Laboratorio di Algoritmi e Strutture dati
Esercizio 1

Leonardo Guglielmi

September 2023

Indice

1 Introduzione generale

1.1 Obiettivo

L'obiettivo di questo esperimento è quello di confrontare varie implementazioni delle strutture dati per rappresentare gli insiemi disgiunti.
Più precisamente si vogliono confrontare le implementazioni tramite liste concatenate, sia con che senza euristica dell'unione pesata, e tramite foreste di insiemi disgiunti con compressione del cammino.
Queste diverse implementazioni verranno confrontate basandosi sull'efficienza nell'eseguire algoritmi per il calcolo delle componenti connesse nei grafi non diretti.

1.2 Una breve descrizione della relazione

La relazione è suddivisa in 5 parti:

- **Descrizione teorica:** si vanno a descrivere tutti gli aspetti teorici che interessano l'esperimento svolto.
- **Prestazioni attese:** si delineano le prestazioni attese delle varie parti dell'esperimento.
- **Descrizione degli esperimenti:** si va a descrivere in che modo gli esperimenti vengono svolti ed il codice che li esegue.
- **Risultati sperimentali:** si espongono i risultati ottenuti dall'esecuzione dell'esperimento, sia in modo descrittivo che graficamente.
- **Deduzioni finali:** una volta osservati i risultati, si vanno a dare le considerazioni finali su quanto i risultati coincidano con le ipotesi fatte.

1.3 Specifiche tecniche

A seguire si elencano tutte le caratteristiche della macchina su cui verrà eseguito l'esperimento:

- modello: Asus vivobook S15
- sistema operativo: fedora (linux)
- processore: Intel Core i7-8565U 1.80 GHz
- RAM: 8 GB DDR-4
- disco: ssd Kingstone A2000 256 GB

Inoltre il linguaggio usato sarà Python 3.11.5, mentre il codice è stato scritto e testato usando l'IDE Pycharm Professional 2023.2.1
Questa relazione è stata scritta usando l'editor online Overleaf.
Per la stesura di questa relazione e per la realizzazione di questo esperimento è stato preso come riferimento il libro [1] *Introduzione agli algoritmi e strutture dati* di Thomas Cormen.

2 Descrizione teorica

2.1 Strutture dati per insiemi disgiunti

Le strutture dati per insiemi disgiunti, dette anche strutture Union-Find, sono utilizzate per rappresentare un insieme di collezioni disgiunte di elementi distinti, ognuna delle quali avrà uno dei suoi elementi come rappresentante. Su queste strutture vengono svolte le generiche operazioni di:

- *make_set(e)*: crea un nuovo insieme $\{e\}$ contente il solo elemento e, il quale sarà rappresentante del nuovo inseme.
- *union(A,B)*: unisce gli insiemi A e B, il nuovo insieme avrà come rappresentante il rappresentante di A.
- *find(e)*: restituisce il rappresentante dell'insieme in cui è contenuto e.

2.2 Rappresentazioni per gli insiemi disgiunti

Benché il costo dell'operazione di *make_set* risulti sempre $O(1)$, il costo delle altre operazioni varia il base alle varie rappresentazioni, in particolare in questo esperimento verranno usate le seguenti:

2.2.1 Lista concatenata

In questa rappresentazione, detta anche *QuickFind*, ciascun insieme è rappresentato da una lista concatenata.
Ogni lista ha un puntatore *head* che punta al primo elemento, il quale è anche rappresentante dell'insieme, ed un puntatore *tail* che invece punta all'ultimo elemento della lista.
Ciascun elemento della lista invece contiene l'identificativo dell'elemento dell'insieme, un puntatore all'oggetto lista ed un puntatore al successivo elemento.
Tramite questa rappresentazione l'operazione di *find* ha un'implementazione molto efficiente, in quanto deve soltanto seguire i puntatori.
Per quanto riguarda l'operazione di *union* questo tipo di rappresentazione risulta meno efficiente, essa infatti si realizza andando ad accodare una lista all'altra, dovendo quindi andare ad aggiornare il puntatore all'oggetto lista di tutti gli elementi appartenenti alla lista accodata, impiegando quindi un tempo al più lineare $\theta(n)$.

2.2.2 Foresta di insiemi disgiunti

Con questa rappresentazione, detta anche *QuickUnion*, ogni insieme è rappresentato da un albero.
La radice dell'albero sarà il rappresentante dell'insieme e ogni elemento dell'insieme è rappresentato da un nodo, il quale contiene l'identificativo dell'elemento ed un puntatore al nodo padre *father*.
Nella radice il puntatore *father* punterà a se stessa.
Tramite tale rappresentazione si ha che l'implementazione dell'operazione di *union* risulta molto semplice, in quanto si andrà a rendere uno dei due insiemi un sotto-albero della radice dell'altro: per implementare ciò si va a modificare il puntore *father* della radice dell'insieme che vogliamo accodare, facendolo puntare alla radice dell'altro insieme.
Tuttavia l'implementazione di *find* risulta più problematica poiché si deve risalire dal nodo di partenza tutto l'albero fino alla radice tramite i puntatori *father*, risultando più costosa rispetto a *find* e *make_set*.
Il cammino che si forma durante la ricerca viene detto *cammino di ricerca*.

