# Relazione Progetto B.4: Me lo potevate dire Prim!

## Premessa

Quanto segue è necessario alla comprensione in toto della relazione. Verranno analizzati immediatamente gli “Array Dinamici” . La loro espansione geometrica è fondamentale nella maggior parte dei grafi, sia densi che sparsi, da relazionare. Cercheremo di comprenderne più precisamente il costo ammortizzato e quindi il relativo “Growth Factor” visto a lezione.

**Comparison of list data structures**

| | Linked list | Array | Dynamic array | Balanced tree | Random access list |
|---|---|---|---|---|---|
| Indexing | Θ(*n*) | Θ(1) | Θ(1) | Θ(log n) | Θ(log n) |
| Insert/delete at beginning | Θ(1) | N/A | Θ(*n*) | Θ(log n) | Θ(1) |
| Insert/delete at end | Θ(*n*) when last element is unknown; Θ(1) when last element is known | N/A | Θ(1) amortized | Θ(log *n*) | Θ(log *n*) updating |
| Insert/delete in middle | search time + Θ(1) [10][11][12] | N/A | Θ(*n*) | Θ(log *n*) | Θ(log *n*) updating |
| Wasted space (average) | Θ(*n*) | 0 | Θ(*n*)[13] | Θ(*n*) | Θ(*n*) |

*From Python 2.7 documentation*

Come si può notare dalla documentazione ufficiale di Python 2.7 gli array dinamici paragonati alle liste concatenate hanno un'indicizzazione più rapida (tempo costante VS tempo lineare) e tipicamente anche una più rapida iterazione; tuttavia, gli array dinamici richiedono un tempo lineare per inserire o cancellare su una locazione arbitraria, dal momento che tutti gli elementi seguenti devono essere spostati, mentre le liste concatenate possono farlo in tempo costante. Per una regione di memoria frammentata, può essere dispendioso o impossibile trovare uno spazio contiguo per un grosso array dinamico, mentre le liste concatenate non richiedono che l'intera struttura dati sia memorizzata in maniera contigua. Motivo per il quale è di grande aiuto una Linked_List , o meglio , una Double_Linked_List nell'implementazione di un grafo tramite Liste di Adiacenza.
Notiamo però che con grafi quali CNR-2000 ed EU-2005 (molto sparsi , con fattore di densità $\approx 0$) aumenta il Wasted Space , causando un relativo aumento della memory usage sul calcolatore.
Nel limitare l'uso della memoria , per avere come risultato un programma che si confa con architetture di qualsivoglia tipo , diversi sono stati gli interventi :

- Utilizzo di xrange() [python built_in_method] [ON]

  *Notevole incremento di stabilità ed efficienza. Di seguito una descrizione dalla Python

Documentation :

[This function is very similar to `range()`, but returns an *xrange object* instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of `xrange()` over `range()` is minimal (since `xrange()` still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range's elements are never used (such as when the loop is usually terminated with break).

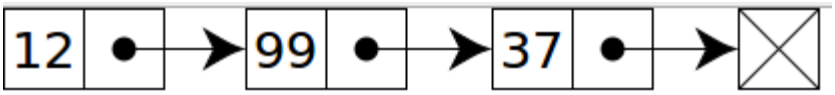
- Pulizia delle liste generate in xrange() ad ogni chiamata di insert / delete iterate. [ON]
- Tentativo di programmazione Multi_Processo e Multi_Thread e quindi relativo scontro con il Global Interpreter Lock (GIL python). [OFF]
- Garbage Collector. [OFF]

Non sono inclusi negli interventi fatti i " non_standard_python_method " visto che come da consegna non è possibile utilizzarli. Tuttavia , a puro scopo di soddisfazione personale , nell'ultimo capitolo della relazione "Possibili miglioramenti al codice .. " verranno trattati insieme all' estensione del codice a varianti Python che rendono possibili gli ultimi due punti.

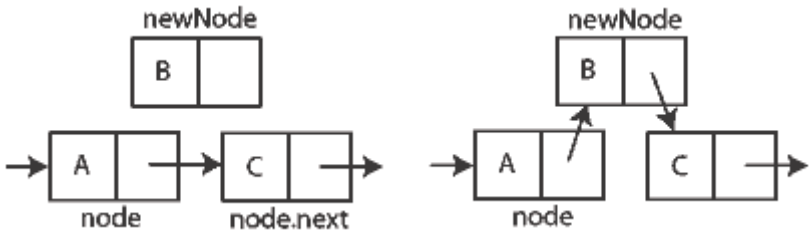
---

Cerchiamo ora di specificare quanto detto in modo teorico tramite degli esempi :

Creiamo una Linked_List utilizzando un collegamento per nodo. Questo collegamento punta al nodo successivo della lista o a un valore nullo o ad una lista vuota se è il valore finale.



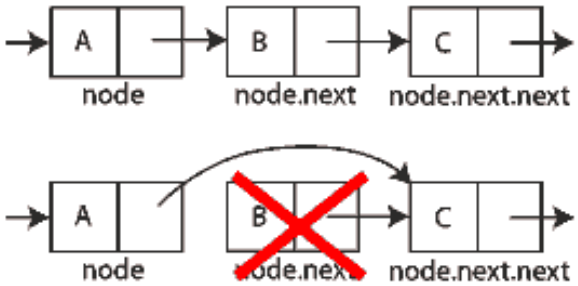

Facciamo ora un collegamento tra la teoria che si cela dietro le Linked_List e il codice sviluppato in python che ci permetterà di realizzare la struttura dati adatta alla creazione di una lista di adiacenza. Notiamo ad esempio come inserire un nuovo nodo nella Linked_List. Non è possibile inserire un nuovo nodo prima di un altro preesistente; è, invece, necessario posizionarlo tenendo conto di quello precedente :



Proseguiamo ora con diversi metodi python per realizzare tutto ciò utilizzando speciali funzioni quali le Classi :

```
class ListaCollegata:
    def __init__(self):
        self.first=None
        self.last=None

    def isEmpty(self):
        return (self.first==None)

    def getFirst(self):
        if self.first==None:
            return None
        else:
            return self.first.elem

    def getLast(self):
        if self.last==None:
            return None
        else:
            return self.last.elem

    def addAsLast(self,elem):
        rec=Record(elem)
        if self.first==None:
            self.first=self.last=rec
        else:
            self.last.next=rec
            self.last=rec

    def addAsFirst(self,elem):
        rec=Record(elem)
        if self.first==None:
            self.first=self.last=rec
        else:
            rec.next=self.first
            self.first=rec
```

Notiamo che l'aggiunta di una lista concatenata ad un'altra è allo stesso modo inefficiente, poiché bisogna attraversare l'intera lista per trovare la coda e, quindi, aggiungere la seconda lista a quest'ultima. Quindi se due liste linearmente concatenate sono di lunghezza n, l'aggiunta di una lista ha una complessità asintotica O(n).
Notiamo inoltre i metodi di rimozione dei nodi. Dimostriamo che è necessario tenere traccia dell'elemento precedente per trovare e rimuovere in un determinato punto :




A destra altri metodi implementati per le Linked_List.
In basso una tabella dalla documentazione python2.7 riguardo le operazioni su Linked_List e Vettori :

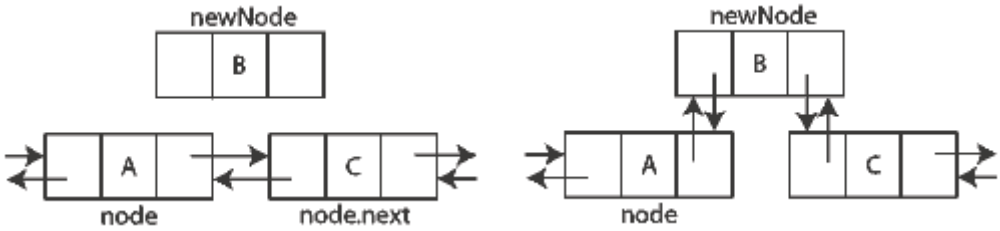
```
def popFirst(self):
    if self.first==None:
        return None
    else:
        res=self.first.elem
        self.first=self.first.next
        if self.first==None:
            self.last=None
        return res

def getFirstRecord(self):
    if self.first==None:
        return None
    else:
        return self.first

def getLastRecord(self):
    if self.first==None:
        return None
    else:
        return self.last
```

| | **Vettore** | **Lista concatenata** |
|---|---|---|
| Indirizzamento | O(1) | O(*n*) |
| Inserimento | O(*n*) | O(1) |
| Cancellazione | O(*n*) | O(1) |
| Persistenza | No | Si singolarmente |
| Località | Ottima | Pessima |

Una volta implementata una struttura di Linked_List possiamo collegarci al concetto più utilizzato nel nostro algoritmo che sono le Liste Doppiamente Collegate.
Nelle liste doppiamente collegate ogni nodo possiede due collegamenti: uno punta al nodo precedente o ad un “null” o ad una lista vuota se è il primo nodo; l'altro punta al nodo successivo o ad un "null” o una lista vuota se è il nodo finale.


Nelle liste doppiamente collegate dobbiamo gestire e aggiornare molti più puntatori che però ci permettono di fornire molte referenze in meno; ad esempio per analizzare elementi precedenti nella lista.

Esaminiamo ad esempio le operazioni di inserimento di un nuovo nodo in una DoubleLinked_List :




```
import LinkedList

class DoubleRecord(LinkedList.Record):
    def __init__(self,elem):
        LinkedList.Record.__init__(self,elem)
        self.prev=None

class ListaDoppiamenteCollegata(LinkedList.ListaCollegata):

    def addAsLast(self,elem):
        rec= DoubleRecord(elem)
        if self.first==None:
            self.first=self.last=rec
        else:
            rec.prev = self.last
            self.last.next = rec
            self.last = rec

    def addAsFirst(self,elem):
        rec=DoubleRecord(elem)
        if self.first==None:
            self.first=self.last=rec
        else:
            self.first.prev = rec
            rec.next = self.first
            self.first = rec
```

Inizializziamo la classe come derivata delle Linked_List impostando il relativo costruttore. Definiamo “prev” come puntatore all'elemento precedente e alla struttura della lista.

```
def popFirst(self):
    if self.first==None:
        return None
    else:
        res = self.first.elem
        self.first = self.first.next
        if self.first!=None:
            self.first.prev = None
        else:
            self.last = None
        return res


def popLast(self):
    if self.first==None:
        return None
    else:
        res = self.last.elem
        self.last = self.last.prev
        if self.last!=None:
            self.last.next = None
        else:
            self.first = None
        return res

def deleteRecord(self,rec):
    if rec==None:
        return  #restituisce None!
    if rec.prev!=None:
        rec.prev.next = rec.next
    else:
        self.first = rec.next
    if rec.next!=None:
        rec.next.prev = rec.prev
    else:
        self.last = rec.prev
```

## Analisi dell'algoritmo di Prim

In questo capitolo analizzeremo accuratamente l'esecuzione dell'algoritmo di Prim, considerando prima a livello teorico i vari costi dell'algoritmo in relazione alla struttura dati considerata e al grafo. In seguito in base ai file di profiling generati durante i vari test, verificheremo se sperimentalmente vengono rispettati gli andamenti asintotici; infine vedremo come la densità di un grafo giochi un ruolo fondamentale nell'analisi delle tempistiche, con vari esempi e considerazioni.

