y) be the first edge taken by the shortest path, where $x \in S$ and $y \in V \setminus S$ (see Fig. 27). (Note that it may be that $x = s$ and/or $y = u$).

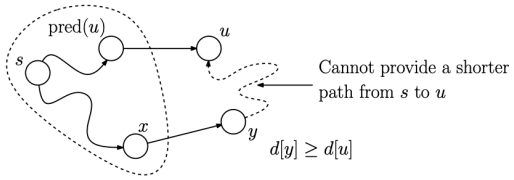


Fig. 27: Correctness of Dijkstra's Algorithm.

Because u is the first vertex where we made a mistake and since x was already processed, we have $d[x] = \delta(s,x)$. Since we applied relaxation to x when it was processed, we must have

$$d[y] \;=\; d[x] + w(x,y) \;=\; \delta(s,x) + w(x,y) \;=\; \delta(s,y).$$

Since y appears before u along the shortest path and edge weights are nonnegative, we have $\delta(s,y) \leq \delta(s,u)$. Also, because u (not y) was chosen next for processing, we know that $d[u] \leq d[y]$. Putting this together, we have

$$\delta(s,u) \;<\; d[u] \;\leq\; d[y] \;=\; \delta(s,y) \;\leq\; \delta(s,u).$$

Clearly we cannot have $\delta(s,u) < \delta(s,u)$, which establishes the desired contradiction.

MST and cut

What is an MST? Given a connected, undirected graph G = (V, E), a **spanning tree** is an acyclic subset of edges T ⊆ E that connects all the vertices together. A spanning tree is also called a **free tree** (i.e. **connected, undirected, and acyclic graph**). The **minimum** spanning tree (**MST**) is a **spanning tree** of minimum weight/cost, where the weight/cost of a spanning tree is calculated as the sum of edges' weights in the spanning tree.

NW. Not always there is exactly one unique minimum spanning tree for a given graph. A graph can have multiple MST trees if edges in a graph have distinct weights, there will be a distinct/unique MST for that graph.

Lemma: (i) A free tree with n vertices has exactly $n-1$ edges.

(ii) There exists a unique path between any two vertices of a free tree.

(iii) Adding any edge to a free tree creates a unique cycle. Breaking *any* edge on this cycle restores a free tree.

The intuition behind the greedy MST algorithms is simple, we maintain a subset of edges A, which will initially be empty, and we will add edges one at a time, until A is a spanning tree. At every iteration we add to the subset an edge which is considered to be **safe**.

What is a safe edge?

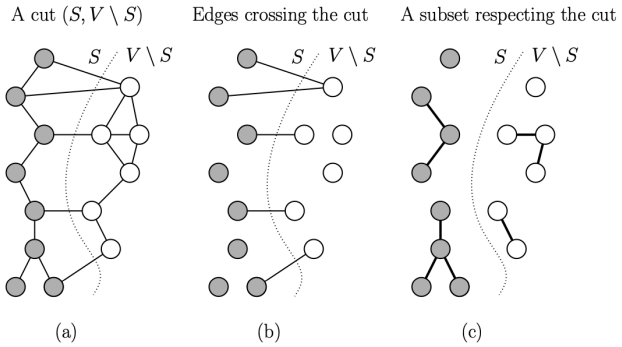


Fig. 29: MST-related terminology.

- A *cut* $(S, V \setminus S)$ is a partition of the vertices into two disjoint subsets (see Fig. 29(a)).
- An edge (u,v) *crosses* the cut if $u \in S$ and $v \notin S$ (see Fig. 29(b)).
- Given a subset of edges A, we say that a cut *respects* A if no edge in A crosses the cut (see Fig. 29(c)).

It is not hard to see why respecting cuts are important to this problem. If we have computed a partial MST, and we wish to know which edges can be added that do not induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate. An edge of E is a light edge crossing a cut, if among all edges crossing the cut, it has the minimum weight. Intuition says that since all the edges that cross a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice. This intuition is the base of the so called **MST lemma** (which is the base of

MST greedy algorithms).

MST Lemma: Let $G = (V, E)$ be a connected, undirected graph with real-valued weights on the edges. Let A be a viable subset of E (i.e. a subset of some MST), let $(S, V \setminus S)$ be any cut that respects A, and let (u, v) be a light edge crossing this cut. Then the edge (u, v) is *safe* for A.

What is a cut-set? The cut-set is the set containing all edges which traverse the cut.

Prim, Kruskal and Reverse-Delete

Kruskal's Algorithm Θ(mlogn) or O((n+m) logn)

Kruskal's algorithm works by attempting to add edges to the A in in- creasing order of weight (lightest edges first). If the next edge does not induce a cycle among the current set of edges, then it is added to A. If it does, then this edge is passed over, and we consider the next edge in order. Note that as this algorithm runs, the edges of A will induce a forest on the vertices. As the algorithm continues, the trees of this forest are merged together, until we have a single tree containing all the vertices.

The only tricky part of the algorithm is how to detect efficiently whether the addition of an edge will create a cycle in A. We could perform a DFS on subgraph induced by the edges of A, but this will take too much time. We want a fast test that tells us whether u and v are in the same tree of A. This can be done by a data structure (which we have not studied) called the disjoint set union-find data structure. This data structure supports three operations:

create(u): Create a set containing a single item v.

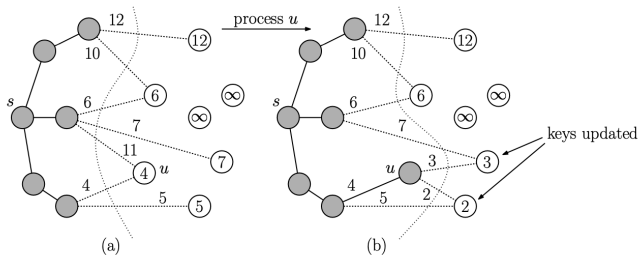
find(u): Find the set that contains a given item

union(u, v): Merge the set containing u and the set containing v into a common set.


```
Kruskal MST(G=(V,E), w) {
    A = {}                                  // initially A is empty
    Place each vertex u in a set by itself
    Sort E in increasing order by w
    for each ((u, v) in this order) {
        if (find(u) != find(v)) {           // u and v in different trees
            add (u, v) to A                 // join subtrees together
            union(u, v)                     // merge these two components
        }
    }
    return A
}
```

Prim's Algorithm Θ(mlogn) or O((n+m) logn)

Prim's algorithm is another greedy algorithm for computing minimum span- ning trees. It differs from Kruskal's algorithm only in how it selects the next safe edge to add at each step. Prim's algorithm looks very much like another greedy algorithm, called Dijkstra: Prim's algorithm builds the tree up by adding leaves one at a time to the current tree. We start with a root vertex s (it can be any vertex). At any time, the subset of edges A forms a single tree (in Kruskal's it formed a forest). We look to add a single vertex as a leaf to the tree. It is easy to see, that the key questions in the efficient implementation of Prim's algorithm is how to update the cut efficiently, and how to determine the light edge quickly. To do this, we will make use of a priority queue data structure, where each item is associated with a key value. The priority queue supports three operations:



insert(u, k)**:** Insert u with the key value k in Q.

extract-min$()$**:** Extract the item with the minimum key value in Q.

decrease-key(u, k')**:** Decrease the value of u's key value to k'.

Knapsack Problem

- We have n objects
- Each with a weight w_i and a value v_i
- We have a sack of max weight W

→ we want to minimize W and maximize v_i

BEST WAY TO VISUALIZE IS TO REASON "TABULARLY"

ITEM (i) \ W (sack weight)	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	6
4	0	0	1	2	5	6	

$V[i, w]$ = value of item i having sack weight w

WE ALWAYS ASSUME ITEMS ORDERED BY WEIGHT

$OPT(i, w) = 0$ IF $i = 0$

$V[0, w] = 0$

$OPT(i, W) = OPT(i-1, W)$ IF $W_i > W$

ITEM i W[illegible] TOO [illegible] VALUE
O[illegible] PREVIO[illegible] WEI[illegible]T E[illegible]T OPTION,
VA[illegible] OF [illegible]AST [illegible] TO BEST FIT.

