Relazione di

Laboratorio di Algoritmi e Strutture Dati

Edoardo Grassi

Settembre 2023

Indice

Premessa

Questa relazione fa parte di un progetto per il corso di Laboratorio di Algoritmi e Strutture ed è composto da due esercizi:

1. Confronto tra diverse strutture dati per insiemi disgiunti nell'ambito della ricerca di componenti connesse in grafi non diretti
2. Confronto tra *insertion sort* e *merge sort*

In questa relazione viene analizzato e discusso solo il primo esercizio, la cui implementazione è stata scritta in codice *Python*.

1 Introduzione al problema

Le strutture dati per insiemi disgiunti vengono utilizzate in algoritmi dove sia necessario raggruppare diversi elementi sotto un unico rappresentante; definito $\mathcal{S}$ l'insieme contenente tutti gli insiemi disgiunti, possiamo definire tre operazioni fondamentali su di esso:

- MAKE-SET(x): aggiunge a $\mathcal{S}$ un nuovo insieme $\mathcal{S}_i = \{x\}$ (x non deve esistere all'interno di alcun altro insieme, altrimenti non sarebbero disgiunti)
- FIND-SET(x): individua l'elemento rappresentante dell'insieme nel quale x è contenuto
- UNION(x, y): dati $x \in \mathcal{S}_x$ e $y \in \mathcal{S}_y$ ($x \neq y$ per quanto detto prima) $\mathcal{S} = (\mathcal{S} - \mathcal{S}_x - \mathcal{S}_y) \cup \mathcal{S}_\cup$ dove $\mathcal{S}_\cup = \mathcal{S}_x \cup \mathcal{S}_y$ il cui rappresentante è dato da uno degli elementi in $\mathcal{S}_x$ o $\mathcal{S}_y$

1.1 Ricerca di componenti connesse in grafi

Un possibile utilizzo di queste strutture dati è nella ricerca di componenti connesse in un **grafo non diretto** (o non orientato), ovvero quando le relative connessioni non hanno direzione e pertanto collegano due vertici bidirezionalmente, e questo verrà utilizzato per confrontare diverse implementazioni della struttura dati.

Dato $G(V, E)$ un grafo non diretto, con V insieme dei vertici del grafo e E l'insieme delle connessioni, l'algoritmo è il seguente:

Algoritmo 1 Ricerca di componenti connesse

```
1. for v ∈ G.V do
2.     MAKE-SET(v)
3. for (u, v) ∈ G.E do
4.     if FIND-SET(u) ≠ FIND-SET(v) then
5.         UNION(u, v)
```

1.1.1 Copertura dei grafi

Nei successivi capitoli useremo il termine *copertura* per indicare una percentuale $c \in [0; 1]$ di connessioni rispetto ad un numero massimo, definito in base al tipo copertura:

- *Copertura lineare*: è la percentuale di connessioni rispetto al numero di vertici del grafo e ci permetterà di stabilire che il numero di connessioni considerato è un $\Theta(v)$ con $v = |V|$;
$$e = |E| = c \cdot v = \Theta(v) \tag{1}$$
- *Copertura quadratica*: è la percentuale di connessioni rispetto al numero massimo di connessioni possibili; essendo il grafo $G(V, E)$ non diretto si ha che
$$\max(|E|) = \frac{v(v-1)}{2} = \Theta(v^2) \text{ con } v = |V| \tag{2}$$
quindi il numero di connessioni data una copertura quadratica c è dato da:
$$e = |E| = c \cdot \frac{v(v-1)}{2} = \Theta(v^2) \tag{3}$$
e per indicare il numero di connessioni nei costi computazionali utilizzeremo $\Theta(v^2)$.

1.2 Implementazioni della struttura dati

Come riportato precedentemente, esistono varie implementazioni delle strutture dati per insiemi disgiunti che permettono di avere variazioni sui costi di complessità computazionale.

1.2.1 Liste concatenate

Questa implementazione consiste nel rappresentare gli insiemi disgiunti tramite liste concatenate. Ogni lista ha due puntatori, uno all'elemento iniziale della lista e uno all'elemento finale, permettendo di ottenere in tempo costante il rappresentante di un insieme, e di aggiungere elementi ad esso sempre in tempo costante. Ogni elemento della lista contiene, oltre ai dati associati, anche un puntatore al suo rappresentante, così da ottenere in tempo costante il rappresentante di qualsiasi elemento, e un puntatore al successivo elemento della lista, come in qualsiasi lista concatenata.

I costi computazionali delle operazioni base nei casi peggiori sono i seguenti:

- MAKE-SET(x) $\rightarrow O(1)$: dato che il costo di creazione di una nuova lista vuota è costante, e per quanto detto nel paragrafo precedente anche l'inserimento ha costo costante;
- FIND-SET(x) $\rightarrow O(1)$: per via del puntatore al rappresentante già presente all'interno di qualsiasi elemento;
- UNION(x, y) $\rightarrow O(n)$ con n numero di elementi totali nella struttura dati: infatti, supponendo di concatenare tutti gli elementi della lista contenente y alla lista contenente x, abbiamo il caso peggiore quando la lista di x contiene solo x stesso, e la lista di y contiene i rimanenti $n-1$ elementi; dato che l'unione prevede di cambiare il puntatore al rappresentante per ogni elemento inserito nella lista finale, questa operazione ha tempo lineare, giustificando il risultato.

Il costo computazionale nel caso peggiore tiene conto del fatto che per n elementi inseriti all'interno della struttura dati potranno essere eseguite massimo $n-1$ UNION il cui numero di operazioni interne tenderà man mano verso $n-1$, quindi si considera il costo pari a:

$$\Theta(n^2)^1 \tag{4}$$

Considerando il problema della ricerca di componenti connesse su un grafo non diretto $G(V, E)$, questo è composto da un ciclo *for* alla riga 1 che esegue $|V|$ operazioni di MAKE-SET, quindi crea $|V|$ elementi. Considerando (4), il costo totale è dato da:

$$\Theta(v^2) \tag{5}$$

con $v = |V|$.

