Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

PLI e MAXSAT:
Analisi Comparativa su Problemi di Ottimizzazione Combinatoria

Relatore:
Chiar.mo Prof.
UGO DAL LAGO

Presentata da:
SARA CASADIO

II Sessione
Anno Accademico 2023/2024

A mamma e papà,
A mio fratello Luca

Indice

Introduzione **6**

1 Modelli **10**

1.1 Problema del Cammino Minimo su grafo pesato 10

1.1.1 Formulazione del Problema del Cammino Minimo in Programmazione Lineare Intera 12

1.1.2 Formulazione del Problema del Cammino Minimo in MaxSAT 14

1.2 Problema dello Zaino 16

1.2.1 Formulazione del Problema dello Zaino in Programmazione Lineare Intera 16

1.2.2 Formulazione del Problema dello Zaino in MaxSAT . 17

1.3 Problema dell'Insieme Indipendente Massimo 18

1.3.1 Formulazione del Problema dell'Insieme Indipendente in Programmazione Lineare Intera 18

1.3.2 Formulazione del Problema dell'Insieme Indipendente in MaxSAT 18

1.4 NP-Completezza dei Modelli Considerati 19

2 Solvers utilizzati **21**

2.1 Solvers per Programmazione Lineare Intera 21

2.1.1 Solver glpsol 21
2.1.2 Solver CBC 22
2.2 Solvers per Soddisfacibilità Booleana Massima 23
2.2.1 Solver RC2 23
2.2.2 Solver Open-wbo 24

3 Scripts e Procedure di Test **26**
3.1 Problema del Cammino Minimo 26
3.1.1 Generazione di Istanze Casuali 26
3.1.2 Produzione di File di Input per i Solvers 27
3.2 Problema dello Zaino 33
3.2.1 Generazione di Istanze Casuali 33
3.2.2 Produzione di File di Input per i Solvers 33
3.3 Problema dell'Insieme Indipendente 36
3.3.1 Generazione di Istanze Casuali 36
3.3.2 Produzione di File di Input per i Solvers 38

4 Valutazione del Costo Computazionale **41**
4.1 Problema del Cammino Minimo 41
4.2 Problema dello Zaino 45
4.3 Problema dell'Insieme Indipendente 47

Conclusioni **49**

Bibliografia **50**

Lista di Algoritmi

1 Shortest Path - Generazione dei dati 27
2 Generazione shortestPath.lp 29
3 Generazione shortestPath.wcnf 31
4 Knapsack - Generazione dei dati 33
5 Generazione knapsack.lp 34
6 Generazione knapsack.wcnf 35
7 Independent Set - Generazione dei dati 37
8 Generazione independentSet.lp 38
9 Generazione independentSet.wcnf 40

Elenco delle figure

4.1 Problema del Cammino Minimo 43
4.2 Problema del Cammino Minimo - Generazione File di Input 43
4.3 Problema del Cammino Minimo - Fitting 44
4.4 Problema dello Zaino 46
4.5 Problema dell'Insieme Indipendente 47

Introduzione

L'ottimizzazione combinatoria rappresenta un elemento fondamentale dell'informatica e trova applicazioni in una vasta gamma di settori, sia in ambito accademico che in situazioni quotidiane. La capacità di risolvere problemi di ottimizzazione in modo efficiente è fondamentale per l'ottimizzazione di processi e risorse in diversi contesti. Questa tesi si propone di valutare sperimentalmente il costo computazionale impiegato per risolvere tre problemi noti di ottimizzazione combinatoria:

- il problema del cammino minimo su grafo pesato
- il problema dello zaino
- il problema dell'insieme indipendente massimo

Ciascuno di questi problemi verrà poi risolto utilizzando quattro solver differenti, `glpsol` e `CBC` per la Programmazione Lineare Intera e `RC2` e `Open-WBO` per i problemi di soddisfacibilità booleana massima. I modelli per la Programmazione Lineare Intera sono stati definiti in linguaggio LP, mentre per MaxSAT sono stati adottati file in formato WCNF. L'utilizzo di più risolutori diversi per ciascuna classe di solver consente di confrontare le loro prestazioni e analizzare punti di forza e di debolezza di ciascun approccio. In particolare, verrà valutata l'efficienza relativa a ciascun solver rispetto ai tre problemi analizzati.

Programmazione Lineare Intera

La programmazione lineare (PL) è una tecnica matematica ampiamente utilizzata per risolvere problemi di ottimizzazione. L'obiettivo è massimizzare o minimizzare una funzione lineare soggetta a una serie di vincoli espressi anch'essi in forma lineare, questa funzione viene chiamata funzione obiettivo. Si parla di Programmazione Lineare Intera (PLI) quando una o più variabili coinvolte nel problema devono assumere valori interi. Questo requisito aggiuntivo aumenta la complessità computazionale del problema, poiché la soluzione non può essere ottenuta direttamente utilizzando i metodi della programmazione lineare che si basano su un trattamento delle variabili come valori continui. La Programmazione Lineare Intera può essere ulteriormente classificata in due categorie: PLI *pura*, quando le variabili devono assumere tutti valori interi, e PLI *mista* quando solo alcune delle variabili coinvolte sono vincolate ad assumere valori interi. Grazie alla sua versatilità, la Programmazione Lineare Intera è utilizzata in molte applicazioni reali, tra cui problemi in ambito industriale o economico. Questa tesi si concentrerà sulla Programmazione Lineare Intera pura.

Soddisfacibilità Booleana Massima

Il problema della soddisfacibilità booleana massima (MaxSAT) invece è una generalizzazione del problema classico della soddisfacibilità booleana (SAT). SAT è uno dei problemi NP-completi più studiati, e la sua variante MaxSAT è altrettanto rilevante nel contesto dell'ottimizzazione combinatoria. Mentre il problema SAT si focalizza sul determinare, se esiste, un'assegnazione di valori di verità che renda vera l'intera formula booleana, MaxSAT si concentra sul massimizzare il numero di clausole soddisfatte,

anche nel caso in cui non tutte possano essere contemporaneamente vere. Questa caratteristica rende MaxSAT particolarmente utile in contesti in cui l'ottimizzazione non implica una soluzione perfetta, ma un compromesso che massimizza il numero di vincoli rispettati. Le formule utilizzate in MaxSAT sono espresse in Forma Normale Congiuntiva (CNF), che rappresenta una congiunzione di clausole, ciascuna delle quali è una disgiunzione di letterali. I letterali possono essere variabili booleane o le loro negazioni. MaxSAT si presta bene a problemi in cui è importante ottimizzare una funzione obiettivo soggetta a vincoli espressi in forma di clausole booleane. Rappresentare i pesi con clausole pesate (WCNF), inoltre, consente di dare maggiore importanza a determinate clausole rispetto ad altre.

Vincoli pseudo-booleani

Un vincolo pseudo booleano è un'equazione o una disequazione che coinvolge una funzione pseudo booleana e un numero intero. Una funzione pseudo-booleana accetta come input un insieme di variabili booleane, $x_i \in \{0, 1\}$, e restituisce un valore numerico, tipicamente intero. Questi vincoli rappresentano una generalizzazione dei vincoli logici utilizzati nella soddisfacibilità booleana e trovano applicazioni in diversi ambiti, tra cui l'ottimizzazione combinatoria. La forma generale di un vincolo pseudo-booleano è la seguente:

$$f(x_1, ..., x_n) \ op \ b$$

dove $op \in \{\leq, =, \geq\}$, f è una funzione pseudo-booleana definita su $(x_1, ..., x_n)$ e b è un numero intero. Questi vincoli permettono di ottimizzare funzioni pseudo-booleane, soggette a vincoli espressi come sistemi di uguaglianze o disuguaglianze pseudo-booleane. l'ottimizzazione consiste

nel trovare il valore ottimale della funzione obiettivo, rispettando i vincoli imposti.

