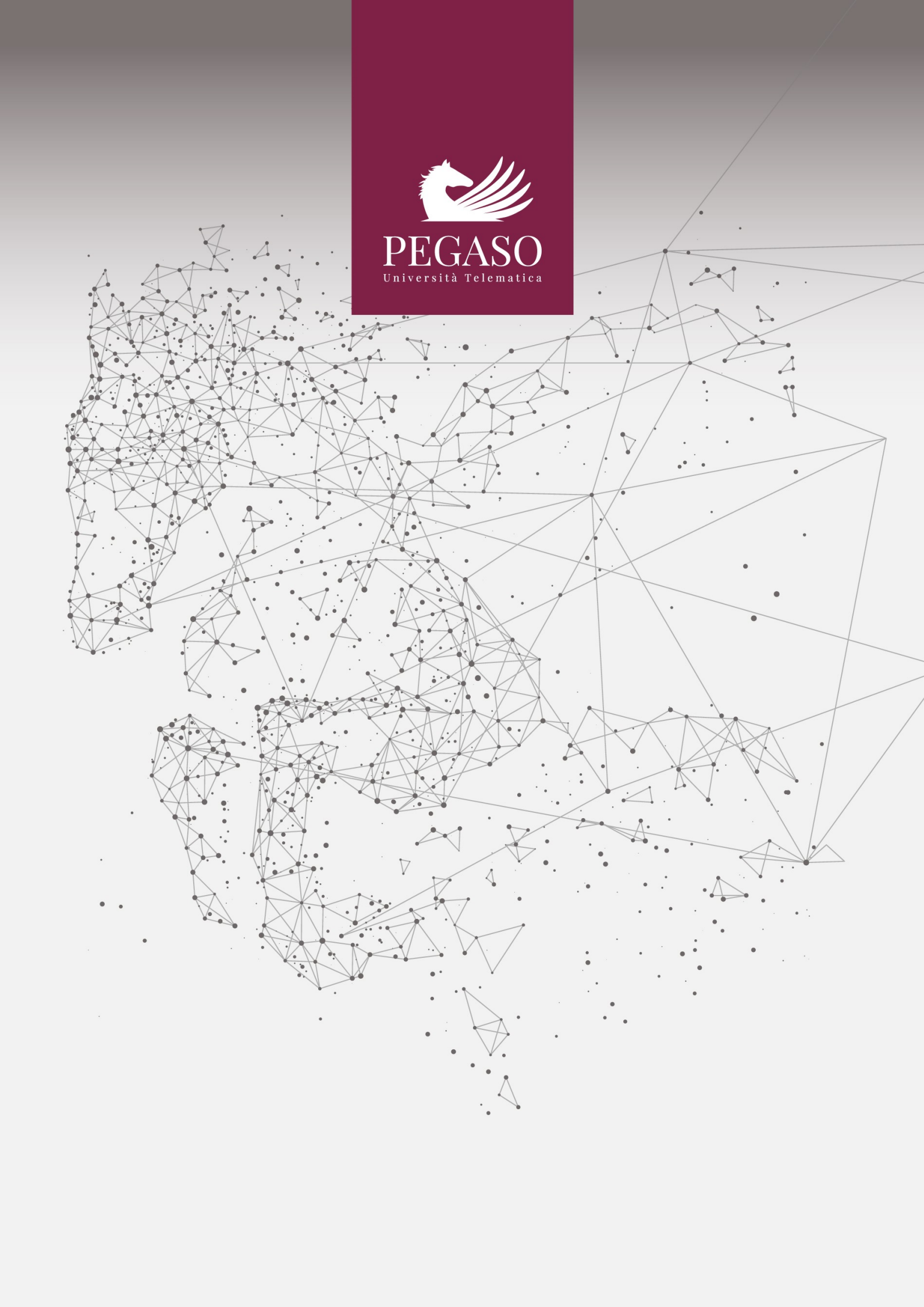
PEGASO
Università Telematica

Indice

1. Attraversamento di un grafo

Dato un grafo $G = (V, E)$ ed un vertice r di V (detto sorgente o radice), il problema dell'attraversamento di un grafo consiste nel visitare ogni vertice raggiungibile nel grafo dal vertice r; ogni nodo deve essere visitato una volta sola. Gli algoritmi di visita di un grafo hanno dunque come obiettivo l'esplorazione di tutti i nodi e gli archi del grafo.

