Minimum Weight Vertex Cover using Tabu Search algorithm

Luigi Priano (1000002081)

Professori: Mario Francesco Pavone, Vincenzo Cutello.

Dipartimento: D.M.I. Informatica LM-18 A.A. 2024/2025

Data documento: 10/02/2025.

Indice

1 Minimum Weight Vertex Cover

Il problema della Copertura dei Vertici a Peso Minimo (MWVC - Minimum Weight Vertex Cover) è un problema fondamentale nella teoria dei grafi con molte applicazioni pratiche, tra cui le comunicazioni wireless, la progettazione di circuiti e l'ottimizzazione dei flussi di rete. Una copertura di vertici di un grafo è un insieme di vertici tale che ogni arco del grafo ha almeno un estremo in questo insieme. Il problema MWVC estende il problema standard della copertura di vertici assegnando un peso positivo a ciascun vertice e cercando una copertura di vertici con il peso totale minimo. Il problema MWVC è NP-completo, il che significa che non esiste un algoritmo in tempo polinomiale noto per trovare la soluzione ottimale in tutti i casi.

1.1 Definizione formale del MWVC

Dato un grafo non direzionato $G = (V, E)$ con i pesi assegnati a ogni vertice, l'obiettivo del Minimum Weight Vertex Cover è quello di cercare tra tutti i possibili Vertex Cover $S \subset V$, l'insieme $S*$ la cui somma dei vertici è minima.
La funzione obiettivo da minimizzare è: $\omega(S) = \sum_{v \in S} \omega(v)$
Sapendo che: $\forall (v_i, v_j) \in E, v_i, v_j \in S, S \in V$
Una soluzione S è valida e si chiama Vertex Cover se ogni arco in G ha almeno un vertice in S. L'obiettivo della MWVC è quello di minimizzare la funzione obiettivo $\omega(S)$.

2 Tabu Search algorithm

L'algoritmo Tabu Search è una tecnica che guida una procedura di ricerca euristica locale per esplorare lo spazio delle soluzioni superando gli ottimi locali. Ad ogni iterazione si considera un intorno dove sono state rimosse alcune soluzioni, definite tabù, sulla base di una memoria del processo di ricerca effettuato fino a quella iterazione e le soluzioni tabù non sono quindi raggiungibili all'iterazione successiva. Per "tecnica euristica" si intende la ricerca di una buona soluzione, accettabile senza necessariamente raggiungere la soluzione migliore assoluta, con un risparmio di risorse ed un risultato comunque soddisfacente. La Tabu Search si basa sulla premessa che la soluzione dei problemi deve incorporare una memoria adattabile e reagire durante l'esplorazione. L'uso di questa memoria contrasta con strutture "senza memoria" con quelle a "memoria rigida". L'esplorazione attiva della Tabu Search, sia in condizione deterministica che probabilistica, deriva dal presupposto che una cattiva scelta strategica può dare più informazioni rispetto a una buona scelta casuale. Il metodo di ricerca più conosciuto nel vicinato è quello di approssimare il valore minimo di una funzione f con valore reale in un insieme S detto metodo di discesa.

2.1 Definizione formale

Data una funzione $f(x)$ da ottimizzare su un insieme X, la Tabu Search inizia allo stesso modo ordinario della ricerca globale, procedendo iterativamente da un punto ad un altro fino a quando è soddisfatto un criterio di terminazione scelto. Ogni $x \in X$ ha associato un intorno di $N(x) \subset X$, e ogni soluzione x di N(x) è raggiungibile da x con un'operazione detta mossa. La ricerca va oltre l'insieme locale utilizzando una strategia di modifica di N(x) andando a sostituire il suo intorno N*(x). Se le soluzioni trovate non sono ottime vengono classificate come "tabù" e vengono rimosse dall'insieme N*(x). La gestione della memoria avviene tramite creazione di una o più tabù list, al loro interno si registrano gli attributi tabù-attivi e implicitamente o esplicitamente identificano il loro stato attuale. La durata di un attributo nel rimanere tabù-attivo si chiama tabù tenure e si misura in numero di iterazioni. Questa misura o livelli di tabù possono variare per i diversi tipi o combinazioni di attributi e possono anche variare nei diversi intervalli di tempo o fasi di ricerca.