2.3 Euristiche

Esistono delle euristiche che permettono di migliorare i tempi di esecuzione delle operazioni più costose di ciascuna rappresentazione e delle relative implementazioni:

2.3.1 Euristica dell'unione pesata

usata nella rappresentazione tramite **lista concatenata**, quando si va a fare una *union* si sceglie di accodare la lista con il minor numero di elementi a quella contenente più elementi, in modo da ridurre il numero di elementi da aggiornare.
Questo richiede che la lista memorizzi anche il numero di elementi al suo interno.

2.3.2 Compressione del cammino

usata della rappresentazione tramite **foresta di insiemi disgiunti**, consiste nel rendere ogni nodo nel cammino di ricerca di una *find* figlio della radice, andando quindi ad aggiornare il puntatore *father* del nodo nel cammino in

modo che punti direttamente alla radice.

2.4 Algoritmo per il calcolo delle componenti connesse

Infine, per costruire le **componenti connesse** verrà usato l'algoritmo per grafi non orientati (Figura 1), il quale eseguendo una serie di operazioni di *make_set, find* e *union* andrà a creare una collezione di insiemi, ognuno rappresentante una componente connessa.

```
CONNECTED-COMPONENTS(G)
  for ogni vertice v ∈ G.V
    MAKESET(v)
  for ogni arco (u,v) ∈ G.E
    if FINDSET(u) ≠ FINDSET(v)
      UNION(u,v)
```

Figura 1: Pseudocodice algoritmo per il calcolo delle componenti connesse

L'esecuzione dell'algoritmo per il calcolo delle componenti connesse ha dei tempi di esecuzione molto differenti in base alla rappresentazione ed alle implementazioni usate:

- **Rappresentazione tramite liste concatenate:** creando n insiemi con *make_set* impiego un tempo $\theta(n)$, se eseguo la union andando ad accodare la lista contente il maggior numero di elementi ho un tempo medio per chiamata dell'operazione di *union* di $\theta(n)$, perciò in totale per una certa sequenza delle operazioni per gli insiemi disgiunti impiego un tempo $\theta(n^2)$.
- **Rappresentazione tramite liste concatenate con euristica pesata del cammino:** si ha che una sequenza di operazioni *make_set*, *union* e *find* impiega un tempo $O(m + n \cdot lg\ n)$, dove n è il numero di operazioni di *make_set* e m è il numero totale di operazioni eseguite.
- **Rappresentazione tramite foresta di insiemi disgiunti con compressione del cammino:** si ha che una sequenza di m operazioni di *make_set*,*union* e *find* richiedono un tempo $\theta(n + f \cdot (1 + log_{2+f/n}\ n))$, dove n è il numero di operazioni *make_set* e f il numero di *find*.

3 Prestazione attese

Ci si aspetta dunque che la rappresentazione tramite *lista concatenata* risulti quella meno efficiente nell'eseguire l'algoritmo per il calcolo delle componenti connesse a causa del suo costo quadratico, mentre per le implementazioni tramite *lista concatenata con euristica dell'unione pesata* e *foresta di insiemi disgiunti* il tempo di esecuzione dipenderà dal numero di operazioni eseguite, per tale motivo si dovrà osservare come il numero di operazioni cresce alla dimensione del problema per avere un'idea di come il costo di queste due rappresentazioni cresca.

4 Descrizione esperimenti e codice

Il codice è stato diviso in 3 moduli Python:

- **linked_list.py** contiene tutte le classi e le funzioni relative alla rappresentazione mediante lista concatenata, sia con che senza euristica dell'unione pesata.
- **set_forest.py** contiene classi e funzioni relative alla rappresentazione mediante foresta di alberi disgiunti.
- **main.py** contiene il codice che inizializza le variabili dell'esperimento, esegue le simulazioni e riporta i risultati dell'esperimento.

4.1 Codice lista concatenata (linked_list)

4.1.1 Classe Element

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

Figura 2: Codice della classe Element per lista concatenata

Un elemento dell'insieme viene rappresentato dalla classe *Element* (Figura 2) il quale contiene un attributo *name* che funge da rappresentante dell'elemento, un attributo *set* che punta alla lista concatenata rappresentante l'insieme a cui l'elemento appartiene e un attributo *next* che punta al successivo elemento della lista. Questa classe verrà usata sia nella rappresentazione con euristica dell'unione pesata sia in quella senza euristica.

