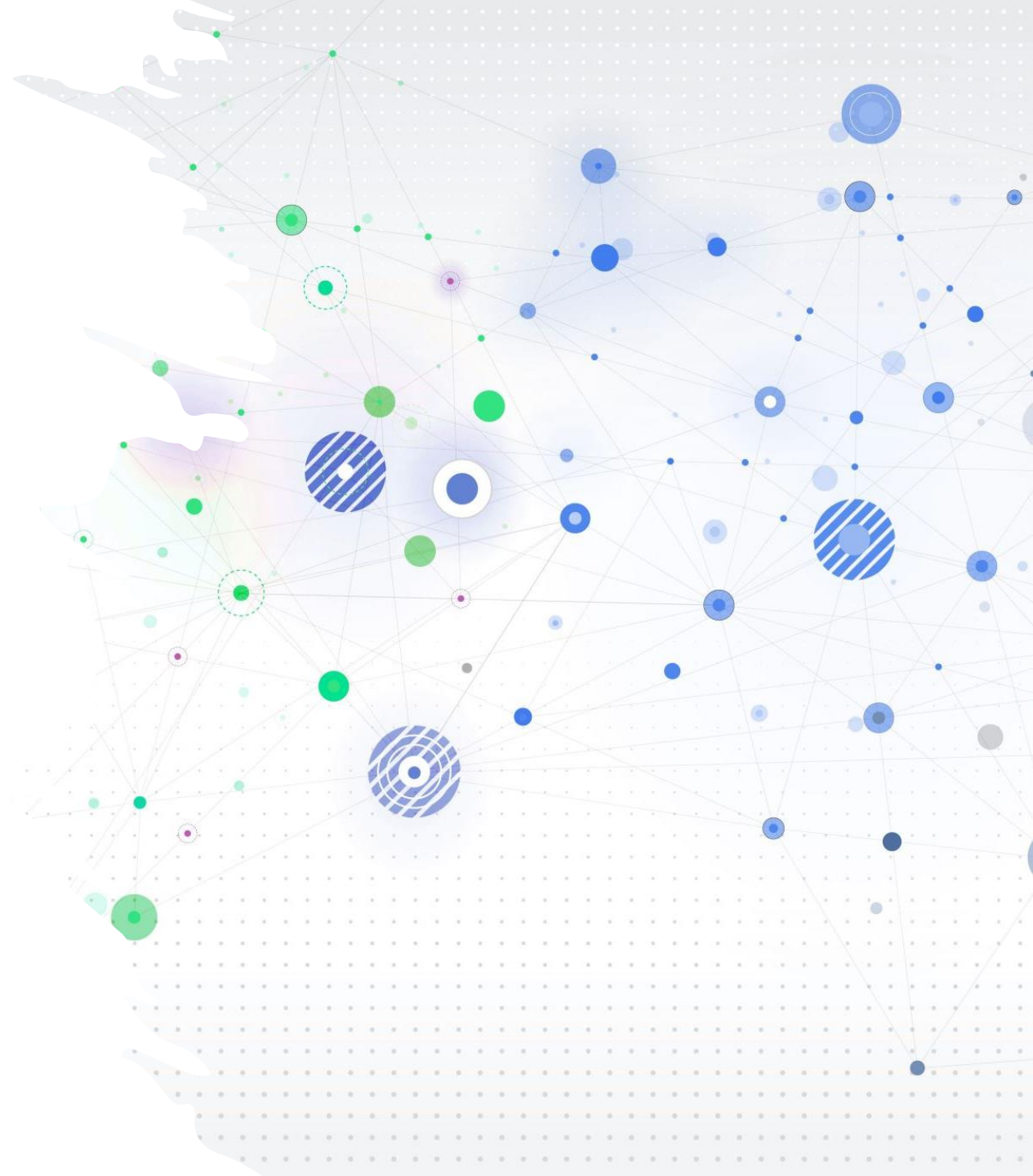


# Analisi del grafo di Wikipedia Italia 2013

Cristian Bargiacchi

Algoritmi per Programmazione e Analisi Dati 2023/2024