Contenuto della Tesi

Il lavoro svolto in questa tesi è organizzato come segue: Nel Capitolo 1 vengono introdotti i problemi sopra citati soffermandosi sia su Programmazione Lineare Intera che su soddisfacibilità booleana massima. Nel Capitolo 2 viene fatta una descrizione esaustiva dei solver utilizzati (`glpsol`, `CBC`, `RC2` e `Open-WBO`). Nel Capitolo 3 vengono presentati gli script e le procedure di test utilizzate per risolvere ciascun problema; viene descritto il processo di modellizzazione per ciascuno dei formati (`LP` e `wcnf`) adottati. Infine nel Capitolo 4 vengono analizzati i risultati ottenuti e fatte le relative considerazioni; vengono discusse le osservazioni emerse dai test condotti valutando le prestazioni dei solver e confrontando i tempi di esecuzione. Infine nelle Conclusioni vengono riassunti i risultati ottenuti.

Capitolo 1

Modelli

Ogni problema di ottimizzazione combinatoria presenta caratteristiche specifiche che richiedono l'uso di modelli matematici appropriati per essere risolti in modo efficiente. In questa sezione verranno presentati e analizzati in dettaglio i modelli matematici impiegati per affrontare i problemi trattati in questa tesi. L'obiettivo è offrire una comprensione approfondita delle formulazioni teoriche che stanno alla base delle soluzioni proposte. In particolare verranno esaminati i modelli per tre problemi classici: il problema del cammino minimo su un grafo pesato, il problema dello zaino, e il problema dell'insieme indipendente massimo. Per ciascun problema, sarà data la formulazione matematica adottata, con particolare attenzione alla definizione delle variabili decisionali, della funzione obiettivo e dei vincoli.

1.1 Problema del Cammino Minimo su grafo pesato

Il problema del cammino minimo su grafo pesato $G = (N, A, c)$ consiste nel trovare, se esiste, il cammino di costo minore tra due nodi $s, t \in N$, dove il costo di un cammino è dato dalla somma dei pesi degli archi che lo compongono, calcolato tramite c.

Per comprendere meglio il problema, è necessario definire alcune nozioni di base relative ai grafi e ai cammini.

Definizione 1 (Grafo) *Un grafo è una coppia ordinata di insiemi* $G = (N, A)$*, dove* N *è un insieme finito di nodi mentre* A *è un insieme finito di archi, ovvero coppie di nodi, tale che* $A \subseteq N \times N$*. Ogni arco è una coppia di nodi* (u, v) *che può essere orientato o non orientato a seconda dalle caratteristiche del grafo.*

Esempio 1.1.1 *G = (N, A)*
con $N = \{1, 2, 3, 4\}$ *e* $A = \{(1, 2), (1, 3), (2, 3), (3, 4)\}$.

Definizione 2 (Grafo orientato) *Un grafo orientato è una coppia ordinata di insiemi* $G = (N, A)$*, dove* N *è un insieme finito di nodi mentre* A *è un insieme finito di archi orientati. Ogni arco è rappresentato come una coppia di nodi* (u, v) *dove l'ordine è importante e indica la direzione dell'arco da* u *a* v.

Definizione 3 (Grafo pesato) *Un grafo pesato* $G = (N, A, c)$ *è una tripla in cui* (N, A) *è un grafo e* $c : A \to \mathbb{R}$ *è una funzione che associa ad ogni arco un peso reale.*

Esempio 1.1.2 *G = (N, A, c)*
con $N = \{1, 2, 3, 4\}$*,* $A = \{(1, 2), (1, 3), (2, 3), (3, 4)\}$ *e* $c = \{1, 2, 3, 4\}$.

Definizione 4 (Cammino) *Un cammino* P *in un grafo* $G = (N, A)$ *è una sequenza di nodi* $P = (v_0, v_1, \dots, v_k)$ *tale che* $(v_i, v_{i+1}) \in A$ *per ogni* $i = 0, \dots, k - 1$.

Definizione 5 (Cammino pesato) *Il costo di un cammino* P *in un grafo pesato* $G = (N, A, c)$ *è la somma dei pesi degli archi che lo compongono,*

ovvero

$$c(P) = \sum_{(u,v)\in P} c_{u,v}$$

.

Definizione 6 (Ciclo) *Un ciclo in un grafo $G = (N, A)$ è un cammino $P = (v_0, v_1, \ldots, v_k)$ tale che $v_0 = v_k$, ovvero il nodo di partenza coincide con il nodo di arrivo.*

Definizione 7 (Costo del cammino minimo) *Il costo del cammino minimo tra due nodi u e v è definito come*

$$\delta(u, v) = \begin{cases} \min\{c(P) : P \textit{ cammino tra } u \textit{ e } v\}, & \textit{se } \exists \textit{ cammino tra } u \textit{ e } v \\ +\infty, & \textit{altrimenti} \end{cases}$$

1.1.1 Formulazione del Problema del Cammino Minimo in Programmazione Lineare Intera

Dato un grafo orientato e pesato $G = (N, A, c)$ e una coppia di nodi $s, t \in N$, per ogni arco $(i, j) \in A$, è possibile definire una variabile decisionale $x_{i,j}$ che assume valore 1 se l'arco è incluso nel cammino minimo, 0 altrimenti.

Il problema del cammino minimo può quindi essere formulato come un problema di Programmazione Lineare Intera (PLI) con la seguente funzione obiettivo e vincoli.

Funzione Obiettivo:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \qquad [1.1]$$

Vincoli:

$$\sum_{(s,j)\in A} x_{sj} - \sum_{(i,s)\in A} x_{is} = 1 \qquad [1.2]$$

$$\sum_{(t,j)\in A} x_{tj} - \sum_{(i,t)\in A} x_{it} = -1 \qquad [1.3]$$

$$\sum_{(k,j)\in A} x_{kj} - \sum_{(i,k)\in A} x_{ik} = 0, \quad \forall k \in N \setminus \{s,t\} \qquad [1.4]$$

Di seguito sono illustrati nel dettaglio i vincoli utilizzati.

Il vincolo [1.2] fa in modo che la somma degli archi uscenti dal nodo di partenza s sia uguale alla somma degli archi entranti in s più 1, assicurando che il cammino abbia inizio da s. Il vincolo [1.3] garantisce che la somma degli archi entranti nel nodo di arrivo t sia uguale alla somma degli archi uscenti da t più 1, assicurando che il cammino termini in t. Infine il vincolo [1.4] assicura che la somma degli archi entranti in un nodo k sia uguale alla somma degli archi uscenti da k, per ogni nodo k diverso da s e t.

Da questi vincoli emerge che il cammino minimo non ha soluzione se non esiste un cammino tra i nodi sorgente s e quelli destinazione t. Inoltre, se nel grafo sono presenti cicli di costo negativo, il problema potrebbe non avere soluzione, in quanto il cammino minimo potrebbe non esistere o non essere unico.

1.1.2 Formulazione del Problema del Cammino Minimo in MaxSAT

Il problema del cammino minimo può essere formulato in MaxSAT convertendo il problema di ottimizzazione in un problema di soddisfacibilità booleana, con l'obiettivo di massimizzare il numero di clausole soddisfatte. In questo caso, si desidera trovare un cammino di costo minimo tra due nodi, rappresentando sia la funzione obiettivo sia i vincoli tramite formule booleane. Per ogni arco $(i, j) \in A$, viene introdotta una variabile booleana y_{ij}, che assume valore vero se l'arco (i, j) è incluso nel cammino e valore falso altrimenti. Dato che il problema del cammino minimo è un problema di minimizzazione, è necessario convertirlo in un problema di massimizzazione facendo in modo di massimizzare la somma dei costi negativi associati agli archi.