4.1.2 Classe LinkedList

```
8   class LinkedList:
9       def __init__(self, e):
10          self.head = e
11          self.tail = e
```

Figura 3: Codice della classe LinkedList

Questa classe rappresenta un insieme (Figura 3), ha come attributi *head* che punta al rappresentante dell'insieme e *tail* che invece punta all'ultimo elemento della lista.

4.1.3 Funzione make_set()

```
14  def make_set(e):
15      s = LinkedList(e)
16      e.set = s
17      e.next = None
```

Figura 4: Codice della funzione make_set() per lista concatenata

La funzione in Figura 4 permette di creare una nuova lista: prende come parametro l'elemento con cui creare l'insieme e ne modifica il puntatore a *set* in modo tale che punti al nuovo insieme creato, inoltre poiché l'insieme contiene un solo elemento imposta il puntatore *next* a None.

```
20  def find(e):
21      return e.set.head.name
```

Figura 5: Codice della funzione find() per lista concatenata

4.1.4 Funzione find()

Permette di ottenere il *name* del rappresentante dell'insieme a cui appartiene l'elemento passato come parametro (Figura 5).
Questa funzione verrà usata per la rappresentazione tramite lista concatenata sia con che senza euristica dell'unione pesata.

4.1.5 Funzione union()

```
24  def union(A, B):
25      if A is not B:
26          A.tail.next = B.head
27          A.tail = B.tail
28
29          i = B.head
30          while i is not None:
31              i.set = A
32              i = i.next
```

Figura 6: Codice della funzione union()

Nella funzione (Figura 6) per prima cosa controllo che i due insiemi dati come parametri siano diversi, in tal caso aggiorna il puntatore dell'ultimo elemento dell'insieme A in modo che punti al primo elemento dell'insieme B, aggiorna il puntatore *tail* dell'insieme A facendolo puntare all'ultimo elemento dell'insieme B. Dopodiché aggiorno tutti i puntatori *set* degli elementi della lista B in modo che puntino al nuovo insieme.

4.1.6 Classe HeuristicsLinkedList

Questa classe (Figura 7) è per lo più identica alla classe LinkedList, l'unica differenza consiste nell'attributo *length* che registra il numero di elementi

```
35  class HeuristicsLinkedList:
36      def __init__(self, e):
37          self.head = e
38          self.tail = e
39          self.length = 1
```

Figura 7: Codice della classe HeuristicsLinkedList

all'interno dell'insieme.

4.1.7 Funzione heuristics_make_set()

```
42  def heuristics_make_set(e):
43      s = HeuristicsLinkedList(e)
44      e.set = s
45      e.next = None
```

Figura 8: Codice della funzione heuristics_make_set()

Come l'equivalente per la rappresentazione senza euristica, questa funzione (Figura 8) crea un nuovo oggetto di tipo HeuristicsLinkedList e imposta il puntatore *set* dell'elemento passato come parametro alla funzione.

4.1.8 Funzione heursistics_union()

```
48  def heuristics_union(A, B):
49      if A is not B:
50          if A.length >= B.length:
51              A.length = A.length + B.length
52              union(A, B)
53          else:
54              B.length = B.length + A.length
55              union(B, A)
```

Figura 9: Codice della funzione heuristics_union()

Come si può osservare dal codice (Figura 9), la funzione per prima cosa controlla che i due insiemi non siano uguali, dopodiché controlla quale dei due sia quello contenente meno elementi tramite l'attributo *length* e va ad

eseguire la funzione *union* definita prima per unire le due liste, andando precedentemente ad aggiornare la lunghezza delle liste.

4.2 Codice foresta di insiemi disgiunti (set_forest)

4.2.1 Classe Element

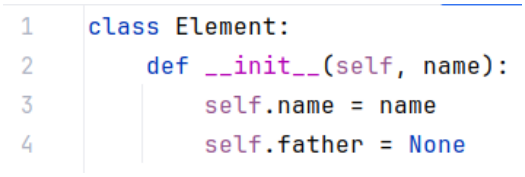

Figura 10: Codice della classe Element per foresta di insiemi disgiunti

Nella rappresentazione tramite foresta di insiemi disgiunti ciascun elemento (Figura 10) contiene un attributo *name* che svolge il ruolo di rappresentante dell'elemento ed un puntatore *father* che punta al nodo padre dell'elemento.

4.2.2 Funzione make_set()

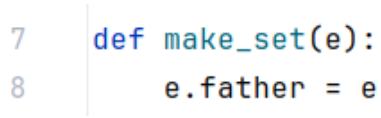

Figura 11: Codice della funzione make_set() per foresta di insiemi disgiunti

La funzione (Figura 11) imposta il puntatore *father* a se stesso, in modo da rendere l'elemento passato come parametro radice di un insieme contente solo se stesso.