# Sommario

## 1. Distribuzione del grado in uscita

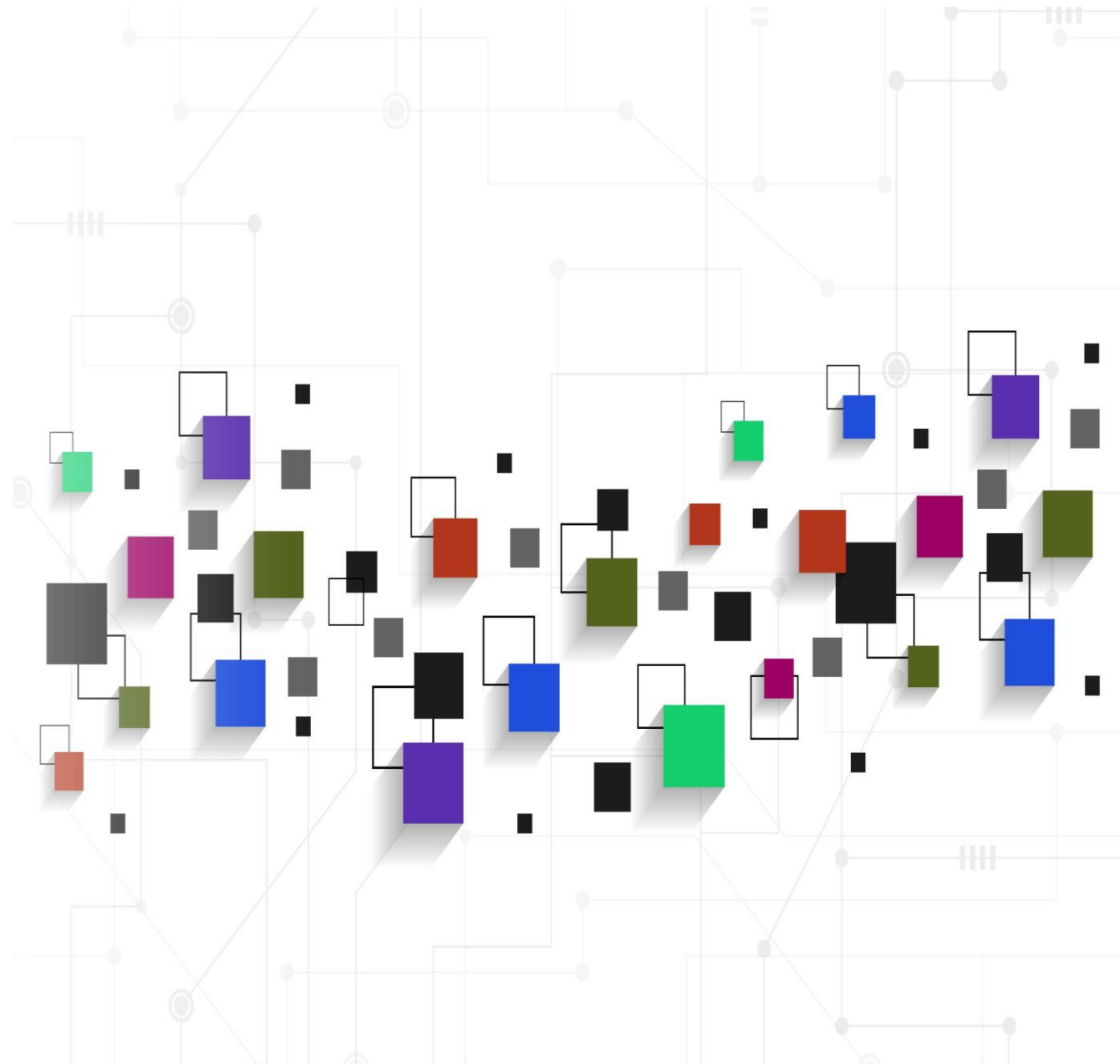
- *Quali sono le 10 pagine con maggiori link in uscita?*