1.2.2 Liste concatenate con euristica dell'unione pesata

Questa implementazione riprende quella delle normali liste concatenate e cerca di ridurre il costo dell'operazione di UNION conservando nelle liste un parametro che indica il numero di elementi contenuti in esse e questo viene usato per unire la lista più corta a quella più lunga, riducendo il numero di operazioni. Inoltre, la gestione di un contatore di elementi all'interno di ogni lista ha costo costante, quindi i costi per le rimanenti operazioni continuano ad essere i medesimi per le liste concatenate.

In generale, data una sequenza di m operazioni di MAKE-SET, FIND-SET and UNION, n delle quali sono operazioni di MAKE-SET (i.e., n è il numero di elementi totali all'interno della struttura dati), il costo computazionale è dato da:

$$O(m + n \log_2 n)^2 \tag{6}$$

Analizzando il costo della struttura dati nell'algoritmo di ricerca di componenti connessi su $G(V, E)$ grafo non diretto, si osserva che: n, il numero di MAKE-SET è dato dal numero di iterazioni del ciclo *for* alla riga 1; m è invece dato dalla somma di n con il numero di FIND-SET e UNION del ciclo *for* alla riga 3, moltiplicate per il numero di iterazioni del ciclo stesso:

[1] Thomas H. Cormen et al. "Introduction to Algorithms". In: Fourth. Cambridge, Massachusetts: The MIT Press, 2022, p. 689

[2] Thomas H. Cormen et al. "Introduction to Algorithms". In: Fourth. Cambridge, Massachusetts: The MIT Press, 2022, pp. 689–690

$$\begin{aligned} n &= |V| = \Theta(|V|) = \Theta(v) \\ m &= n + |E| \cdot (2+1) = |V| + 3|E| = \Theta(|V| + |E|) = \Theta(v+e) \end{aligned} \tag{7}$$

con $v = |V|$ e $e = |E|$. Quindi per (6) il costo complessivo è dato da:

$$\begin{aligned} O(m + n\log_2 n) &= O((v+3e) + v\log_2 v) \\ &= O(v + 3e + v\log_2 v) \\ &= O(3e + v\log_2 v) \\ &= O(e + v\log_2 v) \end{aligned} \tag{8}$$

Considerando il costo rispetto al solo numero dei vertici usando la copertura lineare, per (1) si ha:

$$O(e + v\log_2 v) = O(v + v\log_2 v) = O(v\log_2 v) \tag{9}$$

mentre usando la copertura quadratica, per (3) si ha:

$$O(e + v\log_2 v) = O(v^2 + v\log_2 v) = O(v^2) \tag{10}$$

1.2.3 Foreste di insiemi disgiunti con compressione dei cammini

Questa implementazione prevede di rappresentare ogni elemento di ogni insieme disgiunto come nodo di un albero, quindi è composto dal dato che contiene e da un puntatore al nodo padre. Ogni insieme disgiunto è quindi rappresentato da un albero la cui radice diventa automaticamente il rappresentante dell'insieme. Nella versione base, ovvero senza compressione dei cammini, i costi delle operazioni base sono i seguenti:

- MAKE-SET$(x) \to O(1)$: prevede di creare un nuovo nodo senza padre;
- FIND-SET$(x) \to O(h_x)$ con h_x altezza dell'albero in cui è contenuto x: questo perchè per individuare il rappresentante del nodo è necessario risalire tutto l'albero fino a giungere alla radice;
- UNION$(x,\ y) \to O(h_y)$ supponendo di porre x come padre del rappresentante di y: infatti, per unire due alberi (o i due insiemi) basta porre come padre di uno dei due rappresentanti l'altro elemento, e automaticamente il rappresentante degli elementi dell'albero unito diviene il rappresentante dell'albero espanso.

La compressione del cammino cerca di attenuare i tempi dell'operazione di FIND-SET (che viene utilizzata anche in UNION per individuare il rappresentante sul quale cambiare padre), andando a ridurre i cammini dai singoli nodi verso la radice dell'albero. Ciò viene fatto nell'operazione di FIND-SET che diventa una funzione ricorsiva: man mano che si risale l'albero i nodi attraversati vengono salvati e una volta raggiunta la radice i padri dei nodi salvati vengono aggiornati con la radice individuata.

Con la compressione dei cammini, la struttura dati ha un costo computazionale pari a:

$$\Theta(n + f \cdot (1 + \log_{2+\frac{f}{n}} n)^3 \tag{11}$$

con f il numero di operazioni FIND-SET e n il numero di operazioni UNION.
Nel caso della ricerca di componenti connesse n è dato dal numero delle iterazioni del ciclo *for* alla riga 1, mentre f dipende dal numero di iterazioni del ciclo *for* alla riga 3 abbiamo:

$$\begin{aligned} n &= |V| = \Theta(|V|) = \Theta(v) \\ f &= 2|E| = \Theta(|E|) = \Theta(e) \end{aligned} \tag{12}$$

Quindi, per (11), il costo totale è dato da:

$$\begin{aligned} \Theta(n + f \cdot (1 + \log_{2+\frac{f}{n}} n)) &= \Theta(v + e(1 + \log_{2+\frac{e}{v}} v)) \\ &= \Theta(v + e + 2e\log_{2+\frac{e}{v}} v) \\ &= O(v + e\log_{2+\frac{e}{v}} v) \end{aligned} \tag{13}$$