2.2 Motivazioni della scelta dell'algoritmo

La Tabu Search è stata preferita agli altri algoritmi presenti per l'esperimento a causa di diversi motivi. La preferenza rispetto al Branch and Bound è dovuta all'assenza delle informazioni iniziali sulle istanze, come il numero di nodi e la struttura degli archi. Nonostante si tratti di un metodo esatto, la mancanza di queste informazioni e la sua inefficienza nei grafi di grandi dimensioni ne hanno determinato l'esclusione. Rispetto agli algoritmi genetici, la Tabu Search ha il vantaggio di essere più mirata e di richiedere meno memoria. Questo perchè gli algoritmi genetici lavorano con una popolazione di soluzioni che comporta un costo computazionale elevato, mentre la Tabu Search opera su una singola soluzione alla volta evitando il problema della gestione di molteplici percorsi contemporaneamente. La Tabu Search, invece, si è dimostrata una scelta più adatta grazie alla sua velocità ed efficienza una volta implementata correttamente la memoria. A differenza del Branch and Bound non punta necessariamente a trovare la soluzione ottima, ma è in grado di individuare soluzioni di altà qualità in tempi molto brevi. Nel contesto dei MWVC, la soluzione sarà comunque ottimale, poiché i pesi dei vertici verranno analizzati e minimizzati. Di conseguenza, otterremo un metodo in grado di trovare l'ottimo in tempi brevi. Grazie alla lista Tabù questa ricerca permette di evitare di tornare su soluzioni già visitate, riducendo il rischio di cicli e migliorando l'efficacia della ricerca.

3 Sviluppo dell'algoritmo

L'algoritmo è stato sviluppato in due fasi differenti. La differenza principale tra i due script è l'ottimizzazione delle iterazioni e della memoria e la differente struttura di quest'ultima.

3.1 Primo sviluppo

Entrambi gli algoritmi sono stati sviluppati in Python (versione 3.11.5). Sono presenti diversi metodi, ognuno col proprio scopo, i principali vengono analizzati nei successivi paragrafi.

3.1.1 Lettura del grafo

```
Funzione LetturaGrafo(FileName)
    lines = FileName
    n = lines[0]
    weights = lines[1]
    matrix = lines[2:]
    Return n, weights, matrix
```

Questo metodo legge un file e suddivide i dati in strutture appropriate. In particolare, weights è una lista che contiene i pesi associati ai vertici, mentre matrix è un array NumPy composto da liste, ciascuna rappresentante una riga della matrice di adiacenza. Quest'ultima descrive i collegamenti tra i vertici del grafo.

3.1.2 Generazione dei vicini

```
Funzione GeneraVicini(solution, n, matrix):
    neighbors = lista vuota
    soluzione_set = converti solution in un set
    # Rimozione di un nodo dalla soluzione
    Per ogni v in solution:
        new_solution = soluzione_set - {v}
        Se is_valid_cover(new_solution, matrix, n) allora
            neighbors add new_solution
    # Aggiunta di un nodo alla soluzione
    Per ogni v in {Tutti i nodi} - soluzione_set:
        new_solution = soluzione_set + {v}
        Se is_valid_cover(new_solution, matrix, n) allora
            neighbors add new_solution
    Return neighbors
```

La funzione GeneraVicini genera tutte le possibili soluzioni vicine a solution rimuovendo o aggiungendo un nodo alla soluzione corrente. Solution viene gestita in un set per facilitarne le operazioni di aggiunta e rimozione dei nodi. L'obiettivo dei due cicli è quello di cercare le soluzioni possibili andando prima a rimuovere i nodi e poi aggiungendo quelli assenti, in modo tale da creare una lista valida di vicini.

