**GRUPPO 3: PROGETTO 4**

**ESERCIZIO 1**

**Greedy Vertex Cover**

La funzione non prende nulla in ingresso, mentre in uscita restituisce un dizionario vertexCover contenente non più del doppio dei vertici rispetto alla soluzione ottima per risolvere il problema richiesto.

Viene usato un set di appoggio E che contiene al suo interno tutti gli archi del grafo. Poiché l’algoritmo è di tipo *greedy*, si procede per passi successivi: finché il numero di archi non diviene pari a zero, ad ogni iterazione si seleziona un arco **(u,v)** di E, e si aggiungono i suoi vertici **u** e **v** al dizionario vertexCover; si eliminano dal set E tutti gli archi incidenti dei vertici **u** e **v**, per poi procedere all’iterazione successiva, finché E non è vuoto. Infine viene restituito il dizionario.

Riempire il set di appoggio prende tempo del numero di archi, mentre il while prende al massimo tempo del numero di archi in E e, per ogni vertice inserito prende tempo del grado dei vertici. Rimuovere l’arco dal set prende tempo costante. Quindi l’algoritmo complessivo ha complessità O(n + m) dove n è il numero di vertici ed m il numero di archi.

**Min Vertex Cover**

La funzione non prende nulla in ingresso e restituisce un dizionario vertexCover di dimensione minima. Essa richiama però un’altra funzione ricorsiva che riceve in ingresso un set di archi e restituisce un dizionario con il minimo numero di vertici che servono per coprire tutti gli archi.

La funzione ricorsiva controlla innanzitutto se il set di archi è vuoto, in questo caso ritorna un dizionario vuoto. Se così non è, estrare un arco casuale **(u,v)** e crea due copie della lista degli archi. In una copia, rimuove tutti gli archi incidenti di **u** ed effettua la ricorsione sulla copia della lista di archi più piccola, mentre nell’altra copia rimuove tutti gli archi incidenti di **v** ed effettua la ricorsione sulla seconda copia della lista di archi, sempre più piccola di quella originale. Si suppone in questo modo che la lista del vertex cover sia già disponibile e che manchi soltanto l’ultimo vertice da aggiungere, escludendo una delle due soluzioni ricorsive, ovvero la più lunga.

E.copy() prende tempo O(m), ed anche l’estrazione degli archi incidenti quando si itera su E prende tempo O(m), anche se questo tempo diminuisce molto quanti più archi incidenti sono presenti. Poiché però vengono effettuate due ricorsioni la complessità cresce come O(2^m) perché ad ogni ricorsione aumentano i set di archi che sto esaminando, anche se alla fine si andrà mano mano ad eliminare le soluzioni.

Quindi la complessità finale è di O(m\*2^m).

Dai risultati dei test è emerso infatti che, fissati il numero dei vertici, all’aumentare del numero degli archi, e quindi della densità del grafo, l’algoritmo diventa tremendamente lento, tanto da non riuscire a dare una soluzione in un tempo accettabile per miserabili.txt (ordine delle ore). Per questo motivo si è deciso di scrivere una terza funzione che riuscisse a dare un risultato quanto meno più preciso anche se non minimo.

**Greedy Min Vertex Cover**

La funzione non prende nulla in ingresso e restituisce un dizionario vertexCover di dimensione quasi minima. Si è scritto questa funzione per grafi con molti vertici e molti archi, poiché la soluzione minima non veniva trovata in tempi accettabili (ordine delle ore).

Dopo una fase di inizializzazione di una lista di appoggio e variabili contatore, il ciclo viene iterato n volte, con n numero dei vertici. Ad ogni iterazione il primo arco esaminato sarà diverso in modo tale che si esaminano così n soluzioni diverse: infatti si seleziona un arco, si inserisce nella soluzione il vertice che, tra i due estremi dell’arco, ha il maggior numero di archi incidenti e poi si cancellano dalla lista tutti gli archi incidenti di quel vertice e si procede, ripetendo il procedimento fino a che la lista non rimane vuota. Quindi a questo punto la soluzione dell’ultima iterazione viene confrontata con quella all’iterazione precedente (quest’ultima vale n per la prima iterazione): se la soluzione trovata ha un numero di nodi minore di quella all’iterazione precedente allora viene salvata e si ricomincia. Alla fine viene restituita la soluzione che contiene il minor numero di vertici.

