Foundamentals of Computer Science - Theory

Graph Theory

In Mathematics and Computer Science, an **algorithm** is a finite sequence of well defined, computer implementable instructions, typically to solve a class of problems or to perform a computation.

Algorithms are the ideas behind computer programs, this procedure can be written in many different languages producing the same result. We will see theoretically with SUDO codes. Every algorithm has its own process time, and to solve a problem there can be many different algorithms with many different times of execution.

Graph Theory, it is the language used to describe problems solved by algorithms. In mathematics, graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of **vertices** (also called *nodes or points*) which are connected by **edges** (also called *links or lines*). A distinction is made between **undirected graphs**, where edges link two vertices symmetrically, and **directed graphs**, where edges link two vertices asymmetrically.

In **Computer Science**, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from on page to another. A similar approach can be taken to problems in social media, biology, computer chip design, mapping the progression of neuro-degenerative diseases, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science.

The paper written by **Leonhard Euler** on the "**Seven bridges of Konigsberg**" and published in 1736 is regarded as the first paper in the history of graph theory. From Euler takes the name a class of graphs which are the **Eulerian Graphs**, this term has two common meanings in graph theory. One meaning is a graph with a Eulerian Circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.

What is defined as **Euler's theorem** is the concept that a connected graph has an **Eulerian cycle** if and only if every vertex has even degree, where an Eulerian cycle is a path in a finite graph that visits every edge exactly once (allowing for revisiting vertices).

In the mathematical field of graph theory, a **bipartite graph** is a graph whose vertices can be divided into two disjoint and independent sets X and Y such that every edge connects a vertex in X to one in Y. One of the most common algorithms is the **Matching problem**, a perfect matching occurs when nobody remains without match and also in case of preferences, the highest degree of preferences are matched.

Many problems and theorems in graph theory have to do with various ways of colouring graphs. Typically, one is interested in coloring a graph so that no adjacent vertices have the same color, or

with other similar restrictions. One may also consider edges (possibly so that no two coincident edges are the same color), or other variations.

The problem of **Proper Coloring** a graph states that a color is "proper" if and only if an edge exists such that the color of a vertex A is different from the color of a vertex B, connected by an edge. The **Chromatic Number X(G)** is the minimum number of colors in a proper coloring. Regarding this, exists the **Four Color Theorem** which states that, given any separation of a plane into contiguous regions, producing a figure called a map, **no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color**.

A **Subgraph** G' of a graph G is a graph G' whose vertex set, and edge set are subsets of those of G. If G' is a subgraph of G, then G is said to be a **supergraph** of G'.
A lot of problems in computer science consist of finding a subgraph with certain special properties. A spanning subgraph is defined by the fact that the vertex set of the subgraph (V') is equal to the vertex set of supergraph (V).

An **Induced Subgraph** of a graph is another graph, formed from a subset of the vertices of the graph and all of the edges connecting pairs of vertices in that subset. Or the contrary, when from a subset of the edges of a graph a new graph is obtained defining the missing vertices.

Isomorphism in subgraphs, G1 and G2 are isomorphic if there exist a bijection f: V1 -> V2 such that for each edge in G1, applying a relabeling, you find an edge in G2. Defining an algorithm capable of finding this bijection function was considered an intractable problem.

A **Complete Graph** is composed of n vertices and a set of edges such that every vertex is connected to all the others.

The **Vertex Degree** is the number of edges incidents with that vertex.
In graph theory, the degree of a vertex of a graph is the number of edges that are incident to the vertex, and in a multigraph, loops are counted twice. The degree of a vertex *v* is denoted ***deg(v)*** or *degv*. The **maximum degree of a graph G**, denoted by ***Δ(G)***, and the **minimum degree of a graph G**, denoted by **δ(G)**, are the maximum and minimum degree of its vertices.

An **Incidence Matrix**, **M (V, E)** where we introduce 1 if a vertex € to the edge, and 0 otherwise. From this matrix we are able to see that every column has two 1s, indicating that every edge is connected to two vertices. Instead, if we sum up the rows, we get the degree of the correspondent node.
In **mathematics**, an incidence matrix is a matrix that shows the relationship between two classes of objects. If the first class is X and the second is Y, the matrix has one row for each element of X and one column for each element of Y. The entry in row x and column y is 1 if x and y are related (called incident in this context) and 0 if they are not.

An **Adjacency Matrix**, **A (V, V)**. We introduce a 1 in the matrix If two vertices are adjacent, and a 0 otherwise. This matrix tells us whether two vertices are connected.
In **graph theory** and **computer science**, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is **symmetric**.

Theorem 1

$\sum_{v \in V} d_G(v) = 2|E|$

Proof: Consider the incidence matrix $M(V, E)$

Row v has $d_G(v)$ 1's. So # 1's in matrix M is $\sum_{v \in V} d_G(v)$.

Column e has 2 1's. So # 1's in matrix M is $2|E|$

From which:

$$2|E| = \sum_{v \in V} d_G(v)$$

Corollary In any graph, the number of vertices of odd degree, is even.

proof: Let ODD = {odd degree vertices} and EVEN = V / ODD

$$\sum_{v \in ODD} d(v) = \underbrace{2|E|}_{\text{Even}} - \underbrace{\sum_{v \in EVEN} d(v)}_{\text{Even}}$$

Therefore $\sum_{v \in ODD} d(v) = |ODD|$ is EVEN.