3.1.3 Tabu Search

```
Funzione TabuSearch(n, weights, matrix, max_iter,
tabu_tenure, max_no_improvement):
    # Inizializzazione
    current_solution = lista dei vertici
    best_solution, best_cost = current_solution, costo(current_solution)
    tabu_list = deque di lunghezza tabu_tenure
    no_improvement_count, visited_solution = 0, set vuoto
    # Tabu Search
    Per i in max_iter:
        neighbors = GeneraVicini(current_solution, n, matrix)
        Se neighbors è vuoto allora break
        best_neighbor, best_neighbor_cost = vuoto, infinito
        Per neighbor in neighbors:
            neighbor_tuple = sorted(neighbor)
            Se neighbor_tuple non è in tabu_list e visited_solution:
                cost = costo(neighbor)
                Se cost < best_neighbor_cost:
                    best_neighbor, best_neighbor_cost = neighbor, cost
        Se best_neighbor è vuoto:
            no_improvement_count += 1
        Altrimenti:
            current_solution = best_neighbor
            neighbor_tuple = sorted(best_neighbor)
            tabu_list + neighbor_tuple
            visited_solution + neighbor_tuple
            Se best_neighbor_cost < best_cost:
                best_solution, best_cost, no_improvement_count =
                best_neighbor, best_neighbor_cost, 0
            Altrimenti:
                no_improvement_count += 1
        Se no_improvement_count >= max_no_improvement allora break
    Return best_solution, best_cost
```

Lo pseudocodice soprastante implementa l'algoritmo della Tabu Search. Ad ogni iterazione genera le soluzioni vicine e seleziona la migliore non presente nella lista tabù o nella lista delle soluzioni già visitate. Se la soluzione trovata è

migliore dell'attuale soluzione allora la sostituisce, altrimenti riprova a cercare una soluzione migliore fin quanto il valore di terminazione non raggiunge il limite di iterazioni. La lista degli elementi tabù impedisce di visitare soluzioni già visistate per evitare i cicli. L'algoritmo può terminare quando raggiunge il limite di iterazioni o quando non si migliora il valore attuale per un numero di iterazioni prestabilito.

3.1.4 Risultati ottenuti

Il primo sviluppo ha ottenuto ottimi risultati, trovando soluzioni ottimali in poche iterazioni. Tuttavia, il secondo sviluppo è stato introdotto per migliorarne l'efficienza, poiché il primo risultava troppo lento nel raggiungere l'ottimo globale, soprattutto con grafi di grandi dimensioni e numerosi archi. Questa lentezza è dovuta alle ripetute chiamate a determinate funzioni in ogni ciclo e a un uso non ottimizzato della memoria.

3.2 Secondo sviluppo

Il secondo sviluppo è stato implementato per ottimizzare il primo, concentrandosi sul miglioramento dell'uso della memoria e sulla riduzione delle chiamate superflue e dei passaggi ripetuti. Di seguito verranno analizzati i miglioramenti effettuati.

3.2.1 Lettura del grafo

Durante la lettura del grafo, il cambiamento principale riguarda l'uso di un array NumPy per i pesi dei nodi, al posto di una semplice lista. Questa modifica migliora le prestazioni, poiché gli array NumPy sono più efficienti e veloci rispetto alle liste standard di Python. Inoltre, gli array NumPy permettono di eseguire operazioni come somma e sottrazione senza la necessità di cicli espliciti, semplificando il codice e ottimizzando i calcoli.

3.2.2 Generazione dei vicini

All'interno di questa funzione sono stati apportati pochi ma significativi cambiamenti. La principale differenza riguarda l'uso delle operazioni sui set: nel primo codice veniva creata una copia del set in entrambi i cicli, mentre in questa versione le operazioni vengono eseguite direttamente sugli operatori dei set, senza la necessità di copiare l'intero set.

3.2.3 Tabu Search

Nel metodo principale del codice sono state introdotte diverse ottimizzazioni. In primo luogo, è stato utilizzato frozenset al posto di tuple per rappresentare le soluzioni visitate e i vicini. I frozenset sono immutabili, più efficienti nelle operazioni di ricerca e possono essere usati come chiavi in strutture come set e deque, risolvendo anche il problema dei duplicati nelle soluzioni. Un'altra importante modifica riguarda l'introduzione di una heap per la gestione dei vicini, che permette di selezionare il miglior vicino con il costo più basso in modo più rapido ed efficiente, evitando il confronto manuale tra tutti i vicini. Queste modifiche migliorano la gestione delle soluzioni, riducono il numero di confronti e aumentano l'efficienza dell'algoritmo, rendendolo significativamente più performante.