La creazione del dizionario di appoggio iniziale ha complessità O(n), mentre la creazione della lista di appoggio contenente gli archi (che viene creata n volte) prende tempo O(m) con m numero degli archi. Il while viene ripetuto al massimo m volte ed all’interno per ogni arco si cercano gli archi incidenti, per eliminarli, iterando sulla lista degli archi, che la prima volta sarà grande m, ma poi sarà sempre più piccola. Solo quando viene trovata una soluzione migliore di quelle trovate fino a quel momento, si salva il dizionario in un nuovo dizionario con all’interno al massimo n vertici. La complessità finale è O(n(m^2 +1)) poiché si è notato che l’ultima operazione accade molto raramente.

Dai risultati dei test è emerso che l’algoritmo Greedy diventa molto più veloce rispetto a questo Greedy Min all’aumentare del numero di nodi, ma perde anche di efficienza: infatti nel caso “zachary\_club” con 34 vertici e 78 archi, l’algoritmo Greedy risulta 25 volte più veloce di quello Greedy Min ma ha anche una precisione minore, in quanto vengono inclusi nella soluzione 8 vertici in più, che però comunque sono meno del doppio della soluzione Greedy Min.

A parità del numero di vertici, invece, al diminuire del numero di archi, la velocità dell’algoritmo Greedy rimane stabile, ed è sempre poco preciso.

**N.B.** La soluzione trovata per le prove effettuate restituisce quasi sempre il minimo del numero dei vertici, ma potrebbe non farlo in alcuni casi, non essendo un algoritmo di ricerca esaustiva, poiché non vengono controllate tutte le combinazioni sugli archi, ma solo sui vertici e non essendo stata implementata una funzione di bounding che permetta di scartare apriori delle soluzioni.

**Test effettuati**

**Test grafi piccoli**

**#1: Grafo non diretto non pesato con 8 vertici e 10 archi**

VERTICI:

[A, B, C, D, E, F, G, H, ]

ARCHI:

[ (A,C,None), (A,B,None), (C,E,None), (D,G,None), (E,F,None), (H,G,None), (B,C,None), (C,D,None), (D,F,None), (D,H,None), ]

**Greedy vertex cover:** E, F, D, G, A, C,

GREEDY --> Dim = 6 , Time = 7.599999999999274e-05 s

**Min vertex cover:** B, H, D, A, E,

MIN --> Dim = 5 , Time = 0.00025199999999998834 s

La dimensione dell'algoritmo Greedy è di 1 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 3 volte piu' veloce di quello Min.

**#2: Grafo non diretto non pesato con 12 vertici e 17 archi**

VERTICI:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ]

ARCHI:

[(2,3,None), (11,2,None), (1,2,None), (3,4,None), (10,3,None), (4,5,None), (4,9,None), (5,6,None), (6,7,None), (7,8,None), (12,9,None), (10,11,None), (7,1,None), (9,5,None), (6,8,None), (8,12,None), (12,10,None), ]

**Greedy vertex cover:** 1, 2, 10, 3, 4, 5, 6, 7, 12, 9,

GREEDY --> Dim = 10 , Time = 0.00020399999999999585 s

**Min vertex cover:** 8, 9, 6, 4, 10, 2, 1,

MIN --> Dim = 7 , Time = 0.0017260000000000053 s

La dimensione dell'algoritmo Greedy è di 3 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 8 volte piu' veloce di quello Min.