A sequence of vertices W = (V_1, …, V_k) is a **Walk** in G if (V_i, V_{i+1}) € E for 1 <= i <= k.
A **Walk** is **Closed** if $V_1 = V_k$.
A **Path** is a walk in which the vertices are distinct.
A **Cycle** is a closed walk in which the vertices are distinct except for V_1 and V_k.
A **Tree** is a graph which is **Connected** and has **No Cycles** (Acyclic).

Lemma 1

Suppose W is a walk from vertex a to vertex b and that W minimizes ℓ over all walks from a to b
Then W is a path.

Proof: Suppose W is not a path.

Suppose $W = (a = a_0, a_1, \ldots, a_k = b) = (a = a_0, \ldots, a_i, \ldots, a_j, \ldots, a_k = b)$. Let's assume $a_i = a_j$ where $0 \leq i < j \leq k$

Then I can build a new walk of the form $W' = (a = a_0, a_1, \ldots, a_i, a_{j+1}, \ldots, a_k = b)$ which is also a walk from a to b and $\ell(W') = \ell(W) - (j - i)$. Since $(j - i) > 0 \Rightarrow \ell(W') < \ell(W)$.

This is a contradiction because we have shown that if W is not a path it cannot minimize the length.

In graph theory, a **component**, sometimes called a **connected component**, of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. An alternative way to define components involves the **equivalence classes** of an equivalence relation that is defined on the vertices of the graph. In an undirected graph, a vertex *v* is *reachable* from a vertex *u* if there is a path from *u* to *v*. In this definition, a single vertex is counted as a path of length zero, and the same vertex may occur more than once within a path. Reachability is an equivalence relation, since:

- It is **reflexive**: There is a trivial path of length zero from any vertex to itself.
- It is **symmetric**: If there is a path from *u* to *v*, the same edges form a path from *v* to *u*.
- It is **transitive**: If there is a path from *u* to *v* and a path from *v* to *w*, the two paths may be concatenated together to form a path from *u* to *w*.

The components are then the induced subgraphs formed by the equivalence classes of this relation.

Breadth-first search (**BFS**) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

It uses the opposite strategy as **depth-first search**, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.

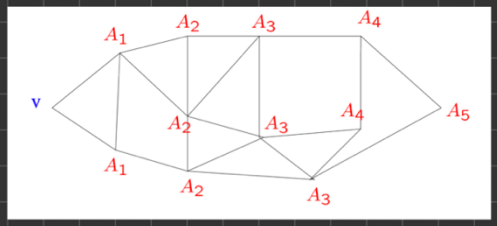
BFS - BREADTH FIRST SEARCH

Fix $v \in V$. For $w \in V$ let $d(v,w)$ = length of shortest path from v to w.

For $t = 0,1,2,\dots$ let $A_t = \{w \in V : d(v,w) = t\}$

In BFS we construct $A_0, A_1, A_2, \dots$ iteratively by following the rule:

$$A_{t+1} = \{w \notin A_0 \cup A_1 \cup \dots \cup A_t : \exists \text{ an edge } (v,w) \text{ such that } v \in A_t\}$$



Note: no edges (a,b) between A_K and A_ℓ for $\ell - K \geq 2$, else $w \in A_{K+1} \neq A_\ell$

In this way we can find all vertices in the same component C as v.
By repeating for $v' \notin C$ we find another component etc.

Theorem 1

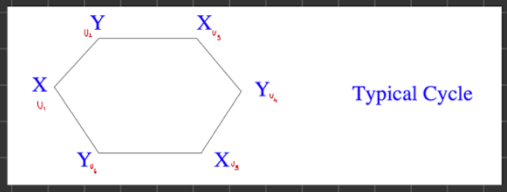
G is bipartite $\iff$ G has no cycles of odd length

Proof:

$G = (X \cup Y, E)$

Suppose $C = (v_1, v_2, \dots, v_K, v_1)$ is a cycle.

Suppose $v_1 \in X$. Then $v_2 \in Y$, $v_3 \in X$, $\dots$, $v_K \in Y$ implies K is even.



Now we want to show that if it has no cycles of odd length then it is bipartite.

So, we start choosing a vertex $v \in V$ and we construct $A_0, A_1, A_2, \dots$ by BFS algorithm.

We call, $X = A_0 \cup A_2 \cup A_4 \cup \dots$ and $Y = A_1 \cup A_3 \cup A_5 \cup \dots$

We need only to show that X and Y contain no edges and then all edges must join X and Y.

Suppose X contains edge (a,b) where $a \in A_K$ and $b \in A_\ell$

- If $K \neq \ell$ then $|K - \ell| \geq 1$ which contradicts the note to the BFS algorithm.
- If $K = \ell$ then

There exist paths $(v = v_0, v_1, v_2, \dots, v_K = a)$ and $(v = w_0, w_1, w_2, \dots, w_K = b)$

Let $J = \max\{t : v_t = w_t\}$ $\Rightarrow$ $(v_J, v_{J+1}, \dots, v_K, w_K, w_{K-1}, \dots, w_J)$ is an odd cycle - length $= 2(K-J)+1$ which is a contradiction.