Funzione Obiettivo:

$$\max \sum_{(i,j)\in A} -c_{ij} y_{ij} \qquad [1.5]$$

Vincoli:

$$\bigvee y_{s,j}, \quad \forall (s,j) \in A \qquad [1.6]$$

$$(\neg y_{s,i} \vee \neg y_{s,j}), \quad \forall (s,i),(s,j) \in A, i \neq j \qquad [1.7]$$

$$\bigvee y_{i,t}, \quad \forall (i,t) \in A \qquad [1.8]$$

$$(\neg y_{i,t} \vee \neg y_{j,t}), \quad \forall (i,t),(j,t) \in A, i \neq j \qquad [1.9]$$

$$(\neg y_{j,i} \vee y_{i,k}), \quad \forall (j,i),(i,k) \in A, \{i,j,k\} \notin \{s,t\} \qquad [1.10]$$

$$\left(\neg y_{i,k} \bigvee_{j\in N(i)} y_{j,i}\right), \quad \forall (i,k),(j,i) \in A, \{i,j,k\} \notin \{s,t\} \qquad [1.11]$$

$$\neg y_{i,j}, \quad \forall \{i,j\} \text{ senza archi uscenti o entranti}, \{i,j\} \neq \{s,t\} \qquad [1.12]$$

Il vincolo [1.6] assicura che esista almeno un arco uscente dal nodo di partenza s, mentre il vincolo [1.7] assicura che non esistano due archi uscenti da s contemporaneamente. Allo stesso modo il vincolo [1.8] assicura che esista almeno un arco entrante nel nodo di arrivo t, mentre il vincolo [1.9] assicura che non esistano due archi entranti in t contemporaneamente. Il vincolo [1.10] assicura che se si utilizza un arco in entrata da un nodo i (dove i non è ne sorgente ne destinazione), allora deve esserci un arco che porta a un altro nodo adiacente (anch'esso non sorgente ne terminale). Ciò impedisce che un nodo intermedio venga "isolato" senza un proseguimento del cammino. Analogamente il vincolo [1.11] assicura che se si utilizza un arco in uscita in un nodo i, allora deve esserci un arco che proviene da un altro nodo adiacente. Infine il vincolo [1.12] assicura che non esistano nel

cammino scelto nodi isolati, ovvero nodi che contengono solamente archi uscenti o entranti (a meno del nodo sorgente o destinazione).

1.2 Problema dello Zaino

Il problema dello zaino è un problema di ottimizzazione combinatoria in cui sono dati n oggetti, ciascuno caratterizzato da un peso p_i e un valore v_i. L'obiettivo è selezionare un sottoinsieme di oggetti da inserire in uno zaino con capacità massima b, in modo da massimizzare il valore totale degli oggetti selezionati rispettando il vincolo di capacità.

1.2.1 Formulazione del Problema dello Zaino in Programmazione Lineare Intera

Dati n oggetti, per ogni oggetto $i = 1, \ldots, n$, è possibile definire una variabile decisionale x_i che assume valore 1 se l'oggetto è selezionato, 0 altrimenti. Il problema dello zaino può quindi essere formulato come un problema di Programmazione Lineare Intera (PLI) con la seguente funzione obiettivo e vincoli.

Funzione Obiettivo:

$$\max \sum_{i=1}^{n} v_i x_i \qquad [1.13]$$

Vincoli:

$$\sum_{i=1}^{n} p_i x_i \leq b \qquad [1.14]$$

Il vincolo [1.14] assicura che la somma dei pesi degli oggetti selezionati non superi la capacità massima dello zaino b.

1.2.2 Formulazione del Problema dello Zaino in MaxSAT

Il problema dello zaino può essere formulato in MaxSAT convertendo il problema di ottimizzazione in un problema di soddisfacibilità booleana con l'obiettivo di massimizzare il numero di clausole soddisfatte. Per completare la formulazione del problema è necessario limitare il peso totale degli oggetti selezionati alla capacità massima dello zaino b. Con l'utilizzo di una libreria Python `PBEnc` di `pySat`, è possibile rappresentare questi vincoli in forma CNF grazie alla sua capacità di tradurre vincoli di ottimizzazione in formule logiche. Per ogni oggetto i, si introduce una variabile booleana y_i, che assume valore vero se l'oggetto è selezionato e falso altrimenti.

Funzione Obiettivo:

$$\max \sum_{i=1}^{n} v_i y_i \qquad [1.15]$$

Vincoli:

$$\sum_{i=1}^{n} p_i y_i \leq b \qquad [1.16]$$

Il vincolo [1.16] assicura che la somma dei pesi degli oggetti selezionati non superi la capacità massima dello zaino b . Grazie alla capacità di `bpEnc` di gestire vincoli lineari, è possibile risolvere il problema dello zaino in MaxSAT senza costruire manualmente tutte le clausole in CNF. La libreria infatti semplifica il processo, garantendo al contempo che il problema venga tradotto correttamente.

1.3 Problema dell'Insieme Indipendente Massimo

Sia dato un grafo non orientato $G = (N, A)$, dove N è l'insieme dei nodi e A è l'insieme degli archi. Un insieme indipendente $I \subseteq N$ è un sottoinsieme di nodi tale che nessun arco in A abbia entrambi i nodi in I.

1.3.1 Formulazione del Problema dell'Insieme Indipendente in Programmazione Lineare Intera

Dato un grafo non orientato $G = (N, A)$ e una variabile di decisione x_i per ogni nodo $i \in N$ che assume valore 1 se il nodo è selezionato, 0 altrimenti. Il problema dell'insieme indipendente massimo può essere formulato come un problema di Programmazione Lineare Intera (PLI) con la seguente funzione obiettivo e vincoli.

Funzione Obiettivo:

$$\max \sum_{i \in N} x_i \qquad [1.17]$$

Vincoli:

$$x_i + x_j \leq 1, \quad \forall (i, j) \in A \qquad [1.18]$$

Il vincolo [1.18] assicura che non siano selezionati contemporaneamente due nodi adiacenti, ovvero che non esista un arco tra i due nodi.

1.3.2 Formulazione del Problema dell'Insieme Indipendente in MaxSAT

Il problema dell'insieme indipendente massimo può essere formulato in MaxSAT convertendo il problema di ottimizzazione in un problema di sod-

disfacibilità booleana con l'obiettivo di massimizzare il numero di clausole soddisfatte. Per tale formulazione, è necessario rappresentare sia la funzione obiettivo sia i vincoli come una formula booleana. Per ogni nodo i del grafo, si introduce una variabile booleana y_i, che assume valore vero se il nodo è selezionato per l'insieme indipendente e falso altrimenti.

Funzione Obiettivo:

$$\max \sum_{i \in N} y_i \qquad [1.19]$$

Vincoli:

$$\neg y_i \vee \neg y_j, \quad \forall (i,j) \in A \qquad [1.20]$$

Il vincolo [1.20] esprime che non possono essere selezionati contemporaneamente due nodi collegati da un arco quindi almeno uno tra i nodi i e j deve essere escluso dall'insieme indipendente, per ogni coppia di nodi adiacenti $(i, j) \in A$.

