



# Graph Theory for Clustering Algorithms

Algorithms & Data Structure for  
Physics of Data

Elisa Stabilini

30 Aprile 2024



# Table of Contents

Nozioni sul clustering

Elementi di teoria dei grafi

Chameleon algorithm

Implementazione



## Cos'è il clustering

- Procedimento di *ricerca* e *scoperta* tipico del data mining [2]
- Esempio di unsupervised-learning [5]
- Raggruppare dati per massimizzare la *somiglianza* all'interno del cluster e minimizzarla tra i differenti cluster [2], [5]
- Principali categorie di algoritmi: density-based, clustering gerarchico, clustering spettrale



## Nozioni sui grafi I

- Un **grafo**  $G$  è una struttura matematica discreta definita da una coppia  $(V, E)$  dove  $V$  è un set finito di vertici (o nodi) e  $E$  è un set di coppie (eventualmente ordinate)  $\{u, v\}$  di nodi, gli elementi di  $E$  sono chiamati archi.
  - **Grafo pesato**: è un grafo  $G = (V, E, w)$  dove  $w : E \rightarrow \mathbb{R}$  è una funzione che associa un numero reale, peso (o costo), ad ogni arco
  - **Grafo diretto**: è un grafo in cui gli archi sono orientati (quindi possono essere uscenti o entranti rispetto a un nodo)
- Grafi **dinamici**: grafi i cui parametri variano durante l'esecuzione del programma
- Grafo **sparso**: il numero di archi è molto inferiore rispetto al massimo numero possibile di archi tra i nodi.

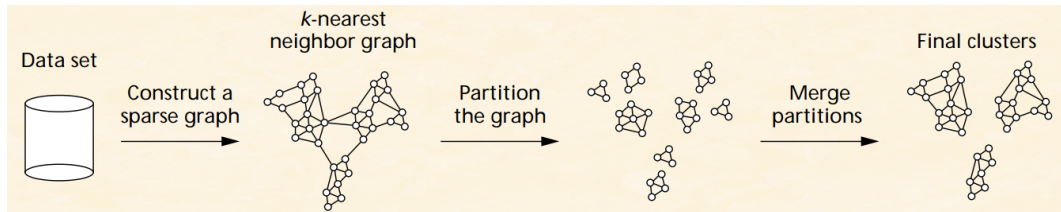


## Nozioni sui grafi II

- Dato un grafo  $G(V, E)$ , si definisce **taglio**  $C(S, T)$  una *partizione* dell'insieme dei vertici  $V$  di  $G$ , con  $S, T \subseteq V$
- Si definisce **insieme di taglio** di un taglio l'insieme degli archi  $\{(u, v) \in E \mid u \in S, v \in T\}$
- Si definisce **dimensione del taglio** il numero di archi nell'insieme di taglio. Se gli archi sono pesati la dimensione del taglio è data dalla somma dei pesi.
- Si definisce **min-cut bisector** una partizione di un grafo in *due* parti di numerosità simile che minimizza la dimensione dell'insieme di taglio.



## Workflow dell'algoritmo



**Figure:** Funzionamento dell'algoritmo  
fonte: immagine tratta da [2]



## Caratteristiche dell'algoritmo

Utilizza un **modello dinamico** per valutare la somiglianza dei punti: due clusters sono uniti solo se la loro *interconnettività* e *vicinanza* sono alte rispetto alla interconnettività e vicinanza interna. Algoritmo graph-based in due fasi:

1. Utilizza un algoritmo di **partizione di grafi**: raggruppa i data points in molti sotto-cluster relativamente piccoli
2. Utilizza un algoritmo di **clustering gerarchico agglomerativo**: trova i veri e propri cluster combinando in maniera iterativa i sotto-cluster



## Definizione parametri

- **Inter-connettività interna assoluta**  $|EC_{\{C_i, C_i\}}|$  di un grafo  $C_i$ : dimensione del min-cut bisector di  $C_i$
- **Inter-connettività assoluta**  $|EC_{\{C_i, C_j\}}|$  di due grafi  $C_i$  e  $C_j$ : Dimensione dell'insieme di taglio che definisce  $C_i$  e  $C_j$ .
- **Vicinanza interna assoluta**  $\bar{S}_{EC_{\{C_i\}}}$ : peso medio degli archi del min-cut bisector
- **Vicinanza assoluta**  $\bar{S}_{EC_{\{C_i, C_j\}}}$ : peso medio degli archi che connettono i vertici di  $C_i$  ai vertici di  $C_j$