## 2. Diametro della LCC in $U(G)$

- *Calcola anche per grafo senza pagine «disambigua»*

## 3. Trovare una clique massimale in $U(G)$

- *Per trovarne 2?*





# 1. Distribuzione grado in uscita

- **Calcolo grado in uscita di un nodo**

*Nodo* → Lista dei vicini uscenti (grado è la lunghezza della lista) («**neighbors\_dict**»)

- **Calcolo distribuzione**

*Grado* → Frequenze

- **10 pagine con grado maggiore?**

*Grado* → Lista dei nodi con quel grado

*Grado* → [Lista dei nodi con quel grado, Frequenze]  
(«**distribution\_dict**»)

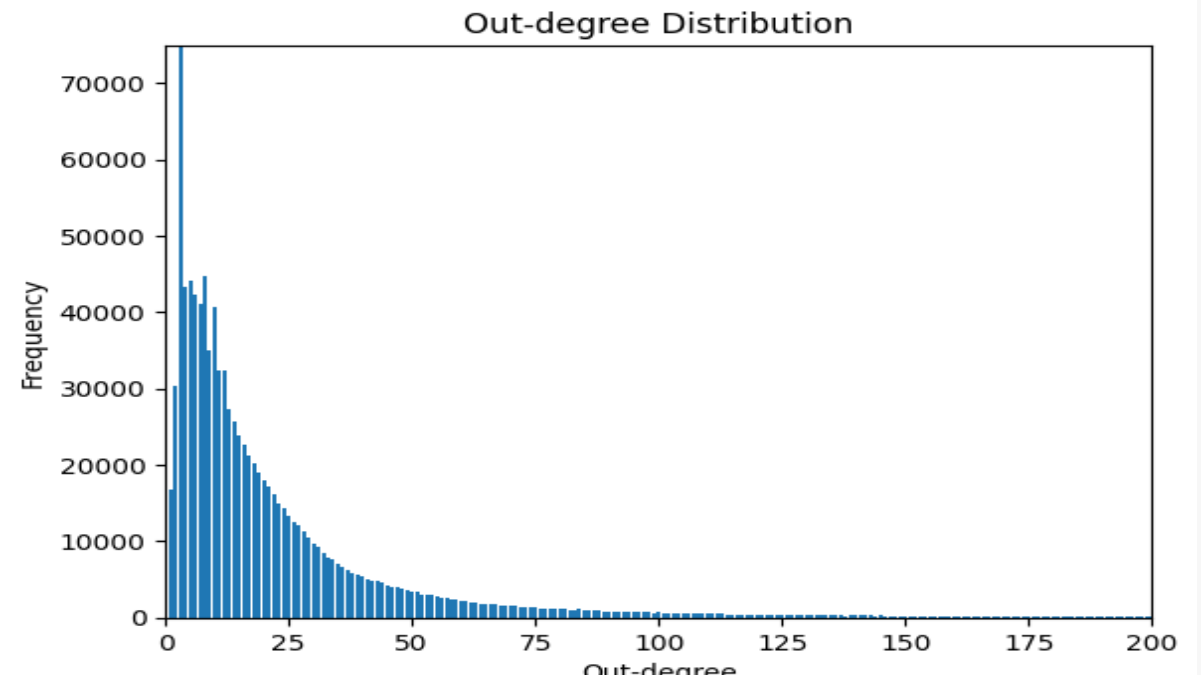
```

def outDegree_distribution (g:nx.DiGraph) -> dict:
    neighbors_dict = {node: list(g.successors(node)) for node in g.nodes()}
    distribution_dict = {}
    for node in neighbors_dict.keys():
        degree_node = len(neighbors_dict[node])
        if degree_node in distribution_dict:
            distribution_dict[degree_node][0].append(node)
            distribution_dict[degree_node][1] += 1
        else:
            distribution_dict[degree_node] = [[node], 1]
    return distribution_dict
g_outDegree = outDegree_distribution(wiki_g)

