



UNIVERSITÀ
degli STUDI
di CATANIA

Intelligenza Artificiale e Laboratorio Vertex Cover Solver

Giuseppe Napoli - 1000012802

A.A. 2024/25

Docenti

- Prof. Vincenzo Cutello
- Prof. Mario Francesco Pavone

Indice

1	Introduzione	3
2	Algoritmo	4
2.1	Panoramica Generale	4
2.2	Struttura Della Tabu Search	4
2.3	Pseudocodice	5
2.4	Esecuzione e Parametri Principali	6
3	Scelte Progettuali	7
3.1	Struttura della Soluzione e Gestione della Memoria Tabu	7
3.2	Generazione del Vicinato e Riflessioni sulle Scelte Progettuali	8
3.3	Scelte di Implementazione e Linguaggio	8
4	Risultati Sperimentali	9
4.1	Setup Sperimentale	9
4.2	Confronto tra Diverse Configurazioni di Parametri	9
4.3	Commenti sui Risultati	10
5	Conclusioni	11

1 Introduzione

Il *Vertex Cover* è un problema classico della teoria dei grafi e dell'ottimizzazione combinatoria, in cui si cerca un insieme di vertici tale da *coprire* (o intercettare) tutti gli archi di un grafo. Nella sua variante pesata (*Weighted Vertex Cover*), ad ogni vertice è associato un costo (o peso) e l'obiettivo consiste nel minimizzare la somma dei pesi dei vertici selezionati, garantendo al contempo la copertura di tutti gli archi. Questo problema risulta *NP-hard*, rendendolo particolarmente sfidante dal punto di vista computazionale, soprattutto per istanze di grandi dimensioni.

Nel presente progetto, si è scelto di affrontare il *Weighted Vertex Cover* mediante una *metaeuristica di tipo Tabu Search*, al fine di ottenere soluzioni di buona qualità in tempi ragionevoli, anche su istanze con dimensioni non trascurabili. La *Tabu Search*, introdotta da Fred Glover, è un approccio iterativo che combina una ricerca locale sistematica con una struttura di memoria (detta *tabu*) in grado di gestire sia la cosiddetta *short-term memory* (con lo scopo di evitare cicli o ritorni su soluzioni già visitate), sia la *long-term memory* (orientata ad aumentare la diversificazione e l'esplorazione dello spazio delle soluzioni).

Nel prosieguo, verranno illustrate le principali caratteristiche dell'algoritmo, le scelte progettuali e implementative adottate — con particolare attenzione ai parametri chiave (*maxIterations*, *tabuTenure* e *maxNoImprovement*) — e i risultati sperimentali ottenuti. Verranno infine discussi possibili sviluppi futuri e riflessioni personali emerse dal lavoro svolto.

2 Algoritmo

2.1 Panoramica Generale

L'algoritmo adottato è una *Tabu Search*, una metaeuristica che evita cicli memorizzando le mosse recenti in una *tabu list*. Come descritto nell'Introduzione, il problema viene affrontato tramite una ricerca locale iterativa, partendo da una soluzione iniziale e modificandola per minimizzare il costo.

Ad ogni iterazione, vengono generate soluzioni vicine valutate tramite una funzione costo. La *tabu list* impedisce di ripetere modifiche recenti per un numero prefissato di iterazioni, bilanciando esplorazione e ottimizzazione. Nei prossimi paragrafi verranno descritti i dettagli implementativi dell'algoritmo.

2.2 Struttura Della Tabu Search

L'algoritmo si basa su una ricerca locale iterativa, mantenendo una soluzione corrente e aggiornandola esplorando il vicinato. La qualità di ogni soluzione è valutata tramite una funzione costo, mentre una *tabu list* impedisce il ripristino di mosse recenti, favorendo l'esplorazione di nuove configurazioni.