4.2.3 Funzione find()

La funzione find() per foresta di insiemi disgiunti (Figura 12) ritorna un puntatore alla radice dell'elemento, per fare ciò va ad eseguire una chiamata ricorsiva sul padre dell'elemento passato come parametro fino a raggiungere la radice.

```
15  def find(e):
16      if e.father is not e:
17          e.father = find(e.father)
18      return e.father
```

Figura 12: Codice della funzione find() per foreste di insiemi disgiunti

Nell'eseguire le chiamate ricorsive va anche ad aggiornare il puntatore *father* degli elementi nel cammino di ricerca in modo che abbiano come nodo padre direttamente la radice (in tal modo si implementa la compressione del cammino).

4.2.4 Funzione union()

```
11  def union(A, B):
12      find(B).father = find(A)
```

Figura 13: Codice della funzione union() per foresta di insiemi disgiunti

Questa funzione (Figura 13) va ad unire i due insiemi passati come parametri andando a richiamare la funzione *find()* per trovare la radice dell'insieme B, dopodiché aggiusta quest'ultima in modo che il proprio puntatore *father* punti alla radice dell'altro insieme A.

4.3 Codice di inizializzazione e simulazione dell'esperimento (main)

4.3.1 Import

Nella figura 14 si riportano i moduli importati per l'esecuzione dell'esperimento, più precisamente:

- **linked_list:** contiene le funzioni e le classi per la rappresentazione tramite lista concatenata, sia con che senza euristica dell'unione pesata.
- **set_forest:** contiene le classi e le funzioni per la rappresentazione con foresta di insiemi disgiunti e compressione del cammino.

```python
import linked_list as ll
import set_forest as sf

import numpy
from timeit import default_timer as timer
import matplotlib.pyplot as plt
import matplotlib.patches as mpatch
```

Figura 14: Moduli importati

- **numpy:** modulo pyhton usato per la gestione dei vettori e delle matrici usate nell'esperimento, oltre alla generazione randomica degli elementi.
- **timeit:** modulo usato per calcolare i tempi di esecuzione delle varie funzioni, più nello specifico verrà usata la funzione *default_timer*.
- **matplotlib.pyplot:** modulo usato per creare e salvare su file i vari grafici relativi ai dati raccolti.
- **matplotlib.patches:** modulo python che permette di aggiungere maggiore chiarezza ai grafici generati tramite l'aggiunta di etichette.

4.3.2 Funzione make_graph

```python
def make_graph(N):
    graph = numpy.random.randint(low=0, high=2, size=(N, N))

    for i in range(N):
        graph[i][i] = 0

    graph = graph + graph.T
    graph = graph > 1
    return graph
```

Figura 15: Codice della funzione che genera il grafo

In figura 15 è mostrato il codice che crea una matrice di adiacenza per rappresentare il grafo inserendovi randomicamente gli archi del grafo, successivamente fa alcuni aggiustamenti alla matrice (elimina gli archi sulla diagonale e rende la matrice simmetrica rispetto alla diagonale principale).
Essa prende come parametro in ingresso il numero di vertici del grafo.

```python
def connect_components_ll(graph, vertex, counter):
    for v in vertex:
        ll.make_set(v)
        counter[0] += 1

    for i in range(graph.shape[0]):
        for j in range(i, graph.shape[0]):
            if graph[i][j]:
                counter[1] += 2
                if ll.find(vertex[i]) != ll.find(vertex[j]):
                    ll.union(vertex[j].set, vertex[i].set)
                    counter[2] += 1
```

Figura 16: Codice della funzione per il calcolo delle componenti connesse con rappresentazione tramite lista concatenata

```python
def connect_components_llh(graph, vertex, counter):
    for v in vertex:
        ll.heuristics_make_set(v)
        counter[0] += 1

    for i in range(graph.shape[0]):
        for j in range(i, graph.shape[0]):
            if graph[i][j]:
                counter[1] += 2
                if ll.find(vertex[i]) != ll.find(vertex[j]):
                    ll.heuristics_union(vertex[i].set, vertex[j].set)
                    counter[2] += 1
```

Figura 17: Codice della funzione per il calcolo delle componenti connesse con rappresentazione tramite lista concatenata con euristica dell'unione pesata

4.3.3 Funzioni connect_componets

Nelle figure 16,17 e 18 è riportato il codice delle funzioni per il calcolo delle componenti connesse con le tre rappresentazioni usate nell'esperimento. Prendono come parametri *graph*, che è un vettore contenente i vertici del grafo, *vertex* che è la matrice di adiacenza dove sono salvati i vari archi del grafo e *counter*, che è un vettore nel quale viene registrato il numero di operazioni eseguite.
Per prima cosa creano per ogni vertice un insieme tramite la corrispettiva funzione *make_set*, dopodiché esaminano ogni cella della matrice di adiacenza e se l'arco tra i due vertici esiste controllano che i due vertici appartengano a due insiemi diversi tramite *find*: in caso positivo si procedono a fare la *union* dei due insiemi.

```python
def connect_components_forest(graph, vertex, counter):
    for v in vertex:
        sf.make_set(v)
        counter[0] += 1

    for i in range(graph.shape[0]):
        for j in range(i, graph.shape[0]):
            if graph[i][j]:
                counter[1] += 2
                if sf.find(vertex[i]).name != sf.find(vertex[j]).name:
                    sf.union(vertex[i], vertex[j])
                    counter[2] += 1
```