[3] Thomas H. Cormen et al. "Introduction to Algorithms". In: Fourth. Cambridge, Massachusetts: The MIT Press, 2022, p. 696

Considerando il costo rispetto al solo numero dei vertici usando la copertura lineare, per (1) si ha:

$$\begin{aligned} O(v + e \log_{2+\frac{e}{v}} v) &= O(v + v \log_{2+\frac{v}{v}} v) \\ &= O(v + v \log_3 v) \\ &= O(v \log_3 v) \end{aligned} \tag{14}$$

mentre usando la copertura quadratica, per (3) si ha:

$$\begin{aligned} O(v + e \log_{2+\frac{e}{v}} v) &= O(v + v^2 \log_{2+\frac{v^2}{v}} v) \\ &= O(v + v^2 \log_{2+v} v) \\ &= O(v^2 \log_{2+v} v) \end{aligned} \tag{15}$$

2 Documentazione del codice

Il codice che contiene le varie implementazioni della struttura dati è contenuta nel modulo `disjoint_sets`. Questo contiene tre sottomoduli: `disjoint_sets.base`, `disjoint_sets.list` e `disjoint_sets.tree`. Il codice utilizzato per rappresentare i grafi e l'operazione di ricerca di componenti connesse è contenuta nel modulo `graphs`, mentre il codice usato per il confronto è nel modulo principale (`__main__`).

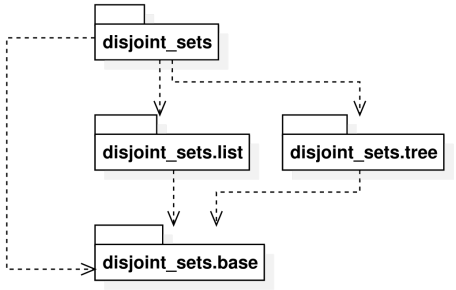

Figura 1: Diagramma dei moduli utilizzati

2.1 Il modulo `disjoint_sets.base`

Questo modulo contiene le interfacce base per la gestione di strutture dati per insiemi disgiunti:

1. `Node`: classe per indicare un generico elemento all'interno di un insieme disgiunto; contiene un solo attributo `data` che viene utilizzato per salvare i dati relativi all'elemento.
2. `DisjointSetsInterface`: interfaccia per le strutture dati che gestiscono insiemi disgiunti, contiene i metodi:
   (a) `find_set(Node)` → `Node`: dato un oggetto `Node` restituisce il suo rappresentante di tipo `Node`;
   (b) `make_set(Any)` → `Node`: data una istanza di un qualsiasi tipo ritorna un oggetto `Node` con `data` impostato al valore passato, creando un nuovo insieme disgiunto; **Nota**: nelle implementazioni contenute nel modulo `disjoint_sets` si presuppone che gli elementi inseriti siano tutti diversi tra loro, condizione necessaria per avere insiemi disgiunti, quindi non è presente alcun controllo su tale requisito;
   (c) `union(Node, Node)` → `Node`: dati 2 oggetti `Node` esegue l'unione tra i relativi insiemi disgiunti, restituendo il `Node` nel quale insieme sono stati aggiunti gli elementi dell'insieme dell'altro `Node` (i.e. restituisce l'oggetto `Node` il cui insieme non è stato distrutto).

2.2 Il modulo `disjoint_sets.list`

Questo modulo contiene le implementazioni delle interfacce tramite liste concatenate con e senza euristica dell'unione pesata. Sono definite al suo interno le seguenti classi:

1. `ListNode`: eredita `Node` aggiungendo gli attributi:
   (a) `next` → `Node`: per indicare il successivo elemento (in una lista concatenata)
   (b) `list` → `List`: anzichè usare un puntatore al rappresentante si usa un puntatore alla lista intera (questo non varia i costi computazionali);

2. `List`: è la struttura dati usata per gestire le liste concatenate e quindi gli insiemi; viene inizializzata vuota ed è composta dagli attributi:

   (a) `first` → `ListNode`: rappresenta il primo nodo della lista e quindi il rappresentante;

   (b) `last` → `ListNode`: rappresenta l'ultimo nodo della lista, fondamentale per gli inserimenti in tempo costante;

   (c) `_size` → `int`: attributo privato che rappresenta la dimensione attuale della lista;

   e dai metodi:

   (d) `add(ListNode)`: aggiunge alla lista un nodo, incrementando la dimensione della lista;

   (e) `get_size()` → `int`: restituisce la dimensione della lista (i.e. elementi contenuti nell'insieme disgiunto);

   (f) `merge(List)`: inserisce tutti gli elementi di una certa lista nella lista corrente, aggiornando i puntatori verso essa;

3. `ListDisjointSets`: implementazione dell'interfaccia `DisjointSetsInterface` tramite liste concatenate **senza** euristica dell'unione pesata, tra i metodi implementati troviamo alcune particolarità: come detto nella classe `List` l'operazione di `find_set` sfrutta il puntatore alla lista per poi ottenere il primo elemento (rappresentante); l'operazione di `union` va a richiamare in modo diretto il metodo `merge` della lista associata al primo nodo passato come argomento;

