
UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

# Laboratorio di Algoritmi e Strutture Dati

*Autore:*
Cristian Mazzarone

*N° Matricola:*
7030329

*Corso principale:*
Algoritmi e Strutture Dati

*Docente corso:*
Simone Marinai

# Indice



# Elenco delle figure

# 1 Introduzione generale

## 1.1 Progetto assegnato

Il progetto assegnato per il superamento della parte di Laboratorio di Algoritmi e Strutture Dati riguarda il seguente esercizio:

- Contronto tra insiemi disgiunti implementati con liste concatenate (con e senza euristica) e con le foreste di insiemi disgiunti con compressione dei cammini

## 1.2 Descrizione dello svoglimento dell'esercizio

Per trattare al meglio l'argomento dell'elaborato, suddivideremo in 4 parti distinte la sua descrizione:

- **Spiegazione teorica del problema** : qui è dove si descrive il problema che andremo ad affrontare in modo teorico partendo dagli assunti del libro di Algoritmi e Strutture Dati e da altre fonti.
- **Documentazione del codice** : in questa parte spieghiamo come il codice dell'esercizio viene implementato
- **Descrizione degli esperimenti condotti** : partendo dal codice ed effettuando misurazioni varie cerchiamo di verificare le ipotesi teoriche
- **Analisi dei risultati sperimentali** : dopo aver svolto i vari esperimenti riflettiamo sui vari risultati ed esponiamo una tesi

## 1.3 Specifiche della piattaforma di test

I test sono stati condotti su un computer portatile con le seguenti carattestiche:

- **CPU** : Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz
- **RAM** : 16,0 GB
- **SSD** : Kingstone A400 da 500 GB

Il linguaggio di programmazione utilizzato sarà Python e la piattaforma in cui il codice è stato scritto è l'IDE **VS Code**. La stesura di questo testo è avvenuta tramite l'utilizzo dell'editor online **Overleaf**.

# Parte I
# Insiemi disgiunti

**Esercizio G**

- Vogliamo confrontare vari modi per implementare strutture dati per insiemi disgiunti (capitolo 21 libro di testo):
  1. tramite liste concatenate (con e senza euristica dell'unione pesata)
  2. tramite foreste di insiemi disgiunti (con compressione del cammino)
- Per confrontare gli algoritmi si utilizzino algoritmi per trovare componenti connesse in grafi non diretti Nota: La lista deve essere implementata considerando strutture collegate con puntatori e non la struttura dati lista di Python
- per fare questo dovremo:
  - Scrivere i programmi in Python che:
    - implementano quanto richiesto
    - eseguono un insieme di test che ci permettano di comprendere i vantaggi e gli svantaggi delle diverse implementazioni
  - scrivere una relazione che descriva quanto è stato fatto

## 2 Spiegazione teorica del problema

### 2.1 Introduzione

Gli insiemi disgiunti sono insiemi che non hanno elementi in comune. Alcune applicazioni richiedono di raggruppare n elementi distinti una collezione di insiemi disgiunti. Queste applicazioni spesso hanno bisogno dli svolgere due operazioni in particolare: trovare l'insieme unico che contiene un dato elemento e unire due insiemi disgiunti.

Una struttura dati per insiemi disgiunti deve mantenere un insieme finito di insiemi disgiunti e dinamici. Ongi insieme è identificato da un rappresentante, che non è altro che un membro dell'insieme stesso.

### 2.2 Operazioni sugli insiemi disgiunti

Rappresentiamo ogni elemento di un insieme come un oggetto. Prendiamo un oggetto $x$, vorremo poter effettuare le seguenti operazioni:

- Make-Set($x$): crea un nuovo insieme formato dal solo elemento $x$, che rappresenterà anche l'insieme. Dato che gli insiemi devono essere disgiunti, dobbiamo assicurarci che non esista già l'elemento $x$ all'interno di un altro insieme.
- Union($x$, $y$): unisce gli insiemi dinamici che contengono $x$ e $y$, rispettivamente $S_x$ e $S_y$, in un nuovo insieme che è l'unione dei precedenti. Nel nostro caso il nuovo rappresentante sarà uno dei precedenti. Dato che gli insiemi devono essere disgiunti, dobbiamo eliminare i vecchi insiemi.
- Find-Set($x$): dato un elemento in ingresso, questa funzione ritorna il rappresentante dell'insieme che lo contiene. Dato che gli insiemi sono disgiunti, il rappresentante trovato è unico.

### 2.3 Una applicazione degli insiemi disgiunti

Gli insiemi disgiunti ci vengono in aiuto quando dobbiamo trovare le componenti connesse di un grafo (non orientato). Per farlo andiamo a utilizzare la funzione Connected-Components, che crea un nuovo insieme per ogni nodo del grafo e poi fa l'unione sugli insiemi che contengono i nodi che sono connessi da archi.

