## Esercizio 2: graph\_coloring(g)

L’algoritmo implementato per la soluzione del problema si articola nei seguenti passi:

1. Scelta di un vertice da cui partire
2. Colorazione del vertice ed aggiunta del vertice alla soluzione
3. Scelta del prossimo vertice
4. Colorazione del vertice in maniera tale da non assegnare un colore già usato per uno dei vertici adiacenti presenti nella soluzione ed aggiunta del vertice corrente alla soluzione
5. Iterare dal punto tre finché tutti i vertici non saranno stati colorati

Per quanto riguarda la scelta dell’ordine in cui considerare i vertici, secondo Welsh-Powell[[1]](#footnote-1) ordinando i vertici in base al grado crescente è possibile porre un limite superiore al numero di colori necessari, pari a:

Dove *x*i sono i vertici del grafo. Ovvero i colori necessari saranno al più uguali al grado massimo dei vertici del grafo più uno.

Per ordinare i vertici in base al grado è stata usata una reverse heap priority queue, mentre per tenere traccia dei colori utilizzati è stato scelto un set, questo perché le operazioni disponibili sugli insiemi risultano particolarmente adatte all’implementazione scelta per l’algoritmo.

Siano *color* il dizionario contenente i vertici già aggiunti alla soluzione ed *u* il vertice valutato all’iterazione corrente: per ogni vertice nella soluzione vengono aggiunti ad un set di appoggio *used* i colori assegnati ad i vertici adiacenti ad u, infine effettuando la differenza fra i due insiemi è possibile conoscere l’insieme dei colori che è possono essere assegnati al vertice corrente. Se questo insieme è vuoto, allora abbiamo bisogno di un nuovo colore, avendo già usato tutti quelli a nostra disposizione, altrimenti è possibile estrarre un qualsiasi elemento dall’insieme dei colori non utilizzati ed assegnarlo al verti corrente, che potrà così essere aggiunto definitivamente alla soluzione.

Il codice relativo al meccanismo appena descritto è mostrato a seguire:

deg, u = pq.remove\_min()  
used = set()   
**for** v **in** color:   
 **if** G.get\_edge(u, v):   
 used.add(color[v])   
unused = ku.difference(used)  
**if** unused:  
 color[u] = unused.pop()  
**else**:  
 k += 1  
 ku.add(k)  
 color[u] = k

Per quanto riguarda la complessità della funzione, siano *n* i vertici del grafo, il primo passo è il riordinamento dei vertici che, effettuato tramite una heap queue, richiede complessità O(nlog(n)). Proseguendo nell’algoritmo, è necessario eseguire n iterazioni per aggiungere tutti i vertici alla soluzione. Per ogni iterazione è necessario controllare l’adiacenza del vertice corrente con quelli già presenti nella soluzione, aggiungendo di volta in volta i colori utilizzati al set. La differenza fra due insiemi *s* e *t* ha complessità O(len(s))[[2]](#footnote-2), ovvero al più pari al numero di colori usati, che per quanto detto sarà inferiore od uguale al grado massimo dei vertici del grafo più uno, dunque molto inferiore al numero dei vertici. Considerando che nella solzuione, all’iterazione finale, ci saranno n-1 vertici da dover analizzare, la complessità complessiva dell’algoritmo può essere stimata come O(n2).

1. Welsh, D. J. A.; Powell, M. B. (1967), "An upper bound for the chromatic number of a graph and its application to timetabling problems", The Computer Journal, 10 (1): 85–86 [↑](#footnote-ref-1)
2. https://wiki.python.org/moin/TimeComplexity [↑](#footnote-ref-2)