Distinguiamo tra:

- **Visita in ampiezza** (**breadth-first search**): visita i nodi "espandendo" la frontiera fra nodi scoperti / da scoprire (ad es. i cammini più brevi da singola sorgente); mi muovo dunque il più possible in ampiezza (di fratello in fratello) e ritorno sui miei passi solo quando non posso più spingermi oltre;
- **Visita in profondità** (**depth-first search**): visita i nodi andando il "più lontano possibile" nel grafo (ad es. componenti fortemente connesse, ordinamento topologico); mi muovo dunque il più possible in profondità (di padre in figlio) e ritorno sui miei passi solo quando non posso più spingermi oltre.

Da un punto di vista "grafico", ecco come si potrebbero rappresentare i 2 modelli di visita:

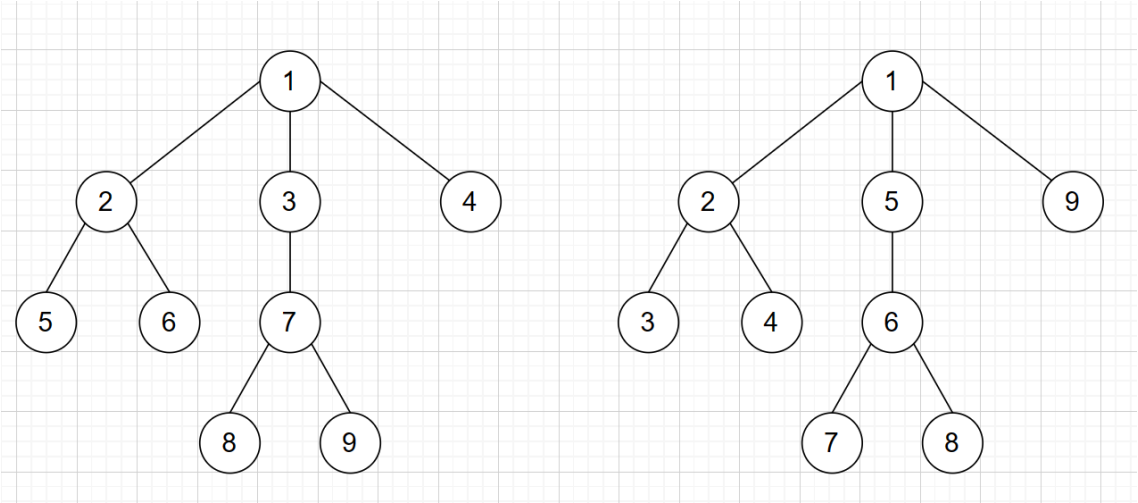

Figura 1: Visita in Ampiezza ed in Profondità

I numeri sui nodi rappresentano l'ordine di "visita" e come si evince dalla figura, nella visita in ampiezza (la prima) si esplorano prima i fratelli e poi si scende verso i figli, mentre nella visita in profondità (la seconda) si esplorano prima i figli e poi i fratelli.

Un approccio naive (ingenuo) alla visita di un grafo potrebbe essere il seguente:

```
{visita il nodo u}
Per ogni nodo v adiacente ad u
   {visita l'arco (u,v)}
```

Questo approccio non tiene conto della struttura del grafo ed itera su tutti i nodi e gli archi senza alcun criterio.

È possibile, tuttavia, adottare strategie simili a quelle adottate per la visita negli alberi, ad es. si potrebbe usare una visita in ampiezza utilizzando una coda e trattare i nodi adiacenti come se fossero figli; tuttavia, una soluzione ingenua (anche questa, dunque, "naive") porterebbe ad un problema:

```
visita(grafo G, nodo r)
queue S = new Queue()
S.enqueue(r)
while not S.isEmpty()
     node u = S.dequeue()
     { visita il nodo u }
     foreach v in G.adj(u)
          S.enqueue(v)
```

Adottando infatti questa strategia si avrebbe il seguente problema nella visita del seguente grafo:

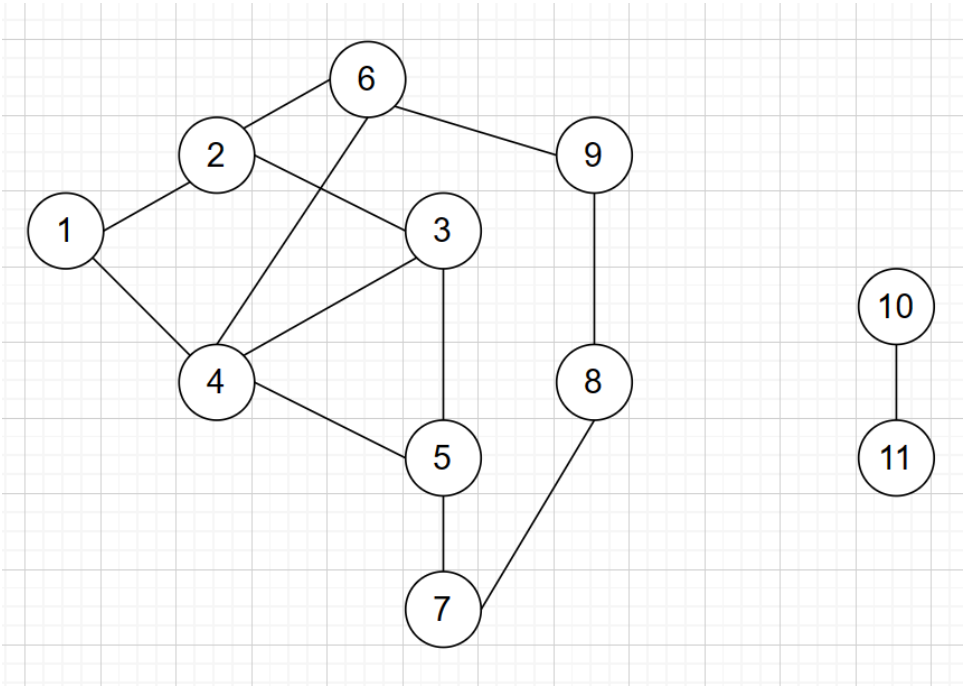

Figura 2: Visita in ampiezza

Supponiamo di partire dal nodo 6 (che diventa la nostra radice); eseguiamo dunque le prime istruzioni:

```
queue S = new Queue()
S.enqueue(r)
```

La nostra coda sarà così costituita:

coda: {6}

Proseguiamo nel nostro programma:

```
while not S.isEmpty()
      node u = S.dequeue()
      { visita il nodo u }
```

Prelevo il nodo dalla coda: $u = \{6\}$ ed inizio ad analizzarlo (eseguo cioè le operazioni che sono previste per un singolo nodo); proseguo poi nel programma:

```
      foreach v in G.adj(u)
            S.enqueue(v)
```

Inserisco dunque nella coda tutti i nodi adiacenti al nodo $u = \{6\}$:

coda: {2, 4, 9}

Prelevo il nodo dalla coda: $u = \{2\}$ ed inizio ad analizzarlo (eseguo cioè le operazioni che sono previste per un singolo nodo); proseguo poi nel programma ed inserisco dunque i nuovi nodi adiacenti nella coda:

coda: {4, 9, 3, 1, 6}

Ma attenzione: il nodo 6 è già stato "visitato". E se continuassi, prelevando dalla coda il nodo: $u = \{4\}$ avrei come scenario:

coda: {9, 3, 1, 6, 5, 6, 1, 3}

E quindi nuovamente avrei dei nodi già "visitati" nella coda.

Come gestire i nodi già visitati? Occorre "marcare" un nodo visitato in modo che non possa essere visitato di nuovo e questo può essere fatto usando un bit di marcatura.

```
visita(grafo G, nodo r)
queue S = new Queue()
S.enqueue(r)
mark(r)
while not S.isEmpty()
      node u = S.dequeue()
      { visita il nodo u }
```

```
    foreach v in G.adj(u)
        if !isMarked(v)
            mark(v)
            S.enqueue(v)
```

Le funzioni $mark(x)$ ed $isMarked(x)$ sono utilizzate rispettivamente per marcare un nodo e verificare se un nodo è stato marcato.

2. Visita in ampiezza

Abbiamo parlato di **visita in ampiezza** (**breadth-first search**), cioè una procedura che visita i nodi "espandendo" la frontiera fra nodi scoperti / da scoprire (ad es. i cammini più brevi da singola sorgente).