## 2.4 Aspetti fondamentali

Le varie implementazioni che andremo a usare, hanno carateristiche diverse:

- **Liste concatenate**: ogni elemento ha un puntatore al prossimo elemento della lista e un puntatore al rappresentante dell'insieme disgiunto. Trovare il rappresentante della lista sarà quindi molto semplice e impiegherà un tempo costante, Le criticità di questa implementazione iniziano a farsi sentire quando andiamo a unire insiemi con un numero diverso di elementi: mettiamo che noi vogliamo unire l'insieme $S_2$ all'insieme $S_1$, dovremo quindi trovare l'ultimo elemento di $S_1$ (una possibile soluzione potrebbe prevedere un puntatore all'ultimo elemento della lista) e collegarlo al primo elemento dell'insieme $S_2$, ovvero al suo rappresentante; a questo punto dovremo aggiornare ogni elemento di $S_2$ in modo tale che il rappresentante del nuovo insieme diventi il rappresentante di $S_1$

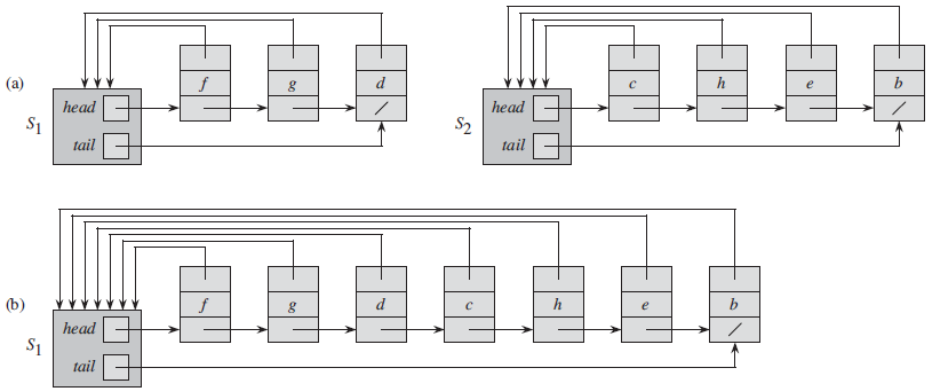



Figura 1: liste concatenate

Ovviamente la complessità di questa operazione dipende dalla lunghezza della lista $S_2$.

- **Liste concatenate con euristica delle pesate**: Questa implementazione è identica a quella riportata sopra, con l'unica differenza che l'operazione di unione andrà a concatenare la lista che, tra le due, presenta la lunghezza inferiore. E' quindi necessario aggiungere un campo nell'header dell'insieme che mantiene il numero di elementi contenuti

- **Foreste con compressione dei cammini**: in questo caso gli insiemi disgiunti sono implementati attraverso degli alberi, la cui radice identifica il rappresentante dell'insieme. ogni nodo ha un padre, anche la radice, che punta a se stessa. La compressione dei cammini viene usata durante l'operazione di Find-Set e permette a ogni nodo di puntare direttamente alla radice. La procedura di Find-Set è detta ***two-pass method***: mentre ricorre, svolge un passo verso l'alto per trovare la radice dell'albero, mano a mano che la ricorsione si svolge, il parent dei nodi viene aggiornato in modo che puntino direttamente alla radice:

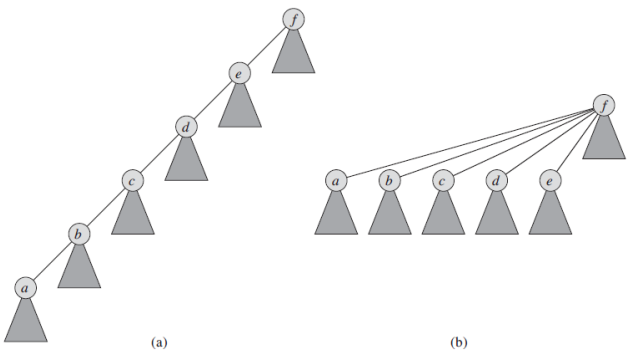



Figura 2: compressione dei cammini

## 2.5 Complessità degli algoritmi

In questa sezione andiamo a analizzare gli algoritmi delle 3 operazioni illustrate sopra, implementate per tutte e tre le modalità che trattiamo. L'obiettivo è farci un'idea della complessità degli algoritmi, in modo da valutare al meglio i risultati che otterremo nella parte pratica del progetto.

Supponiamo di avere $n$ elementi e vogliamo eseguire le varie operazioni su di loro. Per prima cosa dovremo creare dei nuovi insiemi disgiunti, uno per ogni elemento. In seguito potremo eseguire come vogliamo le varie operazioni di unione e ricerca dell'insieme.