**#3: Grafo non diretto non pesato con 15 vertici e 51 archi**

Vertici:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ]

Archi:

[(6,3,None), (4,5,None), (5,14,None), (12,5,None), (9,10,None), (10,15,None), (13,14,None), (7,5,None), (7,13,None), (9,11,None), (10,14,None), (6,5,None), (13,8,None), (11,14,None), (12,2,None), (2,8,None), (3,15,None), (4,10,None), (9,13,None), (14,12,None), (1,5,None), (2,3,None), (3,4,None), (8,5,None), (6,15,None), (9,6,None), (7,8,None), (9,14,None), (10,11,None), (11,12,None), (1,12,None), (1,4,None), (9,7,None), (8,12,None), (9,15,None), (11,13,None), (1,3,None), (9,1,None), (2,5,None), (4,15,None), (7,6,None), (10,13,None), (13,12,None), (1,2,None), (2,4,None), (6,8,None), (7,15,None), (10,12,None), ]

**Greedy vertex cover:** 4, 5, 6, 3, 9, 10, 13, 14, 12, 2, 7, 8,

GREEDY --> Dim = 12 , Time = 0.0003170000000000117 s

**Min vertex cover:** 3, 11, 7, 2, 10, 5, 12, 13, 9, 6, 4,

MIN --> Dim = 11 , Time = 0.028913999999999995 s

La dimensione dell'algoritmo Greedy è di 1 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 91 volte piu' veloce di quello Min.

**#4: Grafo non diretto non pesato con 2 vertici e 1 arco**

VERTICI:

[A, B, ]

ARCHI:

[(A,B,None), ]

**Greedy vertex cover:** A, B,

GREEDY --> Dim = 2 , Time = 1.8000000000004124e-05 s

**Min vertex cover:** A,

MIN --> Dim = 1 , Time = 1.5000000000001124e-05 s

La dimensione dell'algoritmo Greedy è di 1 vertici in più rispetto all'algoritmo Min.

Sono veloci allo stesso modo.

**Zachary Club**

**#5 : Grafo non diretto non pesato con 34 vertici e 78 archi**

**Greedy vertex cover:** 3, 8, 33, 34, 1, 9, 2, 20, 4, 13, 6, 11, 5, 7, 25, 28, 27, 30, 29, 32, 24, 26,

GREEDY --> Dim = 22 , Time = 0.0005399999999999988 s

**Min vertex cover:** 9, 7, 32, 24, 5, 6, 4, 2, 27, 34, 1, 25, 33, 3,

MIN --> Dim = 14 , Time = 1.7606680000000001 s

La dimensione dell'algoritmo Greedy è di 8 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 3260 volte piu' veloce di quello Min.

**Testing50**

**#6 : Grafo non diretto non pesato con 51 vertici e 20 archi.**

**Greedy vertex cover:** 4, 15, 1, 2, 37, 38, 12, 5, 7, 6, 13, 14, 19, 22, 9, 11, 47, 49, 20, 23, 42, 43,

GREEDY --> Dim = 22 , Time = 0.00015899999999999248 s

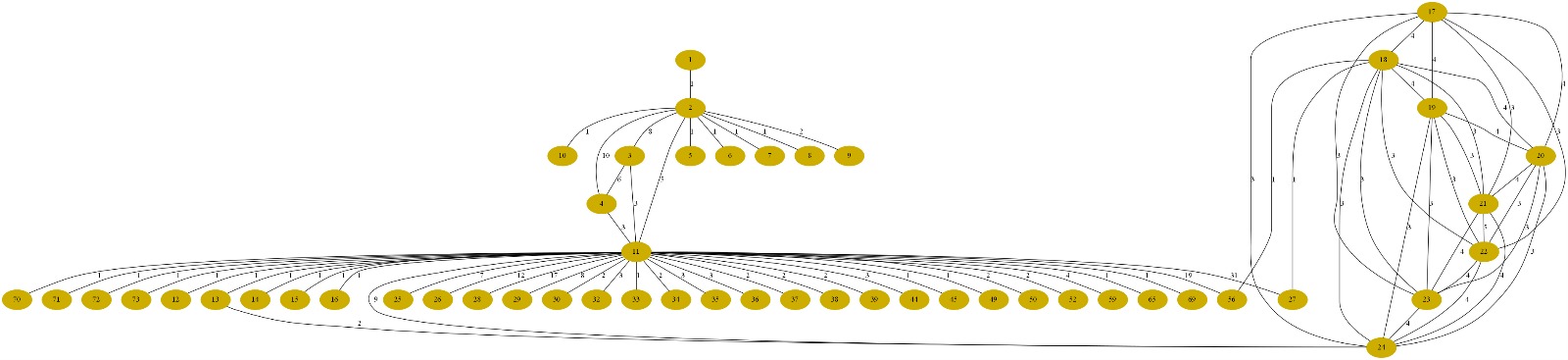
**Min vertex cover:** 47, 9, 7, 1, 23, 19, 42, 13, 12, 38, 4,

MIN --> Dim = 11 , Time = 0.050066 s

La dimensione dell'algoritmo Greedy è di 11 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 314 volte più veloce di quello Min.