## Descrizione dell'algoritmo I

### Costruzione del grafo kNN

- Input: dataset
- Ciascun nodo del grafo rappresenta un data-point
- Tra due vertici esiste un arco se il dato corrispondente a uno dei nodi è tra i k-elementi più simili al data-point corrispondente all'altro nodo



## Descrizione dell'algoritmo II

### Algoritmo di partizione

- Partizione con min-cut bisector (minimizzazione edge-cut)
- Bilanciamento: ciascuno dei due sotto-cluster contiene almeno il 25% dei nodi in  $C$
- Iterativamente viene partizionato il cluster più grande
- La partizione si interrompe quando la dimensione del cluster più grande raggiunge 2% del dataset



## Descrizione dell'algoritmo III

### Clustering agglomerativo

Valutazione della somiglianza tra due clusters tramite

1. Interconnettività relativa  $RI(C_i, C_j)$ :

$$RI(C_i, C_j) = \frac{|EC_{\{C_i, C_j\}}|}{\frac{1}{2}(|EC_{\{C_i, C_i\}}| + |EC_{\{C_j, C_j\}}|)} \quad (1)$$



## Descrizione dell'algoritmo IV

2. Vicinanza relativa  $RC(C_i, C_j)$ :

$$RC(C_i, C_j) = \frac{\bar{S}_{EC\{C_i, C_j\}}}{\frac{|C_i|}{|C_i|+|C_j|} \bar{S}_{EC\{C_i, C_i\}} + \frac{|C_j|}{|C_i|+|C_j|} \bar{S}_{EC\{C_j, C_j\}}} \quad (2)$$

dove:

- $\bar{S}_{EC\{C_i, C_j\}}$  e  $\bar{S}_{EC\{C_i, C_i\}}$  sono rispettivamente la vicinanza e la vicinanza interna assoluta.
- $|C_i|$  e  $|C_j|$  rappresentano la cardinalità rispettivamente dell' $i$ -esimo e  $j$ -esimo



## Descrizione dell'algoritmo V

### Unione dei clusters:

- Vengono uniti i cluster con inter-connettività e vicinanza relativa superiori rispettivamente a  $T_{RI}$  e  $T_{RC}$
- $T_{RI}$  e  $T_{RC}$  sono soglie definite dall'utente
- termina quando non ci sono più coppie di cluster con valori superiori alla soglia



## Performance I

Complessità totale  $\mathcal{O}(nm + n \log n + m^2 \log m)$

Dipende da quanto tempo è richiesto per costruire il grafo dei kNN e quanto per completare i due passaggi dell'algoritmo

- Costruzione kNN:
  - dataset piccolo-dimensionali:  $\mathcal{O}(n \log n)$ <sup>1</sup>
  - dataset multi-dimensionali tempo medio di ricerca dei kNN è  $\mathcal{O}(n^2)$

---

<sup>1</sup>utilizzo algoritmi basati su *k-d trees* [11]. Per questi algoritmi il costo medio di inserimento e ricerca dei k-NN è  $\mathcal{O}(\log n)$  [6]



- Tempo richiesto dall'algoritmo in due step: dipende da numero iniziale  $m$  di sub-cluster. Per semplificare i calcoli assumo:
  - a. ogni sub-cluster ha un numero  $n/m$  di nodi
  - b. durante ciascuno step di *merging* unisco *al più una coppia* di cluster
- 1. **Algoritmo di partizione:** Il numero di step dipende dalla complessità in tempo della partizione con *metis*:  $\mathcal{O}(n \log(n/m))^2$
- 2. **Clustering agglomerativo** La complessità in tempo di questa fase dipende da:
  - tempo impiegato per calcolare inter-connettività e vicinanza dei grafi  $\mathcal{O}(nm)$
  - tempo impiegato per selezionare i cluster più simili da fondere:  $\mathcal{O}(m^2 \log m)$ <sup>3</sup>

---

<sup>2</sup>In realtà *metis* impiega  $\mathcal{O}(|V| + |E|)$  per calcolare la bisezione ma dato che *chameleon* lavora su grafi kNN  $|E| = |V|$  quindi la complessità computazionale è  $\mathcal{O}(|V|)$