Gli elementi principali dell'algoritmo includono:

- **Soluzione corrente:** insieme di vertici selezionati nel *vertex cover*.
- **Migliore soluzione:** la configurazione con costo minimo trovata finora.
- **Tabu list:** insieme di mosse vietate per un numero limitato di iterazioni.
- **Vicinato:** soluzioni ottenute aggiungendo o rimuovendo vertici.

L'algoritmo prosegue fino al raggiungimento di un criterio di arresto, basato su un numero massimo di iterazioni o sull'assenza di miglioramenti per un intervallo predefinito. Il funzionamento dettagliato, inclusa la gestione della *tabu list* e dei parametri principali (*tabuTenure*, *maxIterations*, *maxNoImprovement*), è illustrato nello pseudocodice della sezione successiva.

Tra i parametri fondamentali troviamo:

- ***maxIterations*:** numero massimo di iterazioni complessive.
- ***tabuTenure*:** numero di iterazioni per cui una mossa rimane nella *tabu list*.
- ***maxNoImprovement*:** numero massimo di iterazioni consecutive senza miglioramenti, oltre cui fermare l'algoritmo.

2.3 Pseudocodice

L'algoritmo segue un ciclo iterativo in cui si generano soluzioni vicine e si applicano criteri di selezione basati sulla *tabu list* e sulla qualità della soluzione. La funzione di valutazione guida la ricerca, mentre i criteri di aspirazione permettono di ignorare vincoli tabu se si trova una soluzione migliore.

Il funzionamento è riassunto nel seguente pseudocodice:

Listing 1: Pseudocodice della Tabu Search

```
FUNCTION TABUSEARCH(Graph, maxIter, tabuTenure,
maxNoImprovement):
    bestSolution = INITIAL_SOLUTION(Graph)
    currentSolution = bestSolution.copy()
    tabuList = emptySet
    iterations = 0
    noImprovement = 0

    WHILE iterations < maxIter AND noImprovement <
maxNoImprovement:
        neighbors = GENERATE_NEIGHBORS(currentSolution)
        bestNeighbor = SELECT_BEST(neighbors, tabuList,
bestSolution)

        IF bestNeighbor is NULL:
            BREAK

        UPDATE_TABU_LIST(tabuList)
        tabuList.add(DETERMINE_MOVE(currentSolution,
bestNeighbor), tabuTenure)

        currentSolution = bestNeighbor.copy()

        IF cost(currentSolution) < cost(bestSolution):
            bestSolution = currentSolution.copy()
            noImprovement = 0
        ELSE:
            noImprovement += 1

        iterations += 1
```

RETURN bestSolution

Le principali operazioni sono:

- INITIAL_SOLUTION: genera una configurazione iniziale ammissibile.
- GENERATE_NEIGHBORS: produce soluzioni vicine.
- SELECT_BEST: sceglie la soluzione migliore rispettando le regole tabu.
- UPDATE_TABU_LIST: aggiorna la memoria tabu rimuovendo mosse scadute.
- DETERMINE_MOVE: identifica la modifica necessaria per la transizione tra soluzioni.

L'algoritmo si arresta se il numero massimo di iterazioni è raggiunto o se non si verificano miglioramenti per un certo numero di iterazioni consecutive.

2.4 Esecuzione e Parametri Principali

In fase di esecuzione, l'algoritmo arresta la ricerca in presenza di due condizioni:

1. Il *numero totale di iterazioni* raggiunge *maxIterations*.
2. Non si osservano miglioramenti alla *bestSolution* per più di *maxNoImprovement* iterazioni consecutive.

Le prestazioni (sia in termini di tempo di esecuzione che di qualità delle soluzioni) dipendono fortemente dalla scelta di parametri come *maxIterations*, *tabuTenure*, *maxNoImprovement* e dalla definizione delle mosse nel *vicinato*.

In questo lavoro, si è analizzato un range di valori per i parametri chiave (ad esempio, *tabuTenure* variato tra 5 e 7, *maxNoImprovement* tra 10 e 30), confrontando i risultati sia in termini di costi ottenuti sia di stabilità dell'algoritmo. I dettagli di tali sperimentazioni sono riportati nel Capitolo 4.