$V[1] = 0$ [illegible] $W_1 = 2$, bu[illegible]
we [illegible]e $W = 1$, so we [illegible]
to p[illegible]viou[illegible] which
is $V[0, 1] = 0$

ITEM i	WEIGHT W_i	VALUE V_i
1	2	1
2	3	2
3	4	5
4	5	6

$OPT(i, w) = \max\{OPT(i-1, w), V_i + OPT[i-1, w-w_i]\}$

We take the max betw. previous item value and current value + value of item that best fits in remaining space.

$V[3, 6] = \max\{V[2, 6], V_3 + V[2, 6-W_3]\}$

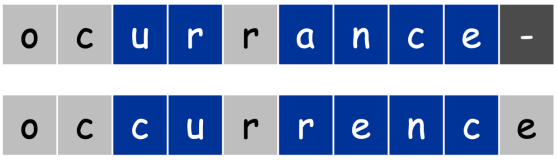
$\max\{3, 5+1\}$

↓

$V[2, 2]$

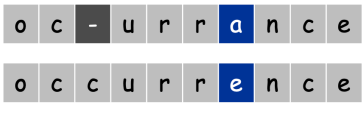
Sequence Alignment Θ(mn)

The problem of sequence alignment can be described as, given two strings, to find the alignment of min cost. Were by "alignment of minimum cost" we refer to a re-arrangement of the two sentences so that the number of gaps and mismatches is minimized.
Let's see an example:

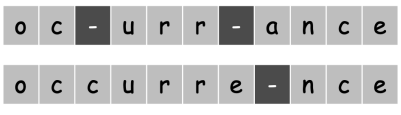


5 mismatches, 1 gap

Suppose we want to align these two words, **occurrence** and **occurrence**. When we compare them, without any re-arrangement, we find to have **5 mismatches** (i.e. xi≠yi) and one **gap** (i.e. blank space). Is there a way to re-arrange these two sentences (by re-arrange I mean to align the sequences as they are, by introducing gaps where necessary, rather than by rearranging the sequence elements). Thus, two different alignments of these sentences can be:



1 mismatch, 1 gap



0 mismatches, 3 gaps

Formal Definition of Alignment

Goal: Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find the **alignment of minimum cost**.

L... An ...gnment M is ...set of ...dered pairs x_i-y_j su... that each i... occ... in **at most** ...**e pai**... nd **no two pairs cro**...

Def. Th... air x_i-y_j and ... $y_{j'}$ cros... f i < i', but j > j'.

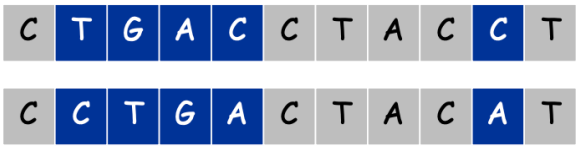
Thes... defini...ons simply mea... that a align...ent m...ns basically ass...ning...n item ... (item ... first sequence) ...th ... item ... (of th... second sequence ...sc... that:

- **we ca... align an i...m of ... with AT M... XIMUM O... : iter... sequen...**.
  The alignment of a letter can be with a blank space or just one letter.
- alignments **cannot cross**

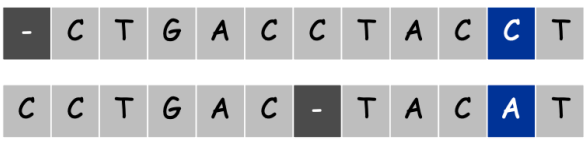
Formal Definition of Cost and Edit Distance

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

- **Gap penalty** δ; **mismatch penalty** α_{pq}.
- **Cost** of a given alignment **= sum of gap and mismatch penalties**



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$



$2\delta + \alpha_{CA}$

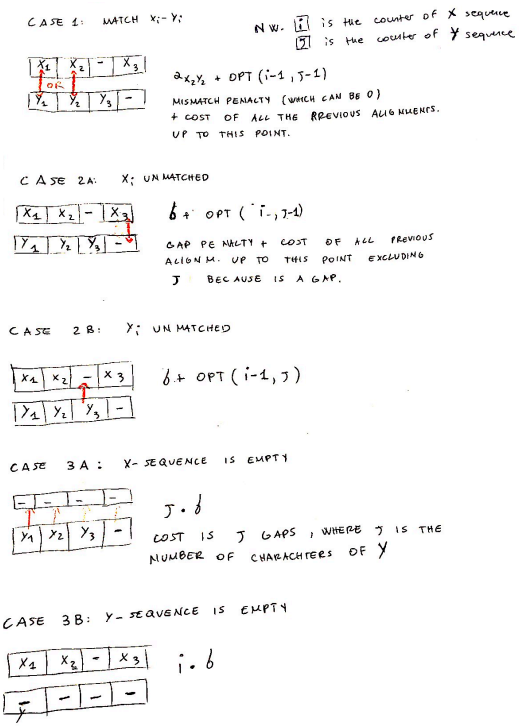
Each alignment is linked with a cost, which is the sum of all mismatch penalties and gap penalties. The **edit distance** between two strings is the minimum cost of any possible alignment between them. In other words, it is the smallest sum of gap and mismatch penalties among all possible alignments that transform one sequence into another.

Problem Structure and DP Formula

As with very DP formulation, we need to define some cases:

Def. **OPT(i, j)** = min cost of aligning strings $x_1\ x_2\ \ldots\ x_i$ and $y_1\ y_2\ \ldots\ y_j$.
Goal. Find **OPT(m,n)**

- Case 1: **Match x_i-y_j**.
  - pay mismatch for x_i-y_j + min cost of aligning strings $x_1\ x_2\ \ldots\ x_{i-1}$ and $y_1\ y_2\ \ldots\ y_{j-1}$,. OPT(i-1,j-1) + $\alpha_{x_i y_j}$
- Case 2a: leave **x_i unmatched**.
  - pay **gap for x_i** + min cost of aligning $x_1\ x_2\ \ldots\ x_{i-1}$ and $y_1\ y_2\ \ldots\ y_j$
- Case 2b: leave **y_j unmatched**.
  - pay **gap for y_j** + min cost of aligning $x_1\ x_2\ \ldots\ x_i$ and $y_1\ y_2\ \ldots\ y_{j-1}$



which leads to this formulation:

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

We take the minimum of the three options because we are basically [illegible] all possi[illegible]ent (i.e. matchin[illegible] characters, add[illegible] a sp[illegible]e to the X sequ[illegible]ce or ad[illegible]g a s[illegible]ace t[illegible]he Y sequence[illegible] find th[illegible]best[illegible]e. the one[illegible]ith m[illegible]imum cost). *Rec[illegible] that in [illegible] we a[illegible]ays [illegible] every possible [illegible]lution [illegible] cho[illegible]e the "bo[illegible]".*

Sequen[illegible] Alignme[illegible] Algo[illegible]hm (using memoi[illegible]tio[illegible])

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

Line up an i-letter and 0-letter word: use i gaps

[illegible]iven [illegible]e matrix M, we ca[illegible] [illegible]etriev[illegible] [illegible] [illegible]quence alignment (in term of words, not cost) by starting from M[m,n] and choosing the possibility among the three that matches with the cost we have. Or we can leave some hints while compiling, like storing which option we choose.

Analysis. $\Theta(mn)$ time and space.