Figura 18: Codice della funzione per il calcolo delle componenti connesse con rappresentazione tramite foresta di insiemi disgiunti

```python
def main():
    max_dim = 510
    min_dim = 10
    range_dim = max_dim - min_dim
    num_iter_for_dim = 100

    ll_time = numpy.zeros(range_dim)
    llh_time = ll_time.copy()
    sf_time = ll_time.copy()

    ll_tot_time = 0
    llh_tot_time = 0
    sf_tot_time = 0

    ll_operation_counter = numpy.zeros((3, range_dim))  # 0 make_set, 1 find, 2 union
    llh_operation_counter = numpy.zeros((3, range_dim))
    sf_operation_counter = numpy.zeros((3, range_dim))
```

Figura 19: Inizializzazione dell'esperimento

4.3.4 Inizializzazione dell'esperimento

In questa sezione del modulo dell'esperimento (Figura 19) si inizializzano tutte quelle variabili usate nel corso dell'esperimento, più precisamente:

- *range_dim*: contiene la dimensione dell'insieme dei vertici del grafo.
- *num_iter_for_dim*: contiene il numero di volte che si esegue l'esperimento per ogni dimensione dell'insieme dei vertici.
- *ll_time, llh_time, sf_time*: verranno usati per registrare i tempi di esecuzione dell'algoritmo per le componenti connesse delle rispettive metodologie di rappresentazione.
- *ll_tot_time, llh_tot_time, sf_tot_time*: permettono di registrare i tempi totali di esecuzione dell'algoritmo per le componenti connesse delle rispettive metodologie.

- *ll_operation_counter, llh_operation_counter, sf_operation_counter*: registrano il numero di operazioni per ciascun tipo eseguite durante il calcolo delle componenti connesse.

4.3.5 Cicli di simulazione dell'esperimento

```
81      for i in range(range_dim):
82          print(">> inizio iter ", i)
83          for j in range(num_iter_for_dim):
84
85              ll_vertex = [ll.Element(k) for k in range(i + min_dim)]
86              llh_vertex = ll_vertex.copy()
87              fi_vertex = [sf.Element(k) for k in range(i + min_dim)]
88
89              graph = make_graph(i + min_dim)
90
91              # run all connect-components algorithms ad record their time
92              ll_begin = timer()
93              connect_components_ll(graph, ll_vertex, ll_operation_counter[:, i])
94              ll_time[i] += timer() - ll_begin
95              ll_tot_time += ll_time[i]
96
97              llh_begin = timer()
98              connect_components_llh(graph, llh_vertex, llh_operation_counter[:, i])
99              llh_time[i] += timer() - llh_begin
100             llh_tot_time += llh_time[i]
101
102             sf_begin = timer()
103             connect_components_forest(graph, fi_vertex, sf_operation_counter[:, i])
104             sf_time[i] += timer() - sf_begin
105             sf_tot_time += sf_time[i]
106
107         ll_time[i] /= num_iter_for_dim
108         llh_time[i] /= num_iter_for_dim
109         sf_time[i] /= num_iter_for_dim
110
111         ll_operation_counter[0, i] /= num_iter_for_dim
112         ll_operation_counter[1, i] /= num_iter_for_dim
113         ll_operation_counter[2, i] /= num_iter_for_dim
114         llh_operation_counter[0, i] /= num_iter_for_dim
115         llh_operation_counter[1, i] /= num_iter_for_dim
116         llh_operation_counter[2, i] /= num_iter_for_dim
117         sf_operation_counter[0, i] /= num_iter_for_dim
118         sf_operation_counter[1, i] /= num_iter_for_dim
119         sf_operation_counter[2, i] /= num_iter_for_dim
120
121     ll_avg_time = ll_tot_time/(num_iter_for_dim * range_dim)
122     llh_avg_time = llh_tot_time/(num_iter_for_dim * range_dim)
123     sf_avg_time = sf_tot_time/(num_iter_for_dim * range_dim)
```

Figura 20: Cicli esperimento

Come mostrato in figura 20, la simulazione è composta da due cicli: quello più esterno va ad incrementare la dimensione del problema, aumentando il numero di vertici, mentre quello più interno ripete l'esperimento tante volte quante specificate nella variabile *num_iter_for_dim*.