Vediamo il costo computazionale nel caso di liste concatenate:

- Make-Set($x$): deve semplicemente creare un insieme disgiunto composto dal solo elemento $x$. Questo algoritmo ha tempo costante $O(1)$. Quindi se devo creare $n$ insiemi, il costo computaizonale sarà $O(n)$
- Find-Set($x$): Questa è ovvia. Dato che tutti hanno il puntatore al rappresentante, questo algoritmo ha costa costante: $O(1)$
- Union($y$, $y$): Questo algoritmo unisce due insiemi, appendendo l'insieme y all'insieme x. Supponiamo di voler unire $n$ elementi; dovrò fare in tutto $n-1$ unioni (Union($x_2$, $x_1$), Union($x_3$, $x_2$), ..., Union($x_n$, $x_{n-1}$). L'operazione di unione deve aggiornare il puntatore al rappresentante di tutti gli elementi della seconda lista e aggiornare il puntatore all'ultimo elemento della prima lista. Dato che in tutto abbiamo $n$ elementi, nel peggiore dei casi (concatenazione di un insieme con n-1 elementi a un insieme con un solo elemento) vengono eseguiti $n-1$ aggiornamenti per il puntatore al rappresetnante e un un aggiornamento per il puntatore all'ultimo elemento, ottenendo un totale di $n$ operazioni. Quindi l'esecuzione della unione, dati n elementi, ha una complessità nel caso peggiore di $\theta(n)$
- Union($y$, $y$) con euristica delle pesate: in questo caso evitiamo il caso peggiore, in quanto concateniamo sempre la lista più piccola alla lista più grande. Supponiamo di avere n insiemi disgiunti e di volerli unire. Verranno eseguite al massimo $n-1$ unioni. Osserviamo che ogni volta che il puntatore al rappresentante di un oggetto viene aggiornato, la lista contenente aumenta di dimensione di almeno 2 volte la dimensione originale, quindi il puntatore al rappresentante di un oggetto viene aggiornato al più $log(n)$ volte. Quindi, la complessità delle unioni su $n$ insiemi disgiunti, in questo caso, è $O(nlog(n))$

Nel caso di insiemi disgiunti con foreste e compressione dei cammini, avrò una complessità totale di $O(n\alpha(n))$ se devo eseguire delle unioni su un gruppo di n elementi. $\alpha(n)$ è una funzione di n che cresce molto lentamente.

# 3 Documentazione del codice

## 3.1 Diagamma UML

Di seguito è riportato il diagramma UML delle classi.

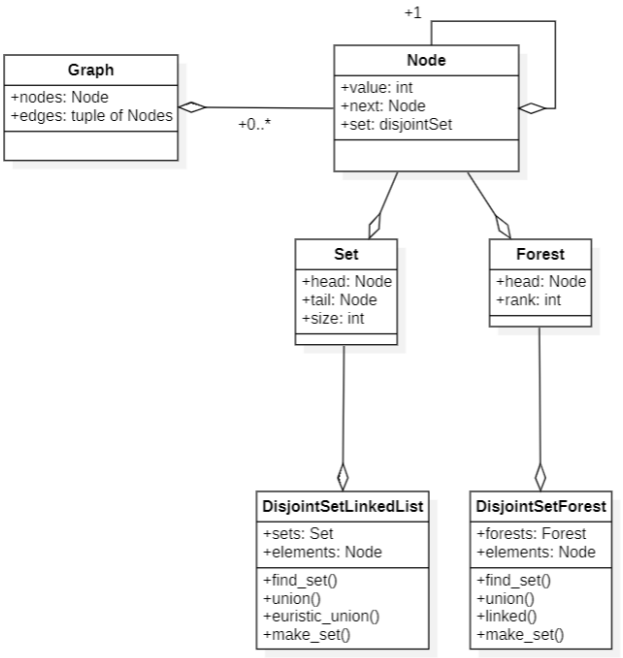



Figura 3: diagramma UML delle classi

Per la creazione dei plot è ho usato la libreria pyplot di Python, implementata attraverso la classe Plot sotto riportata

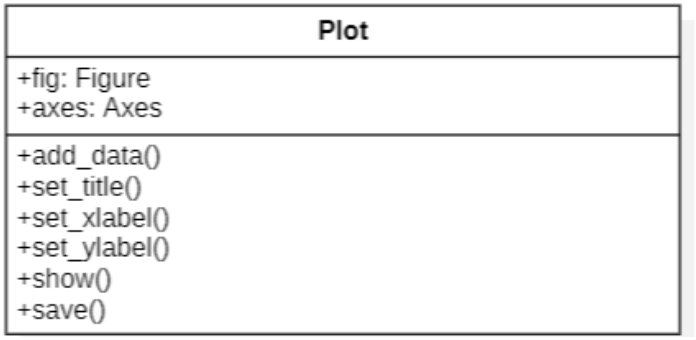
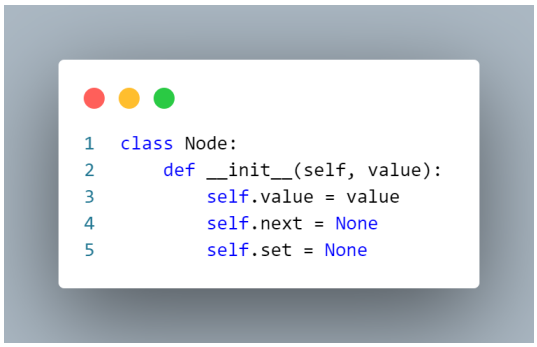



Figura 4: diagramma UML della classe Plot

## 3.2 Implementazione degli algoritmi

Come possiamo vedere dal diagramma UML, entrambe le implementazioni degli insiemi disgiunti utilizzano la classe Node:

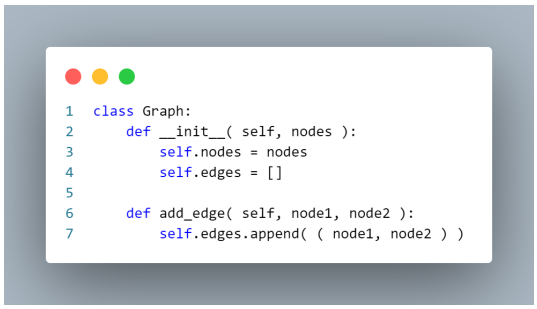


```
1  class Node:
2      def __init__(self, value):
3          self.value = value
4          self.next = None
5          self.set = None
```

Figura 5: classe Node

L'attributo next punterà al nodo successivo nel'insieme di appartenenza: l'attributo set punta all'insieme di appartenenza, viene definito alla creazione del'insieme e modficato quando si eseguono le unioni. Quest'ultimo attributo è stato inserito per semplificare la gestione degli insiemi.

Vediamo adesso la classe Graph:



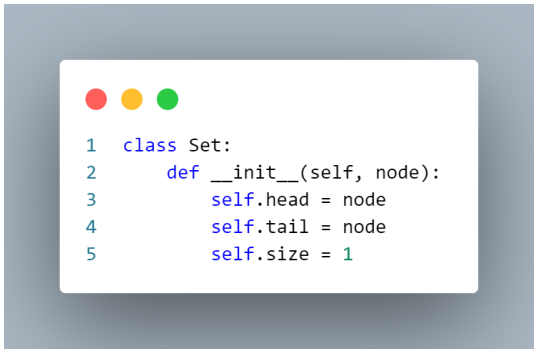
```
1  class Graph:
2      def __init__( self, nodes ):
3          self.nodes = nodes
4          self.edges = []
5
6      def add_edge( self, node1, node2 ):
7          self.edges.append( ( node1, node2 ) )
```

Figura 6: classe Graph

Alla creazione di un grafo, vengono passati tutti i nodi; atttraverso il metodo add_edge(node1, node2) viene aggiunto un arco non orientato che collega node1 con node2.

### 3.2.1 Insiemi disgiunti con liste concatenate

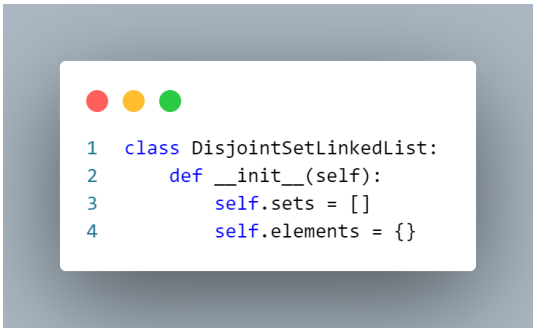
Un insieme un'istanza della classe Set:



```
1  class Set:
2      def __init__(self, node):
3          self.head = node
4          self.tail = node
5          self.size = 1
```

Figura 7: classe Set

e gli insiemi disgiunti sono implementati con la classe DisjointSetLinkedList, che ha come attributi un array di insiemi Set e una lista di elementi contenuti negli insiemi per rendere più semplice la ricerca:

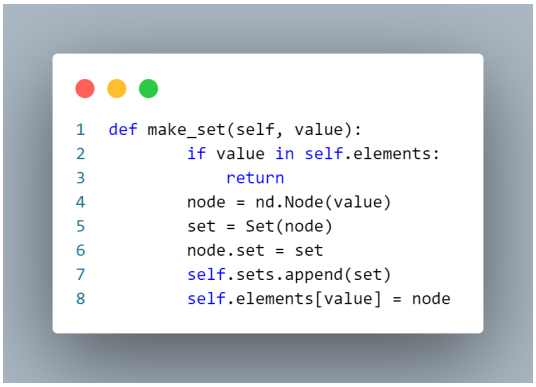


```
1  class DisjointSetLinkedList:
2      def __init__(self):
3          self.sets = []
4          self.elements = {}
```

Figura 8: classe DisjointSetLinkedList

I metodi implementati in questa classe sono:

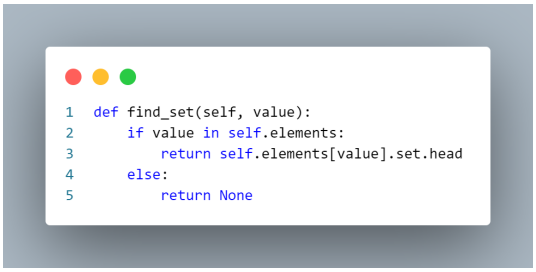
- make_set(value): crea un nodo con il valore che riceve in ingresso; successivamente crea un set composto dal signolo elemento appena creato e lo inserisce nella lista di insiemi disgiunti gestiti. Nel nodo creato passo il suo set di appartenenza:



```
1  def make_set(self, value):
2          if value in self.elements:
3              return
4          node = nd.Node(value)
5          set = Set(node)
6          node.set = set
7          self.sets.append(set)
8          self.elements[value] = node
```