L'algoritmo Greedy impiega 0.04990700000000001 secondi in meno rispetto a quello Min.

**#7: Tratto dai miserabili.txt: Grafo non diretto non pesato con 77 vertici e 77 archi**

Alla fine i vertici realmente contenuti nel grafo sono 51.

**Greedy vertex cover:** 2, 11, 18, 22, 19, 20, 21, 23, 13, 24, 3, 4,

GREEDY --> Dim = 12 , Time = 0.00042000000000008697 s

**Min vertex cover:** 20, 22, 17, 3, 13, 18, 21, 19, 11, 2,

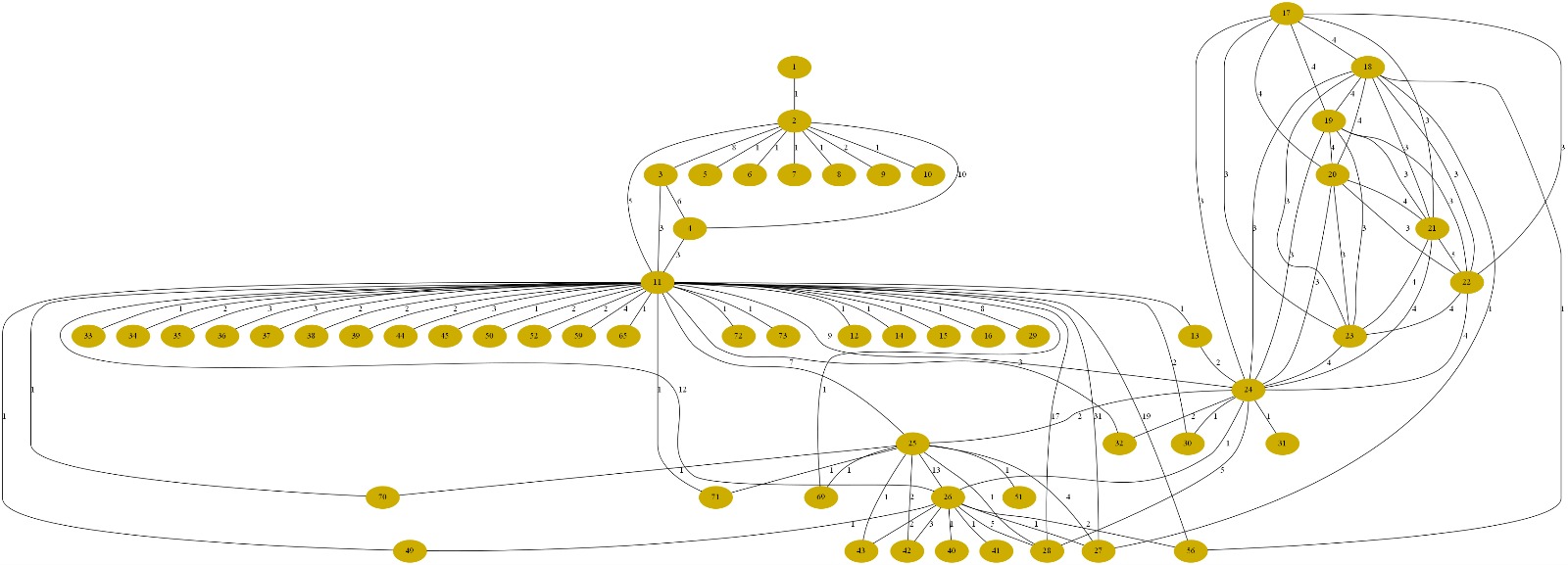
MIN --> Dim = 10 , Time = 0.10302299999999853 s

La dimensione dell'algoritmo Greedy è di 2 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 245 volte piu' veloce di quello Min.