3 Scelte Progettuali

In questo capitolo vengono descritte e motivate le principali scelte progettuali e implementative che sono state adottate nello sviluppo dell'algoritmo di Tabu Search per il *Weighted Vertex Cover*. Tali scelte riguardano sia la configurazione dei parametri, sia la modalità di gestione della memoria *Tabu* (short-term e/o long-term), fino alle strategie di generazione del vicinato.

3.1 Struttura della Soluzione e Gestione della Memoria Tabu

La soluzione è rappresentata come un vettore binario di dimensione pari al numero di nodi del grafo, dove un valore "1" indica un nodo incluso nel *vertex cover*, mentre "0" lo esclude. Il costo di una soluzione è dato dalla somma dei pesi dei nodi selezionati.

L'algoritmo utilizza una *tabu list* per gestire la memoria a breve termine e prevenire cicli. Questa viene implementata come una mappa che associa ogni mossa proibita (es. aggiunta/rimozione di un nodo) a un contatore di scadenza. Ad ogni iterazione:

1. Si decremetano i contatori delle mosse presenti nella *tabu list*.
2. Si rimuovono le mosse scadute.
3. Si aggiunge la mossa appena effettuata con una durata pari a `tabuTenure`.

I principali parametri di controllo dell'algoritmo sono:

- `maxIterations`: numero massimo di iterazioni.
- `tabuTenure`: durata di una mossa nella *tabu list*.
- `maxNoImprovement`: numero massimo di iterazioni senza miglioramenti prima di interrompere la ricerca.

Questa gestione consente all'algoritmo di evitare soluzioni già esplorate e favorire la diversificazione della ricerca. L'uso della memoria *long-term*, basata sulla frequenza con cui un nodo è selezionato, è stato considerato ma non implementato in modo esteso.

3.2 Generazione del Vicinato e Riflessioni sulle Scelte Progettuali

Il vicinato è generato rimuovendo un nodo dal *vertex cover*, purché la soluzione resti ammissibile, o aggiungendo un nodo se necessario per coprire archi scoperti. Si adotta un criterio *strictly feasible*, evitando mosse che renderebbero la soluzione non valida.

Questa strategia consente di mantenere soluzioni ammissibili e di esplorare lo spazio in modo controllato, evitando modifiche eccessivamente distruttive. La Tabu Search, grazie alla *tabu list*, permette di bilanciare esplorazione e intensificazione della ricerca.

Le scelte progettuali adottate si sono dimostrate efficaci nel garantire stabilità e qualità delle soluzioni, evitando minimi locali evidenti e mantenendo tempi di esecuzione contenuti anche su istanze di grandi dimensioni.

3.3 Scelte di Implementazione e Linguaggio

L'algoritmo è stato implementato in Java per sfruttare strutture dati efficienti e garantire buone prestazioni su istanze di grandi dimensioni. La *tabu list* è gestita con una `HashMap<String, Integer>`, in cui la chiave rappresenta una mossa (es. "REMOVE 5") e il valore indica il numero di iterazioni rimanenti prima della sua rimozione.

L'uso di un linguaggio compilato garantisce tempi di esecuzione più rapidi rispetto a linguaggi interpretati, riducendo l'overhead di gestione della memoria. La scelta di Java ha permesso inoltre una gestione semplice delle strutture dati e della modularità del codice.

4 Risultati Sperimentali

In questo capitolo vengono presentati e discussi i risultati ottenuti dall'implementazione dell'algoritmo *Tabu Search* descritto nei capitoli precedenti. Per valutare il comportamento dell'algoritmo, si sono condotti test su diverse istanze del problema, variando i parametri chiave come *tabuTenure*, *maxNoImprovement* e la dimensione dei grafi.

