Il metodo min\_vertex\_cover non prende input e restituisce un ricoprimento dei vertici del grafo su cui è chiamato (vertex cover)  di dimensione minima.

In genere, quando *min\_vertex\_cover* è eseguito su un grafo con *n* vertici e *m* archi, per trovare il minimo insieme di vertici ricoprenti si genera un albero decisionale composto di 2n nodi. Se ci comportassimo nello stesso modo anche nel caso in cui il grafo sia non connesso, andremmo a generare un albero decisionale più grande del dovuto.

Se avessimo un grafo G con n vertici non connesso, composto ad esempio da due grafi G1 e G2 connessi con rispettivamente n1 e n2 vertici, l’albero decisionale associato a G avrebbe 2n nodi, mentre quelli associati a G1 e G2 avrebbero 2n1 + 2n2 nodi. È immediato dire che 2n > 2n1 + 2n2.

Se un grafo è non connesso, conviene quindi dividerlo in grafi connessi e chiamare su ognuno di loro *min\_vertex\_cover* e unire le singole soluzioni parziali così da ottenere la soluzione per il grafo di partenza.

Per compiere l’operazione di divisioni in grafi connessi è stato utilizzato il metodo *\_sconnect\_graph.* Tale metodo verifica inizialmente che il grafo sia non connesso e in seguito lo divide in grafi connessi. *\_sconnect\_graph* utilizza:

* + - *grafi*, una list in cui salvare i grafi che compongono il grafo di partenza
    - *disc\_vert*, contiene tutti i vertici del grafo di partenza ed alcune informazioni riguardanti essi. È un dict che ha per chiave i vertici e per valore una list composta da due campi:
      * + Il primo campo è un valore intero, inizialmente settato a zero, che rappresenta il grafo di appartenenza del vertice
        + Il secondo campo è inizialmente settato a None e serve ad invalidare il vertice

Tale metodo, dopo una fase iniziale d’inizializzazione di *disc\_vert*, chiama per ogni vertice in *disc\_vert,* che non appartiene ancora a un grafo, il metodo *\_DFS*, il quale partendo da tale vertice marca con uno specificato valore tutti i vertici presenti in *disc\_vert* che riesce a raggiungere. In questo modo tutti i vertici in *disc\_vert* apparterranno a un grafo. Se tutti i vertici appartengono allo stesso grafo, significa che questo è connesso. In caso contrario il grafo è non connesso e quindi bisogna dividerlo in grafi connessi. Per far ciò si scorre l’insieme degli archi del grafo di partenza, si ottengono i suoi endpoint e si verifica a quale grafo appartengano attraverso *disc\_vert*. Stabilito il grafo di appartenenza, si aggiungono a esso insieme all’arco che li congiunge.

Una volta terminata l’operazione di divisione in grafi connessi è eseguita la chiamata a \_*min\_vertex\_cover* per ognigrafoin *grafi*, in seguito, si uniscono le soluzioni di ogni grafo connesso e la si fornisce come soluzione del grafo non connesso.

*\_sconnect\_graph* ha complessità O(n+m\*k), dove k è il numero di grafi connessi che compongono il grafo di partenza.

Il metodo *\_min\_vertex\_cover* ha il compito di inizializzare i parametri della funzione ricorsiva di backtracking per ottimizzazioni: *\_backtrack\_min\_vertex\_cover*.

L’algoritmo di ricerca esaustiva implementato nel metodo *\_backtrack\_min\_vertex\_cover* prevede cinque parametri:

* *vertex\_list*: lista dei vertici da esaminare;
* *current\_status*: un dizionario che tiene traccia delle scelte fatte scendendo all’interno dell’albero di decisione;
* *sol*: la soluzione finale dell’algoritmo;
* *current\_solution*: tiene traccia della soluzione corrente;
* *k*: il numero del vertice da analizzare, inizializzato a 0.

All’interno di *\_min\_vertex\_cover* è calcolato *vertex\_list* ordinato per degree massimo dei vertici, è inizializzato *current\_status* che per ogni arco (key) varrà due (value) e infine è impostata come *solution* quella contenente tutti i vertici del grafo (sol conterrà nella posizione 0 *solution*).

Il dizionario *current\_status* è inizializzato a due perché all’inizio nessun vertice è stato scartato quindi ogni arco può essere coperto da due vertici (i suoi endpoints).

La complessità temporale della *\_min\_vertex\_cover* dipenderà quindi dal metodo *\_backtrack\_min\_vertex\_cover*, poiché la complessità del passo d’inizializzazione è pari a O (n\*log (n) + m), molto minore della complessità esponenziale del metodo di backtracking.

Il metodo *\_backtrack\_min\_vertex\_cover* si occupa di ricercare la soluzione ottima evitando di esaminare tutte le 2^n (con n come numero dei vertici) possibili scelte.

Ad ogni passo è considerato il k-esimo (da zero a n-1) vertice della lista (*vertex\_list*[k]), se non considerando tale vertice nella soluzione, si riesce a coprire lo stesso tutti gli archi (condizione verificata dalla funzione di bounding *\_could\_be\_a\_sol*), allora sarà generato il nuovo stato della soluzione e richiamata ricorsivamente la *\_backtrack\_min\_vertex\_cover* con *current\_status* pari al nuovo stato calcolato e passo pari a k+1. Dato che considerare il vertice nella soluzione, permette lo stesso di coprire tutti gli archi andando avanti, sarà richiamata ricorsivamente sempre la *\_backtrack\_min\_vertex\_cover* con passo k+1.

Ricapitolando ad ogni passo è richiamato il metodo di backtrack sullo stato corrente con passo k+1 e inoltre se la funzione di bounding è vera per lo stato corrente e il vertice considerato si chiama ricorsivamente nuovamente il metodo di backtrack sul nuovo stato e con passo k+1.

Quando la ricorsione arriverà alle foglie dell’albero decisionale, queste ultime conterranno una soluzione, se è migliore di quella attuale, la soluzione finale sarà aggiornata.

Nel caso peggiore la funzione di backtracking considererà tutti i nodi dell’albero decisionale, inoltre dovendo eseguire una copia di *current\_solution* prima del passo ricorsivo, la complessità temporale dell’algoritmo è pari a: O(n\*2n ).

L’algoritmo greedy per la ricerca approssimata del vertex cover minimo compie la scelta migliore localmente in funzione del grado massimo dei vertici. In particolare ad ogni passo sarà aggiunto alla soluzione il vertice con più archi incidenti, e sarà diminuito il grado di uno per tutti i vertici adiacenti a quello aggiunto (per simulare l’eliminazione degli archi). A ogni iterazione sarà cercato il massimo tra i vertici ancora non considerati e con grado diverso da zero. L’algoritmo termina quando tutti i vertici avranno grado nullo, in altre parole quando tutti gli archi saranno stati coperti.

L’implementazione prevede una fase d’inizializzazione per ottenere il grado di tutti i vertici (costo O(n)), ed un ciclo che sarà iterato k volte, indicando con k il numero di vertici all’interno della soluzione approssimata del vertex cover, all’interno del quale sarà cercato il massimo, tra i soli vertici restanti, e aggiornato lo stato del problema.

La complessità temporale dell’algoritmo nel caso peggiore di grafo completo, grazie all’utilizza di un dizionario per aggiornare lo stato della soluzione, è pari a O(k\*n + m) (per un grafo completo O(n\*(n-1)).