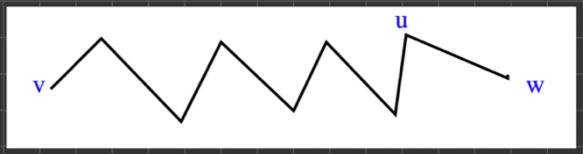
Theorem 2

$A^K(v,w)$ = number of walks of length K from v to w with K edges. (A is an adjacency matrix)

Proof: By induction on K. Trivially true for $K = 1$. Assume it true for some $K \geq 1$.

Let $N_t(v,w)$ be the number of walks from v to w with t edges.

Let $N_t(v,w;u)$ be the number of walks from v to w with t edges whose penultimate vertex is u.



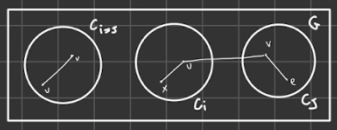
$$N_{K+1}(v,w) = \sum_{u \in V} N_{K+1}(v,w;u) = \sum_{u \in V} N_K(v,u) \cdot A(u,w) = \sum_{u \in V} A^K(v,u)\, A(u,w) = A^{K+1}(v,w)$$

By Induction

Lemma 1 Let the components of G be $C_1, C_2, \ldots, C_r$. Suppose $e = (u,v) \notin E$, $u \in C_i$, $v \in C_j$

a) $i = j \Rightarrow w(G+e) = w(G)$

b) $i \neq j \Rightarrow w(G+e) = w(G) - 1$



Proof: Every path P in $G+e$ which is not in G must contain e. Also, $w(G+e) \leq w(G)$.
Suppose $(x = v_0, v_1, \ldots, v_k = u, v_{k+1} = v, \ldots, v_\ell = y)$ is a path in $G+e$ that uses e. Then clearly $x \in C_i$ and $y \in C_j$

a) Follows as now no new relations $x \sim y$ are added.

b) Only possible new relations $x \sim y$ are for $x \in C_i$ and $y \in C_j$. But $u \sim v$ in $G+e$ and so $C_i \cup C_j$ becomes a single new component.

Introduction to Algorithms

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate. Does the algorithm suddenly become incredibly slow when the input size grows? Does it mostly maintain its quick run time as the input size increases? Asymptotic Notation gives us the ability to answer these questions.

O - Notation: $f(n) = O(g(n))$ if for all $n \geq n_0$ $0 \leq f(n) \leq c\,g(n)$ with $c > 0$ "upper bound"

Ω - Notation: $f(n) = \Omega(g(n))$ if for all $n \geq n_0$ $0 \leq c\,g(n) \leq f(n)$ with $c > 0$ "lower bound"

Θ - Notation: $f(n) = \Theta(g(n))$ if for all $n \geq n_0$ $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ with $c_1, c_2 > 0$
$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ "grows asymptotically as"

o - Notation: $f(n) = o(g(n))$ if for all $n \geq n_0$ $0 \leq f(n) < c\,g(n)$ with $c > 0$ "upper bound"

ω - Notation: $f(n) = \omega(g(n))$ if for all $n \geq n_0$ $0 \leq c\,g(n) < f(n)$ with $c > 0$ "lower bound"

- **Master Method**

The Master Method applies to recurrences of the form $T(n) = a\,T(n/b) + f(n)$
where $a \geq 1$, $b > 1$ and $f(n)$ is asymptotically positive.
There are 3 Typical Cases which are to be chosen through a comparison of $f(n)$ with $n^{\log_b a}$.

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
$f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^ε factor) $\quad T(n) = \Theta(n^{\log_b a})$

CASE 2: $f(n) = \Theta(n^{\log_b a} \cdot \lg^k n)$ for some constant $k \geq 0$
$f(n)$ and $n^{\log_b a}$ grow at similar rates $\quad T(n) = \Theta(n^{\log_b a} \cdot \lg^{k+1} n)$

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$
$f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^ε factor) $\quad T(n) = \Theta(f(n))$.
and satisfies the regularity condition that $a\,f(n/b) \leq c\,f(n)$ for some constant $c < 1$

Insertion Sort is an effective algorithm for sorting a small number of elements. Insertion Sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

The algorithm takes as a parameter an array A[1,..., n] containing a sequence of length n that is to be sorted. The algorithm sorts the input numbers in-place: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time. The input array A contains the sorted output sequence when the Insertion Sort procedure is finished.

The algorithm written in **Pseudocode** is the following:

```
INSERTION-SORT (A, n)  ▷ A[1 . . n]
for j ← 2 to n
    do key←A[j]
       i←j–1
       while i > 0 and A[i] > key
          do A[i+1] ← A[i]
             i←i–1
       A[i+1] = key
```

Analysis of Insertion Sort:

The time taken by Insertion Sort depends on the input: sorting a thousands numbers takes longer than sorting three numbers. Moreover, Insertion Sort can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they are.

We are often interested in considering the worst - case, that in this case is the reverse ordered sequence:

Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$

Divide and Conquer Approach, many useful algorithms are **recursive** in structure, to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a **divide – and – conquer** approach, i.e. they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the problem recursively, and then combine these solutions to create a solution to the original problem.

The **divide – and – conquer** paradigm involves three steps at each level of the recursion:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into the solution for the original problem.