4.1 Setup Sperimentale

Le prove sono state eseguite su una macchina con le seguenti specifiche:

- **CPU:** Intel Core i7-1065G7 (1.30 GHz)
- **RAM:** 16 GB
- **Sistema Operativo:** WSL Ubuntu 20.04 (64 bit)
- **Linguaggio:** Java (OpenJDK 17)

Le istanze di test comprendono grafi con numero di nodi (n) variabile tra 20 e 800, e numero di archi (m) tra 60 e 10000. I pesi dei nodi sono interi positivi.

4.2 Confronto tra Diverse Configurazioni di Parametri

Per capire come i parametri influenzino le prestazioni, sono state confrontate diverse configurazioni di *tabuTenure* (5, 7), *maxNoImprovement* (10, 20, 30) e *maxIterations* (500, 1000, 2000). In ciascun caso, l'algoritmo è stato eseguito 5 volte su ogni istanza, per ridurre la variabilità stocastica. A titolo di esempio, la tabella 1 riporta i risultati medi su un insieme di 3 istanze (piccola, media e grande).

Dall'osservazione dei dati in tabella 1 emergono due punti essenziali:

- **Dimensioni del grafo come fattore dominante:** passando da (20, 60) a (800, 10000) nodi/archi, si registra un salto netto dei tempi di esecuzione (dell'ordine di microsecondi-millisecondi a svariati secondi).
- ***maxNoImprovement* come parametro più incisivo:** a parità di *maxIterations* e *tabuTenure*, valori minori di *maxNoImprovement* (in particolare 10) mostrano i migliori tempi d'esecuzione, mentre configurazioni con *maxNoImprovement* più alto fanno salire i tempi, seppur meno drasticamente del puro aumento di dimensione del grafo.

Table 1: Risultati medi (5 run) per diverse configurazioni di *maxIterations*, *tabuTenure* e *maxNoImprovement*.

Config.	n	m	maxIter.	tabuTenure	maxNoImpr.	Time (s)
C1	20	60	500	5	10	0.0092
C4			500	7	20	0.0148
C7			1000	5	20	0.0156
C10			1000	10	30	0.021
C13			2000	7	10	0.0134
C16			2000	10	20	0.0168
C2	100	500	500	5	10	0.0608
C5			500	7	20	0.108
C8			1000	5	20	0.0836
C11			1000	10	30	0.0818
C14			2000	7	10	0.062
C17			2000	10	20	0.0926
C3	800	10000	500	5	10	6.1008
C6			500	7	20	6.7292
C9			1000	5	20	6.7994
C12			1000	10	30	7.2512
C15			2000	7	10	6.856
C18			2000	10	20	7.54

4.3 Commenti sui Risultati

Nel complesso, la Tabu Search così configurata è risultata:

- **Affidabile:** produce soluzioni di buona qualità con bassa varianza fra diversi run.
- **Poco sensibile** ai parametri: variazioni dei parametri non cambiano in modo significativo i risultati, come visto in tabella 1.
- **Scalabile** fino a grafi di alcune migliaia di archi, con tempi di esecuzione che rimangono in un range accettabile (alcuni secondi).

Una eventuale *long-term memory*, basata sul conteggio di frequenza con cui i nodi appaiono o spariscono dal cover, potrebbe migliorare ulteriormente la diversificazione su istanze molto grandi, sebbene ciò comporti una maggiore complessità implementativa. Nel prossimo capitolo (5) verranno esposti alcuni spunti di sviluppo e riflessioni conclusive sull'algoritmo e le sue prospettive future.