<sup>3</sup>Se si usa una *heap-based priority queue*



## Librerie utilizzate

- `numpy` [4]: gestione array ecc.
- `networkx` [7]: costruzione dei grafi
- `tqdm` [3]: visualizzazione avanzamento algoritmo
- `scikit-learn` [10]: Utilizzo tecniche di clustering differenti
- `metis` [9]: algoritmi di partizione e merging dei grafi
- `seaborn` [12]: costruzione e gestione grafici





## Classi e funzioni I

```
def knn_graph(df, k, verbose=False):
    points = [p[1:] for p in df.itertuples()]
    g = nx.Graph()
    for i in range(0, len(points)):
        g.add_node(i)
    if verbose:
        print("Building kNN graph (k = %d)..." % (k))
    iterpoints = tqdm(enumerate(points), total=len(
        points)) if verbose else enumerate(points)
    for i, p in iterpoints:
        distances = list(map(lambda x: euclidean_distance(p, x), points))
        closests = np.argsort(distances)[1:k+1] # second trough kth closest
        # print(distances[0])
        for c in closests:
            g.add_edge(i, c, weight=1.0 / distances[c], similarity=int(
                1.0 / distances[c] * 1e4))
        g.nodes[i]['pos'] = p
    g.graph['edge_weight_attr'] = 'similarity'
    return g
```

**Figure:** Funzione per la costruzione del grafo KNN



## Classi e funzioni II

```
def part_graph(graph, k, df=None):  
    edgecuts, parts = metis.part_graph(  
        graph, 2, objtype='cut', ufactor=250)  
    print(edgecuts)  
    for i, p in enumerate(graph.nodes()):  
        graph.node[p]['cluster'] = parts[i]  
    if df is not None:  
        df['cluster'] = nx.get_node_attributes(graph, 'cluster').values()  
    return graph
```

**Figure:** Funzione per la partizione del grafo



## Classi e funzioni III

```
#compute internal closness
def internal_closeness(graph, cluster):
    cluster = graph.subgraph(cluster)
    edges = cluster.edges()
    weights = get_weights(cluster, edges)
    return np.sum(weights)

#compute relative closness
def relative_closeness(graph, cluster_i, cluster_j):
    edges = connecting_edges((cluster_i, cluster_j), graph)
    if not edges:
        return 0.0
    else:
        SEC = np.mean(get_weights(graph, edges))
    Ci, Cj = internal_closeness(
        graph, cluster_i), internal_closeness(graph, cluster_j)
    SECci, SECcj = np.mean(bisection_weights(graph, cluster_i)), np.mean(
        bisection_weights(graph, cluster_j))
    return SEC / ((Ci / (Ci + Cj) * SECci) + (Cj / (Ci + Cj) * SECcj))
```

**Figure:** Funzioni per il calcolo della vicinanza all'interno del singolo grafo e tra grafi diversi



## Classi e funzioni IV

```
# compute internal_interconnectivity
def internal_interconnectivity(graph, cluster):
    return np.sum(bisection_weights(graph, cluster))

#compute relative interconnectivity
def relative_interconnectivity(graph, cluster_i, cluster_j):
    edges = connecting_edges((cluster_i, cluster_j), graph)
    EC = np.sum(get_weights(graph, edges))
    ECci, ECcj = internal_interconnectivity(
        graph, cluster_i), internal_interconnectivity(graph, cluster_j)
    return EC / ((ECci + ECcj) / 2.0)
```

**Figure:** Funzioni per il calcolo della connettività all'interno del singolo grafo e tra grafi diversi



## Classi e funzioni V

```
def merge_best(graph, df, a, k, verbose=False):
    clusters = np.unique(df['cluster'])
    max_score = 0
    ci, cj = -1, -1
    if len(clusters) <= k:
        return False

    for combination in itertools.combinations(clusters, 2):
        i, j = combination
        if i != j:
            if verbose:
                print("Checking c%d c%d" % (i, j))
            gi = get_cluster(graph, [i])
            gj = get_cluster(graph, [j])
            edges = connecting_edges(
                (gi, gj), graph)
            if not edges:
                continue
            ms = merge_score(graph, gi, gj, a)
```