L'algoritmo Greedy impiega 0.10260299999999845 secondi in meno rispetto a quello Min.

**#8: Tratto dai miserabili.txt: Grafo non diretto non pesato con 77 vertici e 100 archi**



Alla fine i vertici realmente contenuti nel grafo sono 57.

**Greedy vertex cover:** 11, 52, 18, 24, 2, 3, 19, 20, 25, 70, 26, 27, 22, 23, 17, 21,

GREEDY --> Dim = 16 , Time = 0.0005759999999987997 s

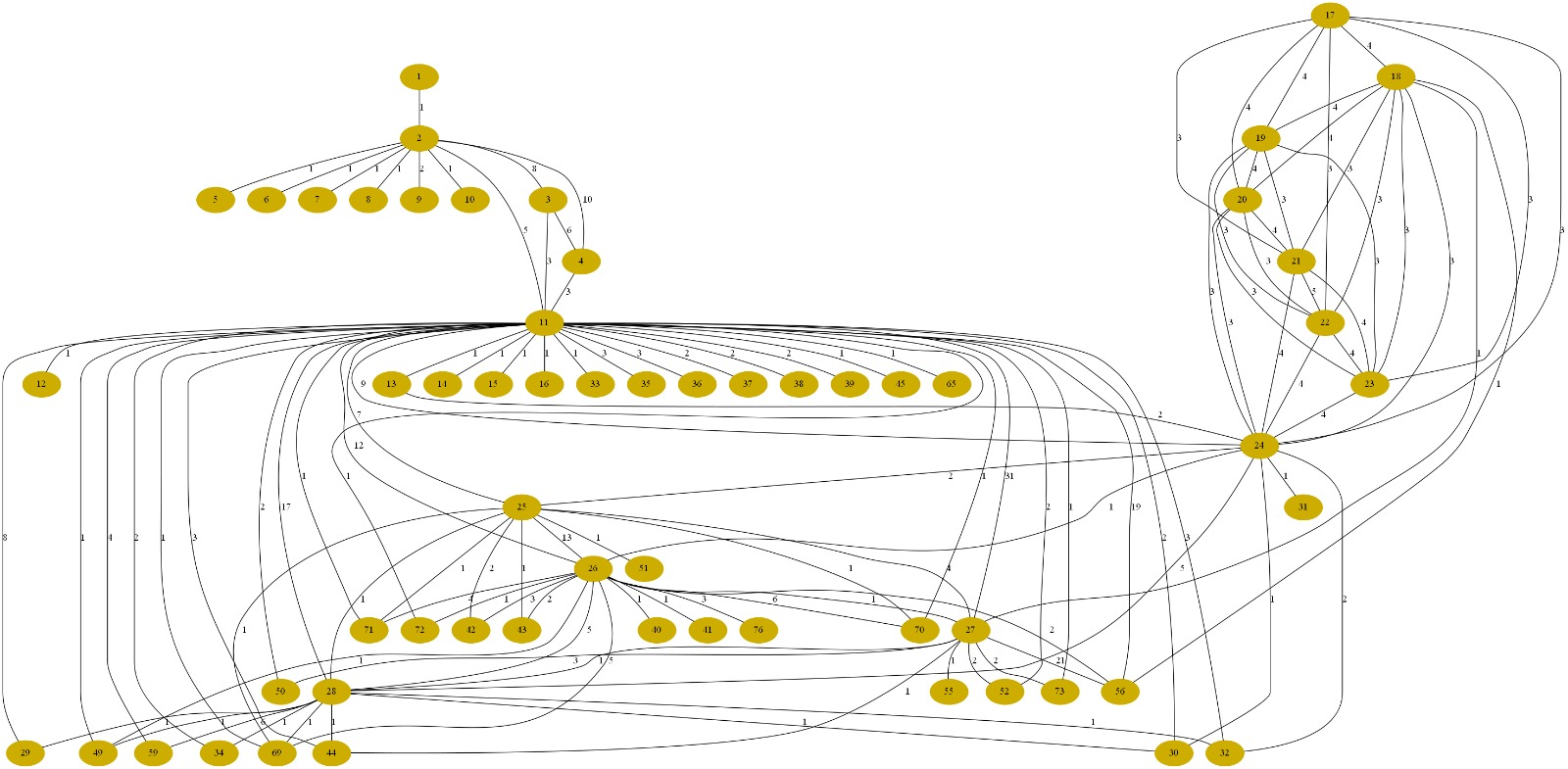
**Min vertex cover:** 17, 3, 25, 26, 20, 22, 24, 21, 2, 19, 18, 11,

MIN --> Dim = 12 , Time = 7.662136 s

La dimensione dell'algoritmo Greedy è di 4 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 13302 volte piu' veloce di quello Min.