Sequence Alignment in Linear Space

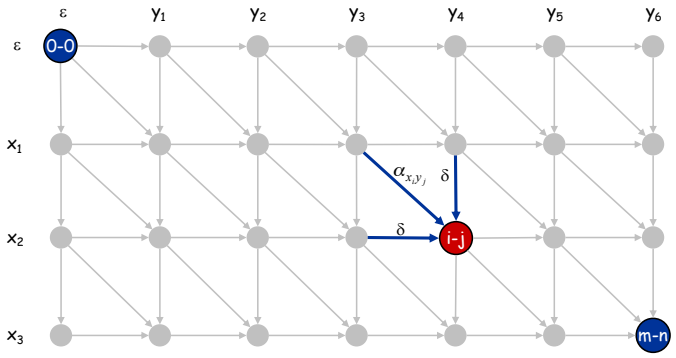
We Combine Dynamic Programming with Divide and Conquer to **avoid quadratic space** usage, i.e. to reduce the **space size** from **O(mn)** to **O(m+n)**. In other words, instead of using a **man** matrix to store the values, we can rely only on a **2xn** matrix. **NW.** The **time complexity is still O(mn).**

The key lays in the fact that every **DP matrix** can be rtperesented as a **DAG** (Directed Acyclic Graph), where each node is an entry, each column is a specific **j** and each row a specific **i**. In simpler terms, each column **j** contains all the possible alignments of letter **Yj** with all possible letters $\mathbf{X_1,X_2,...X_m}$.

Edit distance graph.

- A node for every M[i,j]
- An edge from M[i-1,j-1], M[i-1,j], M([i.j-1] to M[I,j] with weights

- Let **f(i, j)** be (length of) **shortest path from (0,0) to (i, j).**
- **Observation: f(i, j) = OPT(i, j).**



Each edge is assigned with a weight. As you see, for each node we have 3 incoming edges, which represent the three cases, i.e. having **i** matched with **j**, or having **i** matched with a **blank**, or having **j** matched with a **blank**.

Observation: we define **f(i,j)** , i.e. the **shortest path** from 0-0 to the specific matching (i,j) we are considering. This shortest path is proven to be equal to **OPT(i,j)**. This means that **all the nodes along the shortest path** from **0-0** to **m-n**, will represent the **optimal matching.** This means that **for every column, there is just one i** (the one on the shortest path) which is the optimal, i.e. the optimal matching of i-Yj that leads to the best edit distance of the entire sentences.

Now, let's introduce **g(i,j)**, which is exactly as f(i,j) but starting from m and going backwards to i-j. We know that **OPT(m,n) = min$_i$ {f(i, j) + g(i,** for **y** column.

This means that to find the optimal alignment cost $OPT(m, n)$, you should consider the sum of the cost to reach and the cost to continue optimally to (m, n), for every possible row i in the fixed column j.

In other words, to get the final cost of the best alignment, you just need one column j, and you need to find among all entries of this column, the one with the minimum f(i,j) + g(i,j). Recall that each f(i,j) and g(i,j) are costs of a shortest path. To simplify even better, we can say that:
The **divide and conquer** approach is used to efficiently reconstruct the **entire optimal alignment** between the sequences, not just to find the minimum cost. While the minimum cost can be found by examining a single column, the actual alignment (i.e., the specific matching of characters from the two sequences) requires more steps.

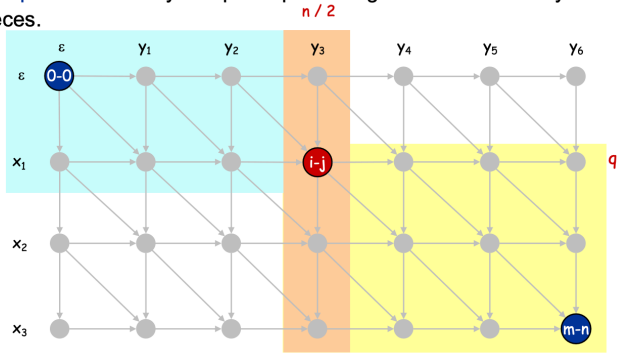
How do we proceed then?

1. We choose column **j=n/2**, i.e. the middle column (since later we want to apply a divide and conquer algorithm, so we want to split perfectly the problem in 2 sub-problems).
2. From this column, we compute f(•,n/2) + g (•,n/2) for every **i** element of the j=n/2 column. Then we find the item **q** which minimises {f(•,n/2) + g (•,n/2)}. In other words, q represents the node of the graph (inside the j=n/2 column) that shows the minimum cost, i.e. the OPT(m,n) (the cost of the best alignment of the two sequences). We can say that **q.value = min{f(•,n/2) + g (•,n/2)}.**
3. We know the i index of the minimum cost node in column j, which is q. So we know that the aligned pair x_q-$y_{n/2}$ (or x_q-gap or $y_{n/2}$- gap) is contained in

Divide: Compute the **value of f(i,n/2)** and **g(i,n/2)**, for **all i**, using **two** space-efficient DP algorithms (time O(mn), space O(m+n)).

- Find the index **q** that minimizes **f(i, n/2) + g(i, n/2)**.
- **Store pair x_q - $y_{n/2}$** (align x_q - $y_{n/2}$ or x_q-gap or $y_{n/2}$- gap).

Conquer: recursively compute optimal alignment in blue and yellow pieces.



Ford Fulkerson Algorithm (DM.18)

Why Greedy Fails. SLIDES

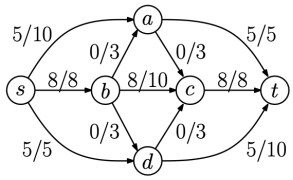
Residual Network: The key insight to overcoming the problem with the greedy algorithm is to observe that, in addition to increasing flows on edges, it is possible to decrease flows on edges that already carry flow (as long as the flow never becomes negative). A residual network is an exact copy of a flow network with the following edge variations:

- forward edges

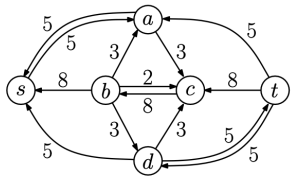
- backward edges

Basically each edge whose capacity is bigger than the current flow, is split into:

- a foward edge of **capacity = capacity - flow**. Basically the forward edge has as capacity the "residual", i.e. the possible fluid that could still pass trough the edge/ the "capacity left".
- a backward edge of **capacity = flow.** Conceptually, by pushing positive flow along the reverse edge (v, u) we are decreasing the flow along the original edge (u, v).



(a): A flow f in network G



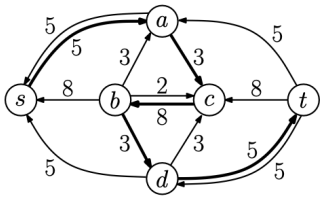
(b): Residual network G_f

NW. Observe that every edge of the residual network has **strictly positive capacity**. Note that each edge in the original network may result in the generation of up to two new edges in the residual network. Thus, the residual network is of the same asymptotic size as the original network.

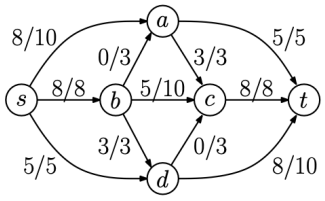
N 2. Th capacity o each dge in the residu network s calle its re dual capacity. T key o serva n about t resic l network is that *we ca ush flo thro h the residual ne work then we ush a addi nal amount of flow hrou the ori nal ne work*.

Augme ng Paths: CHEC SLIDES Consider a work G, le be a ow in G, and let G the associa residual wor **ting pat a simple th P Gf**. i.e is a path in the residual graph**.** The residual capacity (also called the bottleneck capacity) of the path is the minimum capacity of any edge on the path. I.e. we call **bottleneck capacity** the edge in the residual graph with the lowest capacity. It is denoted $\mathbf{c_f(P)}$. What does all of this means? Recall that all the edges of G_f are of strictly positive capacity, so $\mathbf{c_f(P) > 0}$. By pushing $\mathbf{c_f(P)}$ units of flow along each edge of the path, we obtain a valid flow in G_f, and by the previous lemma, adding this to f results in a valid flow in G of strictly higher value.