Il costo dell'algoritmo di Prim dipende fortemente dalle operazioni sulla coda con priorità. Infatti i tempi d'esecuzione dell'algoritmo variano in base alla struttura dati che implementa la coda:

- Con il Binomial Heap il tempo d'esecuzione è $O(m+n\log n)$ nel caso peggiore;
- Con il Binary Heap il tempo d'esecuzione è $O(m\log n)$ nel caso peggiore;
- Con il d-Heap il tempo d'esecuzione è $O(n\,d\log_d n+m\log_d n)$ ;

Queste tre affermazioni possono essere facilmente dimostrate considerando che durante l'esecuzione dell'algoritmo vengono eseguite n operazioni di *deleteMin*, n operazioni di *insert* e (m – n) operazioni di *decreaseKey*. I tempi d'esecuzione delle singole operazioni sono indicati nella tabella sottostante, grazie ai quali è possibile dedurre i tempi dell'algoritmo.

| | Binary Heap | d-Heap | Binomial Heap |
|---|---|---|---|
| insert( ) | $O(\log n)$ | $O(\log_d n)$ | $O(\log n)$ |
| findMin( ) | $O(1)$ | $O(1)$ | $O(\log n)$ |
| deleteMin( ) | $O(\log n)$ | $O(d\log_d n)$ | $O(\log n)$ |
| decreaseKey( ) | $O(\log n)$ | $O(\log_d n)$ | $O(\log n)$ |

*La tabella in figura mostra i tempi d'esecuzione delle funzioni implementate nelle strutture dati e utilizzate dall'algoritmo di Prim*

Inoltre scegliendo con il d-Heap un valore di d che bilancia il numero di nodi e il numero di archi (ad esempio $d=\lceil 2+m/n\rceil$ ) , il tempo d'esecuzione diventa pari a $O(m\log_{(2+m/n)} n)$ . Per $m=\Omega(n^{1+k})$ con $0<k\leq 1$ , $O(m\log_{(2+m/n)} n)$ diventa $O(m/k)$ . In pratica l'algoritmo di Prim aumenta la propria efficienza con l'aumentare della densità del grafo (il concetto di densità verrà analizzato più nel dettaglio in seguito).

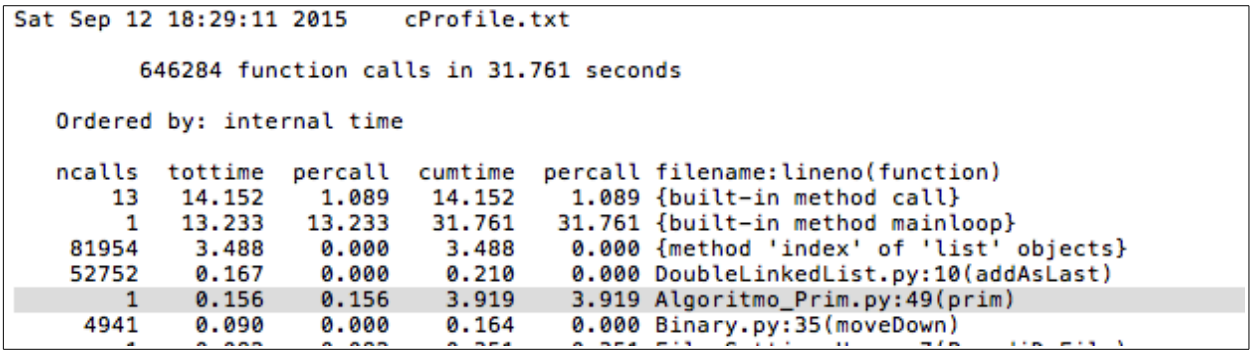
Analisi del grafo USpowerGrid

Nei file di profiling del codice viene indicato il tempo d'esecuzione della funzione **prim** presente nel file **Algoritmo_Prim.py**. Prendiamo in esame un grafo connesso come **grafo_USpowerGrid.txt**: dopo un'approfondita fase di test sperimentali è possibile constatare che vengono verificate le affermazioni precedenti.

Ricordiamo che il grafo in questione contiene 4941 nodi e 13188 archi.

Confrontiamo passo passo i vari profiling del codice (è stato scelto un profiling che meglio rappresenti i vari test effettuati, quindi avente tempistiche nella media) :

1. Binary Heap:



```
Sat Sep 12 18:29:11 2015    cProfile.txt

         646284 function calls in 31.761 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       13   14.152    1.089   14.152    1.089 {built-in method call}
        1   13.233   13.233   31.761   31.761 {built-in method mainloop}
    81954    3.488    0.000    3.488    0.000 {method 'index' of 'list' objects}
    52752    0.167    0.000    0.210    0.000 DoubleLinkedList.py:10(addAsLast)
        1    0.156    0.156    3.919    3.919 Algoritmo_Prim.py:49(prim)
     4941    0.090    0.000    0.164    0.000 Binary.py:35(moveDown)
```

*Screen del file di cProfile eseguendo prim sul grafo USpowerGrid con il Binary Heap*

E' possibile vedere come in questo screen il tempo di esecuzione dell'algoritmo di Prim sia di 3.919 secondi (in generale il tempo d'esecuzione per questa struttura dati si stabilizzava attorno ai 3.9 s).

2. 2-Heap:

Concettualmente parlando, un 2-Heap non è altro che un Binary Heap. A livello pratico però, le differenze di implementazione tra le due strutture dati portano a una differenza di tempistiche. Vediamo come i metodi delle strutture dati variano i loro tempi su uno stesso grafo:

- **insert()**: la insert non risente delle differenze di implementazione, infatti sperimentalmente si attesta che il *cumtime* con 4941 *ncalls* è di 0.016 s in entrambe le strutture dati. Ciò è essenzialmente dovuto al fatto che viene implementata esattamente allo stesso modo. In

realtà le cose sono lievemente diverse: la insert() chiama al proprio interno la funzione moveUp(), che nel d-heap impiega 0.001 s in più a causa della variabile self.d che nel Binary Heap non è presente poiché è fisso il numero 2, ma tale differenza viene poi approssimata nel complessivo delle tempistiche.

- **findMin()**: anche la findMin() non ha differenze tra le due strutture dati, essendo implementate alla stessa maniera. Hanno entrambe un cumtime di 0.003 s in 4941 *ncalls*.

- **deleteMin()**: la deleteMin() presenta tempistiche differenti tra le due strutture dati. La causa di ciò è da attribuirsi alla funzione moveDown(), che al suo interno richiama la funzione swap(), che rimane però invariata, e la minSon(), che invece è implementata diversamente. Nel d-Heap è necessario un ciclo for che scorre tra i figli del nodo per poter poi ritornare il figlio più piccolo, cosa non necessaria nel Binary Heap. Teoricamente i tempi sarebbero gli stessi ( $O(\log n)$ ), ma sperimentalmente il costrutto iterativo della d-Heap porta a un ritardo di 0.046 s in totale rispetto al Binary Heap. Anche i tempi delle due minSon() dimostrano quanto detto, impiegando un *cumtime* 0.053 s nel Binary Heap in 46107 *ncalls* e un *cumtime* di 0.097 s in 45517 *ncalls* nel 2-Heap.

- **decreaseKey()**: la decreaseKey() infine impiega lo stesso tempo, poiché è implementata nello stesso modo, chiamando la moveUp() che, come detto in precedenza, ha una lieve differenza di tempo che non viene considerata.

```
Sat Sep 12 18:29:45 2015    cProfile.txt

         644299 function calls (644281 primitive calls) in 29.349 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    57/46   14.932    0.262   14.932    0.325 {built-in method call}
        1   10.007   10.007   29.349   29.349 {built-in method mainloop}
    81786    3.472    0.000    3.472    0.000 {method 'index' of 'list' objects}
    52752    0.166    0.000    0.208    0.000 DoubleLinkedList.py:10(addAsLast)
        1    0.155    0.155    3.949    3.949 Algoritmo_Prim.py:49(prim)
    45517    0.097    0.000    0.097    0.000 D_Heap.py:18(minSon)
```

*Screen del file di cProfile eseguendo prim sul grafo USpowerGrid con il 2-Heap*

Anche sperimentalmente si può notare come l'esecuzione con il 2-Heap rispetto al Binary Heap richieda più tempo (benché la differenza sia molto contenuta).

3. 3-Heap:

Utilizzando il 3-Heap teoricamente si dovrebbe avere un costo pari a $O(3n\log_3 n+m\log_3 n)$ .

```
Sat Sep 12 18:30:45 2015    cProfile.txt

         606699 function calls (606681 primitive calls) in 26.958 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    57/46   14.489    0.254   14.489    0.315 {built-in method call}
        1    7.982    7.982   26.958   26.958 {built-in method mainloop}
    81966    3.581    0.000    3.581    0.000 {method 'index' of 'list' objects}
    52752    0.178    0.000    0.225    0.000 DoubleLinkedList.py:10(addAsLast)
        1    0.158    0.158    4.008    4.008 Algoritmo_Prim.py:49(prim)
        1    0.089    0.089    0.378    0.378 File_Setting_Up.py:7(PrendiDaFile)
    32974    0.088    0.000    0.088    0.000 D_Heap.py:18(minSon)
```

*Screen del file di cProfile eseguendo prim sul grafo USpowerGrid con il 3-Heap*

Effettivamente anche nella pratica i tempi utilizzando il 3-Heap sono superiori rispetto alle due strutture dati precedenti.