Quando si parla dunque di visita in ampiezza, si intende dunque la visita di nodi a distanze crescenti dalla sorgente: visitare i nodi a distanza k prima di visitare i nodi a distanza $k+1$.

L'algoritmo di una visita in ampiezza è di fatto del tutto analogo a quanto visto in precedenza:

```
BFS(grafo G, nodo r)
queue Q = new Queue()
Q.enqueue(r)
boolean[] visited = new boolean[G.V()]
foreach u in G.V()-{r}
     visited[u] = false;
visited[r] = true;
while not Q.isEmpty()
     node u = Q.dequeue()
     { visita il nodo u }
     foreach v in G.adj(u)
          if !visited[v]
               visited[v]=true
               Q.enqueue(v)
```

Di seguito l'implementazione in Python:

```
def BFS(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(vertex, end=' ')

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

3. Il numero di Erdos

Il numero di Erdős è un valore numerico assegnato ad un matematico in base alla sua **distanza collaborativa** dal matematico ungherese Paul Erdős.

Paul Erdős era noto per la sua collaborazione con un gran numero di altri matematici e il suo numero di Erdős è 0. Gli altri matematici che hanno collaborato direttamente con Erdős hanno un numero di Erdős 1. I matematici che hanno collaborato con qualcuno con un numero di Erdős di 1, ma non direttamente con Erdős, hanno un numero di Erdős 2 e così via.

Il numero di Erdős è stato introdotto per la prima volta negli anni '50 ed è diventato un modo comune per misurare la collaborazione tra i matematici. È stato anche esteso ad altre discipline accademiche, come la fisica e l'informatica teorica.

Per calcolare il proprio numero di Erdős, è necessario individuare un percorso di collaborazione attraverso pubblicazioni e altri lavori accademici fino ad Erdős.

Ci sono anche database online come il Mathematical Genealogy Project che possono aiutare a determinare il proprio numero di Erdős.

Il **coefficiente di clustering di Erdős** (anche chiamato "coefficiente di clustering" o "coefficiente di raggruppamento") è una misura di quanto i nodi di un grafo sono collegati tra di loro. In particolare, il coefficiente di clustering di Erdős misura il grado di coesione delle interconnessioni tra i nodi di un grafo.

In un grafo non diretto, il coefficiente di clustering di Erdős di un nodo è definito come la frazione dei suoi vicini (i nodi direttamente collegati al nodo in questione) che sono a loro volta connessi tra di loro. Ad esempio

- un nodo A è connesso ai nodi B, C e D
- esiste una connessione diretta tra i nodi B e C
- non esiste una connessione diretta tra B e D
- non esiste una connessione diretta tra C e D

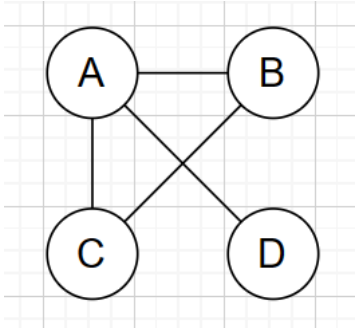

Figura 3: Coefficiente di Erdos di A=1/3

Il coefficiente di clustering di Erdős di A sarebbe 1/3.

In altre parole: il coefficiente di clustering di Erdős di A indica la frazione di coppie di vicini di A che sono a loro volta vicine tra di loro (vicine è da intendere come "collegate direttamente"); tale coefficiente è dato dunque dal rapporto tra il numero di connessioni tra le coppie di vicini di A e il numero massimo possibile di connessioni tra queste coppie.

Nel nostro esempio:

- i vicini di A sono B, C e D (i nodi direttamente connessi)
- i vicini di A direttamente connessi tra loro sono solamente BC (esiste una connessione diretta tra loro) → 1
- il numero massimo possibile di connessioni è dato da: BC, BD e CD → 3

Per questo motivo il coefficiente è pari a 1/3.

Se consideriamo il seguente scenario invece:

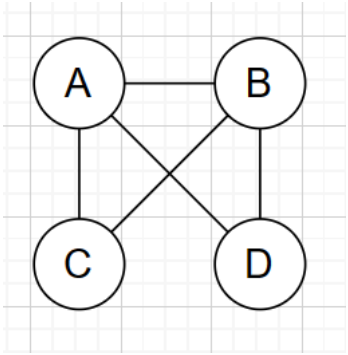

Figura 4: Coefficiente di Erdos di A=2/3

In questo esempio:

- i vicini di A sono B, C e D (i nodi direttamente connessi)
- i vicini di A direttamente connessi tra loro sono BC e BD → 2
- il numero massimo possibile di connessioni è dato da: BC, BD e CD → 3

Per questo motivo il coefficiente è pari a 2/3.

Il coefficiente di clustering di Erdős è una misura utile per caratterizzare la struttura dei grafi e capire come sono organizzati i loro nodi. Ad esempio, i grafi con un alto coefficiente di clustering di Erdős indicano che i nodi sono fortemente interconnessi tra di loro, mentre i grafi con un basso coefficiente di clustering di Erdős indicano che i nodi sono più sparsi e isolati.

L'algoritmo per il calcolo del numero di Erdős per un nodo in un grafo può essere indicato come segue:

```
Erdos(grafo G, nodo r)
queue Q = new Queue()
Q.enqueue(r)
int[] distance = new int[G.V()]
foreach u in G.V()-{r}
     distance[u]=∞;
distance[r]=0;
while not Q.isEmpty()
     node u = Q.dequeue()
     foreach v in G.adj(u)
          if distance[v]== ∞
               distance[v]=distance[u]+1
               Q.enqueue(v)
```

Come si può vedere tale algoritmo è del tutto simile a quello per la BFS.

Di fatto, l'algoritmo per il calcolo della distanza di Erdős utilizza una visita in ampiezza del grafo, ovvero esplora i nodi del grafo a partire da un nodo sorgente (in questo caso, il nodo corrispondente a Paul Erdős), visitando prima i nodi a distanza 1 dal nodo sorgente, poi quelli a distanza 2, e così via, fino a quando non ha visitato tutti i nodi del grafo.

La visita in ampiezza viene implementata utilizzando una coda (Queue) per mantenere l'ordine di visita dei nodi. Inoltre, l'algoritmo aggiorna la distanza di ogni nodo durante la visita, utilizzando la regola che la distanza di un nodo è pari alla distanza del nodo padre + 1.

Quindi, il calcolo della distanza di Erdős coincide con una visita in ampiezza del grafo.

Esempio di calcolo del numero di Erdos

Consideriamo il seguente grafo:

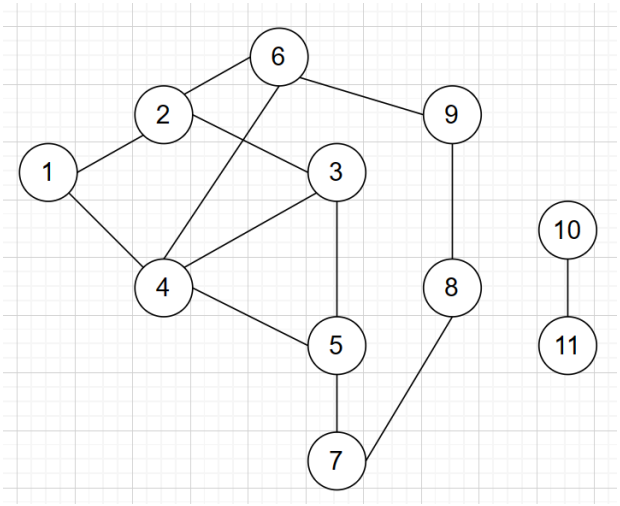

Figura 5: Grafo

Considerando {6} come radice, lo inseriamo nella coda ed inizialmente poniamo tutte le distanze ad ∞ tranne per il nodo radice stesso, la cui distanza è pari a 0.

Eliminiamo il {6} dalla coda ed analizziamo i nodi adiacenti {2, 4, 9}; questi sono tutti a distanza infinita, quindi, devono settare la loro distanza come pari alla distanza del nodo appena rimosso dalla coda (il nodo {6} ha distanza 0) +1 e pertanto la loro distanza sarà pari ad 1; successivamente devono essere inseriti in coda:

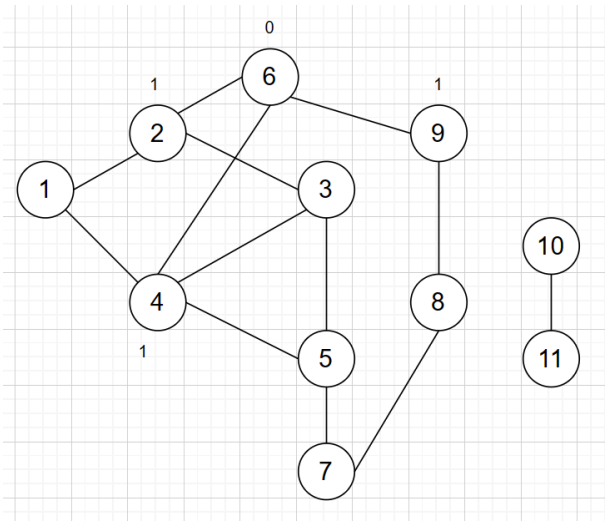

Figura 6: Distanza 1

Nella coda abbiamo:

coda: {2, 4, 9}

Proseguiamo estraendo 2 dalla coda, incrementando la distanza dei vicini di 2 (ponendola pari a 2) ed inserendo in coda i nuovi vicini:

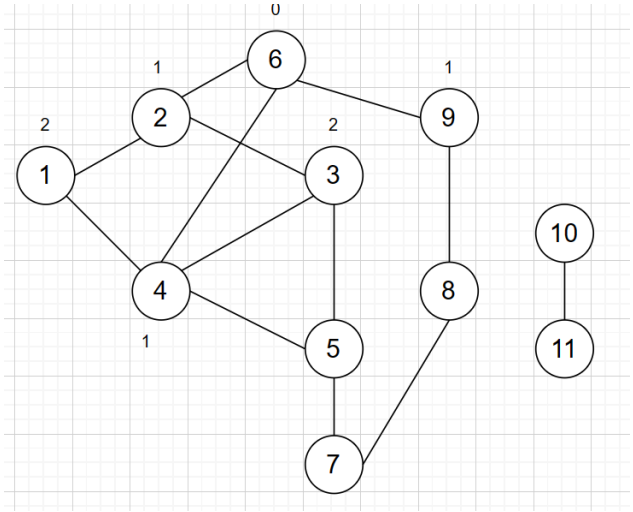

Figura 7: Distanza 2

Nella coda abbiamo:

coda: {4, 9, 3, 1}

Proseguiamo estraendo 4 dalla coda, incrementando la distanza dei vicini di 4 (ponendola pari a 2) ed inserendo in coda i nuovi vicini:

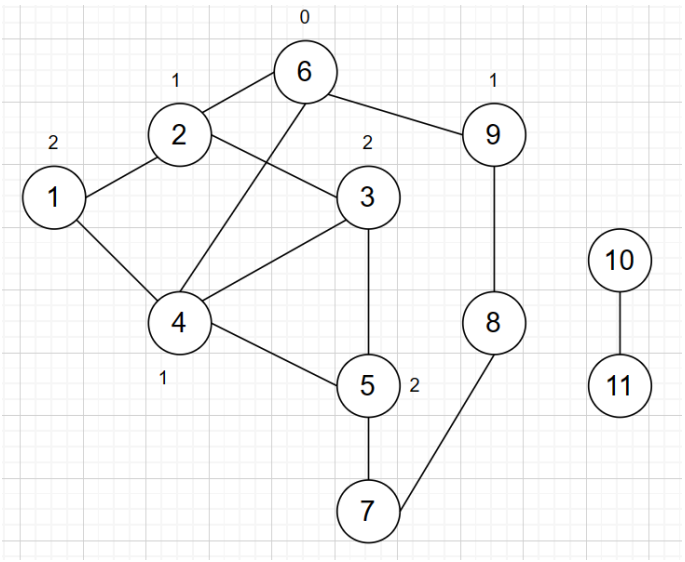

Figura 8: Distanza 2

Nella coda abbiamo:

coda: {9, 3, 1, 5}

Proseguiamo estraendo 9 dalla coda, incrementando la distanza dei vicini di 9 (ponendola pari a 2) ed inserendo in coda i nuovi vicini:

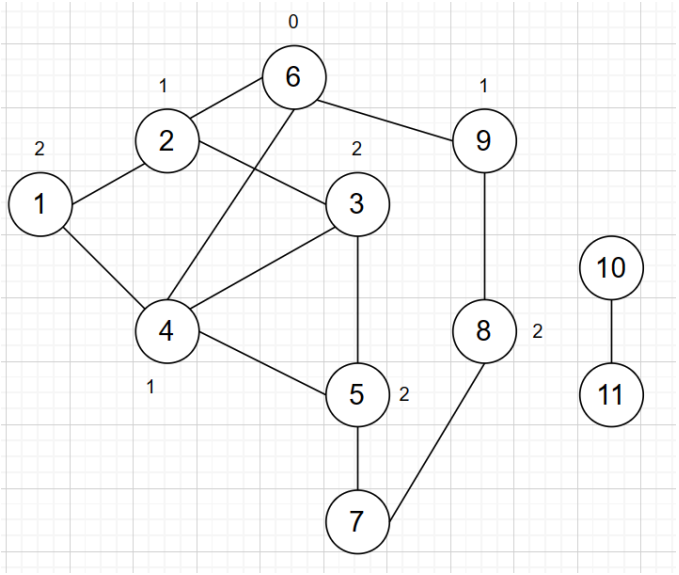

Figura 9: Distanza 2

Nella coda abbiamo:

coda: {3, 1, 5, 8}

Proseguiamo estraendo 3 dalla coda, ma 3 non ha vicini con distanza infinita quindi proseguiamo con 1 che però si trova nella stessa situazione ed arriviamo dunque a 5:

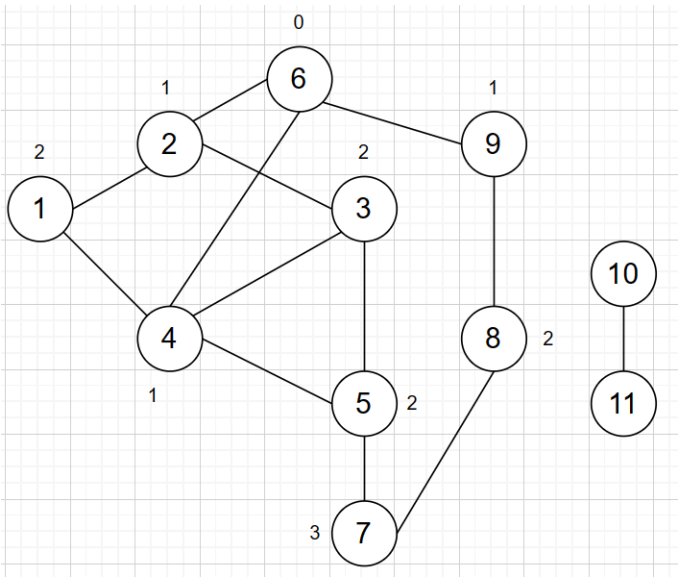

Figura 10: Distanza 3

Nella coda abbiamo:

coda: {8, 7}

Si può proseguire estraendo 8 e 7 dalla coda ed abbiamo completato il processo.

4. Albero dei cammini (BFS-TREE)

L'**albero dei cammini** (breadth-first search tree / **BFS tree**) è una struttura dati utilizzata nell'algoritmo di ricerca in ampiezza (**BFS**) per rappresentare i cammini scoperti durante la ricerca.

L'algoritmo di BFS esplora gradualmente il grafo a partire dal nodo di partenza, visitando tutti i nodi che possono essere raggiunti in un numero limitato di passi. Durante questa esplorazione, l'algoritmo mantiene traccia dei nodi visitati e dei cammini scoperti, costruendo un albero radicato nel nodo di partenza.