NW. By adding on forward edges, I add up to the total flow, by adding on backward edges, I reduce on the total flow. Look a the image. Here, suppose we have the augmenting path s-a-c-b-d-t (an augmenting path is a path from s to t in the residual graph, which can include backward edges). If i augment this path by the bootleneck (3): the edge s->a (originally 5/10) becomes 8/10, the edge b->c (originally 8/10) becomes 5/10.



(a): Augmenting path of capacity 3



(b): The flow after augmentation

Ford-Fulkerson: Start with a flow of weight 0, and then repeatedly find an augmenting path. Repeat this until no such path exists. This, in a nutshell, is the simplest and best known algorithm for computing flows, called the Ford-Fulkerson method (We do not call it an "algorithm," since the method of selecting the augmenting path is not specified. We will discuss this later.)

Ford-Fulkerson Network Flow

```
ford-fulkerson-flow(G = (V, E, s, t)) {
    f = 0 (all edges carry zero flow)
    while (true) {
        G' = the residual-network of G for f
        if (G' has no s-t augmenting path)
            break                                   // no augmentations left
        P = any-augmenting-path of G'               // augmenting path
        c = minimum capacity edge of P              // augmentation amount
        augment f by adding c to the flow on every edge of P
    }
    return f
}
```

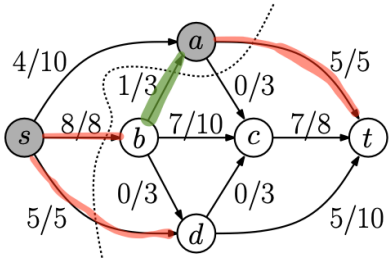
1.set flow on each edge to 0
2. Compute the residual network G'. **O(n+m) n=|V|, m=|E|**
3.If you find any augmenting (**O(n+m)**) path P in G', identify the bottleneck (= **c**), i.e. what is the capacity of the edge with the minimum capacity. If you find no augmenting path, stop.
4. For all edges of G, augment their flow by **c**. In the first iteration all flows = 0, so they will become f=0+c.
Repeat 2.,.3,.4.

To find the augmenting path, we can do perform either a DFS or BFS in the residual network starting at s and terminating as soon (if ever) t is reached.

Cuts: SLIDES(INCLUDE FLOW-VALUE LEMMA). To show that Ford-Fulkerson leads to the maximum flow, we need to formalize the notion of cuts. Let's first introduce the concept of **bottleneck**. Intuitively, the flow cannot be increased forever, because there is some subset of edges, whose capacities eventually become saturated with flow. Every path from s to t must cross one of these saturated edges, and so the sum of capacities of these edges imposes an upper bound on size of the maximum flow. Thus, these edges form a **bottleneck**. Their removal defines a partition separating the vertices that s can reach from the vertices that s cannot reach.
A cut is a partition of the graph G int two **disjointed sets X** and **Y**, where **X** contains the **source s** and **Y** contains the **sink t**. Note that you will have edges crossing from X to Y. All these edges form th e can **t fl from X t** as sur **> Y** the su o ows **<- Y** . Ma mat lly:

$$(X,Y) \quad \sum_{x \in X} \sum_{y \in} f(x,y) \quad \sum_{y \in Y} \sum_{x \in X} f(y,x)$$

Visually



$$X = \{s, a\}$$
$$Y = \{b, c, d, t\}$$
$$f(X,Y) = 5 + 0 - 1 + 8 + 5 = 17$$

Fl alue le a. Let f be any flow, and A, B) be s ut.
Th et flow across any cut is equal to amoun aving

$$\sum_{\text{out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = \quad).$$

Pf

$$v(f) = \sum_{e \text{ out of } s} f(e)$$
$$= \sum_{e \text{ out of } s} f(e) + 0$$

by flow conservation, all terms except v = s are 0

$$\rightarrow = \sum_{e \text{ out of } s} f(e) + \sum_{v \in A \setminus \{s\}} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right)$$
$$= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right)$$
$$= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e).$$

red ones: edges from **X –> Y**, **green ones:** edges from **Y –> X** (i.e. **X <– Y**). You subtract the sum of flows on X –> Y edges with the sum of flows on X <– Y edges. The result is the **flow(A,B).**

These definitions led to two **lemmas** (and relative proofs). *Both definitions relies on the "crossing edges" of the cut*, so the edges that goes from X –> Y and from Y –> X.

Lemma1. This lemma proves that the flow(A,B) = flow of the graph G.

Lemma: Let (X, Y) be any s-t cut in a network. Given any flow f, the value of f is equal to the net flow across the cut, that is, $f(X, Y) = |f|$.

Proof: Recall that there are no edges leading into s, and so we have $|f| = f^{\text{out}}(s) = f^{\text{out}}(s) - f^{\text{in}}(s)$. Since all the other nodes of X must satisfy flow conservation it follows that

$$|f| = \sum_{x \in X} (f^{\text{out}}(x) - f^{\text{in}}(x))$$

Now, observe that every edge (u, v) where both u and v are in X contributes one positive term and one negative term of value $f(u, v)$ to the above sum, and so all of these cancel out. The only terms that remain are the edges that either go from X to Y (which contribute positively) and those from Y to X (which contribute negatively). Thus, it follows that the value of the sum is exactly $f(X, Y)$, and therefore $|f| = f(X, Y)$.

Explanation of the proof. Basically, understand two things:

1. **s** is the only vertex which does not respect the **flow conservation** (i.e. number of flow-in = number of flow-out), while all the other nodes respect this rule.
2. All flow within nodes inside the same subset cancels out. If I have two nodes a,b in X, a sends 10 flow (+10) and be receives 10 flow (-10). So the flow cancels out. This is ensured by the flow conservation rule. Edges that cross from X to Y are the only edges that contribute to the net flow from X to Y because they represent flow leaving the set X and entering the set Y(or vice versa). These edges are not subject to internal cancellation within X or Y because the flow does not return to the same set—it moves across the cut to the other set.

That is why f(X,Y) = |f|.

Capacity of the cut: the capacity of the cut is the sum of capacity of crossing edges, but only the edges **X -> Y.**

Define the *capacity* of the cut (X, Y) to be the sum of the capacities of the edges leading from X to Y, that is,

$$c(X, Y) = \sum_{x \in X} \sum_{y \in Y} c(x, y).$$

So the **minimum cut** of a graph, is the graph in which the crossing edges (**X->Y**) are the one with lowest capacity.

Before introducing the famous **Max-Flow/Min-Cut Theorem** we need to introduce the **Weak Duality Corollary.**

Weak D[illegible]eak du[illegible] pr[illegible]es that **v(f) = ca[illegible]A,B).** [illegible] **o[illegible]eak d[illegible]lity** [illegible]oves, inste[illegible], the[illegible]llowing: If v(f)= [illegible]**p(A,B)**, **f[illegible] max[illegible]ow a[illegible] cut(A,B)** is th[illegible]**in cu**t. N[illegible] We[illegible] duality co[illegible]llary [illegible]oves only one di[illegible]ction, M[illegible]x-Flow[illegible]Min-C[illegible] proves both di[illegible]ctions [illegible] wea[illegible]uality p[illegible]es tha[illegible] **(A,B) [illegible] f = m[illegible]x flow[illegible]ut(A,[illegible].** Whi[illegible] Max Flow/Mi[illegible]Cut prove[illegible]hat **if [illegible](f) = cap(A,B) <=>[illegible] = m[illegible]x flow,[illegible]t(A,B[illegible] = min cut.**

We want to p[illegible] that $|f| = v(f) =$ [illegible](A, B). [illegible]*e: f and $v(f)$ are interchangeably, [illegible]b[illegible]efer to the flow of the ne[illegible]k.*

1. From flow-value lemma we get that

$$v(f) = \sum_{\text{out } A} f(e) - \sum_{\text{in } A} f(e)$$

i.e., the flow of a graph (or graph subset A) is the sum of all flow going out - all flow coming in.