1.4 NP-Completezza dei Modelli Considerati

I problemi di decisione possono essere classificati in base alla loro complessità computazionale in classi, tra cui: P, NP e NP-Completo. Un problema è definito in P se può essere risolto in tempo polinomiale con algoritmi deterministici; questo significa che esiste un algoritmo che, per un'istanza del problema di dimensione n, trova la soluzione in un tempo massimo limitato da una funzione polinomiale di n. D'altra parte un problema è definito in NP se può essere risolto in tempo polinomiale con algoritmi non deterministici; quindi un problema è in NP se la soluzione può essere verificata in tempo polinomiale, ma non necessariamente trovata in tempo polinomiale.

La classe di linguaggi P è contenuta nella classe NP, dato che qualsiasi algoritmo deterministico è anche un algoritmo non deterministico, tuttavia non e ancora stato dimostrato se queste due classi coincidano, ossia se P = NP, e questa rimane una delle questioni aperte più importanti nell'informatica teorica.

Un problema è NP-completo se è in NP e ogni problema in NP può essere ridotto a esso in tempo polinomiale. Questo implica che se esistesse un algoritmo polinomiale per risolvere un problema NP-completo, allora tutti i problemi in NP potrebbero essere risolti in tempo polinomiale, portando a P = NP.

Due dei problemi di ottimizzazione combinatoria affrontati in questa tesi sono NP-completi, ovvero il problema dell'insieme indipendente massimo e il problema dello zaino. Quest'ultimo solamente nella sua forma discreta, cioè quando un oggetto può essere considerato o meno ma non può essere frazionato. Il problema del cammino minimo su grafo pesato, invece, è un problema risolvibile in tempo polinomiale e fa parte della classe P.

Capitolo 2

Solvers utilizzati

In questa sezione verranno presentati i solvers utilizzati per risolvere i problemi di ottimizzazione combinatoria considerati in questa tesi. In particolare, verranno descritte le caratteristiche principali di quattro solvers: glpsol e CBC per la Programmazione Lineare Intera e RC2 e Open-WBO per la soddisfacibilità booleana massima.

2.1 Solvers per Programmazione Lineare Intera

I solvers per la Programmazione Lineare Intera (PLI) sono strumenti software che consentono di risolvere problemi di ottimizzazione combinatoria in cui alcune o tutte le variabili coinvolte devono assumere valori interi. Questi solvers implementano algoritmi di ottimizzazione specifici per la risoluzione di problemi di PLI, garantendo la convergenza verso la soluzione ottima o una soluzione approssimata entro un tempo ragionevole.

2.1.1 Solver glpsol

Il solver glpsol è un solver contenuto all'interno del pacchetto GLPK (GNU Linear Programming Kit), un pacchetto di librerie open-source per

la risoluzione di problemi di programmazione lineare, intera e mista. Il pacchetto GLPK è composto da una serie di routine scritte in ANSI C e organizzate sotto forma di libreria richiamabile. Questo programma può essere utilizzato direttamente da riga di comando per risolvere modelli scritti in linguaggio MathProg o LP, un linguaggio di modellazione per problemi di programmazione lineare. glpsol supporta vari algoritmi per la risoluzione di problemi di PLI, tra cui il metodo del simplesso (simplex), il metodo duale, il metodo dei punti interni (interior-point), e l'algoritmo branch-and-bound. GLPK include anche il metodo branch-and-cut, utilizzato principalmente per la risoluzione di problemi di Programmazione Lineare Intera mista (MILP).

2.1.2 Solver CBC

Il solver COIN Branch and Cut (CBC) è un solver open-source scritto in C++ adatto per la risoluzione di problemi di Programmazione Lineare Intera e mista. Può essere utilizzato sia come libreria, integrabile in altre applicazioni, sia come programma autonomo eseguibile da riga di comando. CBC è parte del progetto COIN-OR (Computational Infrastructure for Operations Research), un'iniziativa comunitaria che fornisce una raccolta di software open-source per la ricerca operativa. Il solver implementa l'algoritmo branch-and-cut, una tecnica di risoluzione che combina la ricerca esaustiva di branch-and-bound, che suddivide il problema in sottoproblemi, con la generazione di piani di taglio (cutting planes) per migliorare i rilassamenti e ridurre lo spazio di ricerca.

2.2 Solvers per Soddisfacibilità Booleana Massima

I solver per la soddisfacibilità booleana massima (MaxSAT) sono strumenti software che consentono di risolvere problemi di soddisfacibilità booleana (SAT) in cui l'obiettivo non è solo trovare una soluzione che soddisfi tutte le clausole, ma massimizzare il numero di clausole soddisfatte. Questi solvers implementano algoritmi avanzati per individuare soluzioni ottimali o approssimate entro un tempo ragionevole, in particolare per problemi in cui alcune clausole non possono essere soddisfatte simultaneamente.

I problemi MaxSAT sono suddivisi in due categorie principali: MaxSAT non pesato (Unweighted MaxSAT) e MaxSAT pesato (Weighted MaxSAT). Nel primo caso tutte le clausole hanno lo stesso peso, mentre nel caso di MaxSAT pesato, ogni clausola ha un peso che rappresenta l'importanza relativa della sua soddisfazione.

2.2.1 Solver RC2

RC2 è un solver open-source core-guided per la soddisfacibilità booleana massima, scritto in Python e basato su un framework PySAT. L'acronimo sta per "Relaxable Cardinality Constraints" poiché è progettato per risolvere sia clausole hard che clausole soft; le prime devono essere necessariamente soddisfatte per ottenere una soluzione valida, mentre le seconde possono essere violate ma con un costo associato. In questo modo è possibile affrontare problemi in cui non si riescono a soddisfare intermente tutte le clausole, ma è importante soddisfarne il maggior numero possibile. L'approccio core-guided utilizzato dal solver consiste nel trovare un insieme di clausole dette core, che non possono essere soddisfatte simultaneamente e rilassare temporaneamente alcune di esse per trovare una soluzione ottima. RC2 utilizza

un algoritmo specifico per MaxSAT chiamato OLLITI (Optimized Lexicographic Linear Integer Programming Totalizer). Questo algoritmo è particolarmente efficiente per risolvere istanze complesse e pesate del problema MaxSAT. RC2 ha partecipato alle competizione internazionali di MaxSAT Evaluation dal 2018 al 2020 ottenendo risultati notevoli, in particolare nel 2019 ha vinto il primo posto in due categorie: "Unweighted", dove le clausole hanno tutte lo stesso peso, e "Weighted", in cui ogni clausola ha un peso. Gli sviluppatori di questo solver sono Antonio Morgado, João Marques-Silva, e Alexey Ignatiev, ricercatori specializzati nell'ambito SAT e MaxSAT.

2.2.2 Solver Open-wbo

Open-WBO è un solver open-source per la soddisfacibilità booleana massima scritto in C++ che supporta vari algoritmi e risolutori MaxSAT. La sua efficienza è particolarmenre riconosciuta per i problemi non pesati, tanto da essere uno dei migliori risolutori nelle competizioni MaxSAT dal 2014 al 2018. Per quanto riguarda i problemi "Unweighted", questo solver utilizza una variante dell'algoritmo MSU3 che è un metodo iterativo per definire progressivamente un limite superiore al numero di clausole soft non soddisfatte, fino a trovare una soluzione ottimale. MSU3 divide il problema in sottoproblemi più gestibili, incrementando gradualmente la soglia di violazione consentita, fino a quando non viene raggiunta la soluzione ottimale. Per i problemi Weighted MaxSAT, Open-WBO utilizza l'algoritmo OLL (Objective Linear Lexicographic), che adatta la metodologia del MaxSAT non pesato alla gestione di clausole con pesi differenti. OLL continua a ridefinire iterativamente un limite inferiore al numero di clausole soft violate, tenendo conto del loro peso, fino a quando non viene trovata la soluzione ottimale. Questa combinazione di tecniche rende Open-WBO altamente

versatile e adatto sia a problemi pesati che non pesati. Open-WBO è stato sviluppato da Ruben Martins, Vasco Manquinho, e Inês Lynce presso l'Università di Lisbona in Portogallo.