Analysis of Divide and Conquer Algorithms, when an algorithm contains a recursive call to itself, we can often describe its running time by a **recurrence**, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.
There are two common ways of solving this kind of recurrences:

- **Recursion Trees**
- **Master Theorem Method**, if the recurrence has the form T(n) = aT(n/b) + f(n)

Merge Sort is a sorting algorithm that follows the divide – and – conquer approach to sort an array.

- **Divide** the n-element sequence to be sorted into two sub-sequences of n/2 length.
- **Conquer**, sort the two subsequences recursively using Merge Sort.
- **Combine**, merge the sorted subsequences to produce the sorted final array.
-

The **Pseudocode** of the algorithm is the following:

```
MERGE-SORT (A[1 . . n])
{ if n>1
      MERGE-SORT (A[ 1 . . ⌈n/2⌉ ])
      MERGE-SORT (A[ ⌈n/2⌉ +1 . . n ])
      MERGE(A[1 . . n])
}
```

Analysis of Merge Sort running time:

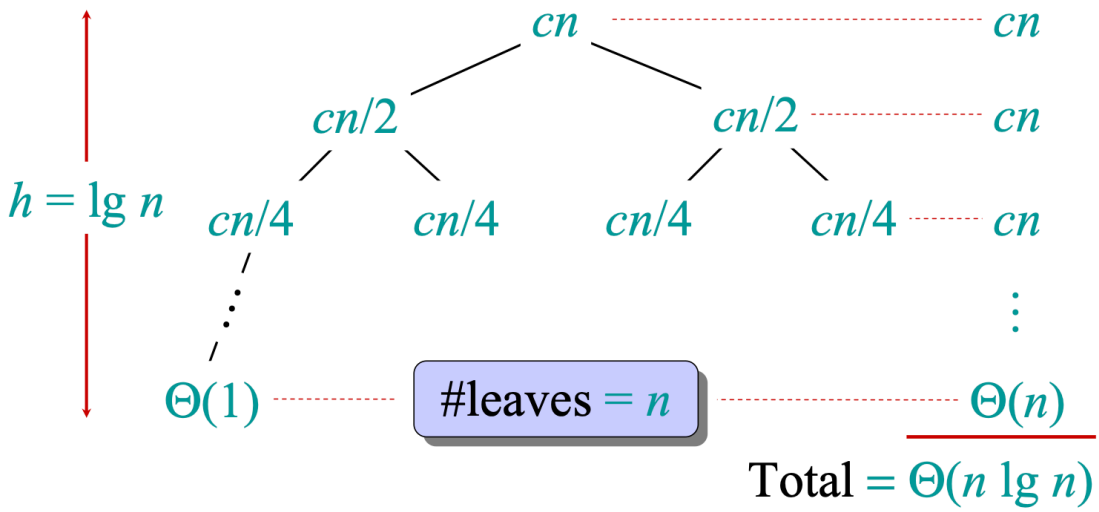
To analyze the running time of the Merge Sort algorithm we start analyzing the time required by each step:

- **Divide**, just computes the middle of the subarray, which takes constant time, therefore **Θ(1).**
- **Conquer**, recursively solve two subproblems, each of size n/2, therefore contributes **2T(n/2)** to the running time.
- **Combine**, the merge procedure on a n-element subarray takes time **Θ(n).**

Adding them up: T(n) = **Θ(1)** + **2T(n/2)** + **Θ(n).**

When we are considering the asymptotic growth of the function, **Θ(1)** can be omitted.

To know the upper bound to this recurrence we can now apply one of the two methods above discussed and discover that: **T(n) = O(n lg n)**.



Binary Search. Looking for a specific information in memory is a key operation in computing. There are several ways to carry out this operation, the simplest one is **Selection Sort**, which consists of simply scan through all the list and search for the object.
However, especially if the list is already sorted, there are some clever and faster ways like **Binary Search**, this algorithm implements the **divide-and-conquer** approach to reach the goal faster. Binary search compares the target value to the middle element of the array: if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful or the remaining half is empty.

- **Divide**, compare the middle element with the searched one.
- **Conquer**, recursively search one subarray.
- **Combine**, trivial.

Binary Search Worst Case Analysis:
We know that there are two ways to approach this kind of analysis:

- **Recurrence Tree**, for n items, log(2,n) steps are needed to fully reduce the input; as each step involves a fixed number of operations the algorithm is O(log n).
- **Master Theorem Method**, we can define the recurrence which describes the algorithm by analysing the three main steps:
  - **Divide**, Θ(1)
  - **Conquer**, 1T(n/2)
  - **Combine**, Θ(1)

Hence the recurrence is **T(n) = 1T(n/2) + Θ(1).** And therefore using the **case 2** of master theorem we can see that **T(n) = O(log n).**

In **Randomized Algorithms**, instead of assuming a distribution of inputs, we impose a distribution, that is the random one, to enforce the property that every permutation is equally likely. For many problems a randomized algorithm is the **simplest** or the **fastest** or even **both**. For many randomized algorithms, no particular input elicits its worst-case behavior. Even your worst enemy cannot produce a bad input array, since the random permutation makes the input order irrelevant. The randomized algorithm performs badly only if the random-number generator produces an "unlucky" permutation.

There are many ways of obtaining randomness, some natural ones (DNA mutations and Quantum mechanics) and artificial ones like **pseudo-random number generators** that generate sequences of number with a big enough period to not see the difference in most of the cases.