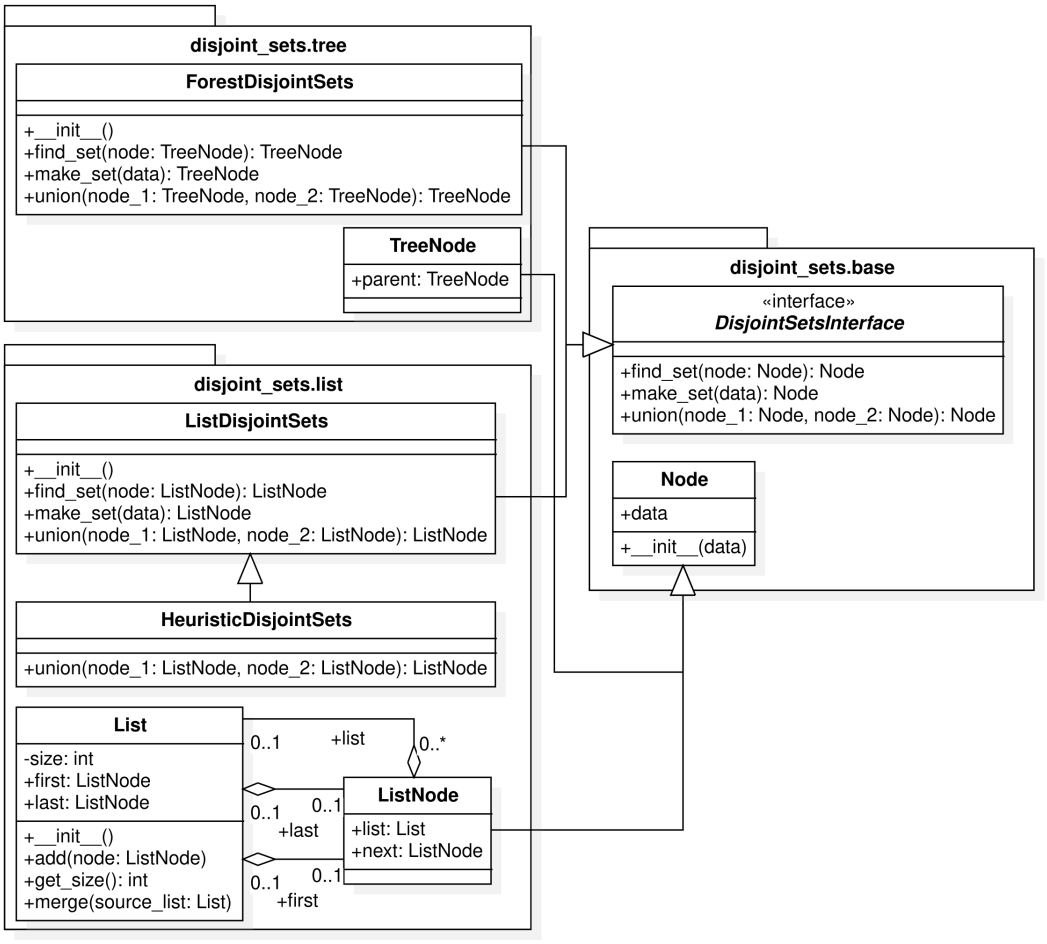

Figura 2: Diagramma UML del modulo `disjoint_sets`

4. `HeuristicDisjointSets`: eredita `ListDisjointSets`, quindi implementa `DisjointSetsInterface` per gestire gli insiemi disgiunti tramite liste concatenate **con** euristica dell'unione pesata: questo implica che il metodo `union` controlli prima le dimensioni delle liste tramite il metodo `get_size` e sulla base dei esse unire la lista più corta a quella più lunga.

2.3 Il modulo `disjoint_sets.tree`

Questo modulo contiene l'implementazione delle interfacce tramite foreste con compressione del cammino. A differenza del modulo precedente dove abbiamo `List` che rappresenta un insieme disgiunto, in questa implementazione non si ha una classe equivalente (e.g. `Tree`), e si riduce agli unici elementi degli alberi. Infatti le classi che lo compongono sono:

1. `TreeNode`: identifica un nodo nell'albero e eredita `Node` aggiungendo l'attributo `parent` di tipo `TreeNode` per individuare il padre del nodo; un nodo senza padre diventa automaticamente rappresentante di quell'albero/insieme;

2. `ForestDisjointSets`: implementazione dell'interfaccia `DisjointSetsInterface` per la gestione di insiemi disgiunti con foreste di nodi, con compressione dei cammini; tra le operazioni implementate la più degna di nota è la `find_set` che viene implementata in forma ricorsiva: il caso base si ha quando il nodo passato come argomento non ha padre e quindi è il rappresentante, il caso induttivo richiama lo stesso metodo con il padre del nodo e nella fase di post-ricorsione si imposta il padre di ogni nodo incontrato con il rappresentante dell'insieme.

2.4 Il modulo `graphs`

Questo modulo definisce le classi per la gestione dei grafi non diretti tramite indici, e fa grande uso delle strutture dati base messe a disposizione dal linguaggio *Python*:

1. `Vertex`: identifica un vertice di un grafo; viene inizializzato con l'indice rispetto alla lista di vertici contenuta in nell'oggetto `Graph` associato, e con il dato associato; dato che `index` è privato, per accedere all'indice del vertice si utilizza il metodo `get_index()` $\rightarrow$ `int`;