L'albero dei cammini BFS è costituito dai nodi visitati durante l'esecuzione dell'algoritmo di BFS e dai loro archi, che collegano ogni nodo al suo predecessore nel cammino che lo ha scoperto per primo. L'albero dei cammini BFS può essere utilizzato per rappresentare la struttura del grafo esplorato e per determinare i cammini più brevi tra il nodo di partenza e gli altri nodi del grafo.

L'albero dei cammini generato dall'algoritmo di BFS è un tipo particolare di albero di copertura, chiamato **albero di copertura BFS** (**BFS spanning tree**).

Un albero di copertura è un sotto-grafo che collega tutti i nodi di un grafo non orientato, senza formare cicli. In altre parole: è un albero che copre tutto il grafo. Formalmente, in un grafo non orientato $G = (V, E)$, un albero di copertura T è un albero libero $T = (V, E')$ composto da tutti i nodi di V e da un sottoinsieme degli archi $(E' \subseteq E)$, tale per cui tutte le coppie di nodi del grafo sono connesse da una sola catena nell'albero.

L'algoritmo di BFS genera un albero di copertura perché, durante la sua esecuzione, esplora tutti i nodi raggiungibili dal nodo di partenza in un numero limitato di passi, generando un albero che connette il nodo di partenza a tutti gli altri nodi visitati, senza formare cicli.

L'albero dei cammini BFS è quindi un tipo particolare di albero di copertura, che è radicato nel nodo di partenza e in cui gli archi rappresentano i cammini più brevi tra il nodo di partenza e gli altri nodi del grafo.

Riportiamo un esempio di albero di copertura:

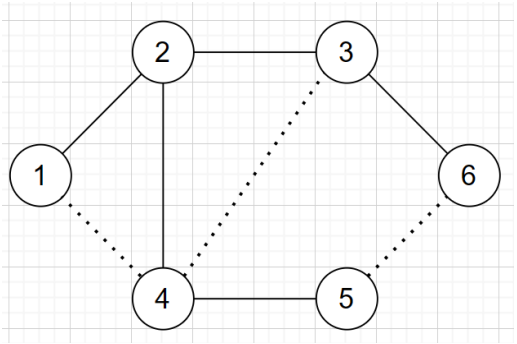

Figura 11: Albero di copertura

Riprendendo l'algoritmo di Erdos:

```
Erdos(grafo G, nodo r)
queue Q = new Queue()
Q.enqueue(r)
int[] distance = new int[G.V()]
foreach u in G.V()-{r}
     distance[u]=∞;
distance[r] = 0;
while not Q.isEmpty()
     node u = Q.dequeue()
     foreach v in G.adj(u)
          if distance[v]== ∞
               distance[v]=distance[u]+1
               Q.enqueue(v)
```

Possiamo utilizzare una struttura dati array dei padri (parent) per memorizzare l'albero di copertura:

```
BFS_TREE(graph G, node r)
queue Q = new Queue()
Q.enqueue(r)
int[] distance = new int[G.V()]
node[] parent = new node[G.V()]
foreach u in G.V()-{r}
     distance[u] = ∞
     parent[u] = nil
distance[r] = 0
parent[r] = nil
while not Q.isEmpty()
     node u = Q.dequeue()
     foreach v in G.adj(u)
          if distance[v] == ∞
               distance[v] = distance[u] + 1
               parent[v] = u
               Q.enqueue(v)
```

Alcune considerazioni relative all'albero T individuato:

- L'albero T contiene i vertici visitati
- $S \subseteq T$ contiene i vertici aperti: vertici i cui archi uscenti non sono ancora stati percorsi
- $T - S \subseteq T$ contiene i vertici chiusi: vertici i cui archi uscenti sono stati tutti percorsi
- $V - T$ contiene i vertici non visitati
- Se u si trova lungo il cammino che va da r al nodo v, diciamo che:
    - u è un antenato di v
    - v è un discendente di u
- I nodi vengono visitati al più una volta (marcatura)
- Tutti i nodi raggiungibili da r vengono visitati
- T contiene esattamente tutti i nodi raggiungibili da r

In termini di complessità, questa è pari a $O(m + n)$

- ognuno degli n nodi viene inserito nella coda al massimo una volta
- ogni volta che un nodo viene estratto, tutti i suoi archi vengono analizzati una volta sola

Il numero di archi analizzati è quindi:

$$m = \sum_{u \in V} d_{out}(u)$$

dove $d_{out}(u)$ è l'out-degree del nodo u

La procedura per stampare il cammino:

```
printCammino(graph G, node r, node s, node[ ] p)
     if r == s then
          print s
     else if p[s] == nil then
          print "nessun cammino da r a s"
     else
          printCammino(G, r, p[s], p)
          print s
```

In particolare, la visita BFS può essere utilizzata per individuare il cammino più breve tra 2 nodi (in termini di numero di archi).

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.