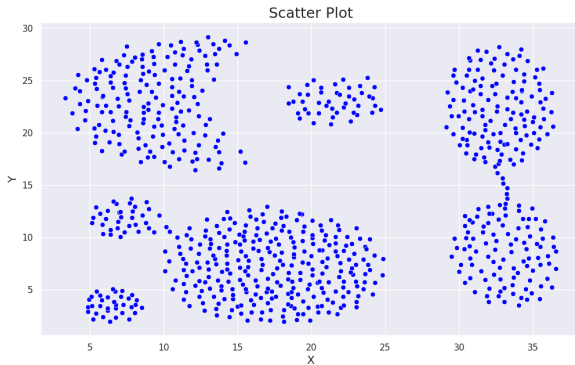
```
        if verbose:
            print("Merge score: %f" % (ms))
        if ms > max_score:
            if verbose:
                print("Better than: %f" % (max_score))
            max_score = ms
            ci, cj = i, j

    if max_score > 0:
        if verbose:
            print("Merging c%d and c%d" % (ci, cj))
        df.loc[df['cluster'] == cj, 'cluster'] = ci
        for i, p in enumerate(graph.nodes()):
            if graph.node[p]['cluster'] == cj:
                graph.node[p]['cluster'] = ci
    return max_score > 0
```

**Figure:** Funzione per il merging dei grafi



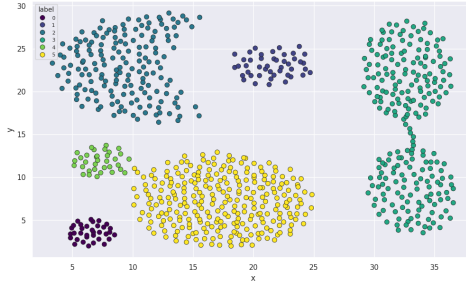
## Analisi correttezza I



**Figure:** Scatterplot del dataset su cui è stato applicato l'algoritmo (dataset: [1])



## Analisi correttezza II



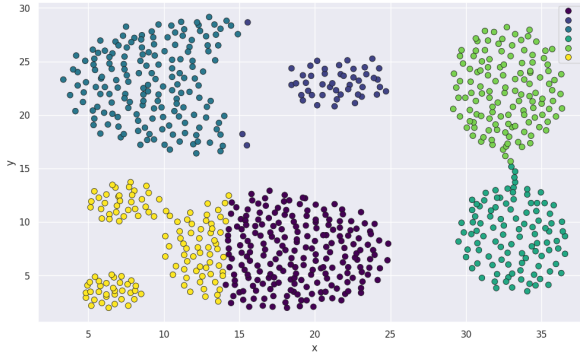
Parametri:

- $k=10$
- $\text{MinSize}=2\%$
- $T_{RI}=0.75$
- $T_{RC}=0.75$

**Figure:** Risultato ottenuto applicando Chameleon



## Analisi correttezza III



**Figure:** Risultato ottenuto applicando KMeans

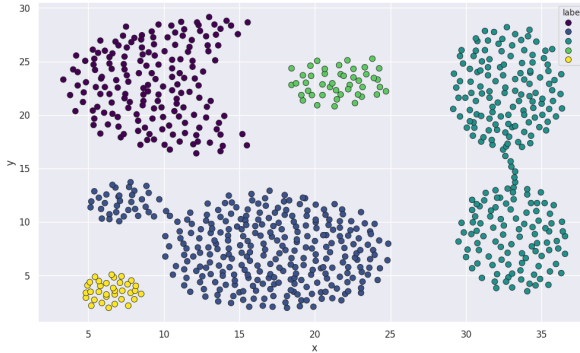
Parametri:

- `n_clusters = 6`
- `random_state = 42`





## Analisi correttezza IV



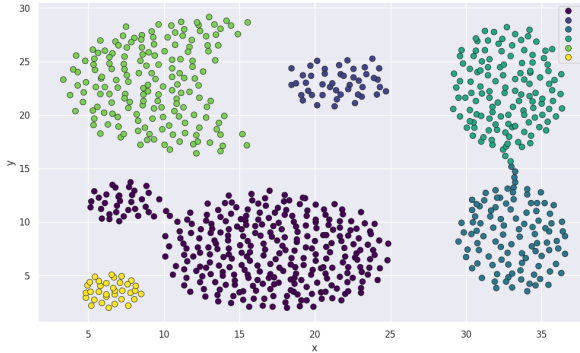
**Figure:** Risultato ottenuto applicando DBSCAN

Parametri:

- $\text{eps}=2$
- $\text{min\_samples}=5$



## Analisi correttezza V



Parametri:

- `n_clusters=6`
- `random_state=42`

**Figure:** Risultato ottenuto applicando clustering spettrale



## References I



Aggregation, 2018.