Randomized Algorithms belong to 2 **Classes**:

- **Monte Carlo**, algorithms that run for a fixed number of steps, but produce an output that is correct with a certain level of probability.

- **Las Vegas**, algorithms that produce always the correct output, but its running time is a random variable whose expectation is bounded by a polynomial.

Karger Min – Cut Algorithm

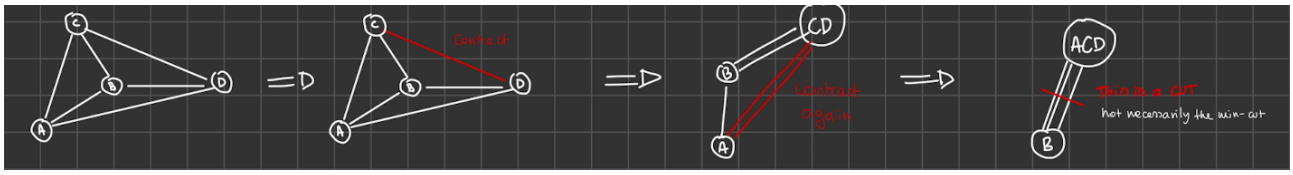
Min – Cut, given an undirected graph, a global min – cut is a cut (S, V – S) minimizing the number of crossing edges, where a crossing edge is an edge (u, v) s.t. u ∈ S and v ∈ V – S.

Graph Contraction, for an undirected graph G, we can construct a new graph G' by contracting two vertices u, v in G as follows:

- U and V become one vertex {u,v} and the edge (u, v) is removed;
- The other edges incident to u or v in G are now incident on the new vertex {u, v} in G';

Note: there may be multi-edges between two vertices. We just keep them.

Karger Min – Cut Algorithm, this algorithm is a randomized one, indeed it exploits the randomness to reach a min-cut.
We apply contraction at random, by associating at each edge the same probability of being chosen. Hence, the probability of choosing vertices that are connected by multiple edges is higher.



The **Pseudocode** of the algorithm is the following:

```
Karger Min-Cut
input  : A graph G = (V, E)
output: A cut C ⊆ E
begin
    repeat
        Choose a random edge e ;
        Contract e and merge its endpoints into a single vertex ;
    until there are only two vertices a, b left ;
    Let C be the set of edges between a and b ;
    return C ;
end
```

Karger observed that by doing this procedure repeatedly, the chance to find a Minimum Cut is **Ω(1/n^2).** Clearly C is a cut, what is surprising is that with reasonably high probability C is the smallest one.

To see this: let's consider **Cmin = |k|**. Let's now consider the step of the algorithm were **t** vertices are left, then each vertex must have at least k edges, otherwise a lower cut could be found. From which we know that the number of edges is **k/2 * t**.

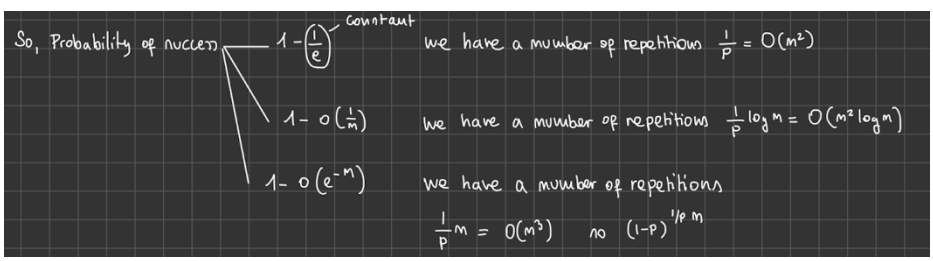
We can now calculate the probability that none of the k edges of the min cut are contracted in that step, which is:

$$1 - \frac{k}{\#\text{edges}} \geq 1 - \frac{k}{t \cdot \frac{k}{2}} = \frac{t-2}{t}$$

Now we have to evaluate the probability that this edges survives till the last step:

$$P \geq \prod_{t=3}^{n} \frac{t-2}{t} = \frac{1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-2)}{3 \cdot 4 \cdot 5 \cdot \dots \cdot n} = \frac{2}{(n-1)n} \quad \text{So, } P \geq \frac{2}{(n-1)n} = \Omega\left(\frac{1}{n^2}\right)$$

If we want to increase the probability of success we can reuse the algorithm and the probability levels can be calculated from the fact that each event is independent.



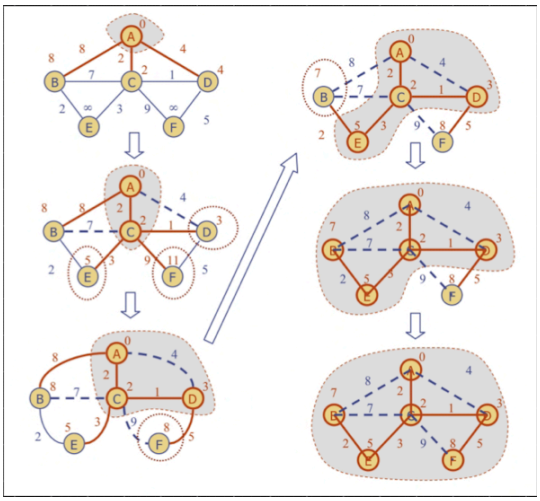
Dijkstra's Algorithm is a **greedy algorithm**. Which is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Dijkstra's Algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative.