4. Binomial Heap:

Con il binomial heap i tempi d'esecuzione vengono ridotti: infatti dall'analisi asintotica sappiamo che il tempo totale per l'algoritmo è $O(m+n\log n)$ .

```
Sun Sep 13 12:58:02 2015    cProfile.txt

         575835 function calls in 62.168 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       13   46.863    3.605   46.863    3.605 {built-in method call}
        1   10.970   10.970   62.168   62.168 {built-in method mainloop}
    81992    3.517    0.000    3.517    0.000 {method 'index' of 'list' objects}
    52752    0.162    0.000    0.204    0.000 DoubleLinkedList.py:10(addAsLast)
        1    0.132    0.132    3.894    3.894 Algoritmo_Prim.py:49(prim)
        1    0.080    0.080    0.342    0.342 File_Setting_Up.py:7(PrendiDaFile)
```

*Screen del file di cProfile eseguendo prim sul grafo USpowerGrid con il Binomial Heap*

5. 5-Heap:

Utilizzando un valore di d pari a $\lceil 2+m/n \rceil$ (nel nostro caso 5), avremmo un tempo di esecuzione $O(m\log_5 n)$ .

```
Sat Sep 12 18:33:09 2015    cProfile.txt

         583938 function calls (583920 primitive calls) in 40.601 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    57/46   29.086    0.510   29.086    0.632 {built-in method call}
        1    7.303    7.303   40.601   40.601 {built-in method mainloop}
    82010    3.393    0.000    3.393    0.000 {method 'index' of 'list' objects}
    52752    0.163    0.000    0.206    0.000 DoubleLinkedList.py:10(addAsLast)
        1    0.147    0.147    3.759    3.759 Algoritmo_Prim.py:49(prim)
        1    0.085    0.085    0.350    0.350 File_Setting_Up.py:7(PrendiDaFile)
    25457    0.078    0.000    0.078    0.000 D_Heap.py:18(minSon)
```

*Screen del file di cProfile eseguendo prim sul grafo USpowerGrid con il 5-Heap*

Come mai allora il tempo d'esecuzione sperimentale è minore di tutti gli altri? La risposta si trova nella *densità* del grafo in oggetto.

Dato un grafo non orientato G = (V, E), la **densità** del grafo viene definita come il rapporto

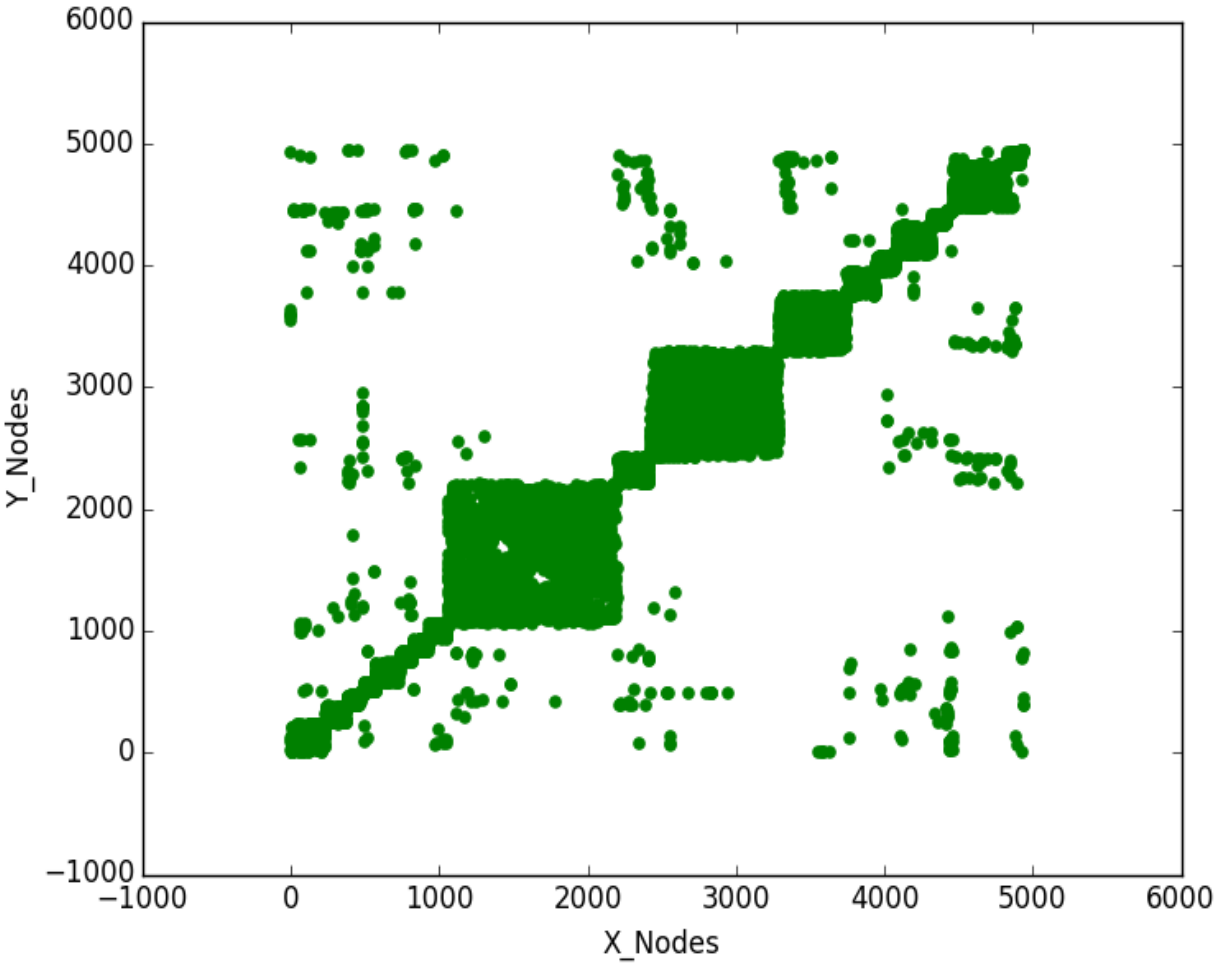
$$\Delta = \frac{2m}{n(n-1)}$$

dove n è il numero di nodi, m il numero di archi.

Essa misura la probabilità che una coppia di nodi sia adiacente (due nodi si dicono adiacenti se sono collegati tra loro tramite un solo arco), infatti la densità può assumere valori compresi tra 0 e 1. C'è una relazione tra la densità del grafo e la condizione $m=\Omega(n^{1+k})$ : all'aumentare della densità, a parità di nodi, aumenta il numero di archi del grafo.

Il grafo *USpowerGrid* è sufficientemente denso ( $\Delta \simeq 0.00108$ ) da garantire la condizione di m. Per k = 0.115 si ha $m=n^{1+k}$ , quindi il tempo d'esecuzione del 5-Heap per questo genere di grafo non è $O(m\log_5 n)$ , ma $O(m/k)$ , il che giustifica il tempo d'esecuzione minore rispetto agli altri.

Il grafico a lato mostra le connessioni del grafo *USpowerGrid*: è possibile così visualizzare la densità del grafo.




Per concludere l'analisi, presentiamo ora una tabella riassuntiva dei tempi sperimentali rilevati durante i test per ogni chiamata delle strutture dati:

| | Binary Heap | 2-Heap | 3-Heap | 5-Heap | Binomial Heap |
|---|---|---|---|---|---|
| insert( ) | 0.016 s in 4941 calls | 0.016 s in 4941 calls | 0.016 s in 4941 calls | 0.016 s in 4941 calls | 0.044 s in 4941 calls |
| findMin( ) | 0.003 s in 4941 calls | 0.003 s in 4941 calls | 0.003 s in 4941 calls | 0.003 s in 4941 calls | 0.012 s in 4942 calls |
| deleteMin( ) | 0.174 s in 4941 calls | 0.220 s in 4941 calls | 0.180 s in 4941 calls | 0.146 in 4941 calls | 0.127 s in 4941 calls |
| decreaseKey( ) | 0.058 s in 8853 calls | 0.058 s in 8759 calls | 0.043 s in 8849 calls | 0.031 in 8871 calls | 0.037 in 8862 calls |

## Analisi del grafo foldoc

Adesso analizzeremo un altro dei grafi testati durante la progettazione. Il grafo in questione è un grafo connesso, ma di dimensioni maggiori rispetto a USpowerGrid.

Il grafo è formato da 13356 nodi e 120238 archi.

Come già detto in precedenza, è stato scelto un profiling che meglio rappresenti i vari test effettuati.

1. Binary Heap:

```
Sun Sep 13 22:53:55 2015    cProfile.txt

         4126503 function calls in 85.175 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   508835   57.953    0.000   57.953    0.000 {method 'index' of 'list' objects}
       13   10.971    0.844   10.971    0.844 {built-in method call}
        1    8.536    8.536   85.175   85.175 {built-in method mainloop}
   480952    2.873    0.000    3.292    0.000 DoubleLinkedList.py:10(addAsLast)
        1    1.088    1.088   60.000   60.000 Algoritmo_Prim.py:49(prim)
        1    0.771    0.771    4.551    4.551 File_Setting_Up.py:7(PrendiDaFile)
```

*Screen del profiling del grafo foldoc utilizzando la coda con priorità Binary Heap.*

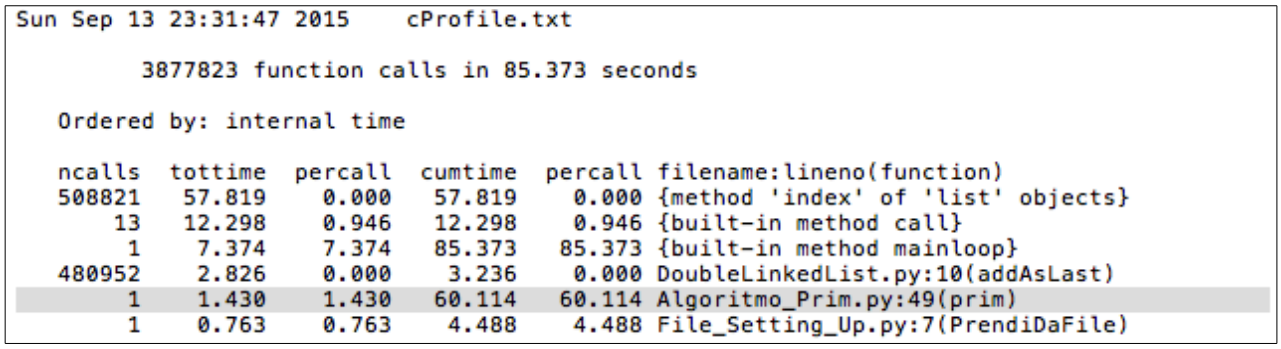
2. 2-Heap:

```
Sun Sep 13 22:55:30 2015    cProfile.txt

         4130591 function calls (4130573 primitive calls) in 90.790 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   509491   58.573    0.000   58.573    0.000 {method 'index' of 'list' objects}
    57/46   15.693    0.275   15.693    0.341 {built-in method call}
        1    8.249    8.249   90.790   90.790 {built-in method mainloop}
   480952    2.950    0.000    3.393    0.000 DoubleLinkedList.py:10(addAsLast)
        1    1.159    1.159   60.914   60.914 Algoritmo_Prim.py:49(prim)
        1    0.845    0.845    4.739    4.739 File_Setting_Up.py:7(PrendiDaFile)
```

*Screen del profiling del grafo foldoc utilizzando la coda con priorità 2-Heap.*

Avendo già analizzato in precedenza i vari metodi di entrambe le strutture, qui ci limitiamo a constatare che è verificato quanto sopra enunciato, poiché il 2-Heap impiega un tempo leggermente superiore rispetto al Binary Heap, dovuto alla differente implementazione.

