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. 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 quello 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 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 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 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.

**ESERCIZIO2**

**ESERCIZIO3**