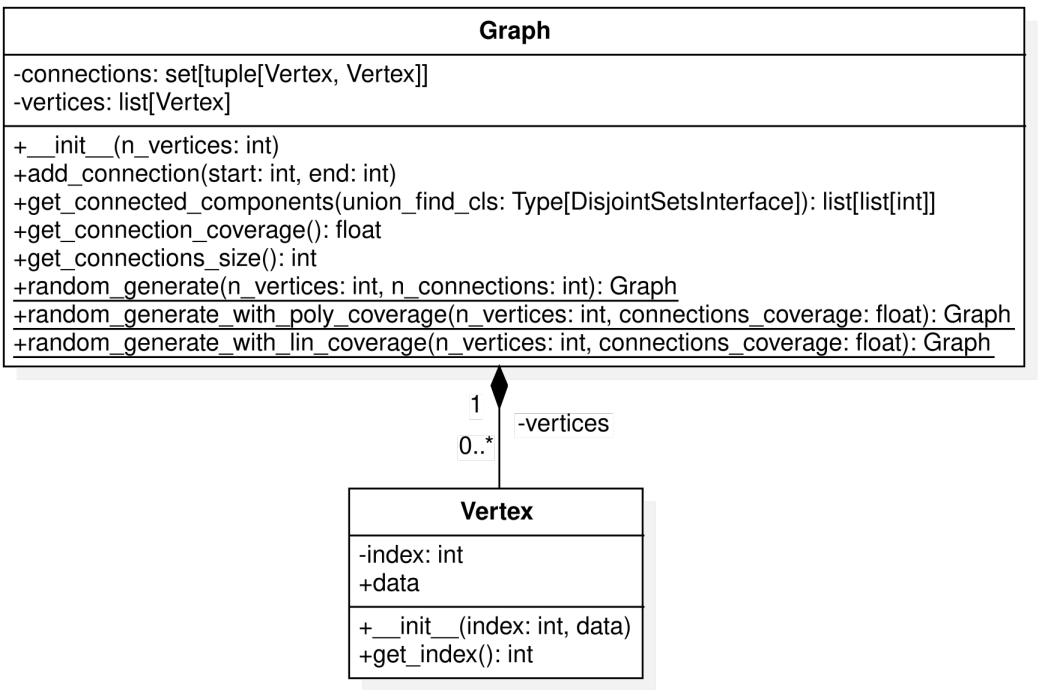

Figura 3: Diagramma UML del modulo `graphs`

2. `Graph`: è la classe che identifica il grafo non diretto ed è composta dagli attributi:

   (a) `_vertices` → `list[Vertex]`: attributo privato che salva tutti i vertici del grafo tramite la struttura dati base `list`; all'istanziazione della classe questa lista viene inizializzata creando un numero di vertici indicato tra i parametri del costruttore;

   (b) `_connections` → `set[tuple[Vertex, Vertex]]`: attributo privato usato per salvare le connessioni tra vertici usando le strutture dati base `tuple` usata per combinare coppie di oggetti in un singolo oggetto e `set` per gestire un insieme di queste coppie, evitando eventuali duplicati con inserimenti in tempi costanti;

   e dai metodi:

   (c) `add_connection(int, int)`: aggiunge una nuova connessione non diretta tra i vertici usando i relativi indici; gli indici passati vengo riordinati in modo che la sorgente abbia sempre indice minore alla destinazione (anche se in grafi non diretti i termini *sorgente* e *destinazione* non hanno differenza) per ridurre il numero di connessioni salvate nel grafo;

   (d) `get_connected_components(Type[DisjointSetsInterface])` → `list[list[int]]`: è il metodo che svolge l'algoritmo 1 utilizzando l'implementazione degli insiemi disgiunti passata come argomento (`Type[DisjointSetsInterface]` significa che accetta come argomento una **classe** che implementa `DisjointSetsInterface`), quindi istanzia la classe passata e la usa nell'algoritmo; oltre a elaborare l'algoritmo il metodo deve ritornarne il risultato della ricerca dato da liste di componenti connesse, ovvero altre liste di indici che rappresentano i vertici, e per fare ciò si fa riferimento al seguente pseudocodice:

Algoritmo 2 Conversione da insiemi disgiunti a liste di liste di vertici

Ensure: $G(V, E)$ grafo non diretto
1. $D \leftarrow$ hash table vuota
2. **for** $v \in G.V$ **do**
3. $\quad rep \leftarrow$ FIND-SET(v)
4. $\quad$ **if** rep **non** è tra le chiavi di D **then**
5. $\quad\quad D[rep] \leftarrow$ lista vuota
6. $\quad$ aggiungi v a $D[rep]$
7. **return** tutti i valori di D

Dobbiamo comunque fare attenzione all'eventuale variazione di costo dell'algoritmo complessivo, infatti abbiamo una variazione del numero di FIND-SET eseguite, quindi dato un grafo non diretto $G(V, E)$, si distinguono le implementazioni:

- Liste concatenate (`ListDisjointSets`): in questo caso la FIND-SET ha un costo costante e viene eseguita $|V| = v$ volte, quindi riprendendo (4) e considerando $n = v$:

$$\Theta(n^2 + v) = \Theta(v^2 + v) = O(v^2) \tag{16}$$

  quindi non si ha variazione dei costi;

- Liste concatenate con euristica dell'unione pesata (`HeuristicDisjointSets`): anche in questo caso il costo della FIND-SET è costante, e possiamo aggiungere il numero di queste operazioni a (7):

$$m = n + 3e + n = \Theta(2v + 3e) = \Theta(v + e) \tag{17}$$

  dimostrando che non vi è alcuna variazione dei costi computazionali;

- Foreste di insiemi disgiunti con compressione dei cammini (`ForestDisjointSets`): in questo caso il costo della FIND-SET non è costante e può richiedere nel caso peggiore $O(v)$ con $v = |V|$, quindi considerando (11), (12) e che il numero di FIND-SET aumenta di v volte:

$$\begin{aligned} f &= 2e + v = \Theta(e + v) \\ \Theta(n + f \cdot (1 + \log_{2+\frac{f}{n}} n)) &= \Theta(v + (e + v) \cdot (1 + \log_{2+\frac{e+v}{v}} v)) \\ &= O(e \log_{2+\frac{e+v}{v}} v + v \log_{2+\frac{e+v}{v}} v) \end{aligned} \tag{18}$$

con un leggera variazione rispetto a (13). Ma riconsiderando il costo sulla base del numero di vertici secondo la copertura lineare (1):

$$\begin{aligned} e &= \Theta(v) \\ (18) &= O(v \log_{2+\frac{2v}{v}} v + v \log_{2+\frac{2v}{v}} v) \\ &= O(v \log_4 v + v \log_4 v) \\ &= O(v \log_4 v) \simeq O(v \log_3 v) = (14) \end{aligned} \tag{19}$$

e secondo la copertura quadratica (3):

$$\begin{aligned} e &= \Theta(v^2) \\ (18) &= O(v^2 \log_{2+\frac{v^2+v}{v}} v + v \log_{2+\frac{v^2+v}{v}} v) \\ &= O(v^2 \log_{3+v} v + v \log_{3+v} v) \\ &= O(v^2 \log_{3+v} v) \simeq O(v^2 \log_{2+v} v) = (15) \end{aligned} \tag{20}$$

quindi il costo non varia nemmeno in questo caso;

in generale consideriamo questa operazione come ininfluente nel calcolo dei costi computazionali;