Capitolo 3

Scripts e Procedure di Test

In questa sezione verranno presentati gli script e le procedure di test utilizzate per risolvere i problemi di ottimizzazione combinatoria considerati in questa tesi. Gli script sono stati scritti in Python e sono progettati per generare istanze casuali dei problemi presi in esame, per poi produrre i file di input necessari ai solvers. Per quanto riguarda i problemi di Programmazione Lineare Intera, i file di input sono stati scritti in linguaggio LP, mentre per i problemi di soddisfacibilità booleana massima in formato WCNF. Gli algoritmi implementano i vincoli e le funzioni obiettivo descritte nel Capitolo 1.

3.1 Problema del Cammino Minimo

3.1.1 Generazione di Istanze Casuali

Per generare i dati nel problema di cammino minimo su grafo pesato, è necessario stabilire inizialmente alcuni parametri:

- Il numero n di nodi del grafo
- La probabilità p di avere un arco tra due nodi i e j

- Un nodo iniziale s
- Un nodo finale t

Questi dati verranno utilizzati come parametro per lo script.

Successivamente si crea una matrice di adiacenza contenente gli archi e i relativi costi del grafo, questi dati vengono generati casualmente. Dall'Algoritmo 1 si ottiene l'insieme di nodi V, l'insieme degli archi E del grafo generato e un dizionario c con gli archi del grafo e i loro costi. Questi dati verranno utilizzati per creare il file di input per i solvers.

Algoritmo 1 Shortest Path - Generazione dei dati

```
 1: V ← {1, 2, ..., n}
 2: E ← ∅
 3: c ← ∅
 4: costs ← matrice di adiacenza con costi casuali
 5: for i = 1 to n do
 6:     for j = i + 1 to n do
 7:         if costs[i][j] > 0 then
 8:             E.add((i, j))
 9:             c[(i, j)] = costs[i][j]
10:         end if
11:     end for
12: end for
```

3.1.2 Produzione di File di Input per i Solvers

Dopo aver generato i dati, è necessario convertire il problema in un formato leggibile dai quattro solver considerati.

File di Input per glpsol e CBC

Per generare il file che i solver glpsol e CBC leggeranno, è necessario convertire i dati ottenuti in un problema di Programmazione Lineare Intera. Questo file è scritto in linguaggio LP e contiene funzione obiettivo e vincoli del problema. In input vengono passati V, E, c, s e t ottenuti dall'Algoritmo 1.

Algoritmo 2 Generazione shortestPath.lp

```
1: Apri il file "shortestPath.lp" in scrittura
2: Scrivi "Minimize" nel file
3: Scrivi la somma dei termini c_ij x_ij per (i, j) ∈ E nel file
4: Scrivi "Subject To" nel file
5: for v ∈ V do
6:     in_edges ← ["x_iv" for i ∈ V if (i, v) ∈ E]
7:     out_edges ← ["x_vi" for i ∈ V if (v, i) ∈ E]
8:     if not in_edges and not out_edges then
9:         continue
10:    end if
11:    constraint ← "flow_conservation_v : "
12:    if then out_edges
13:        constraint ← somma di tutti gli archi uscenti con segno +
14:    end if
15:    if then in_edges
16:        constraint ← somma di tutti gli archi entranti con segno -
17:    end if
18:    if then v = s
19:        constraint + " = 1"
20:    else if then v = t
21:        constraint + " = −1"
22:    else
23:        constraint + " = 0"
24:    end if
25: end for
26: Scrivi constraint nel file
27: Scrivi "Binary" nel file
28: Scrivi tutti gli archi nel file
29: Scrivi "End" nel file
```

Il file generato dall'Algoritmo 2 si presenta come segue:

```
Minimize
    c_ij x_ij + c_i'j' x_i'j' + ...
Subject To
    flow_conservation_s: x_si - x_sj = 1
    flow_conservation_k: x_ki - x_kj = 0
    ...
    flow_conservation_t: x_ti - x_tj = -1
Binary
    x_ij x_i'j' ...
End
```

Nella sezione "Minimize" viene specificata la funzione obiettivo del problema, che in questo caso è minimizzare la somma dei costi degli archi. Successivamente nella sezione "Subject To" vengono specificati i vincoli del problema spiegati al Capitolo 1, infine nella sezione "Binary" vengono specificate le variabili decisionali utilizzate come binarie.

File di Input per RC2 e Open-WBO

Per quanto riguarda il file di input per RC2 e Open-WBO, il problema del cammino minimo su grafo pesato è stato convertito in un problema di soddisfacibilità booleana massima scritto in formato WCNF. In input vengono passati V, E, c, s e t ottenuti dall'Algoritmo 1.

Algoritmo 3 Generazione shortestPath.wcnf

```
1: wcnf ← WCNF()
2: var_map ← associo ad ogni arco un numero di variabile univoco
3: outgoing ← archi uscenti da ogni nodo i
4: incoming ← archi entranti in ogni nodo i
5: Scrivo ogni arco (i, j) sul wcnf grazie a var_map[(i, j)]
6: for i ∈ V do
7:     Salvo i nodi che contengono solo archi entranti o uscenti e non siano s o t
8:     if i == s then
9:         wcnf.add([var_map[(i, j)] for j ∈ E non isolati])
10:        if  c'è più di un arco uscente da s then
11:            wcnf.add(−[var_map[(i, j)] for j ∈ E non isolati])
12:        end if
13:    end if
14:    if i == t then
15:        wcnf.add([var_map[(j, i)] for j ∈ E non isolati])
16:        if  then c'è più di un arco entrante in t
17:            wcnf.add(−[var_map[(j, i)] for j ∈ E non isolati])
18:        end if
19:    end if
20:    if i ≠ s and i ≠ t then
21:        if  thenlen(incoming) > 1
22:            wcnf.add([−var_map[(j, i)] for j ∈ E])
23:        end if
24:        if len(outcoming) > 1 then
25:            wcnf.add([−var_map[(i, j)] for j ∈ E])
26:        end if
27:        if incoming and outcoming then
28:            for i  in  incoming do
29:                wcnf.add([inc] + [−edge for edge ∈ outgoing])
30:            end for
31:        end if
32:    end if
33: end for
```

Il file WCNF si presenta come segue:

```
p wcnf n m k
c_1 -1 0
c_2 -2 0
...
c_n n 0
k a b c ... 0
k d e ... 0
...
```

Nella prima riga del file, sono rappresentati il numero n di variabili, il numero m di clausole totali, e il numero k che rappresenta il peso massimo del problema. Le righe seguenti sono le clausole del problema, divise in clausole soft e clausole hard. Le clausole soft sono composte da una lista di variabili precedute da un peso associato, mentre le clausole hard sono composte da un peso iniziale molto alto, seguito da una lista di variabili che devono essere soddisfatte. Nel file WCNF, le variabili booleane sono rappresentate come interi positivi o negativi. Un segno negativo davanti alla variabile indica che quella variabile deve essere falsa affinché la clausola sia soddisfatta.

In questo caso, le variabili coinvolte rappresentano gli archi del grafo e sono scritti con il segno negativo, in modo da massimizzare la somma dei costi negativi associati agli archi selezionati.

3.2 Problema dello Zaino

3.2.1 Generazione di Istanze Casuali

Per la generazione dei dati nel problema dello zaino è necessario stabilire alcuni parametri iniziali:

- Il numero n di oggetti
- Il peso massimo max_w dello zaino

In seguito vengono generati casualmente i pesi e i valori degli oggetti, compresi in un range di valori specificato. Dall'Algoritmo 4 si ottengono i pesi w e i valori v degli oggetti.