L'algoritmo Greedy impiega 7.6615600000000015 secondi in meno rispetto a quello Min.

**#9: Tratto dai miserabili.txt: Grafo non diretto non pesato con 77 vertici e 120 archi**

Alla fine i vertici realmente contenuti nel grafo sono 59.

**Greedy vertex cover:** 27, 55, 21, 22, 11, 36, 17, 20, 26, 28, 24, 25, 1, 2, 18, 19, 3, 4,

GREEDY --> Dim = 18 , Time = 0.001548999999997136 s

**Min vertex cover:** 19, 3, 25, 28, 20, 2, 24, 18, 26, 22, 17, 11, 21, 27,

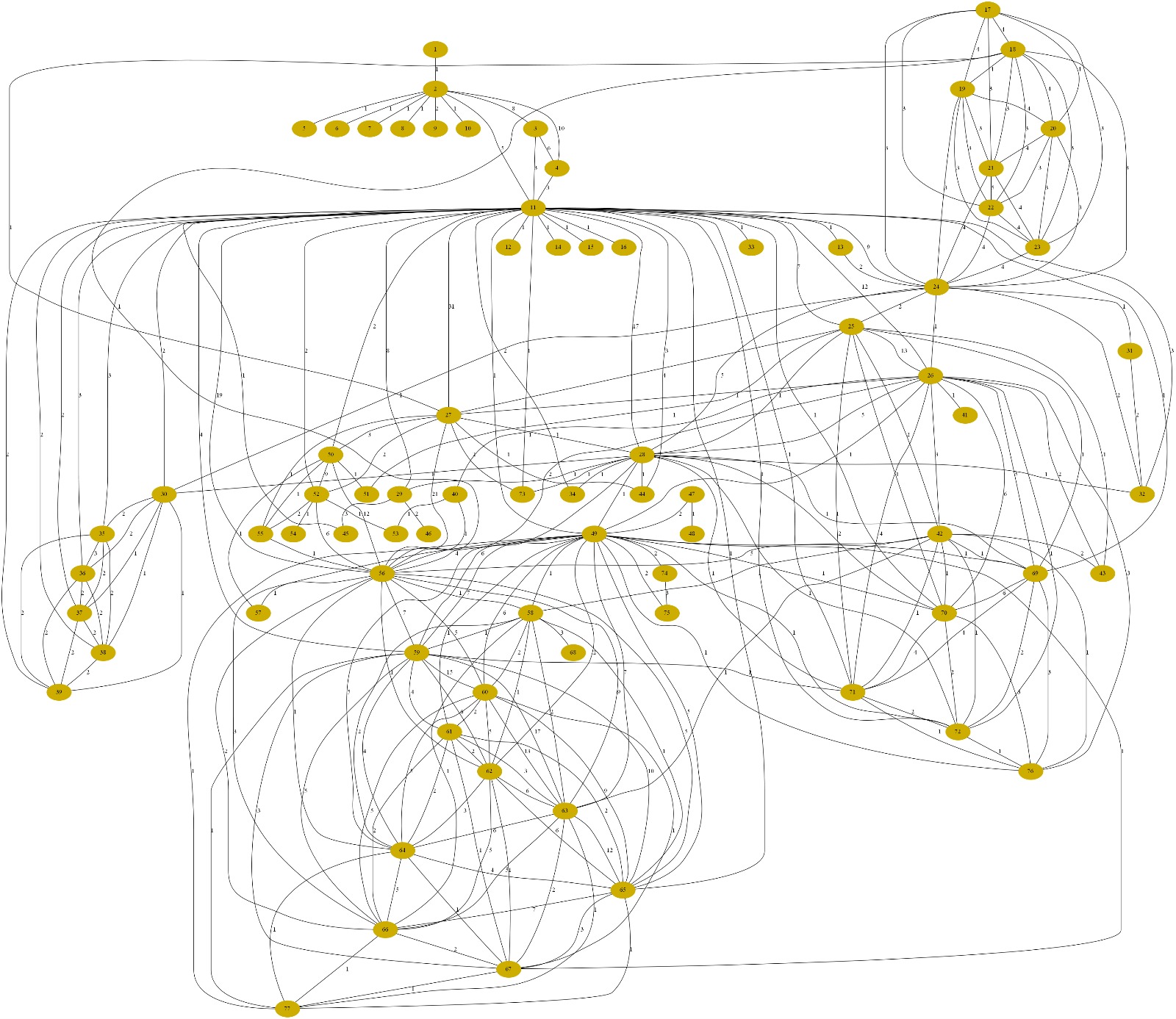
MIN --> Dim = 14 , Time = 11.931045000000001 s