Ming-Syan Chen, Jiawei Han, and P.S. Yu.

Data mining: an overview from a database perspective.

*IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, 1996.



Casper da Costa-Luis et al.

tqdm: A fast, extensible progress bar for python and cli, 2024.



Charles R. Harris et al.

Array programming with NumPy.

*Nature*, 585(7825):357–362, 2020.



## References II



Scitovski et al.

*Introduction.*

Springer International Publishing, 2021.



Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel.

An Algorithm for Finding Best Matches in Logarithmic Time.

*ACM Trans. Math. Software*, 3:209–226, 1977.





Aric Hagberg, Pieter Swart, and Daniel Chult.

Exploring network structure, dynamics, and function using networkx.

01 2008.




## References III

-  George Karypis and Vipin Kumar.  
Metis: A software package for partitioning unstructured graphs, partitioning meshes,  
and computing fill-reducing orderings of sparse matrices.  
<https://conservancy.umn.edu/handle/11299/215346>, 1997.
-  George Karypis and Vipin Kumar.  
MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version  
4.0.  
<http://www.cs.umn.edu/~metis>, 2009.



## References IV

 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay.  
Scikit-learn: Machine learning in Python.  
*Journal of Machine Learning Research*, 12:2825–2830, 2011.

 Hanan Samet.  
*The design and analysis of special data structures*.  
Addison-Wesley series in computer science. Addison Wesley, London, England, September 1989.



## References V



Michael L. Waskom.

seaborn: statistical data visualization.

*Journal of Open Source Software*, 6(60):3021, 2021.



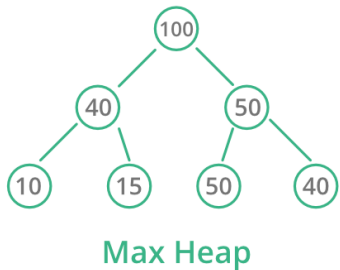
Grazie dell'attenzione!





## Heap data structure

- Struttura dati ad **albero binario completo** che soddisfa la proprietà di heap: per ogni nodo, il valore dei suoi figli è minore (maggiore) o uguale al proprio valore
- Esistono due tipi di heap:
  1. MaxHeap: La radice contiene il valore massimo
  2. MinHeap: La radice contiene il valore minimo



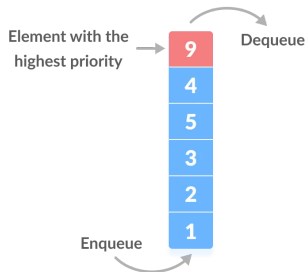
**Figure:** Esempio di MaxHeap DS

fonte: <https://www.geeksforgeeks.org>



# Priority Queue

- Estensione della SD queue: a ciascun elemento è associata una priorità
- Un elemento con alta priorità viene rimosso dalla coda prima di un elemento con bassa priorità
- Se due elementi hanno la stessa priorità, vengono serviti secondo il loro ordine nella coda

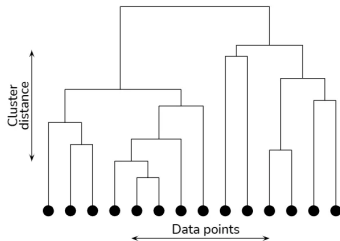


**Figure:** Esempio funzionamento priority queue  
fonte: <https://www.programiz.com/dsa/priority-queue>



## Clustering Gerarchico (HCA)

- Metodo di studio e analisi di cluster che costruisce una gerarchia tra clusters
- Si distinguono due strategie di clustering gerarchico
  1. Gerarchico agglomerativo: metodo bottom-up, inizialmente ciascun dato è un cluster, progressivamente vengono uniti i cluster più simili
  2. Gerarchico divisivo: approccio top-down, inizialmente tutti i dati fanno parte di un unico cluster e vengono progressivamente divisi