3. Binomial Heap:



```
Sun Sep 13 23:31:47 2015    cProfile.txt

         3877823 function calls in 85.373 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   508821   57.819    0.000   57.819    0.000 {method 'index' of 'list' objects}
       13   12.298    0.946   12.298    0.946 {built-in method call}
        1    7.374    7.374   85.373   85.373 {built-in method mainloop}
   480952    2.826    0.000    3.236    0.000 DoubleLinkedList.py:10(addAsLast)
        1    1.430    1.430   60.114   60.114 Algoritmo_Prim.py:49(prim)
        1    0.763    0.763    4.488    4.488 File_Setting_Up.py:7(PrendiDaFile)
```

*Screen del profiling del grafo foldoc utilizzando la coda con priorità Binomial Heap.*

Questa volta possiamo notare come il tempo d'esecuzione con il Binomial Heap sia leggermente superiore rispetto al Binary Heap (al contrario del grafo USpowerGrid in cui l'esecuzione con il Binomial Heap era più veloce). Ciò potrebbe dipendere dalla dimensione e dalla densità del grafo stesso: foldoc è un grafo molto più grande e più denso rispetto al grafo USpowerGrid, il che potrebbe rallentare la costruzione dell'albero binomiale.
La costruzione di un albero binomiale sperimentalmente può risultare più faticosa rispetto alla costruzione di un Binary Heap. Il Binomial Heap quando inserisce un nodo nell'Heap chiama la procedura *ristruttura()*, che per grafi grandi può portare a un leggero rallentamento nell'esecuzione.

Facciamo un'analisi veloce delle funzioni chiamate dalle due strutture dati:

- **insert()**: il test sperimentale conferma quanto detto sopra. La insert del Binomial Heap ha un *cumtime* di 0.134 s in 13356 *ncalls*, mentre quella del Binary Heap ha un *cumtime* di 0.050 s in 13356 *ncalls*.

- **findMin()**: la findMin() presenta differenze anche per quanto riguarda gli andamenti asintotici. Come scritto nella tabella precedente, la findMin() richiede per il Binomial Heap un tempo O(logn), mentre per il Binary Heap richiede O(1), poiché nel Binomial Heap deve scorrere (attraverso un costrutto iterativo *while*) lungo le radici di tutti gli alberi binomiali. Sperimentalmente si ha infatti una differenza di tempistiche: nel Binomial Heap la findMin() ha un *cumtime* di 0.041 s in 13357 *ncalls*, mentre nel Binary Heap ha un *cumtime* di 0.010 s in 13356 *ncalls*.

- **deleteMin()**: asintoticamente la due deleteMin() richiedono lo stesso tempo $O(\log n)$ . Sperimentalmente si nota che la costante moltiplicativa d (che nel Binary Heap è pari a 2) precedente al $\log_d n$ porta a una differenza nel *cumtime* di 0.177 s in 13356 *ncalls*. I *cumtime* delle due chiamate sono di 0.426 s in 13356 *ncalls* per il Binomial Heap e di 0.603 s in 13356 *ncalls* per il Binary Heap.

- **decreaseKey()**: asintoticamente le due decreaseKey() hanno lo stesso andamento $O(\log n)$ . Sperimentalmente però, a causa delle differenze strutturali della classe Binomial da noi creata, si ha una leggera differenza nei tempi, dovuta alle continue inizializzazioni del costruttore della classe ItemRef, necessaria a mantenere un riferimento

dell'albero considerato. Usando la funzione time.time() potremmo appianare questa differenza (che consta di circa 0.76 s) escludendo le chiamate precedentemente specificate.

4. 11-Heap:

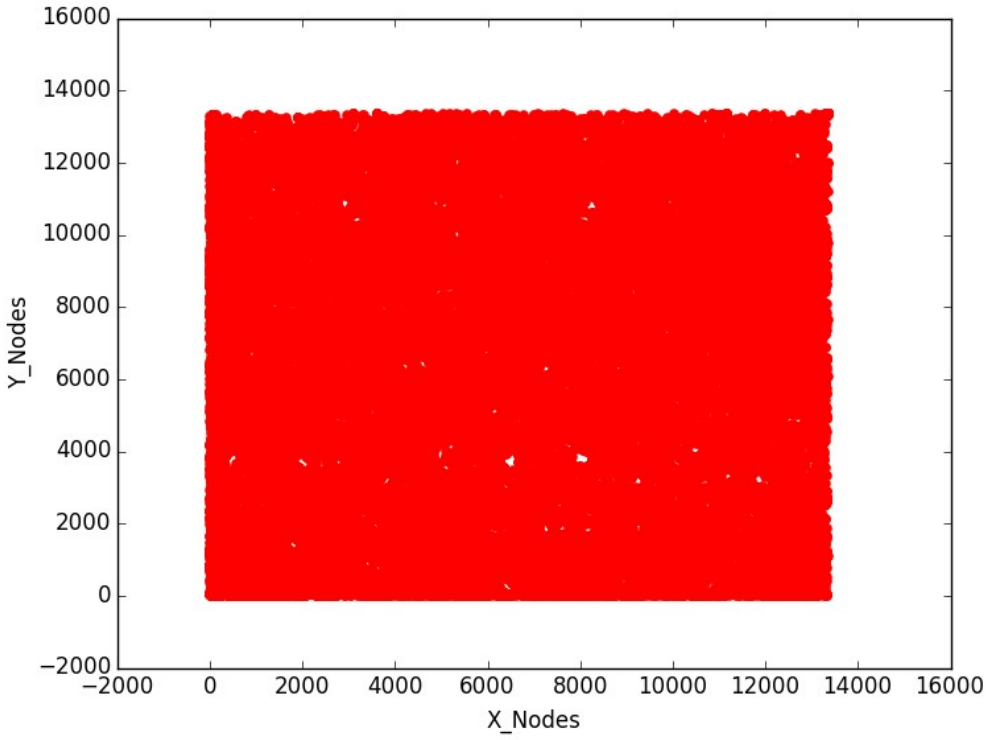
Il valore d = 11 è pari al risultato di $\lceil 2+m/n \rceil$ , che quindi garantisce (grazie alla densità del grafo) un andamento asintotico $O(m/k)$ .

```
Sun Sep 13 23:21:48 2015    cProfile.txt

         3845306 function calls (3845288 primitive calls) in 110.296 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   508975   58.096    0.000   58.096    0.000 {method 'index' of 'list' objects}
    57/46   34.008    0.597   34.008    0.739 {built-in method call}
        1   10.642   10.642  110.296  110.296 {built-in method mainloop}
   480952    2.882    0.000    3.286    0.000 DoubleLinkedList.py:10(addAsLast)
        1    1.116    1.116   59.979   59.979 Algoritmo_Prim.py:49(prim)
        1    0.757    0.757    4.528    4.528 File_Setting_Up.py:7(PrendiDaFile)
```

*Screen del profiling del grafo foldoc utilizzando la coda con priorità 11-Heap.*

Anche sperimentalmente l'esecuzione dell'algoritmo di Prim utilizzando la coda con priorità 11-Heap impiega meno tempo rispetto alle altre.

L'immagine qui sotto rappresenta le connessioni del grafo foldoc. Il grafo risulta essere abbastanza denso, in quanto presenta un gran numero di archi. Il suo valore di densità è $\Delta \simeq 0.00135$ ed è sufficiente per garantire la condizione $m=\Omega(n^{1+k})$ con $0<k\leq 1$ .



Le seguenti tabelle mostrano i dati relativi ai test eseguiti su diversi grafi con diverse strutture dati. Onde evitare di dilungarci troppo, abbiamo deciso di proporre solo alcuni dei grafi testati in esame, allegando il resto delle esecuzioni e dei test insieme alla relazione.

| | grafo_USpowerGrid | grafo_foldoc | grafo_DutchElite |
|---|---|---|---|
| n° nodi, n° archi | 4941, 13188 | 13356, 120238 | 4747, 5220 |
| prim() | 3.919 s in 1 call | 60.000 s in 1 call | 1.587 s in 1 call |
| insert() | 0.016 s in 4941 calls | 0.042 s in 13356 calls | 0.012 s in 3620 calls |
| findMin() | 0.003 s in 4941 calls | 0.010 s in 13356 calls | 0.002 s in 3620 calls |
| deleteMin() | 0.174 s in 4941 calls | 0.603 s in 13356 calls | 0.128 s in 3620 calls |
| decreaseKey() | 0.058 s in 8843 calls | 0.216 s in 40671 calls | 0.031 s in 4069 calls |

*Tabella dei tempi relativa al Binary Heap.*

| | grafo_USpowerGrid | grafo_foldoc | grafo_DutchElite |
|---|---|---|---|
| n° nodi, n° archi | 4941, 13188 | 13356, 120238 | 4747, 5220 |
| prim() | 3.949 s in 1 call | 60.914 s in 1 call | 1.679 s in 1 call |
| insert() | 0.016 s in 4941 calls | 0.043 s in 13356 calls | 0.011 s in 3620 calls |
| findMin() | 0.003 s in 4941 calls | 0.010 s in 13356 calls | 0.002 s in 3620 calls |
| deleteMin() | 0.220 s in 4941 calls | 0.814 s in 13356 calls | 0.174 s in 3620 calls |
| decreaseKey() | 0.058 s in 8759 calls | 0.220 s in 40999 calls | 0.034 s in 4078 calls |

*Tabella dei tempi relativa al 2-Heap*

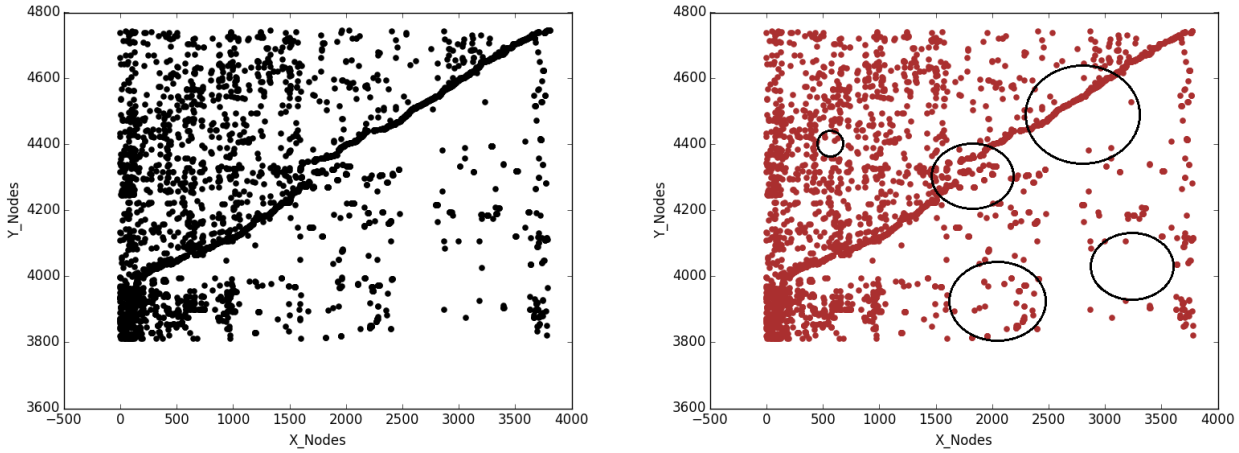
| | grafo_USpowerGrid | grafo_foldoc | grafo_DutchElite |
|---|---|---|---|
| n° nodi, n° archi | 4941, 13188 | 13356, 120238 | 4747, 5220 |
| prim() | 3.894 s in 1 call | 60.114 s in 1 call | 1.581 s in 1 call |
| insert() | 0.044 s in 4941 calls | 0.134 s in 13356 calls | 0.034 s in 3620 calls |
| findMin() | 0.012 s in 4942 calls | 0.041 s in 13357 calls | 0.009 s in 3621 calls |
| deleteMin() | 0.127 s in 4941 calls | 0.426 s in 13356 calls | 0.094 s in 3620 calls |
| decreaseKey() | 0.065 s in 8862 calls | 0.292 s in 40664 calls | 0.042 s in 4055 calls |

*Tabella dei tempi relativa al Binomial Heap*

Ora andremo ad esaminare nel complesso la densità degli altri grafi utilizzati nella fase sperimentale.