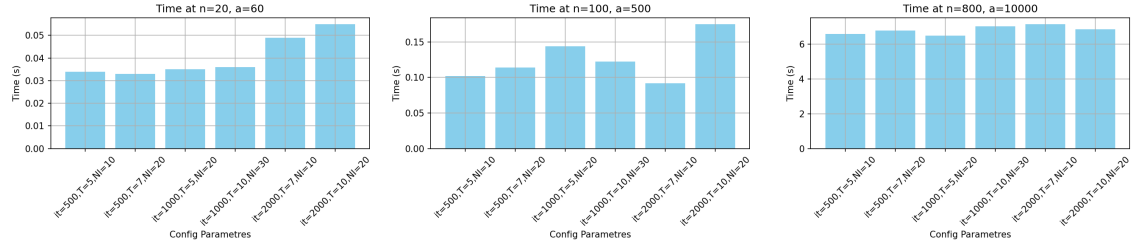


Figure 1: Tempo di esecuzione medio in funzione della dimensione del grafo e della configurazione dei tre parametri.

5 Conclusioni

Nel corso di questo progetto, è stato sviluppato un *solver* per il problema del *Weighted Vertex Cover* basato su una metaeuristica di tipo *Tabu Search*. L'obiettivo principale era esplorare e applicare un approccio combinatorio in grado di trovare soluzioni di buona qualità in tempi computazionali ragionevoli, anche su istanze di dimensioni intermedie o relativamente grandi.

Riepilogo del Lavoro Svolto

Il lavoro ha compreso:

1. La definizione di una *soluzione* come insieme di nodi (o vettore booleano) che copre tutti gli archi del grafo.
2. L'implementazione di una *Tabu Search*, con meccanismi di:
 - **Short-term memory**, per evitare il ritorno immediato a soluzioni o mosse già visitate di recente.
 - **Aspiration criterion**, che consente di ignorare lo stato *tabu* se la soluzione candidata migliora la migliore soluzione globale.
3. L'analisi dei *parametri* chiave dell'algoritmo (*tabuTenure*, *maxIterations*, *maxNoImprovement*) e l'effetto della loro variazione sui risultati.
4. Test sperimentali su istanze di diversa dimensione, con valutazione e confronto delle prestazioni (*Best Cost* trovato, tempi medi, stabilità delle soluzioni).

Valutazione delle Scelte Progettuali

Le strategie adottate — in particolare la definizione del *vicinato*, la gestione *move-based* della Tabu List e il criterio di arresto basato su *maxNoImprovement* — hanno dimostrato di produrre soluzioni di qualità. La Tabu Search, infatti, si è rivelata capace di evitare rapidamente i *minimi locali* più ovvi e di diversificare sufficientemente la ricerca, specialmente quando si ottimizza il valore di *tabuTenure*.

Sviluppi Futuri

Tra le possibili estensioni o linee di lavoro future si possono evidenziare:

- **Integrazione di una Long-term Memory:** per catturare le frequenze di aggiunta/rimozione di ciascun nodo e favorire una *diversificazione* più ampia su istanze di grandi dimensioni.
- **Ibridazione con altre metaeuristiche:** ad esempio, un *framework* che combini una fase di *Genetic Algorithm* per generare popolazioni di soluzioni, seguita da un *local search* Tabu Search per il refinement di ciascun individuo.
- **Parallelizzazione:** molte fasi della Tabu Search (generazione del vicinato, valutazione del costo) possono essere eseguite in parallelo, sfruttando architetture multicore o *GPU*.
- **Benchmark su istanze molto grandi:** sarebbe interessante testare l'algoritmo su grafi con decine di migliaia di nodi e valutare le strategie di pruning o di riduzione del vicinato per abbattere i tempi di calcolo.

Considerazioni Finali

Nel complesso, l'esperienza acquisita con questo progetto evidenzia la validità delle metaeuristiche come strumento efficace per problemi NP-hard. Nonostante la Tabu Search non garantisca l'ottimalità, le soluzioni ottenute risultano competitive rispetto ad approcci euristici più semplici, specialmente su istanze di dimensioni medio-grandi. Il lavoro svolto dimostra come la personalizzazione dei parametri e la cura della struttura *tabu* siano fondamentali per ottenere le migliori prestazioni possibili, aprendo la strada a numerosi sviluppi futuri.