Algoritmo 4 Knapsack - Generazione dei dati

1: $w \leftarrow$ vettore con pesi casuali
2: $v \leftarrow$ vettore con valori casuali

3.2.2 Produzione di File di Input per i Solvers

Per consentire ai solvers di risolvere il problema dello zaino, è necessario convertire i dati ottenuti in modo analogo a quanto fatto prima.

File di Input per glpsol e CBC

Per quanto riguarda il file di input per glpsol e CBC, il problema dello zaino è stato convertito in un problema di Programmazione Lineare Intera scritto in linguaggio LP. Vengono passati in input i parametri n, max_w, w e v ottenuti dall'Algoritmo 4.

Algoritmo 5 Generazione knapsack.lp

1: Apri il file "knapsack.lp" in scrittura
2: Scrivi "Maximize" nel file
3: Scrivi la somma dei termini $v_i\ x_i$ per $i = \{0, \ldots, n\}$ nel file
4: Scrivi "Subject To" nel file
5: Scrivi la somma dei termini $w_i x_i$ per $i = \{0, \ldots, n\}$ " $\leq$ " max_w nel file
6: Scrivi "Binary" nel file
7: Scrivi i termini $x_i \ldots x_n$ nel file
8: Scrivi "End" nel file

L'Algoritmo 5 genera un file LP che si presenta come segue:

```
Maximize
    v_0 x_0 + v_1 x_1 + ... + v_n x_n
Subject To
    w_0 x_0 + w_1 x_1 + ... + w_n x_n <= max_w
Binary
    x_0 x_1 ... x_n
End
```

In questo caso si cerca di la somma dei valori degli oggetti tenendo conto del peso massimo da raggiungere.

File di Input per RC2 e Open-WBO

Per generare il file di input per i solvers RC2 e Open-WBO, il problema dello zaino è stato convertito in un problema di soddisfacibilità booleana massima scritto in formato WCNF. Per fare ciò è stata utilizzata una li-

breria di Python chiamata `PBLib`, che permette di codificare vincoli in forma pseudo booleana. In input vengono passati n, max_w, w e v ottenuti dall'Algoritmo 4.

Algoritmo 6 Generazione knapsack.wcnf

1: $wcnf \leftarrow$ `WCNF()`
2: $var \leftarrow [1 \ldots n]$
3: $weight_constraint = PBEnc.leq(lits = var, weights = w, bound = max_w)$
4: **for** $clause$ in $weight_constraint$ **do**
5: $\quad wcnf.add(clause)$
6: **end for**
7: **for** i in $range(n)$ **do**
8: $\quad wcnf.add([var[i]$ weight=values[i]])
9: **end for**

Il file WCNF si presenta come segue:

```
p wcnf n m k
c_1 1 0
c_2 2 0
...
c_n n 0
k a b 0
k c d 0
...
```

In questo caso le clausole soft rappresentano gli oggetti e sono antici-

pate dal costo dell'oggetto. Le clausole hard rappresentano il vincolo di peso massimo che lo zaino può contenere. Per codificare i vincoli in forma pseudo booleana, la libreria `PBLib` aggiunge delle variabili che consentono di rappresentare i vincoli del problema sotto forma di CNF.

3.3 Problema dell'Insieme Indipendente

3.3.1 Generazione di Istanze Casuali

Per la generazione dei dati nel problema dell'insieme indipendente massimo, bisogna stabilire inizialmente alcuni parametri:

- Il numero n di nodi del grafo
- La probabilità p di avere un arco tra due nodi i e j

Questi dati vengono generati casualmente.

Dall'Algoritmo 7 si ottiene la lista di adiacenza *adjacency_list*, l'insieme di nodi V e di archi E del grafo che verranno utilizzati per creare il file di input per i solvers.

Algoritmo 7 Independent Set - Generazione dei dati

```
1: V ← [1, 2, ..., n]
2: E ← ∅
3: adjacency_list ← lista di adiacenza casuale
4: for i in range n do
5:     for j in range i + 1, n do
6:         if random() < p then
7:             adjacency_list[i].add(j)
8:             adjacency_list[j].add(i)
9:         end if
10:    end for
11: end for
12: for i in range n do
13:    for j in adjacency_list[i] do
14:        if i < j then
15:            E.add(i, j)
16:        end if
17:    end for
18: end for
```

3.3.2 Produzione di File di Input per i Solvers

I dati generati dall'Algoritmo 7 devono essere convertiti in un formato leggibile ai solvers per risolvere il problema dell'insieme indipendente massimo.

File di Input per glpsol e CBC

Per quanto riguarda il file di input per i solver MaxSAT, il problema dell'insieme indipendente massimo è stato convertito in un problema di Programmazione Lineare Intera, utilizzando il linguaggio LP. Le variabili passate in input sono n e *adjacency_list*, ottenute dall'Algoritmo 7.

Algoritmo 8 Generazione independentSet.lp

1: Apri il file "independentSet.lp" in scrittura
2: Scrivi "Maximize" nel file
3: Scrivi la somma dei termini x_i con $i = \{0, \ldots, n\}$ nel file
4: Scrivi "Subject To" nel file
5: **for** $i = 0$ to n **do**
6:     **for** j in *adjacency_list*$[i]$ **do**
7:         **if** $i < j$ **then**
8:             Scrivi $c_{i,j} : x_i + x_j \leq 1$ nel file
9:         **end if**
10:     **end for**
11: **end for**
12: Scrivi "Binary" nel file
13: Scrivi i termini $x_i \ldots x_n$ nel file
14: Scrivi "End" nel file
15: Chiudi il file

Il file LP si presenta come segue:

```
Maximize
    x_0 + x_1 + ... + x_n-1
Subject To
    c_0,1: x_0 + x_1 <= 1
    c_0,2: x_0 + x_2 <= 1
    ...
    c_0,n-1: x_0 + x_n-1 <= 1
    c_1,2: x_1 + x_2 <= 1
    ...
    c_n-2,n-1: x_n-2 + x_n-1 <= 1
Binary
    x_0 x_1 ... x_n-1
End
```

In questo problema si cerca di massimizzare il numero di nodi indipendenti, ovvero nodi che non sono adiacenti tra loro. I vincoli fanno in modo che se un nodo viene selezionato, i nodi adiacenti non possano essere considerati.

File di Input per RC2 e Open-WBO

Per quanto riguarda il file di input per RC2 e Open-WBO, il problema dell'insieme indipendente massimo è stato convertito in un problema di soddisfacibilità booleana massima scritto in formato WCNF. In input si prende l'insieme degli archi E e l'insieme dei nodi V, ottenuti dall'Algoritmo 7.

Algoritmo 9 Generazione independentSet.wcnf

1: $wcnf \leftarrow$ `WCNF`()
2: **for** $(i, j) \in E$ **do**
3: $\quad wcnf.add([-i, -j])$
4: **end for**
5: **for** $i \in V$ **do**
6: $\quad wcnf.add([i])$
7: **end for**
8: $wcnf$.to_file($'independentSet.wcnf'$)

Il file wcnf si presenta come segue:

```
p wcnf n m k
1 1 0
1 2 0
...
1 n 0
...
k -a -b 0
k -a -c 0
...
```

In questo caso i pesi delle clausole soft sono tutti uguali a 1, in modo da massimizzare il numero di nodi indipendenti selezionati. Come vincoli hard vengono inseriti i nodi che non possono essere selezionati contemporaneamente, preceduti da un segno negativo.