Through the **Pseudocode** we can analyze the behavior of Dijkstra's Algorithm.

- In the first line we initialize the graph setting the distance of the source vertex to 0 and infinity to all other vertices.
- In the second line we initialize the set S to an empty set.
- In the third line the set Q is initialized and contains all the vertices of the graph, it is an invariant Q = V - S.
- Then we define the main loop which runs until the set Q is empty: inside the loop, a vertex u is extracted from the min-priority queue and added to the set S. The vertex u has the peculiarity of being the vertex with the smallest distance.
- Finally in lines 7-8 the relaxation of the edges takes place, which means that for each vertex in the neighborhood of u, let's call each of them v, their distance is updated if the path for reaching them is shorter passing through u. The loop runs until Q != 0, which means until all nodes of the graph have been visited.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```



Analysis of Dijkstra's Algorithm
The running time of Dijkstra's algorithm depends on how we implement the min-priority queue.

If we consider Q as a **linear array**: **EXTRACT_MIN** takes **O(V)** time and there are **|V|** such operations. Therefore, a total time for **EXTRACT_MIN** in while-loop is **$O(V^2)$**. Since the total number of edges in all the adjacency list is **|E|**. Therefore for-loop iterates **|E|** times with each iteration taking **O(1)** time. Hence, the running time of the algorithm with array implementation is **$O(V^2 + E) = O(V^2)$**.

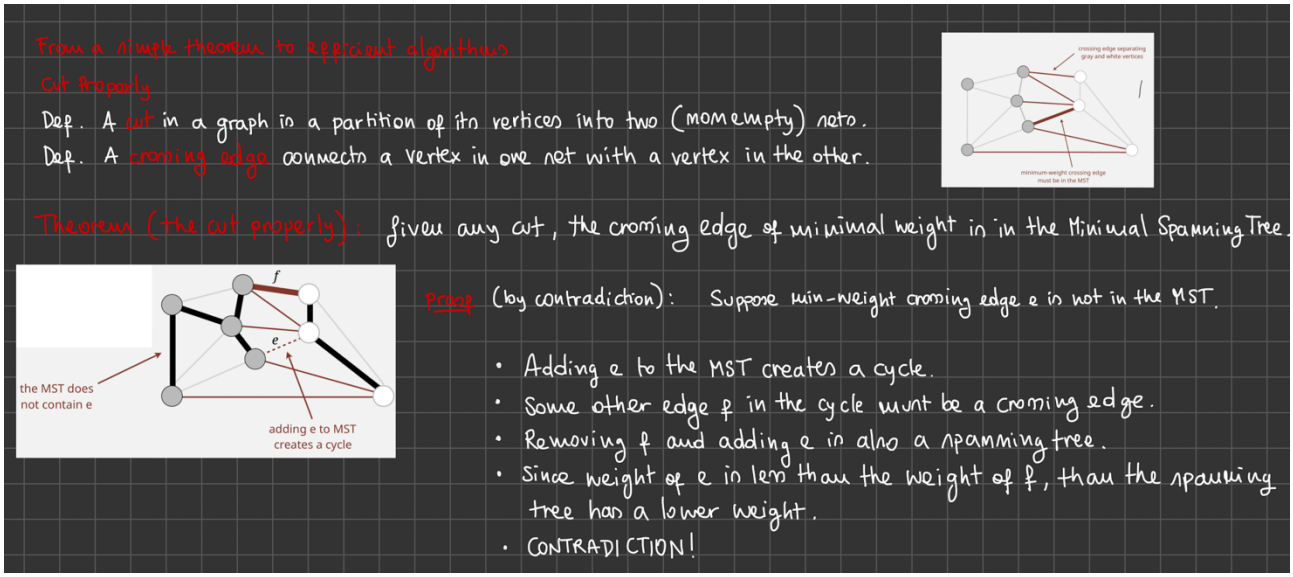
If the graph is **sufficiently sparse**, we can improve the algorithm by implementing the min-priority queue with a **binary min-heap**: **EXTRACT_MIN** operations takes **O(lg V)** time and there are **|V|** such operations. The **binary heap** can be build in **O(V)** time. Each **DECREASE** operation (in the RELAX) takes **O(lg V)** time and there are at most **|E|** such operations. The total running time is therefore **O((V + E) lg V)**. Which becomes **O(E lg V)** if all vertices in the graph is reachable from the source vertices.

Minimum Spanning Tree

A Minimum Spanning Tree is an acyclic subset of a graph that connects all of the vertices and whose total weight is minimized. Since this subset is acyclic and connects all the vertices, it must form a tree, which we call a **Spanning Tree** since it "spans" the graph G. We call the problem of determining the tree the **Minimum Spanning Tree Problem.**

Two are the algorithms that we have seen able to solve the problem, both of them are greedy algorithms: **Kruskal's Algorithm** and **Prim's Algorithm**. In **Kruskal's Algorithm**, the set A (subset of some minimum spanning tree) is a forest whose vertices are all those of the given graph. The edge added to A is always a least-weight edge in the graph that connects two distinct components. While in **Prim's Algorithm**, the Set A form a single tree (Arborescence Tree). The edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

The proof of this algorithm can be found in the **Cut Properly Theorem**:



The algorithm of **Prim's** can be explained through the following **Pseudocode**:

Prim(G)
Input: weighted graph $G(V, E)$
Output: minimum spanning tree $T \subseteq G$

```
begin
    Let T be a single vertex v from G
    while T has less than n vertices
    find the minimum edge connecting T to G – T
    add it to T
    end
end
```

The **Worst-Case Analysis** of the algorithm tells us, that the asymptotic behaviour of the algorithm is: **O (|V| log(|V|) + |E| log(|V|))** where **|V| log(|V|)** is the time needed to extract the next smallest node multiplied by the number of nodes, while **|E| log(|V|)** gives us the time needed to update the node cost multiplied by the number of edges.