All'interno del secondo for vengono generati dei nuovi vettori contenenti i vertici *ll_vertex, llh_vertex, sf_vertex* e un nuovo grafo *graph* tramite la funzione *make_set* definita sopra.
Dopodiché per ogni rappresentazione si esegue il corrispettivo algoritmo per il calcolo delle componenti connesse e si registra il tempo che impiega ad eseguire l'algoritmo con il corrispettivo vettore.
Infine, fuori dal ciclo più interno ma sempre dentro a quello più esterno, si vanno ad aggiustare le variabili dividendo per il numero di iterazioni per quella data dimensione.
All'esterno dei cicli di simulazione vengono calcolati i tempi medi per ogni dimensione e salvati nelle variabili *ll_avg_time, llh_avg_time, sf_avg_time*.

4.3.6 Output risultati

```
125     # printing results
126     print("- lista concatenata")
127     ll_tot_time_minutes = int(ll_tot_time/60)
128     print("tempi totale di esecuzione per ", range_dim, " iterazioni: ", ll_tot_time_minutes, " minuti e ",
129           f'{(ll_tot_time - ll_tot_time_minutes*60):.4f}', " secondi ")
130     print("tempo medio per ciascuna iterazione: ", f'{ll_avg_time:.4f}', "secondi")
131
132     print("- lista concatenata con unione pesata")
133     llh_tot_time_minutes = int(llh_tot_time/60)
134     print("tempi totale di esecuzione per ", range_dim, " iterazioni: ", llh_tot_time_minutes, " minuti e ",
135           f'{(llh_tot_time - llh_tot_time_minutes * 60):.4f}', " secondi ")
136     print("tempo medio per ciascuna iterazione: ", f'{llh_avg_time:.4f}', "secondi")
137
138     print("- foreste di alberi")
139     sf_tot_time_minutes = int(sf_tot_time/60)
140     print("tempi totale di esecuzione per ", range_dim, " iterazioni: ", sf_tot_time_minutes, " minuti e ",
141           f'{(sf_tot_time -sf_tot_time_minutes * 60):.4f}', " secondi ")
142     print("tempo medio per ciascuna iterazione: ", f'{sf_avg_time:.4f}', "secondi")
143
144     # display results with graphs
145
146     x_axis = [i for i in range(min_dim, max_dim)]
147
148     ll_patch = mpatch.Patch(color='orange', label='linked list')
149     llh_patch = mpatch.Patch(color='lime', label='linked list heuristics')
150     sf_patch = mpatch.Patch(color='cyan', label='set forest')
151
152     img_res, plot_res = plt.subplots()
153     plot_res.plot(x_axis, ll_time, color="orange")
154     plot_res.plot(x_axis, llh_time, color="lime")
155     plot_res.plot(x_axis, sf_time, color="cyan")
156     plot_res.set_title("Confronto calcolo algoritmi componenti connesse")
157     plot_res.set_ylabel("tempo impiegato (secondi)")
158     plot_res.set_xlabel("numero di vertici nel grafo")
159     plot_res.legend(handles=[ll_patch, llh_patch, sf_patch])
160     plt.savefig("esercizio1/images/results/result.png")
```

Figura 21: Codice per l'output

Infine i tempi totali e medi di ciascuna rappresentazioni vengono stampati, inoltre si vanno a creare e salvare su file dei grafici che rappresentino in

modo grafico l'andamento dei tempi di esecuzione e del numero di operazioni eseguire all'aumentare del numero di vertici (Figura 21).

5 Risultati sperimentali

Di seguito si mostrano i risultati sui tempi totali e medi ottenuti dall'esperimento:
- lista concatenata
tempo totale di esecuzione per 500 iterazioni: 518 minuti e 58.1180 secondi
tempo medio per ciascuna iterazione: 0.6228 secondi
- lista concatenata con unione pesata
tempo totale di esecuzione per 500 iterazioni: 495 minuti e 25.4427 secondi
tempo medio per ciascuna iterazione: 0.5945 secondi
- foreste di alberi
tempo totale di esecuzione per 500 iterazioni: 548 minuti e 4.3195 secondi
tempo medio per ciascuna iterazione: 0.6577 secondi

ed i risultati grafici [1], mostrando anche una versione normalizzata per il grafico del numero di operazioni, in modo da mettere meglio in evidenza l'andamento delle operazioni di *find* e *union*:

6 Deduzioni finali

Osservando i vari grafici ci si rende conto di come l'andamento sia determinato principalmente dal numero di *find* che vengono eseguiti: infatti risulta che al crescere del numero di vertici il numero di *find* cresca come $\theta(n^2)$, perciò come si può vedere sostituendo il numero di operazioni alle funzioni di costo delle varie rappresentazioni il tempo di esecuzione dell'algoritmo per il calcolo delle componenti connesse cresce più o meno in modo quadratico.
La rappresentazione che si dimostra più efficiente è quella tramite **lista concatenata con euristica dell'unione pesata**, la quale risulta migliore rispetto alla rappresentazione con **lista concatenata** dato il minor costo dell'operazione di *union*.