Capitolo 4

Valutazione del Costo Computazionale

In questo capitolo vengono presentati i risultati delle sperimentazioni condotte con i solver RC2, Open-WBO, glpsol e CBC, utilizzati per risolvere i problemi di ottimizzazione combinatoria affrontati in questa tesi. L'analisi si concentra sulla misurazione dei tempi di calcolo e sull'interpretazione dei dati raccolti, al fine di valutare l'efficienza di ciascun solver. Per garantire una comparazione accurata, i test sono stati eseguiti utilizzando gli stessi set di dati come input per tutti i solver, in funzione del problema specifico. Questo approccio ha permesso di effettuare un'analisi comparativa rigorosa, mettendo in evidenza le prestazioni relative di ciascuno in termini di tempo di esecuzione.

4.1 Problema del Cammino Minimo

I risultati ottenuti dalla sperimentazione condotta sui quattro solver per il problema del cammino minimo, sono stati ottenuti variando il numero n di nodi del grafo e mantenendo costante la probabilità $p = 0.7$ per la

creazione di un arco tra due nodi. Questa metodologia ha permesso di osservare come le performance dei diversi solver cambino in relazione alla dimensione del problema. In Figura 4.1 sono riportati i tempi medi di esecuzione che i quattro solver impiegano per risolvere questo problema. Sull'asse delle ascisse è riportato il numero n di nodi del grafo, mentre sull'asse delle ordinate è mostrato il tempo medio di esecuzione in secondi. L'analisi dei dati evidenzia una differenza significativa tra le performance dei solver per la Programmazione Lineare Intera e i solver MaxSAT.

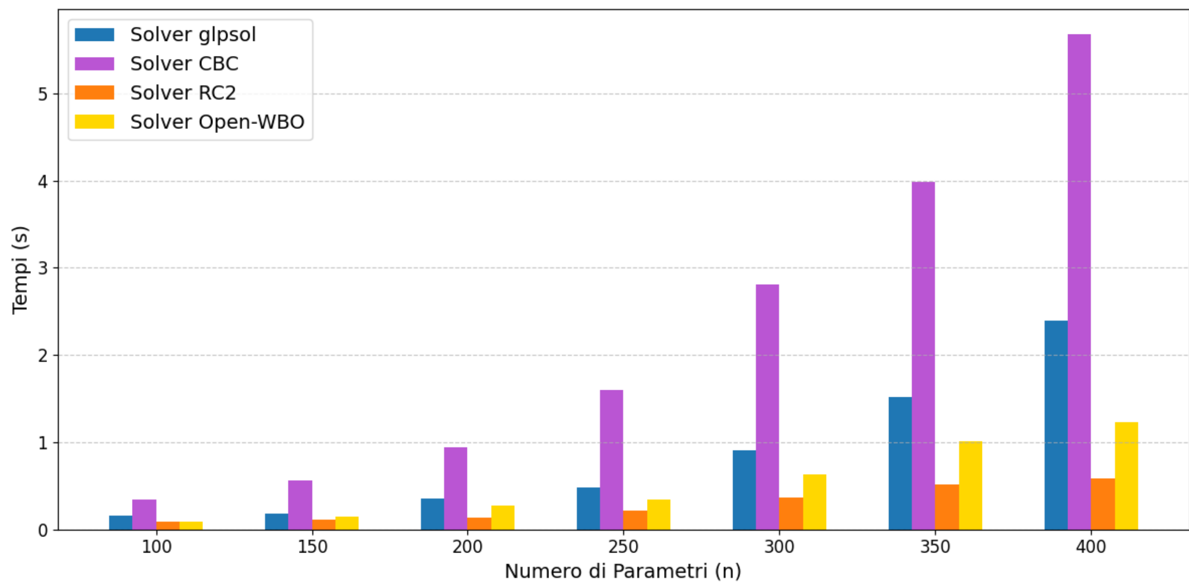

Figura 4.1: Problema del Cammino Minimo

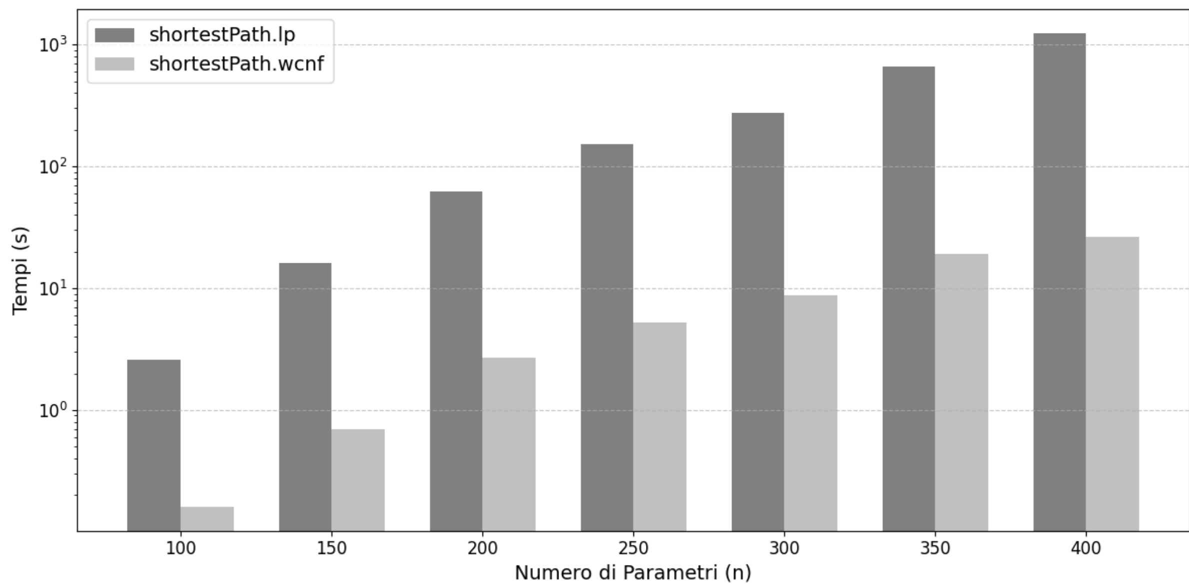

Figura 4.2: Problema del Cammino Minimo - Generazione File di Input

Dalla Figura 4.1 si può notare che i tempi dei solver PLI sono generalmente più elevati rispetto a quelli dei solver MaxSAT per lo stesso problema.

Questa discrepanza aumenta con l'aumentare delle dimensioni del grafo. In Figura 4.2 invece sono riportati i tempi medi di creazione dei file di input per i solver, rappresentati in scala logaritmica. Anche in questo caso si può notare che la creazione dei file di input per i solver PLI risulta più onerosa rispetto a quella dei solver MaxSAT.