2. We know that

$$\sum_{\text{out } A} f(e) - \sum_{\text{in } A} f(e) \leq \sum_{\text{out } A} f(e)$$

Thus we can say that $v(f) \leq \sum_{\text{out } A} f(e)$

3. Since $f(e) \leq c(e)$ we can also say that

$$v(f) \leq \sum_{\text{out } A} c(e)$$

i.e., since we know that the flow of an edge cannot exceed its capacity, we can state that the flow is less or equal to the sum of all edge capacities going out of A.

4. Thus we proved that $v(f) \leq \mathbf{cap}(A, B)$ ↩

5. Then, knowing that $\mathbf{cap}(A, B) = \sum_{\text{out } A} c(e)$ ↩ (recall that the capacity of a cut is the sum of capacities of X -> Y edges)
   We prove that $v(f) \leq \mathbf{cap}(A, B)$

6. The weak duality corollary states that, **if** $v(f) = \mathbf{cap}(A, B)$ then
   - f is a max flow
   - (A, B) is a min cut ↩

   Why?
   - if the value of f is the capacity of the cut, since $f(e) \leq c(e)$, there is no higher value of f which would not exceed the capacity, i.e. any other higher value of f would be $> \mathbf{cap}(A, B)$, which is impossible. So f is a max flow.
   - on the other way around, since $\mathbf{cap}(A, B) \geq f$, meaning that $\mathbf{cap}(A, B) = f$, we get that (A, B) is the cut with the minimum capacity i.e. the min cut.

On the slides is more straightforward:

Pf.

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= \text{cap}(A, B) \end{aligned}$$

Circulation with Demands and Lower Bounds

What if we have lower bounds? That means that the flow on an edge must be at least equal (or higher) than the lower bound. We thus need to slightly redefine the definition of circulation:

Def. A circulation is a function that satisfies:

- For each e ∈ E: $\ell(e) \le f(e) \le c(e)$ (capacity)
- For each v ∈ V: $\sum_{e \text{ in to } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$ (conservation)

Also the problem needs ti be slightly redefined, because we need to take into account these lower bounds:

Circulation problem with lower bounds. Given (V, E, ℓ, c, d), does there exist a circulation?

The key to understand how we can solve the problem, is that we need to **reduce** this problem into a **circulation** problem, so that we can then reduce it to a **max flow** problem.

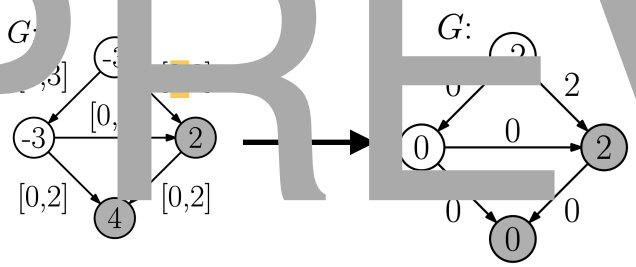
How to proceed:

1. We generate an initial **invalid circulation f_0**, based on the lowest bound on the graph. After that we compute Lv, which basically is **flow entering the node - flow outing the node**, for each node. And we do that for every node. More specifically, we are **generating an initial (invalid) flow f0** base[illegible] bounds[illegible]**ng additional d[illegible]an[illegible]** (exc[illegible]ac[illegible]node

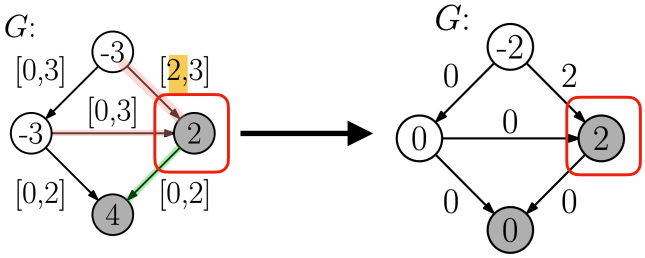
- Generate an initial (invalid) circulation $\mathbf{f_0}$ based on ℓ(e)

$$f_0(u,v) = \ell(u,v)$$

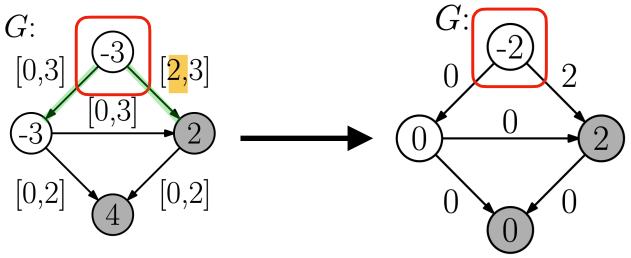
$$L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{(u,v)\in E} \ell(u,v) - \sum_{(v,w)\in E} \ell(v,w)$$



Look[illegible]ow each [illegible]de val[illegible]has changed. The[illegible]dges just e[illegible]ress t[illegible]lower bound. [illegible]e edges w[illegible]ave **f0**[illegible]n the nodes



Here for example, the node had 2+0 entering and 0 outing, thus it becomes 2.



Here for example, the node had nothing entering and 2+0 outing, thus it becomes 0 - 2 = -2.

2. Build a residual circulation problem/network G', by adjusting demands by L(v). We construct this new network G' by ⟶ G': $c'(u,v) = c(u,v) - \ell(u,v)$, $d'(v) = d(v) - L(v)$

So, in short:

- for each **NODE**, we take the original demand - the L value
- for each **EDGE** u-v we use the original capacity - the lower bound

NP-Completeness: Definition of P/NP (slide 8.3 DM 21)

The Class P

Def. P is the set of all languages (i.e., decision problems) for which membership can be determined in (worst case) polynomial time. **P is the set of decision problems which can be solved in polynomial time.** Solved means that we can find a subroutine (i.e. algorithm) takes an input and returns yes or no, in polynomial time. **P is a complexity class**.

Hamiltonian Path

To show that not all languages are in P, we introduce the Hamiltonian Cycle problem, which i believe dot not be P (i.e. not being solvable as a direction problem in polynomial time).

$$\mathrm{HC} = \{G \mid G \text{ has a simple cycle that visits every vertex of } G\}.$$

Given a Graph, has it a Hamiltonian Cycle? This language membership problem seems to not being solvable in polynomial time. As we saw, there is no obviously efficient way to find a Hamiltonian cycle in a graph. Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a very efficient way to verify that a given graph has one. Suppose that a graph did have a Hamiltonian cycle and someone wanted to convince us of its existence. This person would simply tell us the vertices in the order that they appear along the cycle. It would be a very easy matter for us to inspect the graph and check that this is indeed a legal cycle that it visits all the vertices exactly once. The given list of vertices that supposedly forma a cylce is called **certificate:** a certificate is a piece of information which allows us to verify that a given string is in a language in polynomial time.