```

Complessità:  **$O(n)$**

Tempo  $\approx$  1 minuto



```
max_degree = max(degrees)
top_10 = list()

while len(top_10)<10:
    if max_degree in g_outDegree.keys():
        nodes = list(g_outDegree[max_degree][0])
        top_10.extend(nodes)
        max_degree = max_degree - 1
top10 = top_10[:10]
```

```
1          Città dell'India
2  Classificazione Nickel-Strunz
3          Nati nel 1981
4          Nati nel 1985
5          Nati nel 1983
6          Nati nel 1984
7          Nati nel 1986
8          Nati nel 1982
9          Nati nel 1980
10         Nativi del Veneto
```

Complessità: **O(n)**

Tempo  $\approx$  30 sec



## 2. Diametro della LCC nel grafo indiretto

Dopo aver trasformato  $G$  in  $U(G)$  e selezionato la LCC,

1. BFS dal nodo  $u$  con grado più alto nella LCC
2. Dizionario  $B(u)$ , che associa ogni nodo alla sua distanza da  $u$
3. Algoritmo *iFub*

```
def customBFS(LCC, startNode):  
    visited = {}  
    queue = Queue()  
    queue.put(startNode)  
    visited[startNode] = 0  
    while not queue.empty():  
        currentNode = queue.get()  
        for nextNode in LCC.neighbors(currentNode):  
            if nextNode not in visited:  
                queue.put(nextNode)  
                visited[nextNode]=visited[currentNode]+1  
    B_u = defaultdict(list)  
    for key, value in visited.items():  
        B_u[value].append(key)  
    return B_u
```

Complessità:  **$O(m)$**

```

def computeDiameter(LCC, Bu):
    i = lb = max(Bu)
    ub = 2*lb
    while ub > lb:
        eccDict = nx.eccentricity(LCC, Bu[i])
        Bi = max(eccDict.values())
        maxVal = max(Bi, lb)
        if maxVal > 2*(i - 1):
            return print("diametro: ", maxVal)
        else:
            lb = maxVal
            ub = 2*(i - 1)
        i = i - 1
    return print("Diametro iFub: ", lb)

startNode = max(LCC.degree, key=lambda x: x[1])[0]
B_u = customBFS(LCC, startNode)
computeDiameter(LCC, B_u)