**Grafo DutchElite**

DuchElite è un grafo non connesso di modeste dimensioni, la cui componente connessa maggiore è formata da 3620 nodi e 4310 archi. Utilizzando il Binary Heap l'andamento dei tempi (come si può vedere dalla tabella) è concorde con gli altri grafi. Non è un grafo particolarmente denso, infatti $\Delta \simeq 0.00046$ .

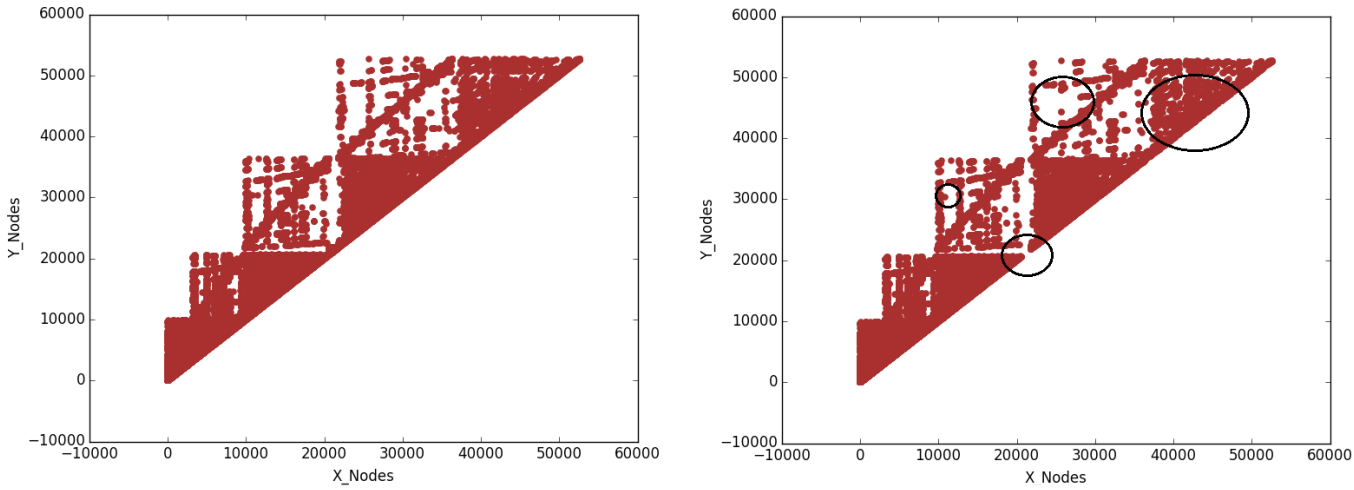



*Grafici del grafo DuchElite: a sinistra il grafo completo, a destra solo la componente connessa maggiore. I punti cerchiati evidenziano l'assenza delle connessioni delle altre componenti del grafo*

I due grafici mettono a confronto il grafo DutchElite completo e la parte connessa maggiore. Dal confronto si evince che la componente connessa maggiore di DutchElite è abbastanza grande e leggermente più densa del grafo completo, con $\Delta \simeq 0.00066$ .

**Grafo DIC28**

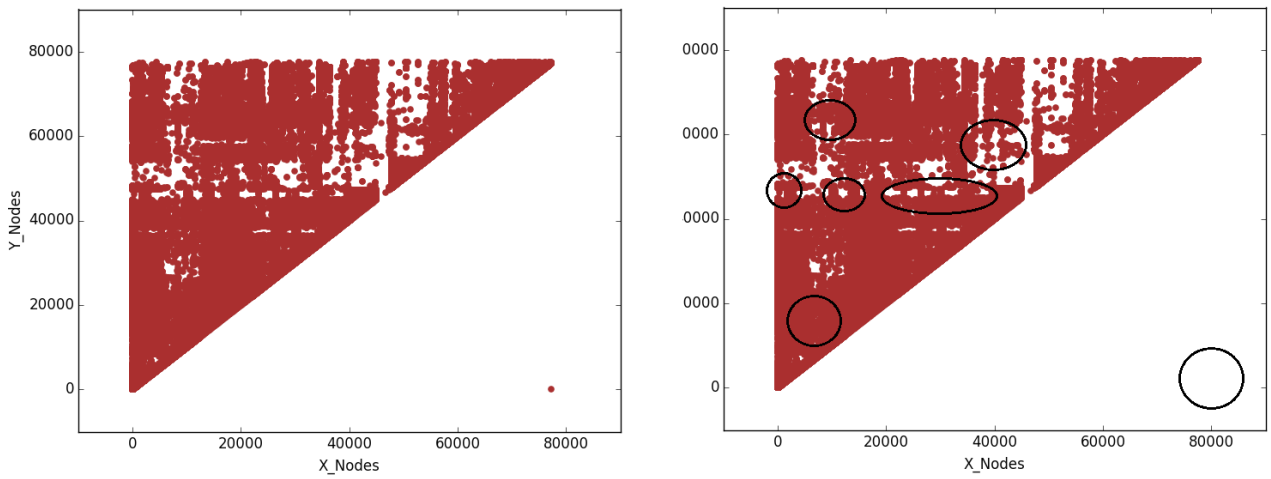
Il grafo DIC28 è un grafo di medie dimensioni non connesso, con 52651 nodi e 89038 archi. La sua componente connessa è formata da 24831 nodi e 71014 archi.



Il grafo è sparso, come si può notare dal grafico: infatti $\Delta \simeq 0.000064$ . La componente connessa è invece più densa, con $\Delta \simeq 0.00023$ , all'incirca un ordine di grandezza maggiore.

**Grafo Wordnet**

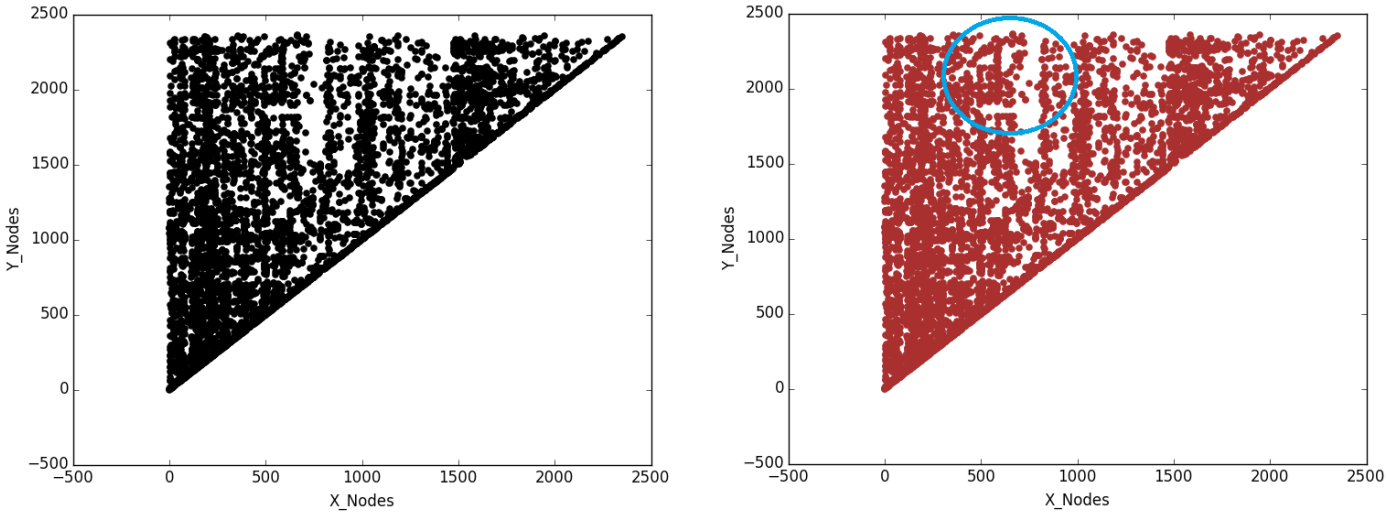
Il grafo Wordnet è un grafo piuttosto grande avente 77594 nodi e 133445 archi. La componente connessa maggiore è formata da 75606 nodi e 132688 archi.




La densità del grafo è $\Delta \simeq 0.000044$ quindi un grafo molto sparso.

**Grafo Yeast**

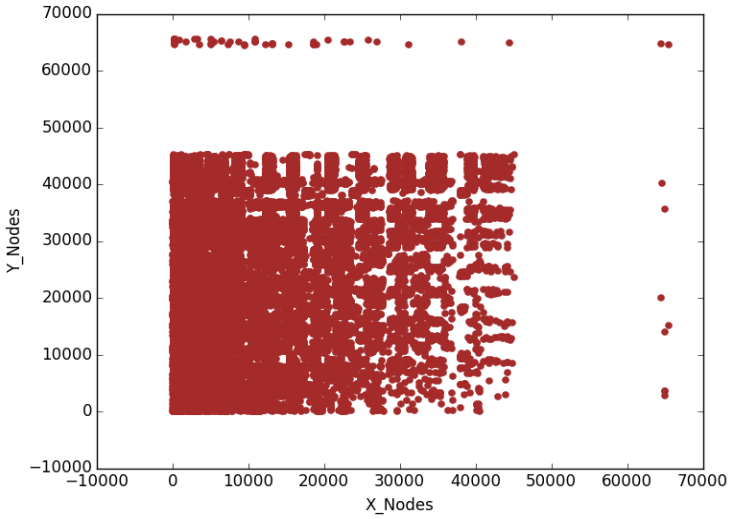
Il grafo Yeast è un grafico di piccole dimensioni, ma discretamente denso. Ha 2361 nodi e 6646 archi, con una densità pari a $\Delta \simeq 0.0024$ .