3.2.4 Altri metodi

- La funzione `is_valid_cover` è migliore della prima implementata perché sfrutta le operazioni vettoriali di numpy, che rendono il processo di verifica della copertura significativamente più veloce ed efficiente. Le operazioni bitwise consentono di eseguire il controllo in parallelo su intere righe e colonne della matrice, riducendo così la complessità computazionale e migliorando le prestazioni, soprattutto con grafi di grandi dimensioni. Inoltre, la funzione np.all() permette di verificare rapidamente se tutte le coperture sono soddisfatte in un unico passaggio;

- La funzione `evaluate_solution` è migliore perché utilizza np.sum(), una funzione ottimizzata di numpy, per sommare gli elementi dell'array dei pesi. Questa versione è più efficiente in quanto esegue la somma direttamente sugli array senza passaggi aggiuntivi, ovvero, ciò che succedeva all'interno del primo sviluppo.

3.2.5 Risultati ottenuti

Le ottimizzazioni apportate hanno reso l'algoritmo più efficiente, garantendo prestazioni adeguate al carico di lavoro. Nonostante il numero di iterazioni rimanga invariato per i diversi grafi, l'algoritmo fornisce comunque la soluzione ottima in tempi ridotti rispetto al primo sviluppo.

4 Risultati finali

Di seguito sono riportati i risultati ottenuti per tutte le istanze dell'esperimento. Per ragioni di efficienza, è stato utilizzato solo il secondo sviluppo, avendo verificato su grafi campione che il primo sviluppo produce gli stessi risultati. Le istanze sono presenti all'interno dei file txt così formati:

- Prima riga: numero di vertici;
- Seconda riga: pesi dei vertici;
- Dalla terza riga: matrice degli archi grande N x N dove N è il numero dei vertici. Sono presenti 0 quando non è presente un arco e 1 quando l'arco è presente.

4.1 Grafo 800 Vertici, 10000 Archi

Questo è il grafo più complesso tra le istanze analizzate, con 800 vertici e 10.000 archi, risultando significativamente più impegnativo rispetto agli altri grafi dell'esperimento. Entrambi gli algoritmi hanno individuato la soluzione ottima, con un costo di 46.393 e un totale di 113 iterazioni, completate in soli 128.0274 secondi (circa 2 minuti e 8 secondi) utilizzando il secondo sviluppo. Nella figura di seguito è mostrato il grafo in tutta la sua complessità. I nodi in blu sono quelli non presenti all'interno della MWVC, mentre quelli in rosso sono presenti.

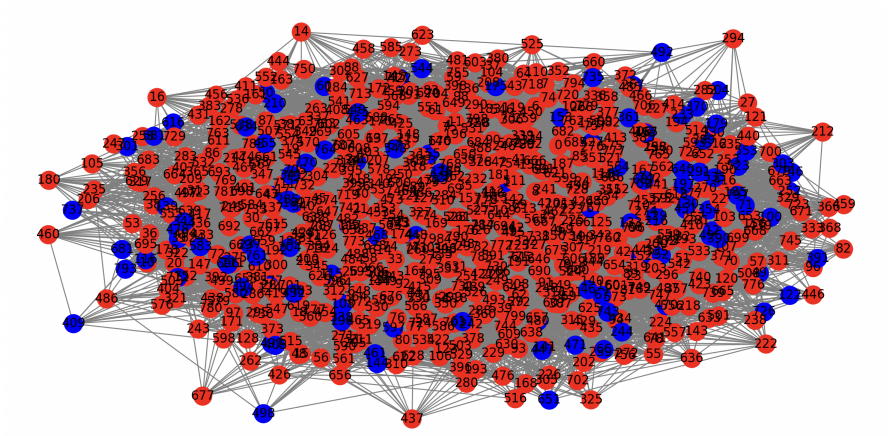
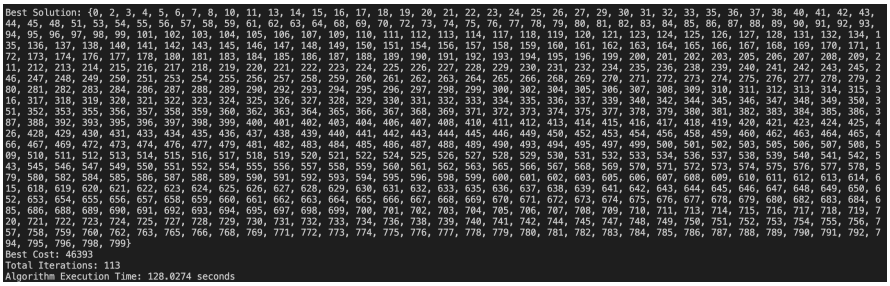


Fig. 1: Grafo vc_800_10000

La soluzione del vertex cover è la seguente:



4.2 Altri grafi

File	Iterazioni	Tempo(Secondi)	Best cost	Iterazioni/Secondo
vc_20_60_01	12	0,0053	774	2264,15
vc_20_60_02	11	0,0043	1035	2558,14
vc_20_60_03	14	0,0059	730	2372,88
vc_20_60_04	11	0,0051	775	2156,86
vc_20_60_05	15	0,0058	871	2586,21
vc_20_60_06	11	0,0044	907	2500
vc_20_60_07	13	0,0052	972	2500
vc_20_60_08	10	0,0043	1085	2325,58
vc_20_60_09	11	0,0044	980	2500
vc_20_60_10	11	0,0044	960	2500
vc_20_120_01	9	0,0041	910	2195,12
vc_20_120_02	11	0,0046	1062	2391,3
vc_20_120_03	12	0,0053	1061	2264,15
vc_20_120_04	9	0,0046	1050	1956,52
vc_20_120_05	9	0,004	997	2250
vc_20_120_06	8	0,0034	961	2352,94
vc_20_120_07	8	0,0034	1063	2352,94
vc_20_120_08	9	0,0038	1162	2368,42
vc_20_120_09	8	0,0035	1271	2285,71
vc_20_120_10	9	0,0038	1133	2368,42
vc_25_150_01	12	0,0075	1403	1600
vc_25_150_02	11	0,0068	1132	1617,65
vc_25_150_03	10	0,0062	1305	1612,9
vc_25_150_04	10	0,0066	1483	1515,15
vc_25_150_05	11	0,0068	1311	1617,65
vc_25_150_06	9	0,0057	1249	1578,95
vc_25_150_07	13	0,0082	1598	1585,37
vc_25_150_08	11	0,0075	1402	1466,67
vc_25_150_09	10	0,0063	1366	1587,3
vc_25_150_10	10	0,0065	1167	1538,46
vc_100_500_01	28	0,2559	4795	109,42
vc_100_500_02	28	0,257	5083	108,95
vc_100_500_03	29	0,2634	4705	110,1
vc_100_500_04	29	0,2636	4803	110,02
vc_100_500_05	29	0,2638	5225	109,93
vc_100_500_06	29	0,2636	5065	110,02
vc_100_500_07	28	0,2581	5050	108,49
vc_100_500_08	28	0,2564	4889	109,2
vc_100_500_09	32	0,2859	4595	111,93
vc_100_500_10	28	0,2552	4777	109,72
vc_100_2000_01	13	0,1301	6260	99,92
vc_100_2000_02	12	0,1186	6110	101,18
vc_100_2000_03	12	0,1183	5930	101,44
vc_100_2000_04	14	0,1373	6077	101,97
vc_100_2000_05	13	0,1279	6533	101,64
vc_100_2000_06	16	0,1593	6158	100,44
vc_100_2000_07	13	0,128	6568	101,56
vc_100_2000_08	17	0,1662	6343	102,29
vc_100_2000_09	13	0,1291	6045	100,7
vc_100_2000_10	14	0,1367	6410	102,41
vc_200_750_01	63	2,4573	9164	25,64
vc_200_750_02	56	2,2155	9014	25,28
vc_200_750_03	66	2,552	9058	25,86
vc_200_750_04	66	2,5659	8330	25,72
vc_200_750_05	64	2,48	9559	25,81
vc_200_750_06	66	2,5477	8626	25,91
vc_200_750_07	63	2,4583	8271	25,63
vc_200_750_08	62	2,4126	8835	25,7
vc_200_750_09	61	2,3909	9122	25,51
vc_200_750_10	63	2,4589	8907	25,62
vc_200_3000_01	24	1,0377	12121	23,13
vc_200_3000_02	28	1,187	11494	23,59
vc_200_3000_03	25	1,0689	12287	23,39
vc_200_3000_04	25	1,0696	13071	23,37
vc_200_3000_05	30	1,258	11763	23,85
vc_200_3000_06	28	1,1889	11725	23,55
vc_200_3000_07	25	1,1459	11985	21,82
vc_200_3000_08	27	1,1823	12042	22,84
vc_200_3000_09	28	1,1949	12203	23,43
vc_200_3000_10	28	1,1871	11684	23,59
vc_800_10000	113	128,0274	46393	0,88

4.3 Differenza di prestazione tra i due sviluppi

In questo paragrafo saranno analizzate le differenze di prestazioni tra i due sviluppi, considerando tre grafi di diversa complessità: uno semplice, uno di media difficoltà e uno complesso.