```

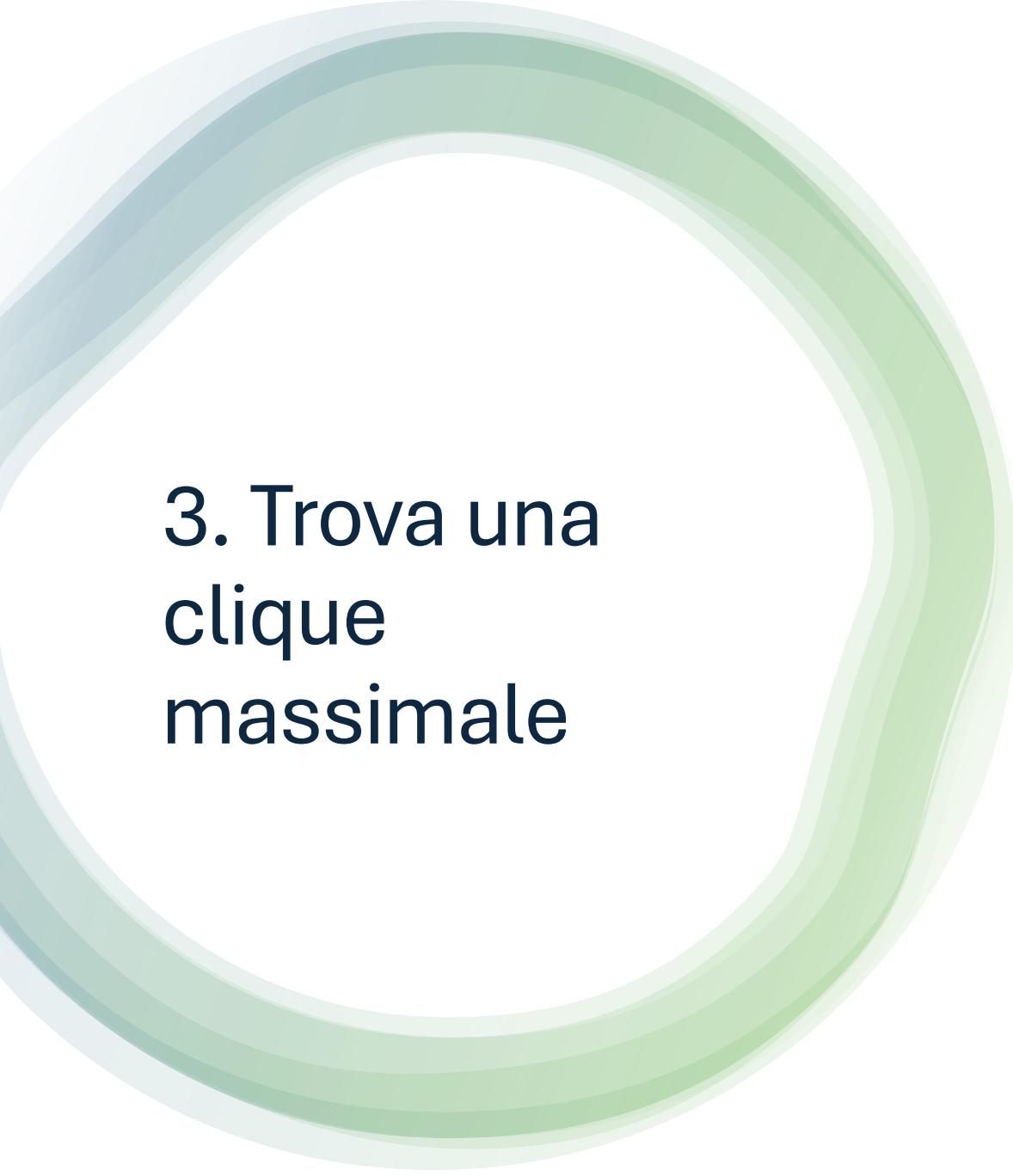
```

Diametro iFub: 8

```

Tempo  $\approx$  2 min





### 3. Trova una clique massimale

**Clique di ordine  $k$ :** sottografo di  $k$  nodi tutti connessi tra loro

**Clique massimale:** clique in cui non è più possibile aggiungere nodi senza perdere la proprietà di essere una clique

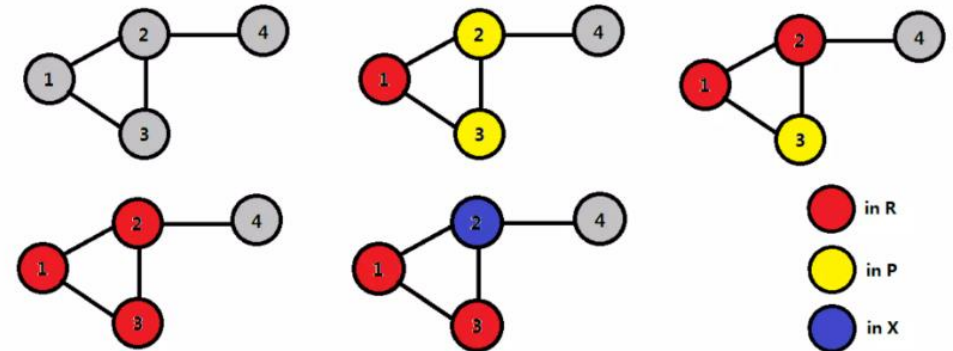
# Algoritmo di Bron-Kerbosch

**R:** insieme dei nodi che formeranno la clique (init: vuoto)

**P:** insieme dei nodi candidati a far parte di una clique (init: V)

**X:** insieme dei nodi già visitati e scartati, o facenti già parte di un'altra clique (init: vuoto)

```
algorithm BronKerbosch1(R, P, X) is
  if P and X are both empty then
    report R as a maximal clique
  for each vertex v in P do
    BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∪ N(v))
  P := P \ {v}
  X := X ∪ {v}
```