To clarif[illegible] language [illegible] **x €** (the senten[illegible]/inp[illegible]), a **v[illegible]go[illegible]hm** is an a[illegible]rithm which, **giv[illegible] x ar[illegible] a string y** calle[illegible] the **certif[illegible]ate**, [illegible] ver[illegible] that x is in the [illegible]nguag[illegible] L[illegible]**sing [illegible]is certific[illegible]e as [illegible]elp**. f there exists [illegible] verifica[illegible]n alg[illegible]thm [illegible]at runs in polyn[illegible]ial [illegible]e, we [illegible]ay that L [illegible]n be [illegible]rified in polynomi[illegible] time. [illegible]om this [illegible]efinit[illegible] follows the one of NP problem

The cla[illegible] NP

Def. NP [illegible] the set of [illegible] lang[illegible]ages [illegible]at [illegible]an be verif[illegible]d in polyn[illegible]mial ti[illegible]e. [illegible]This mea[illegible]s that [illegible]t [illegible] is the set of problems on which we can run a verification algorithm in polynomial time. I.e. the set of decision problem for which exists a subroutine (=algorithm) that uses a **certificate** to check if a given **input** is in a language in polynomial time. It may sounds tricky, but :

- P states if, for a given input, there is a solution
- NP verifies if the solution you pass (the certificate) is true or not for the given input.

P=NP ?

Observe that if we can solve a problem efficently without a certificate, we can certainly solve given the additional help of a certificate. Therefore, **P** ⊆ **NP** (i.e. P is a subset of NP. Among the set of all problems that can be verified with a certificate, P is a subset of these problem which can be solved even without certificate). However, it is not known whether P = NP.

Class EXP

EXP. Decision problems for which there is an **exponential-time algorithm**. We can state (proof not reported) that P ⊆ **NP** ⊆ **EXP**

Example of NP Problem: Composite

We have a **certifier**, which is an algorithm that:

- takes a **certificate**, which is a “non-trivial” (i.e. hard) factor of a number s. (s is the input).
- checks first if t is less than one or higher then s (i.e. it couldn’t be a factor). Then if s is a multiple of t returns true, otherwise false.

```
boolean C(s, t) {
   if (t ≤ 1 or t ≥ s)
      return false
   else if (s is a multiple of t)
      return true
   else
      return false
}
```

Instance. s = 437,669.

Certificate. t = 541 or 809.

Example of NP Problem: 3-SAT
(we will see this later)
A 3-SAT is a problem where, given a CNF formula (divided in clauses of 3 elements), we want to know if it is satisfiable. Here the **input/istance** is the formula, the certificate is a possible solution for this formula, i.e. an assignment of True and False to the boolean variables. The verification algorithms basically checks if the given possible solution (the certificate) is valid.

Ex. $(\bar{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x_1} \vee \bar{x_3} \vee \bar{x_4})$
instance s

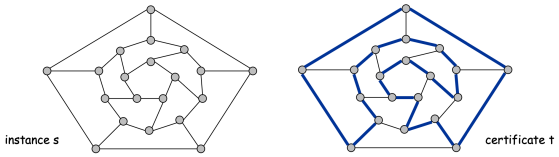
$x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1$
certificate t

Example of NP problem: Hamiltonian Cycle

Problem. Given an undirected graph G=(V,E), does there exist a simple cycle C that visits every node?

- **input/istance:** the graph
- **certificate:** a list of nodes (or path) which is thought to be a cycle

The algorithm checks if the certificate is a valid solution. (Note that a P problem would check only the instance to see if it contains a Ham cycle). More specifically, check that the permutation contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation



N Cor leteness: olyn ial Time Redu on (slide .1 DM 22)

duct :

Before disc ng reductions us jus onsider the following ques . Su ose that th are two pro ns, H and U. know (you strongly believe at least at is *hard*, that it cannot b lved in polyno l time n the other hand, the compl U is *unknow* We want to ove that U is hard this? We e ve want to sh that

$$(H \notin \mathrm{P}) \Rightarrow (U \notin \mathrm{P}).$$

To do this, we could prove the contrapositive,

$$(U \in \mathrm{P}) \Rightarrow (H \in \mathrm{P}).$$

To show that U is not solvable in polynomial time, we will suppose (towards a contradiction) that a polynomial time algorithm for U did existed, and then we will use this algorithm to solve H in polynomial time, thus yielding a contradiction.

To clarify, suppose we have a subroutine (i.e. algorithm) which can solve any istance of U in polynomial time. Now, suppose we have an instance **x** of problem **X** (which we think to be hard). If we are able to transform in some way that istance **x** in an istance **x'** of U, we could pass **x'** directly to the subroutine that solves U in polynomial time and, consequently, solve the translated instance of **x** in polynomial time. This process means that we are **Reducing X to U.**

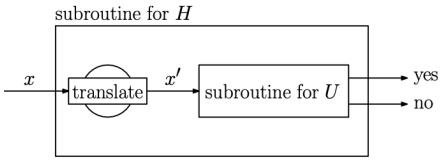


Fig. 77: Reducing H to U.

Assuming that the translation runs in polynomial, we say we have a polynomial reduction of problem H to problem U, which is denoted **$H \leq_P U$.** More specifically, this is called a Karp reduction. **$H \leq_P U$** means that H can be reduced to an instance of U and, thus, solved in at most the same time-complexity (polynomial) as U.

Look at the example having **$L_1 \leq_P L_2$**

L1 can be reduced in istance of L2 and solve in ≤ complexity time.Therefore if L2 is solvable in polynomial time, L1 is soluble in pol time, if L1 is not polynomial, so is L2 (not viceversa). We have also transitivity property

$L_1 \leq_P L_2$

- If L_2 in P, then L_1 in P (why?)
- If L_1 not in P, then L_2 not in P (why?)

"L1 reduces polynomial L2"

- $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ implies $L_1 \leq_P L_3$

Compute the performance ratio (Proof)

Suppose 2-f-1 gives a cost of C, where C = |V'|. And the unknwon optimal solution is C* = |V|. How do we compute now C/C* (since we are in a minimization problem)?.

We **claim** the following: **The 2-for-1 approximation for VC achieves a performance ratio of 2**, i.e. C/C* <= 2

Proof:

- Consider the set C output by two-for-one-VC(G).
- Let C∗ be the optimum vertex cover.
- Let A be the set of edges selected by the alg.
- **|C| = 2|A|**, because both endpoints of each edge of A to C.
- But C∗ must cover the edges in A, which are non-adjacent. Thus, **|C∗| ≥ |A|**

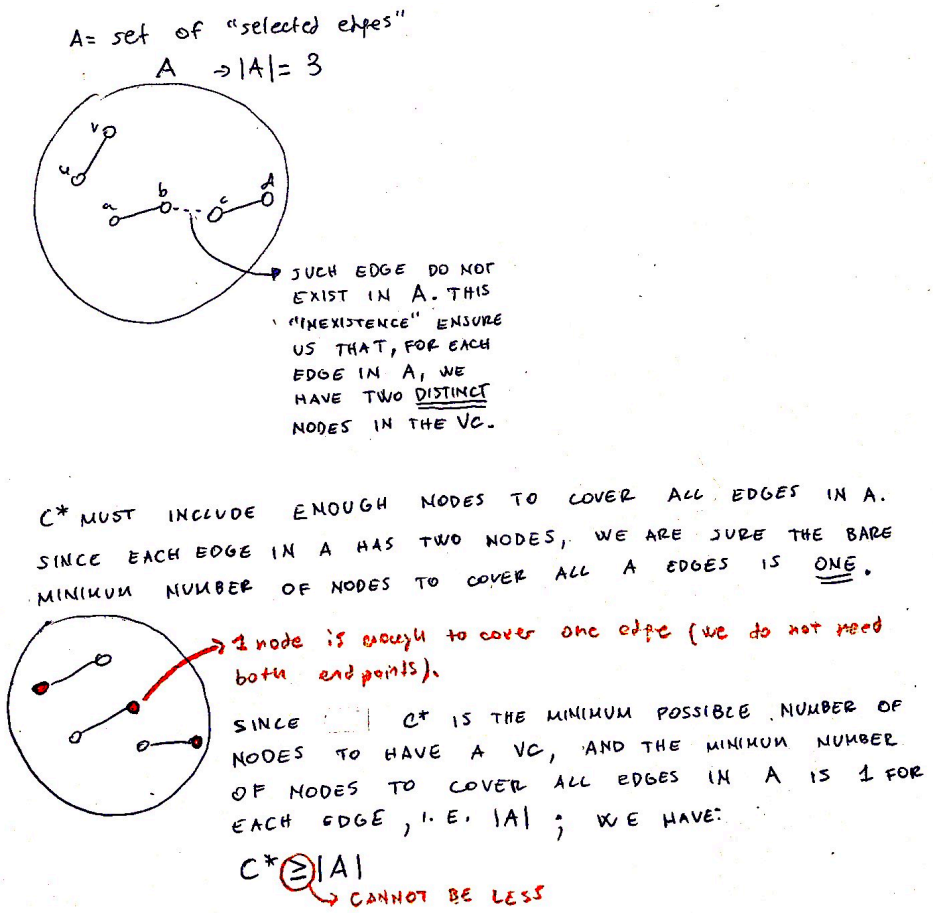
$$|C| = 2|A| \leq 2|C^*| \quad \Rightarrow \quad \frac{|C|}{|C^*|} \leq 2$$