(e) `get_vertices_size()` → `int`: ritorna il numero di vertici contenuti nel grafo;

(f) `get_connections_size()` → `int`: ritorna il numero di singole connessioni contenute nel grafo: se ho una connessione $A \leftrightarrow B$ questa viene contata una sola volta e non come due connessioni distinte $A \to B$ e $B \to A$;

(g) `get_connection_coverage()` → `float`: ritorna la percentuale (compresa tra $[0; 1]$) delle connessioni presenti rispetto al numero massimo;

(h) `random_generate(int, int)` → `Graph`: metodo statico che permette di generare un'istanza di `Graph` aggiungendo vertici e connnessioni in maniera casuale; il primo argomento indica il numero di vertici, il secondo il numero di connessioni e quando non è specificato è pari ad un numero casuale compreso tra 0 e il numero massimo di archi possibile secondo (2); le connessioni vengono scelte casualmente da un pool contenente tutte le possibili connessioni valide utilizzando le librerie `numpy` e `itertools`;

(i) `random_generate_with_poly_coverage(int, float)` → `Graph`: metodo statico che genera un'istanza di `Graph` richiamando `random_generate`; il primo parametro mantiene lo stesso scopo, il secondo, non più opzionale, rappresenta la copertura polinomiale che viene applicata al grafo secondo (3);

(j) `random_generate_with_lin_coverage(int, float)` → `Graph`: metodo statico che si comporta come `random_generate_with_poly_coverage`, eccezione per il secondo parametro che rappresenta la copertura lineare che viene applicata al grafo secondo (1);

2.5 Il modulo `__main__`

Questo modulo rappresenta il file principale del progetto e va ad eseguire i test su diversi grafi utilizzando le diverse implementazioni per gli insiemi disgiunti.

In prima istanza vengono generati i grafi usati per i test sulla base dei valori di copertura assegnati: per la copertura lineare `lin` vengono assegnati i valori `0.75` e `1` per evitare eventuali *outliers* senza dover aumentare spropositamente il numero dei vertici che nei test di copertura lineare assumono i valori $\{1000, 2000, \ldots, 5000\}$; per la copertura quadratica `poly` vengono assegnati gli valori `0.75` e `1` per avere anche un confronto più corretto tra i due metodi di copertura, ma con numeri di vertici che assumono i valori $\{100, 200, \ldots, 1000\}$ per evitare lunghi tempi nella generazione dei grafi. Per comprendere meglio la differenza tra copertura lineare e quadratica e individuare il numero di connesioni effettive nella tabella 1 vengono mostrati i valori di numeri di connessioni rispetto ai nodi. In totale si hanno $2 \cdot 10 + 2 \cdot 5 = 30$ grafi da testare con le 3 implementazioni degli insiemi disgiunti.

Copertura lineare				Copertura quadratica			
75%		100%		75%		100%	
Vertici	Connessioni	Vertici	Connessioni	Vertici	Connessioni	Vertici	Connessioni
1000	750	1000	1000	100	3712	100	3712
2000	1500	2000	2000	200	14925	200	14925
3000	2250	3000	3000	300	33637	300	33637
4000	3000	4000	4000	400	59850	400	59850
5000	3750	5000	5000	500	93562	500	93562
				600	134775	600	134775
				700	183487	700	183487
				800	239700	800	239700
				900	303412	900	303412
				1000	374625	1000	374625

Tabella 1: Numeri di connessioni per nodi basati sui test

I test vengono quindi fatti passando al metodo `get_connected_components` di ognuno dei grafi le classi che implementano l'interfaccia `DisjointSetsInterface` volta per volta e registrando i tempi attraverso la libreria `time`, sottraendo il valore finale di `time.time()` a quello iniziale. Questo processo viene ripetuto per tre volte e alla fine viene calcolata la mediana dei tempi impiegati per evitare ulteriori *outliers*.

Dopodichè i tempi, arrotondati a 3 cifre dopo la virgola, vengono salvati su un file `.csv` insieme ai valori di riferimento dei grafi e tramite la libreria `matplotlib` vengono visualizzati i grafici dei tempi medi, uno per ogni valore di copertura, dove sull'asse x abbiamo i valori dei numeri di vertici mentre sull'asse y i valori dei tempi.

3 Risultati dei test

I test sono stati eseguiti su un emulatore di terminale con sistema operativo basato sulla distribuzione *Linux ArchLinux*, attraverso un *Python virtual environment* di versione `3.11.5`. L'hardware utilizzato è il seguente:

- **CPU**: Intel(R) Core(TM) i3-5005U @ 2.00GHz
- **RAM**: 8GiB @ 1600Mhz
- **SSD**: KINGSTON SA400S3 128GB

Lanciando il seguente comando `time python .` nella directory in cui si trova il file `__main__.py` si ottiene, oltre all'output generato dal codice *Python* stesso anche il risultato del comando `time`:

```
python .  44.33s user 5.51s system 35% cpu 2:20.62 total
```

i dati più significativi sono il `5.51s system` che indica il tempo di CPU usato dal programma e il `35% cpu` che indica il picco massimo di utilizzo della CPU generato per l'esecuzione del codice.