Figura 9: make_set()

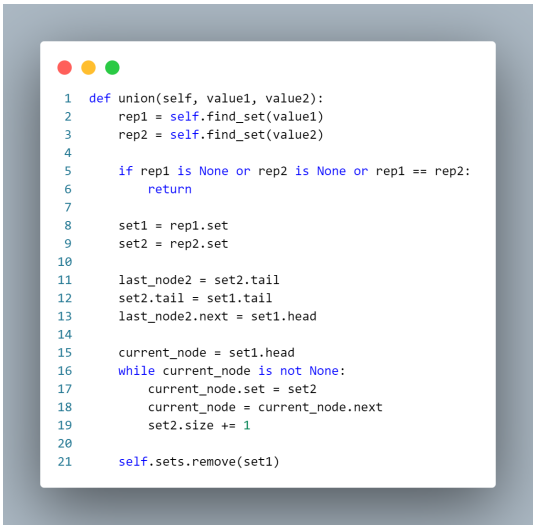
- find_set(value): ricerca il nodo con valore value e ritorna il rappresentante del suo set di appartenenza:



```
1  def find_set(self, value):
2      if value in self.elements:
3          return self.elements[value].set.head
4      else:
5          return None
```

Figura 10: find_set()

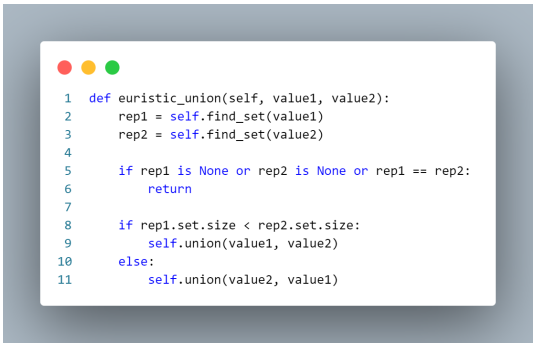
- union(value1, value2): trova i set $S_1$ e $S_2$, che contengono rispettivamente il nodo corrispondente a value1 e il nodo corrispondente a value2; successivamente concatena $S_1$ a $S_2$:



```python
def union(self, value1, value2):
    rep1 = self.find_set(value1)
    rep2 = self.find_set(value2)

    if rep1 is None or rep2 is None or rep1 == rep2:
        return

    set1 = rep1.set
    set2 = rep2.set

    last_node2 = set2.tail
    set2.tail = set1.tail
    last_node2.next = set1.head

    current_node = set1.head
    while current_node is not None:
        current_node.set = set2
        current_node = current_node.next
        set2.size += 1

    self.sets.remove(set1)
```

Figura 11: union()

- euristic_union(value1, value2): confronta la dimensione dei set relativi a value1 e value2 e richiama il metodo union in modo che svolga la concatenzazione dell'insieme più piccolo all'insieme più grande, portando a un'incremento delle prestazioni:

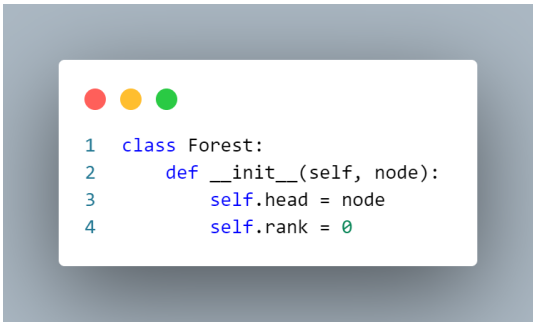


```python
def euristic_union(self, value1, value2):
    rep1 = self.find_set(value1)
    rep2 = self.find_set(value2)

    if rep1 is None or rep2 is None or rep1 == rep2:
        return

    if rep1.set.size < rep2.set.size:
        self.union(value1, value2)
    else:
        self.union(value2, value1)
```

Figura 12: euristic_union()

### 3.2.2 Insiemi disgiunti con foreste

Questa implementazione, a differenza della precedente, non utilizza la classe Set per definire i singoli insiemi, benì utilizza la classe Forest:

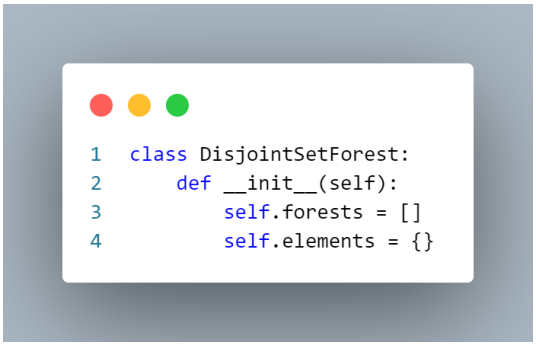


```python
class Forest:
    def __init__(self, node):
        self.head = node
        self.rank = 0
```

Figura 13: classe Forest

Con ogni nodo viene crata una foresta, attravaerso la quale manteniamo un puntatore al rappresentante (head) e il rango (rank), ovvero il numero di nodi presenti sotto.