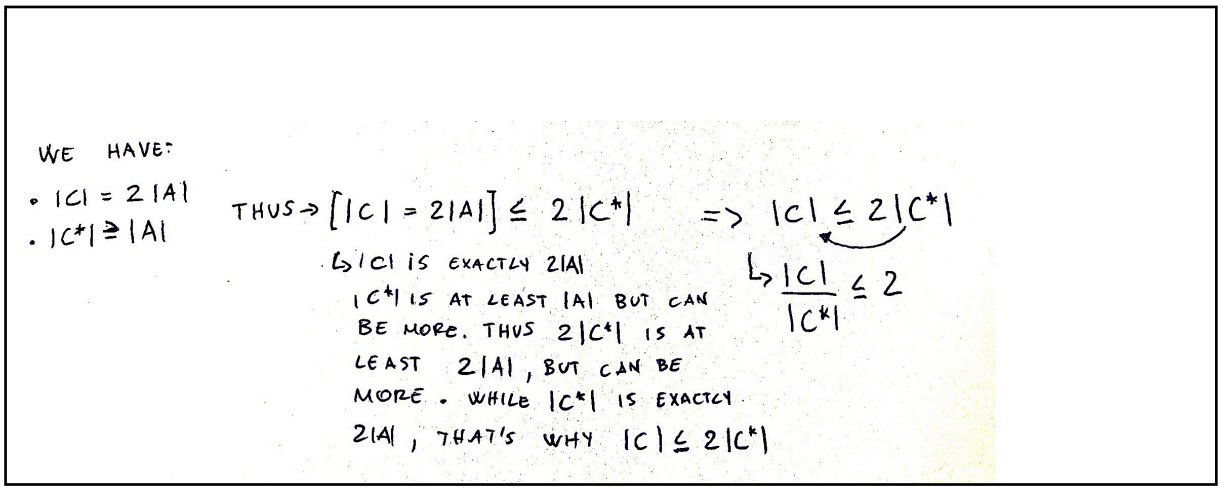
Explanation

- **|C| = 2|A|**, because both endpoints of each edge of A to C.

S e ea n edge has end ints, each time e consid an ed e, we ut 2 nodes in th v ex c er set. Th C=2 |, where |A| is the umber edge nsid ed by the gorith

- But C∗ must co r th dges in A, which r non-adj cent. hus, **|C∗| |A|**





Approximation factors are not preserved
Approximation factors are not preserved by transformations: the fact that a NP-complete problem A can be reduced to an instance of another problem B (thus proving that B is NP-complete), does not guarantee that the performance ratio of B will be the same of A (usually is never like that).
Here we can see an example of reduction from [illegible] to [illegible] instance of IS, [illegible] proving that IS is NP-complete. Despite the reduction is possible, the performance bound of the two problems are [illegible]

E.g. Independent set (V \ V′)
- If V′ is a VC for G, then V \ V′ is an IS for G. we proved this
- If |V'|=k is a min VC for G, then |V\V'|=n-k is a max IS for G
- Heuristic returns a VC |V″| ≤ 2k
- Thus, V\V" is an approximate IS of size ≥ n-2k
- Performance ratio $$\rho(n,k) = \frac{n-k}{n-2k}.$$
- Not constant (arbitrarily large)
- E.g. [illegible] = 500 [illegible] = 500/[illegible]500.

Ex2. Traveling Sale[illegible]an [illegible]h Triangle Inequal[illegible] ---

Recall the **definition of TSP**: Given a complete undirected graph with nonnegative edge edge weights w(u,v) find a cycle that **visits all vertices** and has **minimum cost**.

Often, edge weights in the TSP route satisfy the **triangle inequality.** The triangle inequality states that

For any three points A, B, and C in a metric space, the triangle inequality states that the distance between two points A and C is always less than or equal to the sum of the distances between A and B, and between B and C. Mathematically, it is expressed as:

$$d(A, C) \leq d(A, B) + d(B, C)$$

Applied to edge weights: $w(u, v) \leq w(u, x) + w(x, v)$

- **Direct Path:** $w(u, w)$
- **Indirect Path via** v: $w(u, v) + w(v, w)$

The triangle inequality ensures that the sum of the weights of two sides of a triangle is always greater than or equal to the weight of the third side. In other words, traveling directly from u to w should not cost more than first traveling from u to v and then from v to w.

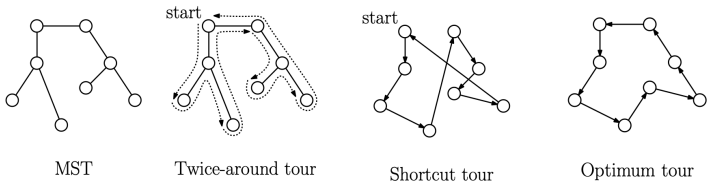
We will show that, When cost function satisfies the triangle inequality there is anapproximation algorithm for TSP with a ratio-bound of 2. This is because triangle inequality allows us to create **shortcuts** (we will see).

Approximation Algorithm for TSP

Recall the definition of tour: is the minimum-weight path that intersects all nodes of a graph and comes back to the first visited, thus creating a cycle. We start by observing that a TSP tour is exactly a **MST tree** with the addition of one edge, the one that connects the last visited node to the first visited one.

Thus, to approximate TSP, we do the following:

1. We compute MST on the graph (using Krystal is the best way).
2. We compute any tree traversal algorithm (i.e. DFS) to compute a “twice-around” tour of the MST tree.
3. This tour is “simplified” by applying shortcuts (possible only if triangle inequality holds)



Let’s understand better steps 2. and 3.

2. The “twice around tour”: To form a tour, we perform a depth-first search (DFS) traversal of the MST. DFS is a way of visiting all the nodes in a tree by starting from a root node and exploring as far down a branch as possible before backtracking. During this DFS traversal, we visit each edge of the MST twice: once when we go down the tree (from parent to child) and once when we come back up (from child to parent). Because we traverse each edge twice, we visit each node twice as well—once when we first arrive at the node, and a second time when we return to that node while backtracking —> This process repeats for all nodes connected to the MST, leading to a situation where e[illegible] visited [illegible] wh[illegible] moving fo[illegible]ard [illegible] the D[illegible] and once wh[illegible] b[illegible]ktrac[illegible]ng.

After [illegible]forming a d[illegible]th-first [illegible]arch (DFS) on the MS[illegible] you obt[illegible] a tour t[illegible] visits each [illegible]e twice. T[illegible]our fol[illegible]s the edges of the MS[illegible]n a s[illegible]uence tha[illegible]ould look s[illegible]ething like t[illegible]

$$A \to B \to C \to B \to D \to B \to A \to E \to A$$

In this sequence:

- Nodes are visited more than once (e.g., B and A are revisited).
- This is not yet a valid TSP tour because nodes should be visited exactly once.