### 3. Trova una clique massimale

```
def find_a_maximal_clique(G:nx.Graph()):
    nodes = list(G.nodes())
    for i in range(len(nodes)):
        start_node = choice(nodes)
        R = {start_node}
        P = set(G.neighbors(start_node))
        X = set()
        myclique = Bron_Kerbosch(G, R, P, X)
        if myclique and len(myclique) >= 3:
            myclique_pages = set()
            for id in myclique:
                myclique_pages.add(id_to_page[id])
            return print("Una clique massimale di ordine 3 o superiore è: ", myclique_pages)
    return "Nessuna clique di ordine 3 trovata"
```

Complessità caso peggiore:  $O(3^{n/3})$

```
find_a_maximal_clique(U_g)
```

Una clique massimale di ordine 3 o superiore è: {'Anders Sandøe Ørsted', 'Danimarca', 'Primi ministri della Danimarca'}

# Per trovare n cliques massimali?

```
def find_n_maximal_cliques(G, n):
    nodes = list(G.nodes())
    maximal_cliques = []
    iter = 0
    while len(maximal_cliques) < n:
        iter += 1
        start_node = choice(nodes)
        R = {start_node}
        P = set(G.neighbors(start_node))
        X = set()
        myclique = Bron_Kerbosch(G, R, P, X)
        if myclique and len(myclique) >= 3:
            duplicate = False
            for old_clique in maximal_cliques:
                if myclique == old_clique:
                    duplicate = True
            if not duplicate:
                maximal_cliques.append(myclique)
                myclique_pages = set()
                for id in myclique:
                    myclique_pages.add(id_to_page[id])
                print("Clique massimale ", iter, ": ", myclique_pages)

    return maximal_cliques
```

Clique massimale 1 : {"Vicariato apostolico dell'Arabia meridionale", "Congregazione per l'Evangelizzazione dei Popoli", '1933', '21 marzo'}

Clique massimale 2 : {'Distretto di Aleksandrów Kujawski', 'Voivodato della Cuiavia-Pomerania', 'Lista dei distretti della Polonia', 'Distretti della Polonia'}  
[{385431, 395488, 397927, 399105}, {421220, 881526, 882082, 882093}]



**GRAZIE PER  
L'ATTENZIONE!**

## Calcolo diametro – LCC grafo senza «disambigua»

```
filtered_page_to_id = {page: id for page, id in page_to_id.items() if "disambigua" not in page}
filtered_ids = set(filtered_page_to_id.values())
filtered_arcs_df = arcs_df_prova[arcs_df_prova['v1'].isin(filtered_ids) & arcs_df_prova['v2'].isin(filtered_ids)]

def create_graph_without_disambigua(df: pd.DataFrame, valid_ids: set) -> nx.DiGraph:
    G = nx.DiGraph()
    for _, line in df.iterrows():
        v1 = int(line['v1'])
        v2 = int(line['v2'])
        if v1 in valid_ids and v2 in valid_ids:
            G.add_edge(v1, v2)
    return G

G_without_disambigua_prova = create_graph_without_disambigua(filtered_arcs_df, filtered_ids)

LCC_dis = get_largest_cc_undirected(G_without_disambigua_prova)
startNode_dis = max(LCC_dis.degree, key=lambda x: x[1])[0]
B_u_dis = customBFS(LCC_dis, startNode_dis)
computeDiameter(LCC_dis, B_u_dis)
```

Diametro iFub: 8

# Implementazione algoritmo di Bron-Kerbosch

```
U_g = nx.to_undirected(wiki_g)
```

```
def Bron_Kerbosch(G, R, P, X):  
    if not P and not X:  
        return R  
    for v in list(P):  
        myclique = Bron_Kerbosch(  
            G,  
            R.union({v}),  
            P.intersection(G.neighbors(v)),  
            X.intersection(G.neighbors(v))  
        )  
        if myclique and len(myclique) >= 3:  
            return myclique  
        P.remove(v)  
        X.add(v)  
    return None
```