Gli insiemi disgiunti con foreste vengono dichiarati attraverso la classe DisjointSetForest:

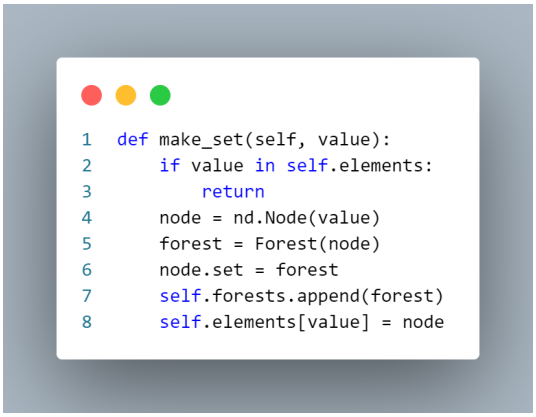


```
1  class DisjointSetForest:
2      def __init__(self):
3          self.forests = []
4          self.elements = {}
```

Figura 14: classe DisjointSetForest

Questa classe mette a disposizioni tutti i metodi per gestire gli insiemi disgiunti con le foreste, vediamoli:

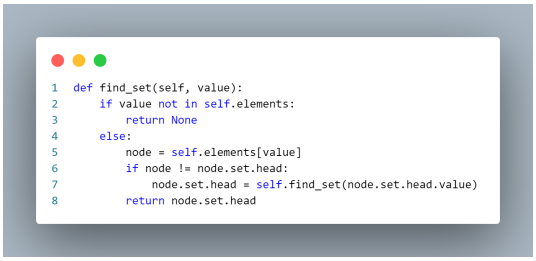
- make_set(value): Questa funzione è analoga a quella già vista. Viene creata una foresta (oggetto di tipo forest) con il valore ricevuto in ingresso (dopo aver creato un nodo):



```
1  def make_set(self, value):
2      if value in self.elements:
3          return
4      node = nd.Node(value)
5      forest = Forest(node)
6      node.set = forest
7      self.forests.append(forest)
8      self.elements[value] = node
```

Figura 15: metodo make_set()

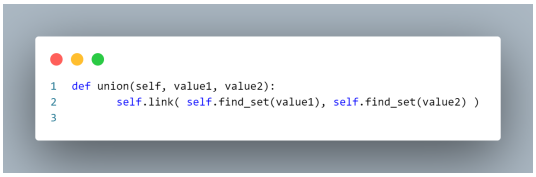
- find_set(value): In questo caso, mentre si risale l'albero, i cammini vengono compressi in modo che tutti i nodi puntino direttamente alla radice, ovvero al rappresetnante dell'insieme. Questo permette di mantenere le performance molto alte, in quanto il cammino tra qualsiasi nodo e il ruo rappresentante è lungo 1:



```
1  def find_set(self, value):
2      if value not in self.elements:
3          return None
4      else:
5          node = self.elements[value]
6          if node != node.set.head:
7              node.set.head = self.find_set(node.set.head.value)
8          return node.set.head
```

Figura 16: metodo find_set()

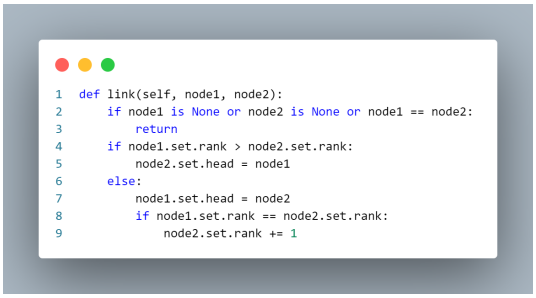
- union(value1, value2): in questo caso, la funzione richiama semplimenete la funzione link(node1, node2):



```
1  def union(self, value1, value2):
2          self.link( self.find_set(value1), self.find_set(value2) )
3
```

Figura 17: metodo union()

- link(node1, node2): esegue l'unione delle foreste secondo il rango delle stesse, in modo da aumentare ulteriormente le performance:



```python
def link(self, node1, node2):
    if node1 is None or node2 is None or node1 == node2:
        return
    if node1.set.rank > node2.set.rank:
        node2.set.head = node1
    else:
        node1.set.head = node2
        if node1.set.rank == node2.set.rank:
            node2.set.rank += 1
```

Figura 18: metodo union()

# 4 Esperimenti condotti

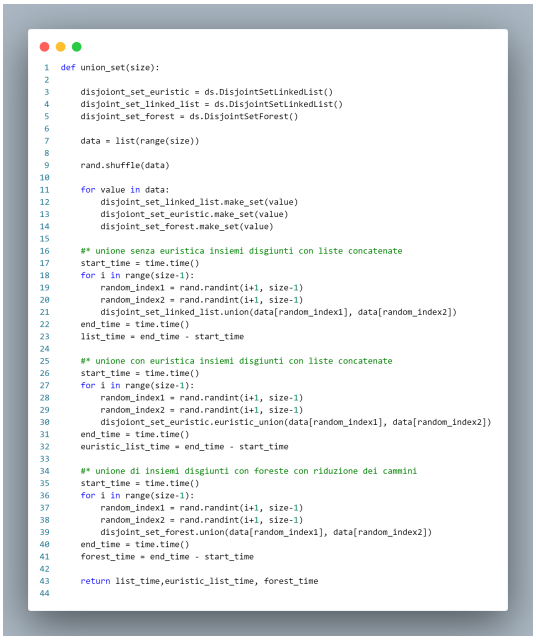
## 4.1 Introduzione

In questa sezione vengono illustrati gli espereimenti condotti al fine di confrontare le varie implementazioni per gli insiemi disgiunti.