Passando ai dati invece otteniamo tre grafici che si differenziano per i valori di copertura dei grafi, con tre curve per ogni grafico che nella legenda sono indicate con:

- `LC`: sta per *Liste Concatenate* e indica i tempi dell'implementazione `ListDisjointSets`;
- `LC/EUP`: sta per *Liste Concatenate con Euristica dell'Unione Pesata* e indica i tempi dell'implementazione `HeuristicDisjointSets`
- `FCC`: sta per *Foreste con Compressione dei Cammini* e indica i tempi dell'implementazione `ForestDisjointSets`.

3.1 Analisi coperture lineari

Con le coperture lineari ci riferiamo ai casi in cui il numero di connessioni è lineare con il numero di vertici, come riportato in (1). Analizzando i grafici dei tempi in figura 4 e 5 possiamo stabilire che:

1. `LC`: sebbene nella figura 4 non sia chiaro, la curva segue il comportamento quadratico definito in (5) che si può notare in parte nella figura 5;
2. `LC/EUP`: anche questa segue il comportamento definito in (9) in tutti i casi; a primo impatto può sembrare che la curva si appiattisca verso il basso con copertura 100% (figura 5) ma in realtà questo è dovuto solo al cambiamento di scala per via dei maggiori valori della curva `LC`;
3. `FCC`: anche questa curva segue la legge definita in (14) e si può notare come eguagli la curva `LC/EUP`, infatti le leggi dei costi sono pressochè identiche ($O(v \log_2 v) \simeq O(v \log_3 v)$); in figura 4 si può anche notare la differenza di base nel logaritmo, che nella curva `FCC` è 3, mentre nella curva `LC/EUP` è 2 e genera una curva più alta per quest'ultima, quindi denota l'implementazione con foreste e compressione dei cammini più adatta nel caso di copertura lineare.

Vertici	LC	LC/EUP	FCC
1000	0.007	0.003	0.002
2000	0.028	0.007	0.004
3000	0.056	0.01	0.007
4000	0.11	0.013	0.008
5000	0.153	0.017	0.011

Tabella 2: Mediane dei tempi (s) per una copertura lineare al 75%

Vertici	LC	LC/EUP	FCC
1000	0.015	0.003	0.002
2000	0.06	0.007	0.005
3000	0.154	0.011	0.007
4000	0.289	0.014	0.01
5000	0.374	0.018	0.017

Tabella 3: Mediane dei tempi (s) per una copertura lineare al 100%

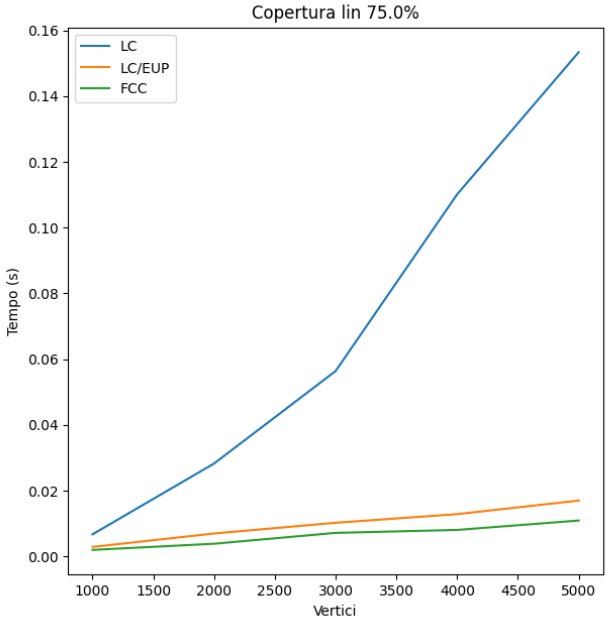

Figura 4: Grafico dei tempi di esecuzione con copertura lineare al 75%

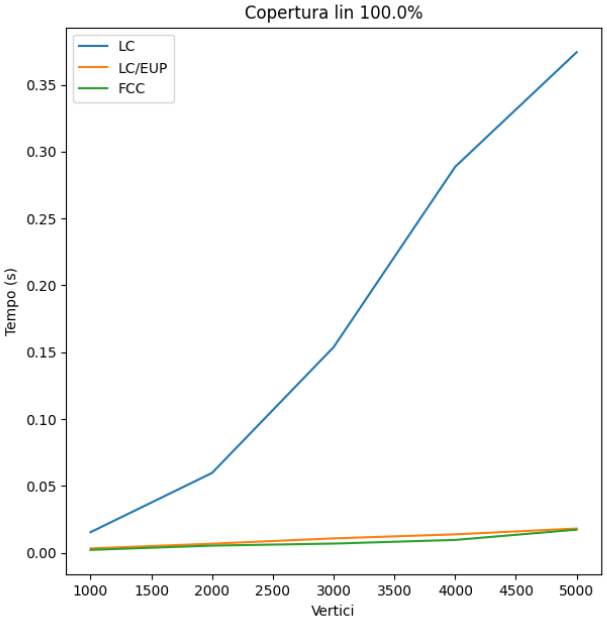

Figura 5: Grafico dei tempi di esecuzione con copertura lineare al 100%

3.2 Analisi coperture quadratiche

Con le coperture quadratiche ci riferiamo ai casi in cui il numero di connessioni è quadratico con il numero di vertici, come riportato in (3).
Analizzando i grafici dei tempi in figura 6 e 7 possiamo stabilire che:

1. `LC`: continua a comportarsi come nel caso delle coperture lineari come previsto in (5), e quindi assume una forma quadratica in entrambe le figure e quindi in entrambi i valori di copertura;

2. `LC/EUP`: è la curva più bassa di tutte e quindi mostra che nel caso di copertura quadratica l'implementazione con liste concatenate e euristica dell'unione pesata è quella che si comporta meglio;nella figura 7 la curva si avvicina alla curva `LC` e questo torna col fatto che le due implementazioni condividono lo stesso costo secondo (5) e (10);

3. `FCC`: è la curva che si comporta in maniera peggiore rispetto alle altre in entrambi i valori di copertura ed è giustificabile secondo (15).