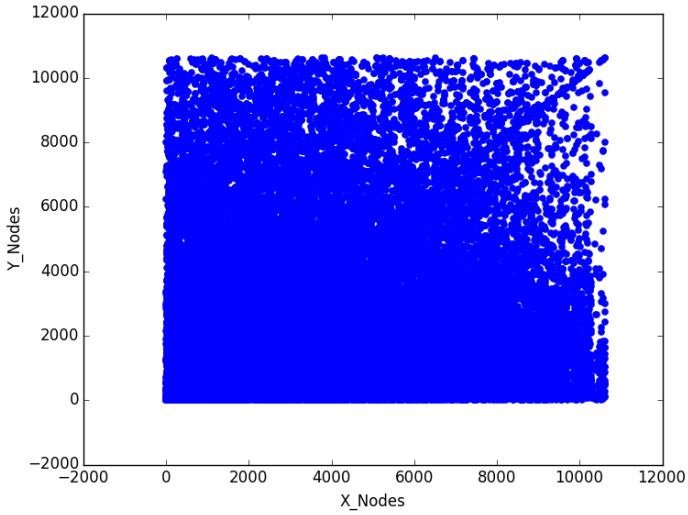

La componente connessa maggiore ha 2224 nodi e 6609 archi, con una densità di $\Delta \simeq 0.0026$

**Grafi con relative densità**

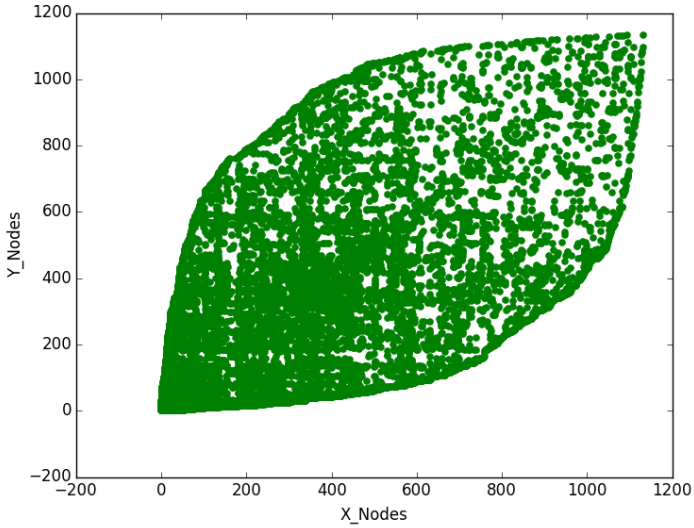
Adesso mostriamo delle immagini dei grafi  rimanenti con il loro valore di densità.



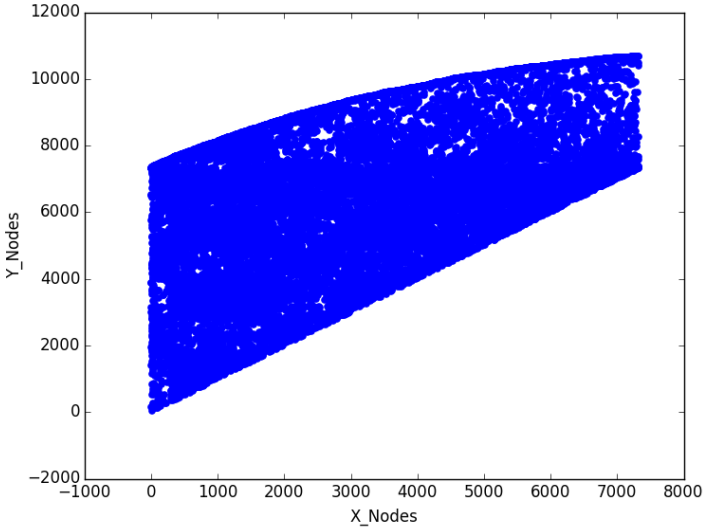

***Grafo AS*** *(  $\Delta \simeq 0.000027$  )*




***Grafo PairsP****(  $\Delta \simeq 0.0012$  )*




***Grafo email*** *(  $\Delta \simeq 0.017$  )*




***Grafo PGP*** *(  $\Delta \simeq 0.00043$  )*

**Grafici tridimensionali dello sviluppo geometrico tra nodi, archi e tempistiche sperimentali**

In questi grafici è possibile visualizzare l'andamento con cui, all'aumentare di nodi ed archi, aumenta il tempo d'esecuzione. I due grafici sono stati costruiti basandosi sui tempi ottenuti utilizzando la coda con priorità Binary Heap.

Per il grafico sui grafi non connessi si può notare come dal DIC28 in poi ci sia un incremento repentino dei tempi d'esecuzione, il che ci fa dedurre che per grafi più grandi come il grafo Wordnet, il Binary Heap non sia la struttura dati ottimale.

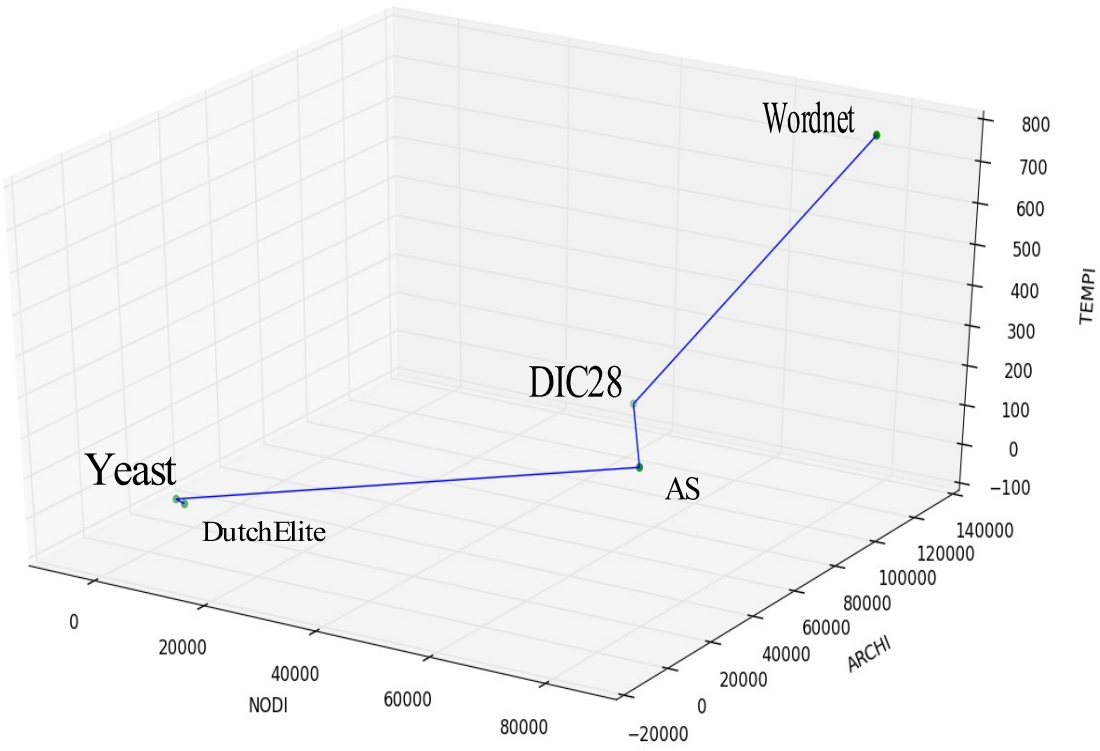



Grafico tridimensionale relativo a grafi non connessi.

Per grafi connessi si ha un risultato molto simile: all'aumentare della coppia (nodi, archi) aumenta il tempo d'esecuzione, anche se in maniera meno repentina. Inoltre si può anche vedere come la ricerca della componente connessa maggiore influisca sui tempi.

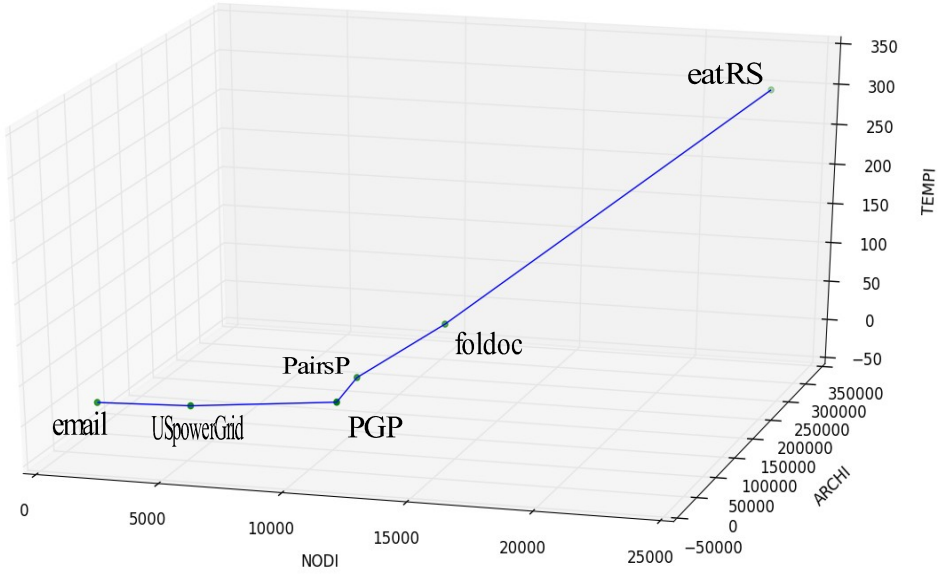



Grafico tridimensionale relativo a grafi connessi.

## Walktrough the code

Procediamo con questo capitolo ad un “walkthrough the code”. Una documentazione relativa all'algoritmo utilizzato è necessaria per la comprensione dei vari step quali :

- Creazione dell'ambiente grafico, button option e relative funzioni.
- Struttura della Main Function e quindi creazione della Lista di Adiacenza per il grafo.
- Soluzione Iterativa al calcolo delle componenti connesse.
- “Cuore” dell'Algoritmo e chiamate alle diverse PQ tramite relativi metodi.
- I/O su file , MST risultante e Profiler dedicato per ogni grafo preso in input.