Per calcolare il tempo è stata usata la funzione time() del modulo time di Python .

## 4.2 Test sulle performance dell'operazione di Unione

Per eseguire il test sulle performance dell'operazione di unione, andiamo a eseguire l'operazione su un insieme di $N$ nodi. Il test inizia creando $N$ insiemi disgiunti formati da un solo elemento (il test non tiene di conto di questa fase); una votla creati tutti gli insiemi, le unioni avvengono prendendo casualmente due valori dallo stesso array usato per la creazione degli insiemi. Questa componente andomica garantisce che gli insiemi che vengono uniti abbiano sempre dimensione casuale, in modo da rendere evidente la differenza tra le varie implementazioni. In tutto vengono eseguite $N$ unioni in modo casuale sugli insiemi. L'algoritmo usato per il test è il seguente:



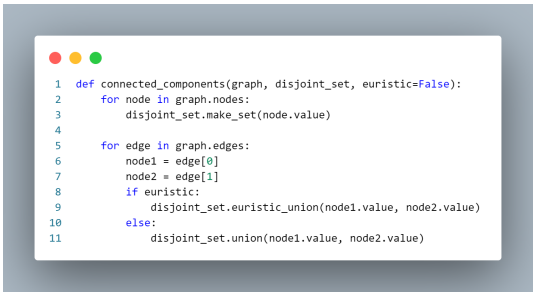
```python
def union_set(size):

    disjoiont_set_euristic = ds.DisjointSetLinkedList()
    disjoint_set_linked_list = ds.DisjointSetLinkedList()
    disjoint_set_forest = ds.DisjointSetForest()

    data = list(range(size))

    rand.shuffle(data)

    for value in data:
        disjoint_set_linked_list.make_set(value)
        disjoiont_set_euristic.make_set(value)
        disjoint_set_forest.make_set(value)

    #* unione senza euristica insiemi disgiunti con liste concatenate
    start_time = time.time()
    for i in range(size-1):
        random_index1 = rand.randint(i+1, size-1)
        random_index2 = rand.randint(i+1, size-1)
        disjoint_set_linked_list.union(data[random_index1], data[random_index2])
    end_time = time.time()
    list_time = end_time - start_time

    #* unione con euristica insiemi disgiunti con liste concatenate
    start_time = time.time()
    for i in range(size-1):
        random_index1 = rand.randint(i+1, size-1)
        random_index2 = rand.randint(i+1, size-1)
        disjoiont_set_euristic.euristic_union(data[random_index1], data[random_index2])
    end_time = time.time()
    euristic_list_time = end_time - start_time

    #* unione di insiemi disgiunti con foreste con riduzione dei cammini
    start_time = time.time()
    for i in range(size-1):
        random_index1 = rand.randint(i+1, size-1)
        random_index2 = rand.randint(i+1, size-1)
        disjoint_set_forest.union(data[random_index1], data[random_index2])
    end_time = time.time()
    forest_time = end_time - start_time

    return list_time,euristic_list_time, forest_time

```

Figura 19: test sulle unioni

Per ogni dimensione, vengono eseguiti 10 test. Il tempo che troveremo sul grafico sarà la media di questi.

## 4.3 Test sulle performance dell'operazione di Connected-Component

Il test sulla performance dell'operazione di ricerca delle componenti connesse va a sfruttare i metodi degli insiemi disgiunti per ricercare le componenti connesse di un grafo: ogni nodo del grafo è usato per creare un insieme; ogni volta che tra due nodi è presente un arco, si esegue l'unione tra gli insiemi disgiunti dei due nodi. Infatti, la proprietà di disgiunzione degli insiemi garantisce che se due nodi si trovano nello stesso insieme, allora si trovano anche nella stessa componente connessa. Vediamo l'algoritmo che implementa il test:



```python
def connected_components(graph, disjoint_set, euristic=False):
    for node in graph.nodes:
        disjoint_set.make_set(node.value)

    for edge in graph.edges:
        node1 = edge[0]
        node2 = edge[1]
        if euristic:
            disjoint_set.euristic_union(node1.value, node2.value)
        else:
            disjoint_set.union(node1.value, node2.value)
```

Figura 20: test sulle componenti connesse

prende in ingresso un grafo e un oggetto che può essere un insieme disgiunto con foreste o con liste concatenate. Per l'euristica delle pesate un flag euristic dice all'algoritmo in che modo svolgere la ricerca.