La dimensione dell'algoritmo Greedy è di 4 vertici in più rispetto all'algoritmo Min.

L'algoritmo Greedy è 7702 volte piu' veloce di quello Min.

L'algoritmo Greedy impiega 11.929496000000004 secondi in meno rispetto a quello Min.

**Miserabili**

**#10: Grafo non diretto non pesato con 77 vertici e 254 archi**

**Greedy vertex cover:** 64, 65, 11, 32, 60, 61, 20, 21, 17, 18, 24, 25, 26, 43, 52, 53, 2, 6, 42, 70, 49, 75, 62, 66, 27, 50, 36, 39, 72, 76, 28, 59, 30, 35, 56, 57, 69, 71, 19, 22, 63, 67, 29, 45, 3, 4, 47, 48, 37, 38, 58, 68,

GREEDY --> Dim = 52 , Time = 0.004092999999999999 s

**Greedy Min vertex cover:** 11, 18, 21, 24, 26, 28, 52, 2, 42, 49, 56, 66, 27, 39, 60, 65, 59, 72, 25, 30, 50, 62, 63, 22, 32, 23, 67, 70, 71, 19, 38, 29, 4, 36, 40, 37, 69, 20, 64, 47, 58,

GREEDY MIN --> Dim = 41 , Time = 0.24397899999999997 s

La dimensione dell'algoritmo Greedy è di 11 vertici in più rispetto all'algoritmo Greedy Min.

L'algoritmo Greedy è 59 volte piu' veloce di quello Greedy Min.

**ESERCIZIO2**

Algoritmo.

La funzione bridge prende in ingresso un grafo connesso e non diretto e restituisce una lista di tutti gli archi che sono bridge, se non sono presenti bridge restituisce una lista vuota.

L'algoritmo utilizza DFS leggermente modificata per ottenere gli archi DISCOVERY e ordine. Ordine è la lista degli archi nell'ordine in cui vengono visitati dalla DFS. Questa chiamata costa O(n+m).

Vengono inizializzati il set discovery che contiene gli archi DISCOVERY ed il set bridge, che inizialmente contiene tutti i potenziali archi BRDIGE, quindi i due set all'inizio sono equivalenti dato che solo un arco DISCOVERY può essere un bridge, un arco BACK se rimosso non disconnette il grafo, dato che ci sono altri archi che portano a quel nodo. Viene quindi inizializzata la lista degli archi back.

Queste operazioni costano O(n).

IDEA CHIAVE: l'aggiunta di ogni arco BACK all'albero ottenuto con la DFS forma un ciclo, tutti i vertici che appartengono ad un ciclo non sono bridge.

L'algoritmo esamina per ogni back gli archi a cui quell'arco back appartiene e li toglie dal set bridge. Iterando questo processo per ogni back, quindi cancellando dal set tutti gli archi che non fanno parte di nessun ciclo, i rimanenti sono necessariamente i BRIDGE del grafo.

Per ogni arco back quindi, sfruttando la list ordine, che contiene il percorso fatto dall'algoritmo, si riesce a capire, dati gli estremi dell'arco BACK, quali sono gli archi che appartengono al ciclo.