Computational Complexity

Computational complexity theory focuses on classifying computational problems according to their inherent difficulty, and relating these classes to each other. A computational problem is a task solved by a computer. A computation problem is solvable by mechanical application of mathematical steps, such as an algorithm.

Optimization and Decision Problems

Many problems of interest are **optimization problems**, in which each feasible solution has an associated value, and we wish to find a feasible solution with the best value. NP – Completeness applies directly not to optimization problems, however, but to **decision problems**, in which the answer is simply “yes” or “no” (or, more formally, “1” or “0”).
Although NP - Complete problems are confined to the realm of decision problems, we can take advantage of a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.
For example, a decision problem related to **shortest - path** is **path**: given an undirected graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?
The relationship between an optimization problem and its related decision problem works in our favor when we try to sow that the optimization problem is "hard". That is because the decision problem is in a sense "easier", or at least "no harder".
As a specific example, we can solve **path** by solving **shortest-path** and then comparing the number of edges in the shortest path found to the value of the decision-problem parameter k. In other words, if an optimization problem is easy, its related decision problem is easy as well. Stated in a way that has more relevance to NP-Completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard.

Satisfiability Problem

This problem has the historical honor of being the first problem ever shown to be NP-Complete. We formulate the **(formula) satisfiability** problem in terms of the language SAT as follows. An instance of SAT is a boolean formula composed of:

- **n** boolean variables (literals): x_1, x_2, ..., x_n
- **m** boolean connectives: and, or, …
- parentheses.

A formula with a satisfying assignment is a **satisfiable** formula. The satisfiability problem asks whether a given boolean formula is satisfiable.

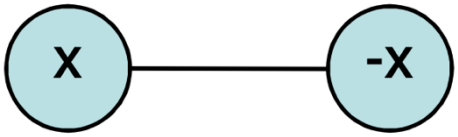
We define **3-CNF Satisfiability** using the following terms. A **literal** in a boolean formula is an occurence of a variable or its negation. A boolean formula is in **Conjunctive Normal Form** or **CNF**, if it is expressed as an AND of **clauses**, each of which is the OR of one or more literals. A boolean formula is in **3-conjunctive normal form**, or **3-CNF**, if each clause has exactly three distinct literals.

SAT to Independent Set:
The natural algorithmic problem is, given a graph, find the largest **Independent Set**. To turn this optimization problem into a decision problem, we define **IND** as:

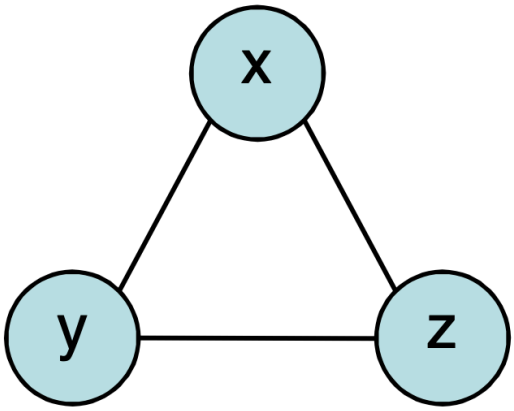
The set of pairs (G,K), where G is a graph, and K is an integer, such that G contains an independent set with K or more vertices. **IND** is in NP. We now show it is NP-complete. We can construct a reduction from **3SAT** to **IND**. A Boolean expression ϕ in 3CNF with m clauses is mapped by the reduction to the pair (G, m), where G is the graph obtained from ϕ as follows: G contains m triangles, one for each clause of ϕ, with each node representing one of the literals in the clause. Additionally, there is an edge between two nodes in different triangles if they represent literals where one is the negation of the other. If this graph has an independent set with m (or more) vertices then the such set would provide a solution for the associated 3-SAT problem.

SAT to Vertex Cover:
Let G be an undirected graph. A *vertex cover* of G is a subset COVER of V such that for every (u, v) ∈ E, at least one of u or v ∈ COVER.
Suppose our boolean formula for 3-SAT, ϕ, contains m variables and l clauses. For our vertex cover we will use k = m + 2l.