# 5 Analisi dei risultati sperimentali

Data la teoria che c'è dietro queste strutture, possiamo analizzare i risultati ottenuti e vedere se corrispondono ai risultati attesi. I test sono eseguiti su un numero di nodi $n$ che arria fino a 100.000.

## 5.1 Risultati del test sulle unioni

Come abbiamo già detto all'inizio della disciussione, l'operazione di unione ha un tempo di esecuzione che varia a seconda dell'implementazione degli insiemi disgiunti. Le liste concatenate senza euristica sulle pesate, eseguono l'unione concatenando due insiemi, aggiornando i puntatori al rappresetnante e il puntatore all'ultimo elemento della lista. Ci aspettiamo di avere una complessita $O(n)$, dove $n$ identifica il numero di nodi. Questo si deve tradurre in un grafico con un **andamento lineare**.

Quando introduciamo l'euristica delle pesate, allora ci aspettiamo di avere un migliormento delle performance, in quanto la lista che viene contatenata è sempre quella più corta, quindi si eseguono meno operazioni di aggiornamento. In particolare ci aspettiamo un caso medio in cui le due liste presentano la stessa lunghezza, portanto a una complessita di $O(nlog(n))$. Questo si traduce in un grafico con un andamento logaritmico.

Gli insiemi disgiunti con le foreste e compressione dei cammini invece eseguono le unioni tenendo di conto del rango delle foreste. La riduzione dei cammini permette di avere un collegametno diretto al rappresentante della lista, garandento una complessità di $O(n\alpha(n))$, dove $\alpha(n)$ è una funzione che cresce molto lentamente all'aumentare di $n$. Questo si traduce in un grafico con una crescita molto lenta:

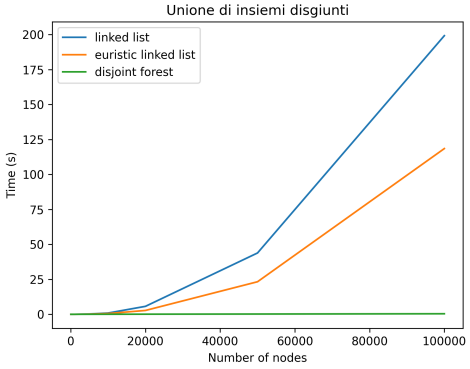



Figura 21: grafico delle unioni

## 5.2 Risultati del test sulle componenti connesse

Il risultato di questo test invece dal numero di archi e di nodi presenti dal grafico. Indichiamo con $v$ il numero di nodi nel grafo, mentre con $e$ il numero di archi. Per ogni nodo viene eseguito un make_set, che ha tempo costante. Quindi la creazione degli insiemi ha complessità $O(v)$. Dopo aver creato gli insiemi, per ogni arco viene eseguita una operazione di unione. Indichiamo con $c$ il costo computazionale dell'operazione di unione (dipende dalla struttura usata per gli insiemi disgiunti), il costo computazionale per l'unione dei nodi connessi è $O(e * c)$. In conclusione l'algoritmo ha costo computazionale totale $O(v + e * c)$.

Per le liste concatenate senza euristica delle pesate mi aspetto di avere un grafico lineare. L'euristica delle pesate deve garantire un miglioramento nelle prestazioni, quindi mi aspetto di avere un grafico con andamento logratmico. Gli insiemi disgiunti con le foreste invece dovrebbere generare un grafico con una crescita molto lenta nel tempo. Vediamo i risultati:

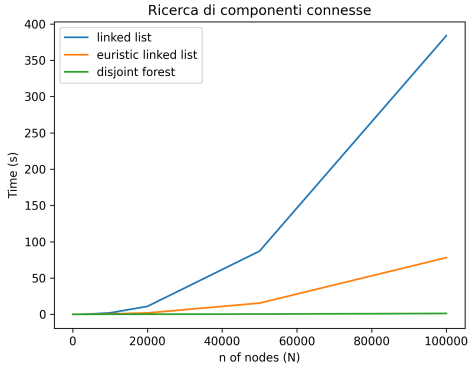



Figura 22: grafico delle componenti connesse

## 5.3 Analisi dei grafici

Dato che gli algoritmi utilizzati sono gli stessi, possiamo discutere i due test insieme.

Notiamo che i risultati ottenuto sono coerenti con la teoria dietro e varie strutture. Come possiamo vedere, nonostante qualche eventuale alterazione, le linee tracciate dai grafici sono coerenti con i risultati previsti, ed è evidente il vantaggio di usare una determinata struttura piuttosto che un'altra. un fatto interessante è la notevole differenza tra le liste concatenate con e senza euristica: una semplice modifica all'algoritmo permette di ottimizzare considerevolmente il codice. Gli insiemi disgiunti implementati con le foreste permette di migliorare ulteriormente le prestazini degli algoritmi: nonostante si aggiunga del lavoro nella funzione di find_set per accorciare i cammini, l'unione degli insiemi è più ottimizzato, in quanto il cammino tra un qualsiasi nodo e il proprio rappresetnante è lungo 1.