[1] si osserva come i grafici del conteggio delle operazioni risultino identici per ogni tipologia di rappresentazione, d'altronde il grafo su cui vengono eseguiti gli algoritmi è lo stesso.
I grafici del numero di operazioni di ciascuna tipologia sono riportati per completezza.

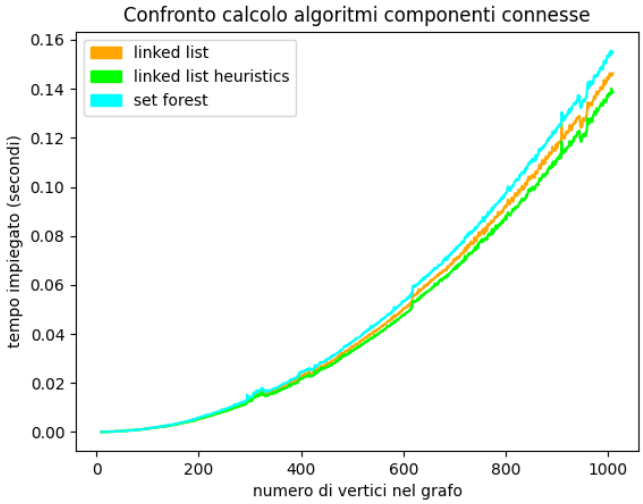

Figura 22: andamento del tempo di esecuzione delle 3 rappresentazioni

La rappresentazione mediante **foresta di insiemi disgiunti** invece risulta quella meno efficiente: a fronte dell'elevato numero di *find* che deve essere eseguito il costo di tale operazione resta troppo alto, perciò la sola euristica di compressione del cammino non risulta sufficiente per avere delle prestazioni migliori rispetto alla rappresentazione mediante lista concatenata.

Riferimenti bibliografici

[1] *Introduzione agli algoritmi e strutture dati / Thomas Cormen ... [et al.] ; edizione italiana a cura di Livio Colussi*. ita. 3. ed. Collana di istruzione scientifica. Serie di informatica. Milano [etc.: McGraw-Hill, 2010. ISBN: 9788838665158.

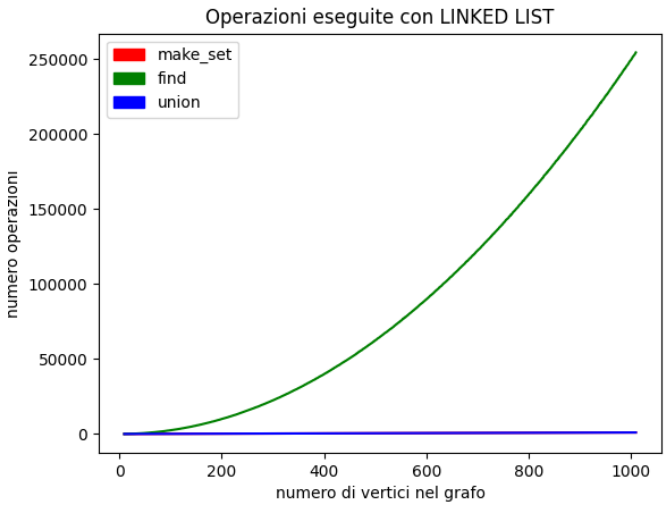

Figura 23: Grafico del numero di operazioni con la rappresentazione tramite lista concatenata

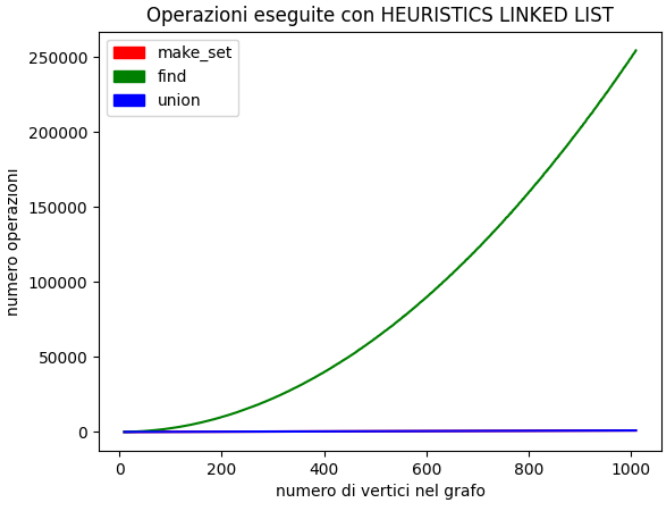

Figura 24: Grafico del numero di operazioni con la rappresentazione tramite lista concatenata ed euristica dell'unione pesata