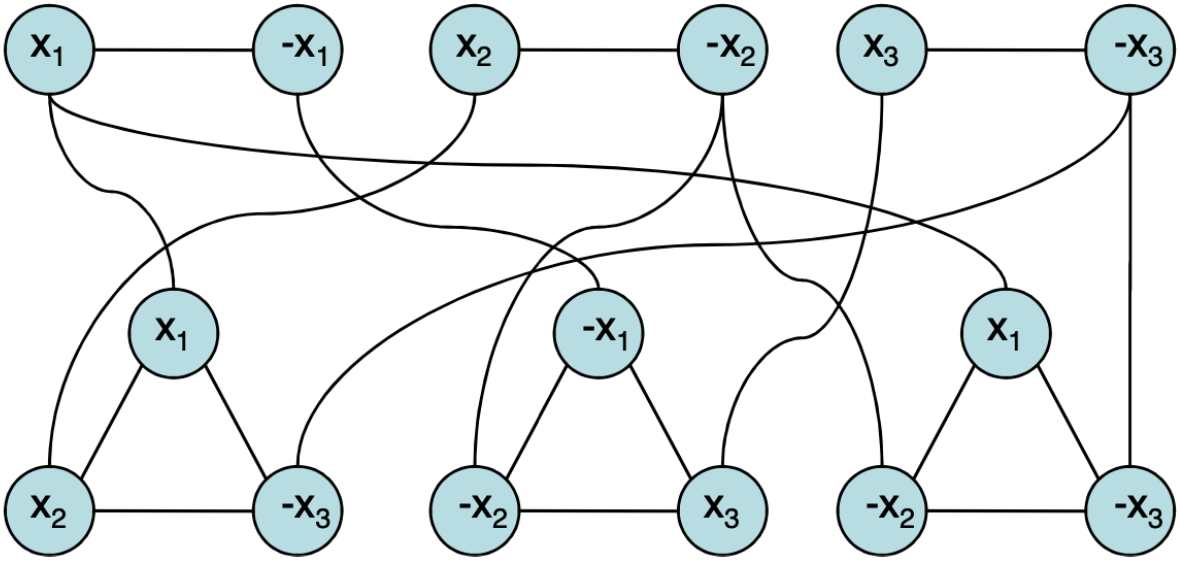
For each variable in ϕ we will use the following variable gadget:



The variable x in a ϕ for 3-SAT is replaced by two nodes in the graph. One represents the literal x, while the other represents the literal ¬x. For each clause in ϕ we will use the clause gadget:



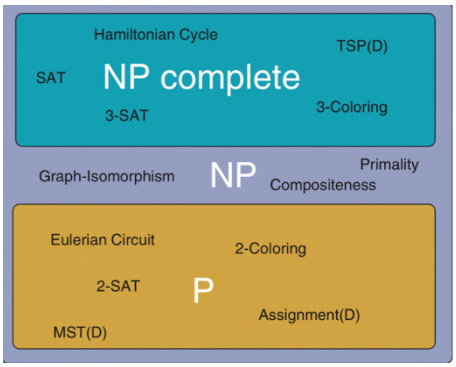
The 3-SAT clause is represented as a clique of 3 nodes, where each node represents a literal in the clause. The clause represented here is (X v Y v Z).
Connect the literals in the variable gadgets to the matching literals in the clause gadgets with an edge.
For example, if we have ϕ = $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \geq x_3)$ then we are looking for a vertex cover of size k = 3 + 2(3) = 9 on the graph in following figure:



At this point the reduction is complete. We still have to prove that this reduction is correct, **meaning that a vertex cover of size k = m + 2l exists if and only if the given is satisfiable**.

Complexity Classes

The branch of theoretical computer science known as computational complexity is concerned with classifying problems according to the computational resources required to solve them.
Almost all the algorithms we have studied thus far have been **polynomial-time algorithms**: on input of size n, their worst-case running time is O(n^k) for some constant k. However, not all problems can be solved in polynomial time.



From this assumption 3 Classes of Problems based on their computational complexity have been created:

- **Class P**, consists of those problems that are **solvable** in polynomial time.

- **Class NP**, consists of those problems that are **verifiable** in polynomial time. That is, if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem. Another way of saying the same thing is defining a problem in Class NP if and only if a nondeterministic algorithm can solve it in polynomial time.

  Any problem in the class P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate.

- **Class NP – Complete**, informally, a problem is in this class if it is in NP and is as “hard” as any problem in NP. Furthermore, if **any** NP-Complete problem can be solved in polynomial time, then **every** problem in NP has a polynomial-time algorithm. This peculiarity is given by the fact that each problem in NP can be polynomial reduced to an NP-Complete problem.

Cook’s Theorem

In computational complexity theory, the **Cook-Levin Theorem**, also known as **Cook's Theorem**, states that the boolean satisfiability is NP Complete. That is, any problem in NP can be reduced in polynomial time, by a deterministic Turing Machine to the problem of determining whether a boolean formula is satisfiable or not.

Reductions

Let us consider a decision problem A, which we would like to solve in polynomial time. We call the input to a particular problem an **instance** of that problem. Now, suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance **alpha of A** into some instance **beta of B** with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for alpha is "yes" if and only if the answer for beta is also "yes".

We call such procedure a polynomial-time **reduction algorithm** and it provides us a way to solve problem A in polynomial time.

Recalling that NP-Completeness is about showing how hard a problem is rather than how easy it is, we use polynomial-time reductions in the opposite way to show that a problem is NP-Complete. Let us take the idea a step further, and show how we could use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B. Suppose we have a decision problem A for which we already know that no polynomial-time algorithm exist. Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B. Now we can use a simple proof by contradiction to show that no polynomial-time algorithm can exist for B. Suppose otherwise, that is, suppose that B has a polynomial-time algorithm. Then we would have a way to solve problem A in polynomial time, which contradicts our assumption that there is no polynomial-time algorithm for A.