La creazione dell'ambiente grafico avviene tramite il pacchetto di GUI strandard Python, Tkinter.
Una volta lanciato il programma da terminale , la nostra Main imposterà una grandezza “strandardizzata” a 800x600.

```
if __name__ == '__main__':

    root = Tk()
    root.resizable(0, 0)
    ws = 800
    hs = 600
    w = root.winfo_screenwidth()
    h = root.winfo_screenheight()
    x = (w/2) - (ws/2)
    y = (h/2) - (hs/2)
    root.geometry('%dx%d+%d+%d'%(800,600,x,y))
    app = App(root)



    cProfile.run("root.mainloop()","cProfile.txt")


    out = open(F + " Stats", "w")
    s = pstats.Stats("cProfile.txt", stream = out)
    s.strip_dirs().sort_stats("time").print_stats()
    out.close()
```

Viene in seguito invocata la classe App la quale provvederà all'utilizzo dei diversi metodi Tkinter per la costruzione del contenuto della finestra (In seguito sarà descritta in dettaglio).
Troviamo subito dopo il frammento di codice necessario allo streamout dei vari tempi relativi alle funzioni facenti parte dell'intero algoritmo. Il cProfile viene utilizzato in questo caso usando la variabile “F”, inizializzata come la directory nella quale si vuole salvare il file prodotto. In seguito viene eseguito un sort delle stats relative alla variabile “s” in funzione del parametro stringato 'time'.
Viene quindi eseguito il metodo “print_stats()” per poi chiudere il file aperto in precedenza.

La classe che segue sarà una derivata del modulo Tkinter dal quale ne andremo ad estrarre il Frame, necessario a lavorare nella finestra stessa. É infatti inizializzato il costruttore della classe tramite il parametro “root” inizializzato nella Main, che sta proprio a rappresentare le dimensioni fisiche che occuperà la nostra finestra. “Root” prenderà il posto del “master” inizializzato come un attributo fondamentale della classe creata. Sarà infatti molto utilizzato nel metodo “initUI” per tenere collegate le diverse creazioni fatte alla finestra.

```
class App(tk.Frame):

    def __init__(self, master):
        tk.Frame.__init__(self,master)

        self.m = master
        self.initUI()
```

E a seguire il metodo initUI() con alcune operazioni di creazione e gestione ambiente grafico :

```
def initUI(self):



    self.m.title("Syr2")
    self.pack(fill=BOTH,expand=1)


    style = Style()
    style.configure("TFrame", highlightbackground='#389')

    self.img = tk.PhotoImage(file = "geo.gif")
    label = Label(self,image=self.img,background='white')
    label.image = self.img
    label.place(x=400,y=180)
```

Impostazione del titolo tramite il metodo “title” con conseguente espansione tramite un packing del master. Lo stile della finestra viene impostato di default e come sfondo è stata scelta un'immagine in formato GIF con wallpaper bianco. Le prossime sono importanti funzioni relative alla scelta dei grafi e delle strutture dati a disposizione :

```
self.m.title('Syr2')
menubar = Menu(self.m)
self.m.config(menu=menubar)

self.fileMenu = Menu(menubar)
self.fileMenu.add_command(label='grafo_eu-2005 (277,3 MB)',
                          command =lambda:
                          self.a('grafo_eu-2005 (277,3 MB)',v,l1))
```

Dopo la creazione di un menu a cascata, utilizziamo il metodo di “add_command()” per rimandare una precisa label ad una funzione che aprirà il file di testo presente all'interno della cartella dell'algoritmo. Per questo genere di processo, torna molto utile la funzione lamda di Python Standard 2.7. Creando ogni volta una “veloce” funzione che al suo interno richiama un self della classe, si ha la possibilità di riutilizzare il programma quante volte vogliamo, continuando a cambiare struttura o grafo a nostro piacimento. Similmente è stato creato un secondo menu per le strutture dati. Il lavoro compiuto è lo stesso :

```
self.fileMenu2 = Menu(menubar)
menubar.add_cascade(label='STRUTTURE DATI',menu=self.fileMenu2)
self.fileMenu2.add_command(label='PQ_DHeap',
                           command =lambda: self.b('PQ_DHeap',u,l2,v))
self.fileMenu2.add_command(label='PQbinaryHeap',
                           command =lambda: self.b('PQbinaryHeap',u,l2,v))
self.fileMenu2.add_command(label='PQbinomialHeap',
                           command =lambda: self.b('PQbinomialHeap',u,l2,v))
```

Arriviamo in fine alla funzione di avvio che, presi i dettagli su cui operare, non farà altro che creare la lista di Adiacenza. Ovviamente tutto questo dopo i dovuti controlli per eventuali errori :

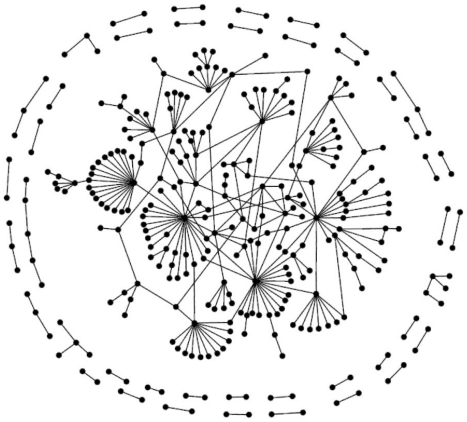
```
def avvio(self,v,u,c):

    gph = v.get()
    data = u.get()
    if not gph:
        t = tkMessageBox.askokcancel('Err',
                                     "Non e' stato selezionato il grafo!", type = "ok")
        if t == True:
            return
    elif c == None:
        t = tkMessageBox.askokcancel('Err',
                                     "Non e' stato scelto il valore per il D Heap!", type = "ok")
        if t == True:
            c = self.creaSpinbox()
    label5 = Label(self,
                   text = "Elaborazione in corso...", font = "Verdana 14 bold")
    label5.place(x=300,y=50)
    F = self.aaskopenfile()
    if F == None:
        return
    else:
        Graph_AdjacencyList.main(gph, data, F, c)
        showinfo("Syr2",
                 "Operazione Completata! Sono stati creati due file nella directory selezionata...
        App.close(self)
```

Come si può ben notare il metodo di “askokcancel” ci aiuta nella determinazione di una svista da parte dell'utente che utilizza il programma.
Il parametro “F” è passato alla Lista di Adiacenza per un controllo sui tempi delle prossime funzioni. Il Profiler scritto in C è pronto per attraversare tutto l'algoritmo per un controllo totale.

---

Nell'analizzare la funzione di partenza presente nel file Graph_AdjacencyList verranno prese in considerazione le porzioni di codice più importanti :

```
def main(q,l, F, c):
        global G

        start = time.time()

        Gl = GraphAdjacencyList()

        G = GraphAdjacencyList()

        PrendiDaFile(q,G)
        s, elapsed2 = max_cc(G,Gl)
```



Immediatamente è inizializzata una variabile “start” per il conteggio del tempo necessario alla terminazione dell'algoritmo. “G1” e “G” sono le due variabili alle quali viene assegnata la classe di Lista di Adiacenza implementata come già spiegato ampiamente nella premessa di questa relazione. Il perché queste istanze siano due e non una sarà spiegato a breve.
“PrendiDaFile()” è delegata alle operazioni di dialogo con i file di testo rappresentanti i grafi :

```
def PrendiDaFile(grafo,G):

    with open(grafo + '.txt','r') as f:
            p = 0
            b = 0
            c = 0
            g = 0
            nn = []
            pp = []

            for line in f:
                line = line.rstrip()
                n = line.split(' ')
                if b == 1:
                    print n[0]
                    for pesi in xrange(1, int(n[0]) + 1):
                        pp.append(pesi)
                    shuffle(pp)
                    c = 1

                if p == 1:
                    c = 2
```

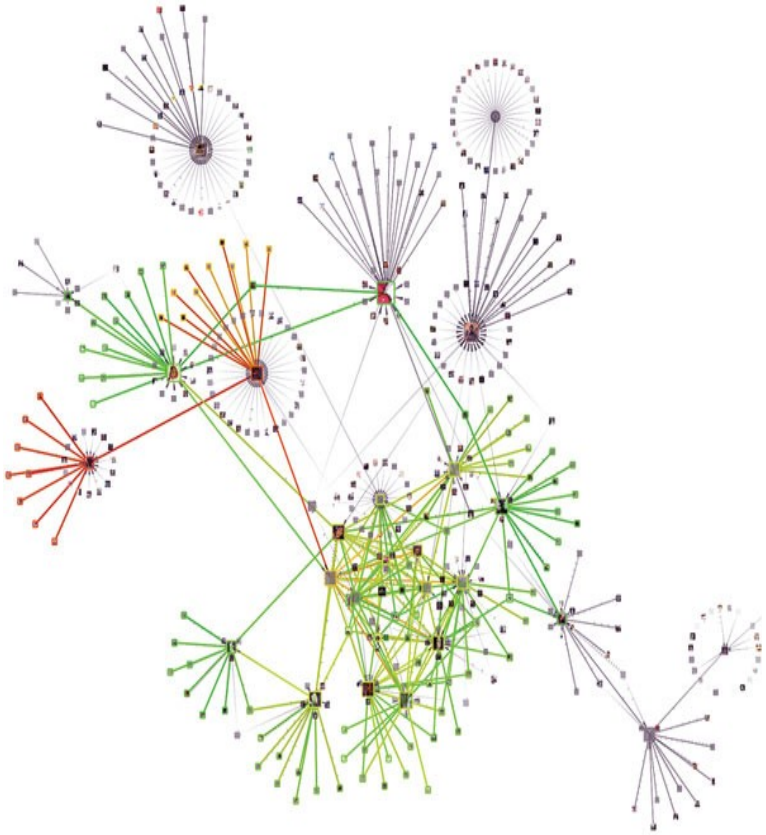
Come da codice, la built_in_function “open()” sarà necessaria per navigare nel file di testo in sola lettura. Essendo il file di testo formato come segue,

```
n       #numero nodi
m       #numero archi
0 1     #nodi che formano un arco
2 3     # “ “ “ “ “ “ “ “ “ “ “ “ “ “ “ “
2 5     # “ “ “ “ “ “ “ “ “ “ “ “ “ “ “ “
```

l'indice “line” dopo il metodo di “rstrip()”, ma soprattutto di split(' '), sarà equivalente a qualcosa come ['node','node'].
Incastrando contatori da incrementare grazie ai costrutti di controllo “if” consecutivi, riusciamo ad eseguire una scansione molto veloce e poco costosa a livello di memoria (precisiamo che gli indici incrementati arriveranno massimo al valore di 2).
Come da consegna l'assegnazione dei pesi è resa random grazie alla creazione di una lista “pp”, che inizialmente è ordinata in modo crescente essendo essa creata in un “for over a range sequences”. Una volta terminato il ciclo l'ordine è reso random dal metodo shuffle() della libreria random di python. Soddisfatte le condizioni per le quali la variabile “c” è uguale a zero e poi uguale a due avverranno le operazioni di insertNode() e insertArcW() relative alla lista di Adiacenza.

```
if c == 0:

    for i in xrange(0,int(n[0])):
        nn.append(G.insertNode(i))
    b = 1

if c == 2:

    if int(n[0]) != int(n[1]):

        G.insertArcW(nn[int(n[0])].index,nn[int(n[1])].index, float(pp[g]))
        G.insertArcW(nn[int(n[1])].index,nn[int(n[0])].index, float(pp[g]))

    g += 1
```

Dopo l'inserimento degli archi è aumentato di uno il contatore a variabile “g” in modo da continuare a scorrere nella lista “pp” per un'assegnazione costante e randomica (come già specificato) dei pesi.