Vertici	LC	LC/EUP	FCC
100	0.001	0.001	0.003
200	0.005	0.004	0.009
300	0.011	0.009	0.015
400	0.021	0.017	0.024
500	0.034	0.027	0.039
600	0.053	0.041	0.057
700	0.074	0.058	0.081
800	0.094	0.085	0.108
900	0.118	0.098	0.137
1000	0.146	0.124	0.17

Tabella 4: Mediane dei tempi (s) per una copertura quadratica al 75%

Vertici	LC	LC/EUP	FCC
100	0.002	0.001	0.002
200	0.006	0.005	0.008
300	0.013	0.012	0.018
400	0.025	0.021	0.032
500	0.041	0.034	0.049
600	0.058	0.049	0.07
700	0.079	0.068	0.096
800	0.108	0.09	0.128
900	0.137	0.115	0.167
1000	0.178	0.147	0.207

Tabella 5: Mediane dei tempi (s) per una copertura quadratica al 100%

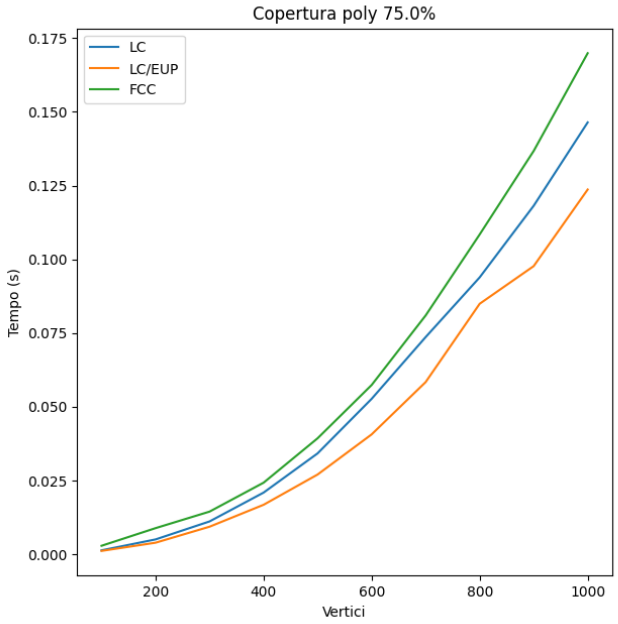

Figura 6: Grafico dei tempi di esecuzione con copertura quadratica al 75%

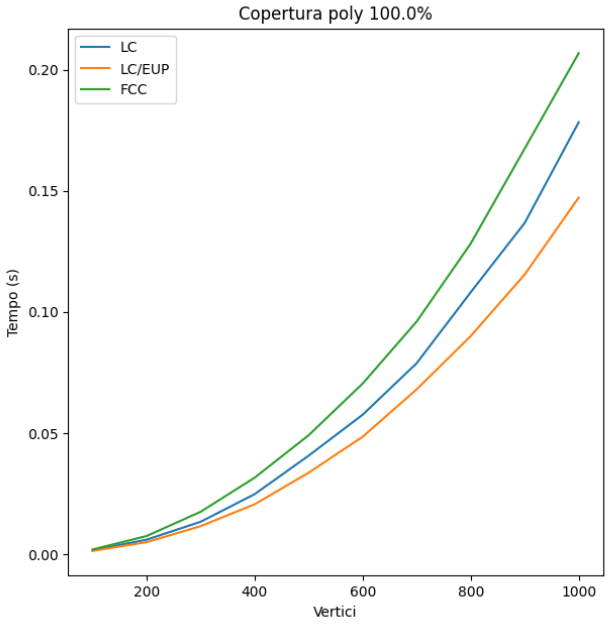

Figura 7: Grafico dei tempi di esecuzione con copertura quadratica al 100%

3.3 Considerazioni finali

Alla luce dei risultati dei test possiamo concludere che le tre implementazioni hanno comportamenti differenti e incidono notevolmente nella ricerca di componenti connesse. In particolare:

- L'implementazione con liste concatenate (senza euristiche) `LC` è quella che si comporta in modo peggiore rispetto alle altre in qualsiasi caso, sia quando il numero di connessioni nel grafo è lineare sia quando è quadratico rispetto al numero di vertici presenti;
- L'implementazione con foreste di nodi con compressione dei cammini `FCC` è la migliore nel caso di un numero di connessioni lineare rispetto al numero dei vertici, ma quando questi diventano quadratici anche l'implementazione risente di questo aumento e tende a comportarsi in maniera peggiore rispetto a tutte le altre implementazioni; solitamente questo tipo di implementazione viene accompagnata da un'ulteriore euristica nota come *union by rank* che, insieme alla compressione dei cammini, tende ad avere costi computazionali migliori in tutti i casi, ma in questa analisi è stata richiesta l'implementazione della semplice compressione dei cammini;

- L'implementazione con liste concatenate attraverso l'euristica dell'unione pesata, nei casi analizzati, è quella che si comporta complessivamente in maniera migliore, quindi considerando sia il caso di numero di connessioni lineare, sia il caso quadratico, rispetto ai vertici; ad ogni modo non è detto che non si possano trovare implementazioni più veloci di questa, e un esempio è la *union by rank* citata nel punto precedente.