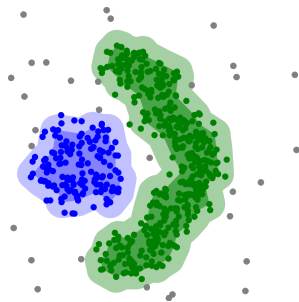
**Figure:** Esempio di dendrogramma da clustering gerarchico

fonte: <https://towardsdatascience.com/>



## Density based clustering

- famiglia di metodi che identifica i cluster sulla base della densità dei data points
- sono in grado di identificare clusters di forma arbitraria anche irregolare
- funzionano bene in caso di spazi piccolo-dimensionali



**Figure:** Esempio di clustering density-based

fonte: <https://en.wikipedia.org/wiki/DBSCAN>



## Confronto tra algoritmi di clustering I

Principali aspetti positivi del clustering gerarchico:

- Non è necessario specificare il numero  $k$  di clusters
- Empiricamente si osserva una migliore performance rispetto ai metodi k-means
- Se servono  $k$  cluster basta considerare i  $k - 1$  rami principali

Principali aspetti negativi del clustering gerarchico:

- non possono mai annullare scelte prese in precedenza
- La complessità in tempo scala sempre almeno come  $O(n^2)$  dove  $n$  è il numero di oggetti



## Confronto tra algoritmi di clustering II

Principali aspetti positivi del clustering density-based:

- Robusti rispetto alla presenza di rumore di fondo
- In grado di identificare clusters con forme particolari

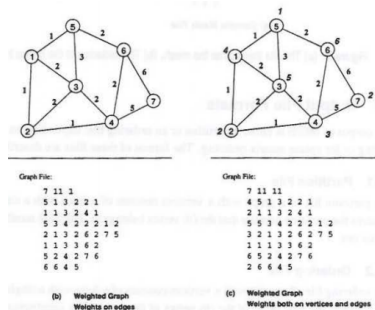
Principali aspetti negativi del clustering density-based:

- Sempre complessità in tempo  $\mathcal{O}(n^2)$
- Particolarmente sensibili all'inizializzazione dei parametri iniziali



## Libreria METIS

- pacchetto software per la partizione di grafi irregolari di grandi dimensioni e il calcolo di ordinamenti che riducono il riempimento di matrici sparse [8]
- formato di salvataggio grafi: file .txt con
  - prima riga: informazioni su lgraofo (numero nodi, archi, tipo di pesi)
  - righe successive: liste di adiacenza dei nodi



**Figure:** Esempio di formato di salvataggio di un grafo. Fonte: [8]



## METIS\_PartGraphKway ( )

METIS\_PartGraphKway(*int* \**n*, *idx*type \**xadj*, *idx*type \**adjncy*, *idx*type \**vwgt*, *idx*type \**adjwgt*, *int* \**wgtflag*, *int* \**numflag*, *int* \**nparts*, *int* \**options*, *int* \**edgcut*, *idx*type \**part*)[8] Descrizione

parametri:

- *n*: numero di vertici nel grafo.
- *xadj*, *adjncy*: La struttura di adiacenza del grafo come descritto nella Sezione 4.1.
- *vwgt*, *adjwgt*: Informazioni sui pesi dei vertici e degli archi come descritto nella Sezione 4.1.
- *wgtflag*: Utilizzato per indicare se il grafo è pesato.
  - 0: Nessun peso (*vwgts* e *adjwgt* SONO NULL)
  - 1: Pesi solo sugli archi (*vwgts* = NULL)





## METIS\_PartGraphKway ( ) ||

- 2: Pesi solo sui vertici (`adjwgt = NULL`)
- 3: Pesi sia sui vertici che sugli archi.
- *numflag*: Utilizzato per indicare quale schema di numerazione è usato per la struttura di adiacenza del grafo. Può assumere i seguenti due valori:
  - 0: Numerazione in stile C che inizia da 0
  - 1: Numerazione in stile Fortran che inizia da 1
- *nparts*: numero di parti in cui dividere il grafo.
- *options*: array di 5 interi utilizzato per passare parametri per le varie fasi dell'algoritmo.
- *edgecut*: Al termine memorizza il numero di archi tagliati dalla partizione.
- *part*: vettore di dimensione  $n$  che al termine memorizza il vettore di partizione del grafo.