To make this a valid TSP tour, we use *shortcutting*. The triangle inequality ensures that the path length won’t increase when we take short-cuts: the order in which vertices are visited during the creation of shortcuts is the preorder traversal of the MST (i.e. the sequence above). To get a valid TSP, any subsequence of the twice-around tour will suffice, as long as this subsequence visits each vertex exactly once (its not necessary to have a subsequence of the twice-around tour that follows exactly the preorder traversal). In short, to **shortcut** means to **skip intermediate nodes without increasing the overall path length** (you can safely do that since the graph satisfies the triangle inequality.

Proof this approximation is <= 2

Claim: Approx-TSP achieves a performance ratio of 2

Proof

- H tour produced by approx-TSP.
- H∗ optimum tour.
- Since T is an MST, $W(T) \leq W(H^*)$.
- The twice around tour of T has cost 2 · W (T),
- By the triangle inequality, short-cuts do not increase cost of tour

$$W(H) \leq 2 \cdot W(T) \leq 2 \cdot W(H^*) \Rightarrow \frac{W(H)}{W(H^*)} \leq 2$$

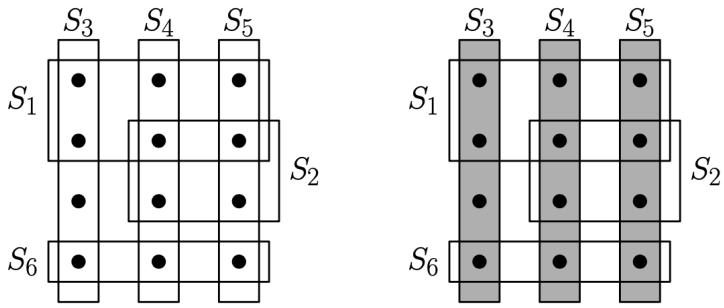
Ex3. Set Cover

Recall the definition of **set cover:**

SET COVER: Given a set U of elements, a collection S_1, S_2, . . . , S_m of subsets of U, and an integer k, does there exist a collection C of ≤ k of these sets whose union is equal to U?

$$U = \bigcup_{S_i \in C} S_i.$$

In short, set cover aims to find the minimum number of subsets so that every element of U is included. In this example, we are given 6 subsets (S1,…,S6). Among these 6 subsets, what is the minimum I can consider so that every element (every dot) is included? The optimal set cover is {S3, S4, S5}, i.e. the three vertical subsets. Any other combination would consider > 3 subsets (e.g. {S1,S2,S6,S3}).



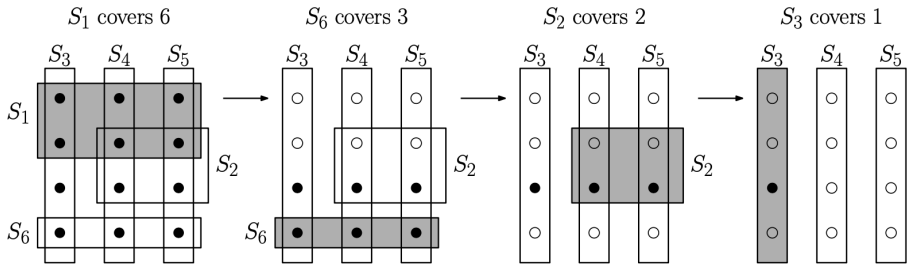
Greedy Heuristic

Greedy heuristic

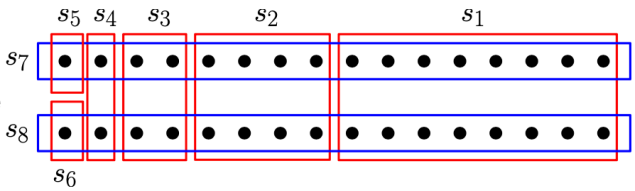
- At each stage select the set that covers the greatest number of uncovered elements
- achieves an approximation factor of ≤ ln m, m = |U|

In short, select the subset with most elements, delete all the elements in the subset and choose the second best subset on the remaining elements

```
Greedy-Set-Cover(U, F)
    X = U                          // X stores the elements covered in this
    C = empty;                     // C stores the sets of the cover
    while (X is nonempty) {
        select S in F that covers the most elements of X;
        add S to C;
        X = X \ S;
    }
    return C
}
```



Greedy heuristic can be easily fooled and compute a highly suboptimal solution. For example, in this case, the optimal choice would be to choose the 2 sets S7 and S8. Instead, the greedy would instantly choose s1, since size of s1 > s7, then, after removing all elements in s1, it would choose s2 (since s2 will have still more elements than s7 or s8).



Opt: {S_7,S_8}

Greedy: $\{s_1, s_2, s_3, s_4, s_5, s_6\}$

Theorem: The greedy set-cover heuristic achieves an approximation factor of at most ln m, where m = |U|.

The bound depends logarithmically on the number of elements in U.

proof not part of the course

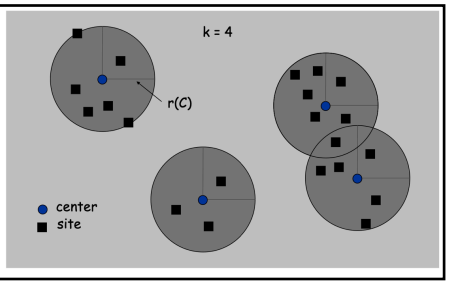
Ex.4-5 Center Selection and k-Center Problem

Center Selection and k-Center

Input. Set of n sites $s_1, \ldots, s_n$.

Center selection problem. Select k centers C so that **maximum** distance from a site to **nearest center** is minimized.

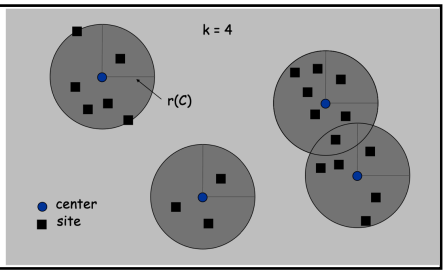
Basically, we want to minimize the largest distance from any site to its nearest selected center. We want any point to be as close as possible to its closer center.



CENTER SELECTION

Input. Set P of n sites $s_1, \ldots, s_n$.

k-center problem. Select k centers **C⊆P** so that the maximum distance from a site to its nearest center is minimized. (C is subset of P)



K-CENTER

To better understand the differences between these two formulations:

Center Selection Problem:

- **Center Can Be Anywhere**: In the center selection problem, you can choose the locations of the centers from any possible points in the space. These centers do not necessarily have to be at the input sites. The goal is to select k centers such that the maximum distance from a site to its nearest center is minimized.

k-Center Problem:

- **Centers Must Be at Input Sites:** In the k-center problem, the centers must be chosen from the given set of input sites. You cannot place a center at an arbitrary point in the space; it must coincide with one of the input sites. The objective remains the same—to minimize the maximum distance from any site to its nearest center—but with the added constraint that centers must be located at the input sites.

Now, let's see each one of these problems more in detail and how they can be approximated. But before, recall the properties of "distance" in this kind of context:

The notion of distance, called **distance function**, has to satisfy several natural properties:

- $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernible)
- $d(p_i, p_j) \geq 0$ (nonnegativity)
- $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

or, more technically:

Positivity: $\delta(u, v) \geq 0$ and $\delta(u, v) = 0$ if and only if $u = v$

Symmetry: $\delta(u, v) = \delta(v, u)$

Triangle inequality: $\delta(u, w) \leq \delta(u, v) + \delta(v, w)$