1. vc_20_60_01: questo grafo è formato da 20 nodi pesati e 60 archi totali:
   - Primo sviluppo: 12 iterazioni in 0.0130 secondi;
   - Secondo sviluppo: 12 iterazioni in 0.0050 secondi.
2. vc_100_500_01: questo grafo è formato da 100 nodi pesati e 500 archi totali:
   - Primo sviluppo: 28 iterazioni in 3.0399 secondi;
   - Secondo sviluppo: 28 iterazioni in 0.2552 secondi.
3. vc_200_3000_01: questo grafo è formato da 200 nodi pesati e 3000 archi totali:
   - Primo sviluppo: 24 iterazioni in 25.1012 secondi;
   - Secondo sviluppo: 24 iterazioni in 1.0292 secondi.

Questi test dimostrano chiaramente che, nei casi più semplici, il secondo sviluppo è 2.6 volte più veloce del primo. Con l'aumentare della complessità, il divario si amplifica: nel caso di media difficoltà, il secondo sviluppo risulta 11.91 volte più rapido, mentre nel grafo più complesso la velocità aumenta fino a 24.4 volte. Questo risultato conferma l'efficacia dell'ottimizzazione effettuata, che ha permesso un significativo miglioramento delle prestazioni dell'algoritmo lasciando inalterata la sua efficacia.

4.4 Differenza nell'uso dell'elemento tabu_tenure

Per completezza, è stata effettuata una prova con diversi valori di `tabu_tenure`. Durante tutti gli esperimenti, il parametro è stato impostato a 8, un valore considerato ottimale in quanto bilancia la possibilità di esplorare soluzioni ottime evitando al contempo di creare una lista tabù troppo grande. È stato anche testato un valore di `tabu_tenure` pari a 2, riscontrando un lieve rallentamento nei grafi con più nodi, con un incremento di circa 10 secondi sull'intera esecuzione. Aumentando ulteriormente il `tabu_tenure` a 20, si è registrato un rallentamento di circa 3 secondi. Tuttavia, in entrambi i casi, il miglior valore è stato ottenuto con lo stesso numero di iterazioni di quando si utilizza il valore di 8.

5 Conclusioni

Il progetto ha analizzato due algoritmi basati su Tabu Search per affrontare il problema del Minimum Weight Vertex Cover (MWVC), ottenendo risultati notevoli. Il primo algoritmo, sebbene più lento, ha dimostrato l'efficacia della Tabu Search nel trovare la soluzione ottima in poche iterazioni, anche se il tempo di esecuzione può rappresentare un limite significativo nei grafi di grandi dimensioni. Il secondo algoritmo, ottimizzato per migliorare le prestazioni, ha ridotto drasticamente i tempi di calcolo, mantenendo comunque la capacità di raggiungere il valore ottimo in poche iterazioni. Quest'ottimizzazione si è rivelata particolarmente vantaggiosa per grafi di dimensioni elevate. Durante l'esperimento, è emerso che, nei grafi con pochi nodi, la differenza tra i due algoritmi è netta, ma anche il primo algoritmo ha mostrato ottime prestazioni, con tempi di esecuzione nell'ordine dei centesimi di secondo. Tuttavia, al crescere del numero di nodi, la differenza diventa più evidente, con l'algoritmo ottimizzato che garantisce prestazioni notevolmente migliori, riducendo il tempo di risposta da secondi a frazioni di secondo. In particolare, l'algoritmo ottimizzato potrebbe essere impiegato in applicazioni quasi in tempo reale per grafi con meno di 100 vertici, mentre per grafi superiori a 200 vertici si osserva un aumento del ritardo, con un tempo di risposta minimo di circa 2 secondi. Questi risultati evidenziano come la scelta dell'algoritmo Tabu Search sia una soluzione particolarmente efficace per il problema del MWVC, rivelandosi una scelta vincente per l'esecuzione dell'esperimento.

6 Riferimenti

[1] Vertex Cover (2024)

[2] L'algoritmo "Tabu Search" per l'ottimizzazione, teoria ed applicazioni (2012)

[3] IA-Project, Luigi Priano (2025)