Necessario il controllo if sui valori di n[0] e n[1] per non considerare nella costruzione del grafo i self loops. Costruito a dovere il grafo da file, proseguiamo subito con la ricerca delle componenti connesse. É necessario eseguire l'algoritmo di Prim sulla componente connessa più grande qualora il grafo non sia connesso. In virtù di ciò la funzione “max_cc()” eseguirà con un percorso iterativo le operazioni base di Simple_Search e costruzione della componente connessa più grande :

```
def max_cc(G,Gl):

    l = G.simpleSearch(0)
    if len(l) == len(G.nodes.keys()):
        print 'The Graph is connected'
        return 1, 0
    else:
        instant = time.time()
        showinfo('GrafoInfo',"Il grafo non e' connesso, verra' estratta la componente connessa piu'
                             "grande ed eseguito l'algoritmo di Prim. "
                             "E' possibile controllare l'avanzamento da terminale.")
        elapsed = time.time() - instant
        c = 0
        h = 1
        nl = len(G.nodes.keys())-1
        for i in G.nodes.keys():

            if not i in l:
                ll = G.simpleSearch(i)
                if len(ll) > len(l):
                    l = ll
            if c == 100*h:
                prog = 100*float(G.nodes.keys().index(i))/float(nl)
                print round(prog,2), '%'
                h += 1
            c += 1
        BuildGraph(G,Gl,l)
        return 0, elapsed
```

Sfruttiamo la semplicità della “simpleSearch()” a partire dal nodo 0 per poi confrontare le lunghezze della variabile creata e del numero di nodi appartenenti al grafo. Nel caso di “return true” il grafo sarà visualizzato come connesso. In caso contrario sarà necessario far ripartire una “simpleSearch()” da un nodo non incluso nella precedente “simpleSearch()”.
Un confronto delle lunghezze ci segnalerà poi la componente connessa più grande che sarà passata come parametro per la costruzione del nuovo grafo.
Il controllo booleano finale è necessario all'avanzamento della percentuale da terminale.

Terminata l'esecuzione di questa funzione, la variabile di ritorno “s” specificherà se il nostro grafo è connesso oppure no. Inizia da adesso in poi la “rampa di lancio” verso l'Algoritmo di Prim. Un semplice controllo sul valore booleano di “s” e verrà eseguito il frammento di codice che segue :

```
instant = time.time()
elapsed = time.time() - instant
w, mst, d = MST.prim(G, 0, l,c)
end = time.time() - elapsed - start
end2 = round(end,3)
out_file = open(str(F), "w")
out_file.write("Struttura dati utilizzata: " + l + "   ")
if l == 'PQ_DHeap':
    out_file.write("D = " + str(d) + "\n")
out_file.write("Tempo impiegato: "+ str(end2) + " s \n \nPeso totale: "
               + str(w) + "\n \nMST: \n \n")
for elee in mst:
    out_file.write(str(elee)+'\n')

out_file.close()
```

L'algoritmo di Prim ritornerà a tre variabili: peso dell'MST, MST ed il valore di ritorno del metodo “creaSpinbox()” del pacchetto Tkinter. Dalla variabile “end” fino alla chiusura del file “out_file” il codice controlla i parametri scelti dall'utente creando documento che attesterà che l'esecuzione è andata a buon fine, specificando i risultati ottenuti. L'operazione di sottrazione tra i risultati dei metodi utilizzati di “time” ci permetterà di visualizzare il tempo totale dell'esecuzione. Ci teniamo a ricordare che come esempio è stato presa la porzione di codice per i grafi connessi che è assolutamente identica per i grafi non connessi. L'unica differenza si noterà nella chiamata della funzione “MST.prim()”, nella quale è specificata la componente più grande del grafo insieme ad una sua radice.

---

Analizziamo infine quello che è stato da noi chiamato il cuore del programma. L'algoritmo di Prim è stato implementato sfruttando le funzioni di struttura delle PQ richieste dalla consegna.
Un veloce controllo di errore riguardo la radice e subito viene paragonato il parametro “l” che porta con se la scelta dell'utente. Viene quindi scelta la struttura dati con una particolarità nella PQ_Dheap(). Il parametro “d” come spiegato precedentemente risulta essere il risultato della Spinbox creata con Tkinter per la scelta del valore di “d”.
Procediamo con i metodi di inserimento delle strutture dati scorrendo la lunghezza delle chiavi di “g” (il nostro grafo). In altre parole inizializziamo la radice a peso float 0.0 mentre il resto degli archi a peso Infinito. Prima di definire la classe basterà dichiarare una variabile di questo tipo per utilizzare “Infinite” come nel codice :

```
Infinite = float("inf")
```

Le diverse istanze della PQ scelta sono memorizzate in “nodes” che è una lista python

```
@staticmethod
def prim(g, root, l, d):


    n = len(g.nodes)
    if root < 0 or root >= n:
        return
    nodes = n * [None]


    #pq = PQ_DHeap(2)
    if l == 'PQ_DHeap':
        pq = PQ_DHeap(d)
    if l == 'PQbinaryHeap':
        pq = PQbinaryHeap()
    if l == 'PQbinomialHeap':
        pq = PQbinomialHeap()

    lui = g.nodes.keys()


    for i in xrange(len(g.nodes.keys())):
        if lui[i] == root:


            nodes[i] = pq.insert(0.0, lui[i])

        else:

            nodes[i] = pq.insert(Infinite, lui[i])
```

avente “n” ( numero nodi) None al suo interno.
A seguire due costrutti “while” inseriti l'uno nell'altro che spiegheremo in modo separato .

```
while not pq.isEmpty():

    inode = pq.findMin()
    if inode == None:
        break

    if l == 'PQbinomialHeap':
        mst_weight += nodes[lui.index(inode)].ref.key \
            if nodes[lui.index(inode)] != None else 0

    else:
        mst_weight += nodes[lui.index(inode)].key \
            if nodes[lui.index(inode)] != None else 0


    if n2e[lui.index(inode)] != None:
        mst.append(n2e[lui.index(inode)])
    nodes[lui.index(inode)] = None

    pq.deleteMin()
    n = n-1

    curr = g.adj[inode].getFirstRecord()
```

Tenendo sott'occhio la presenza di elementi nella nostra PQ cerchiamo di soddisfare la necessità di marcare i nodi già inseriti nell'MST. Settiamo l'i-esimo nodo come il risultato dell'operazione di findMin() potendo così costruire il nosto MST scegliendo correttamente il nodo dalla lista n2e (nodes to edges). Ovvio specificare che ad ogni iterazione aggiorniamo il peso dell'MST per poi ritornarlo alla fine dell'algoritmo. Il controllo riguardo la PQ del Binomial Heap è necessario per una scelta stilistica. La coda con priorità è stata revisionata più volte per l'ottimizzazione del metodo di decreaseKey() fino alla creazione di un metodo di riferimento (.ref()) per l'indicizzazione degli alberi formati nella PQ (Rimandiamo alla premessa di questa relazione per i dettagli).
Infine dichiariamo l'elemento corrente utilizzando la struttura di LinkedList ed il metodo di getFirstRecord(). Possiamo ora passare al secondo costrutto “while” interno al primo. Questo utilizzerà la variabile “curr” dell'elemento corrente per decrementare la chiave relativa ad un singolo nodo nella PQ e per comparare gli archi. Qui di seguito il secondo costrutto while :

```
while curr != None:
    peso = curr.next
    el = curr.elem
    if l == 'PQbinomialHeap':

        if nodes[lui.index(el)] != None and \
                (peso.elem == None or peso.elem < nodes[lui.index(el)].ref.key):

            nodes[lui.index(el)].decrease(peso.elem
                                          if peso.elem != None else 0)
            n2e[lui.index(el)] = comparableArc((inode,el,peso.elem))

    else:

        if nodes[lui.index(el)] != None and \
                (peso.elem == None or peso.elem < nodes[lui.index(el)].key):

            pq.decreaseKey(nodes[lui.index(el)], peso.elem
            if peso.elem != None else 0)
            n2e[lui.index(el)] = comparableArc((inode,el,peso.elem))


    curr = curr.next
    curr = curr.next
```

**Conlusioni**

Con il seguente capitolo si vogliono discutere a livello di curiosità i “pro” e i “contro” relativi al linguaggio di programmazione utilizzato. Come già spiegato nella premessa sono stati adottati miglioramenti sostanziali grazie ad alcune potenzialità che offre il linguaggio Python2.7(Standard). Sorge spontanea però una domanda .. “Si può fare di meglio?” cit.
È possibile una computazione in parallelo senza un incremento sostanziale del memory usage ?
Può un processore MIPS R3000 (PlayStation , primi anni novanta) alimentare la sonda New Horizons che farà qualche scatto a Plutone dopo 5 miliardi di chilometri ?
Implementazioni multi_thread & multi_process abbinate ad una gestione ottimale della memoria porterebbero a notevoli migliorie , sia nella comprensione dell'algoritmo stesso , sia nell'utilizzo (performance) del programma.

Global Interpreter Lock (GIL) - Problema , Si o No ? -

Effettuare una release del Python_GIL tramite estensioni C sarebbe un punto a favore nel lavoro con Dataset di grandi dimensioni. Prendiamo come esempio l'utilizzo degli array multidimensionali e dell'analisi numerica per le quali è nato NumPy. Una semplice release del GIL ad un secondo thread (assunto uno stile di programmazione a più livelli) tramite NumPy, porterebbe un miglioramento dell'efficienza a tutto il programma.
Possiamo inoltre mappare direttamente su file array di grandissime dimensioni con tool come `“numpy.memmap”`.
Mettere in atto un CPython Fork tramite PyParallel per migliorare il conteggio dei riferimenti e quindi una gestione del Garbage Collector di python in contesti paralleli.

Link Utili :

Travis Oliphant, CEO of Continuum Analytics, kicks off PyData with a talk on Python in Big Data. Topics addressed include what Python has to offer the world of Big Data, specific use-cases, as Well asking why Hadoop is considered the de-facto standard. Additionally, Travis gives an overview of NumPy and SciPy :

https://www.youtube.com/watch?v=i0FCn889ucs

Githun PyParallel comprehension :

https://github.com/pyparallel/pyparallel