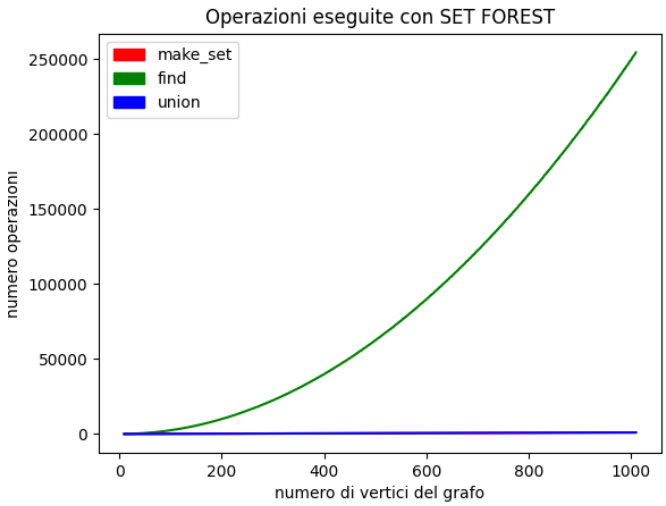

Figura 25: Grafico del numero di operazioni con la rappresentazione tramite foreste di insiemi

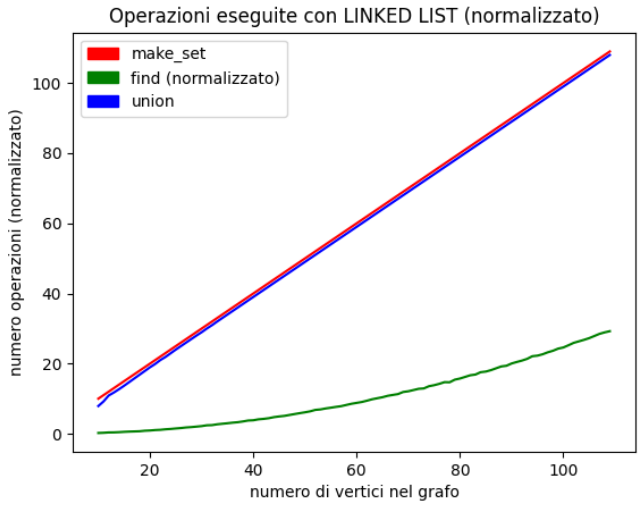

Figura 26: Grafico del numero di operazioni con la rappresentazione tramite lista concatenata (normalizzato)

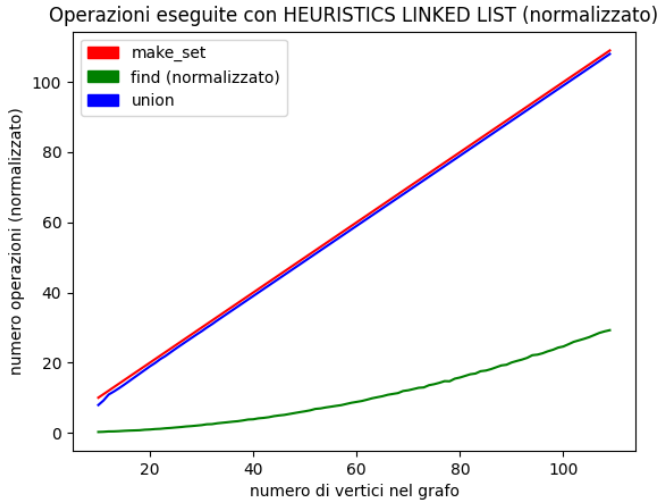

Figura 27: Grafico del numero di operazioni con la rappresentazione tramite lista concatenata ed euristica dell'unione pesata (normalizzato)

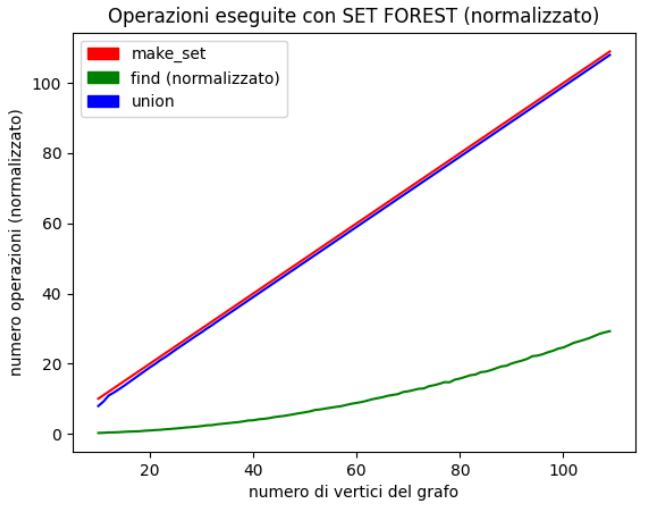

Figura 28: Grafico del numero di operazioni con la rappresentazione tramite foreste di insiemi (normalizzato)