Il primo for viene ripetuto per un numero di volte pari al numero di archi BACK, quindi M-(N-1) volte. Dove N-1 sono gli archi discovery ed M sono gli archi totali del grafo. Il secondo for annidato viene ripetuto nel worst case N-1 volte, ma questo succede solo quando l'arco back parte dal primo nodo visitato all'ultimo, solitamente questo numero è più basso. Alla fine viene fatta un'operazione di differenza tra insiemi, ma, dato che l'insieme dei bridge viene sempre ridotto, e questa operazione ha una complessità che è data dalla somma delle dimensioni dei due set, il suo costo è sostanzialmente ammortizzato ad O(n) nel worst case. Si conclude quindi che la complessità totale nel caso peggiore è O((m-n)\*n), ma i tempi di esecuzione in pratica sono buoni dato che mediamente l’algoritmo itera il secondo for per un numero di volte minore.

Test.

Per il test è stata implementata una funzione che genera grafi random, la funzione non assicura che il grafo sia connesso ed aggiunge archi in maniera casuale. La funzione bridge viene testata solo su grafi connessi generati da questa funzione. Per ogni arco indicato come bridge si rimuove quest'ultimo dal grafo e si verifica se è disconnesso. Nel caso in cui il grafo da connesso diventa disconnesso il test si ritiene superato, altrimenti la funzione fallisce.

La funzione viene testata su 100 grafi generati casualmente, tutti connessi, riportando il numero di bridge trovati e il numero di archi che in realtà non erano bridge, quindi gli errori commessi. La funzione testata più volte non ha mai commesso nessun errore. Lo script calcola il tempo totale, che solitamente è dell'ordine di pochi e dipende anche dal tempo impiegato dalla funzione randomGraph per generare grafi connessi. Inolte, riporta il tempo medio impiegato dalla singola chiamata della funzione.

Sulla base di questo tempo si può vedere che testando l'algoritmo su 100 grafi con m=70 ed n=50, il tempo medio è 0.0058 secondi. Mentre testandolo su 100 grafi con m=250 ed n=50, il tempo medio è 0.0313 secondi. Si può vedere che, pur avendo moltiplicato per 10 il fattore m-n, il tempo di esecuzione non è aumentato di un ordine di grandezza.

Si consiglia di non eseguire il test su grafi con n = m = numero elevato, poiché la funzione randomGraph potrebbe impiegare troppo tempo a generare casualmente un grafo connesso.

**ESERCIZIO3**

File : ./pkg3/emergency.py

La funzione *emergency\_call(G, pos, v, k)* prende in input un grafo diretto connesso *G* che rappresenta la rete stradale cittadina (i vertici sono gli incroci e

gli archi sono le strade) un dizionario *pos* che contiene la posizione delle volanti, il luogo *v* dove intervenire ed il numero *k* di volanti richieste per l’intervento, e restituisce un array con le k volanti più vicine al punto di intervento.

La funzione *calling911()* inizializza i parametri pos,v, k passati alla funzione *emergency\_call();* dato G, il dizionario di posizioni delle volanti e il luogo di intervento sono scelti random tra i vertici di G, evitando che più volanti siano situate nello stesso punto. Il parametro k, invece, si è assunto che sia pari a 1/3 dei vertici di G.

L’obiettivo è quello di individuare un percorso minimo tra le volanti e il punto di emergenza. Essendo il grafo diretto si è deciso di applicare l’algoritmo di Dijkstra partendo dal punto di emergenza ma invertendo gli archi del grafo in modo da trovare tutti percorsi minimi che consentono le volanti di raggiungere il punto di intervento.

Una volta ottenuto il *cloud* (dizionario che mappa v in d[v]) si individuano le k volanti più vicine.

La complessità della funzione emergency\_call() è O(mlogn), dato che l’inversione degli archi ha complessità O(1), la complessità di Dijkstra è O(mlogn), per individuare le volanti occorre iterare al più su tutti i vertici del grafo ma dato n è un numero inferiore a m la complessità totale è O(mlogn).

I test sono stati effettuati su grafi letti da file .txt attraverso la funzione readFile().

Inoltre nella cartella Graphs\_images sono contenute le immagini dei grafi testati.