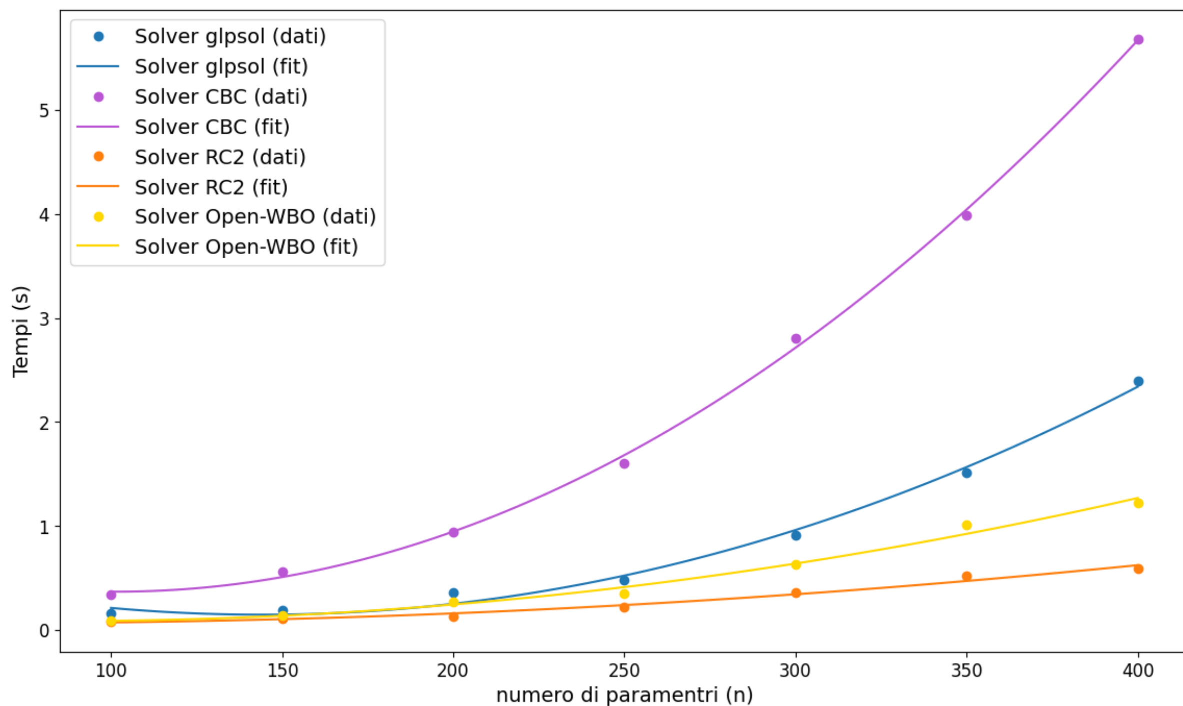

Figura 4.3: Problema del Cammino Minimo - Fitting

La Figura 4.3 mostra il fitting dei tempi medi di esecuzione per la risoluzione del problema del cammino minimo. Le funzioni che meglio descrivono i dati raccolti per ciascun solver sono le seguenti:

- **Solver glpsol**: $y = 3.344 \cdot 10^{-5}x^2 - 9.634 \cdot 10^{-3}x + 0.842$
- **Solver CBC**: $y = 5.944 \cdot 10^{-5}x^2 - 1.205 \cdot 10^{-2}x + 0.980$
- **Solver RC2**: $y = 4.819 \cdot 10^{-6}x^2 - 5.752 \cdot 10^{-4}x + 0.083$
- **Solver Open-WBO**: $y = 1.183 \cdot 10^{-5}x^2 - 1.980 \cdot 10^{-3}x + 0.169$

Le funzioni di tipo quadratico è una funzione semplice che evidenzia l'andamento crescende del tempo impiegato in funzione del numero di nodi del grafo e, allo stesso tempo, si adatta bene ai dati osservati.

Come si può osservare dalle equazioni sopra riportate, i solver per PLI presentano coefficienti sia lineari che quadratici più elevati rispetto ai solver MaxSAT. Questo fatto sottolinea che, sebbene il problema del cammino minimo sia risolvibile in tempo polinomiale e appartenga alla classe di complessità P, i solver PLI introducono una complessità maggiore. Ciò può essere attribuito al modo in cui questa tipologia di solver gestisce i vincoli lineari e le variabili binarie.

4.2 Problema dello Zaino

Dai test condotti sui solver considerati per il problema dello zaino, sono stati ottenuti i tempi di esecuzione medi mostrati dal grafico in Figura 4.4, variando il numero n di oggetti. Questa analisi permette di valutare l'efficienza dei diversi approcci al variare della complessità del problema.

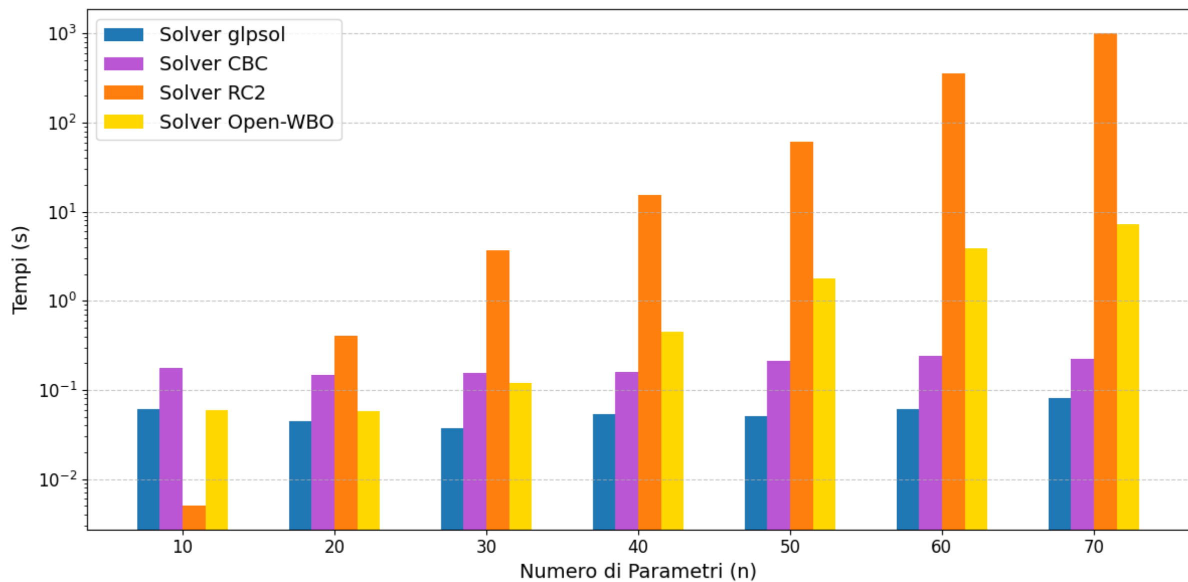

Figura 4.4: Problema dello Zaino

Dal grafico emerge che i solver per la Programmazione Lineare Intera, come glpsol e CBC, presentano tempi di risoluzione ridotti rispetto a solver MaxSAT per questo tipo di problema. Questa differenza è dovuta principalmente al modo con cui i solver gestiscono i vincoli specifici del problema. La formulazione del problema dello zaino in MaxSAT richiede una trasformazione dei vincoli di peso e capacità in clausole pseudo booleane grazie all'utilizzo della libreria `PBLib` di `PySat`. L'uso di questo tipo di clausole consente di rappresentare direttamente i vincoli del problema con maggiore precisione, ma può introdurre un carico computazionale maggiore rispetto a PLI. D'altra parte i solver PLI sono in grado di risolvere il problema in tempi più brevi, poiché possono gestire direttamente i vincoli di peso e capacità. Questi solver sfruttano tecniche efficienti come branch-and-bound, cutting planes o rilassamenti lineari, che consentono di ridurre significativamente lo spazio di ricerca e, di conseguenza, il tempo di risoluzione. Grazie a queste tecniche i solver PLI possono affrontare i vincoli lineari in modo

più efficiente per il problema dello zaino, risultando più veloci rispetto ai solver MaxSAT.

4.3 Problema dell'Insieme Indipendente

I risultati dei test condotti sui quattro solver per questo problema sono stati ottenuti facendo variare il numero n dei nodi del grafo e mantenendo costante la probabilità $p = 0.5$ per la creazione di un arco tra due nodi.

In Figura 4.5 sono riportati i tempi di esecuzione medi dei quattro solver per risolvere il problema dell'Insieme Indipendente Massimo. Sull'asse delle ascisse è riportato il numero n di nodi del grafo, mentre sull'asse delle ordinate è indicato il tempo di esecuzione medio in secondi, visualizzati su scala logaritmica.

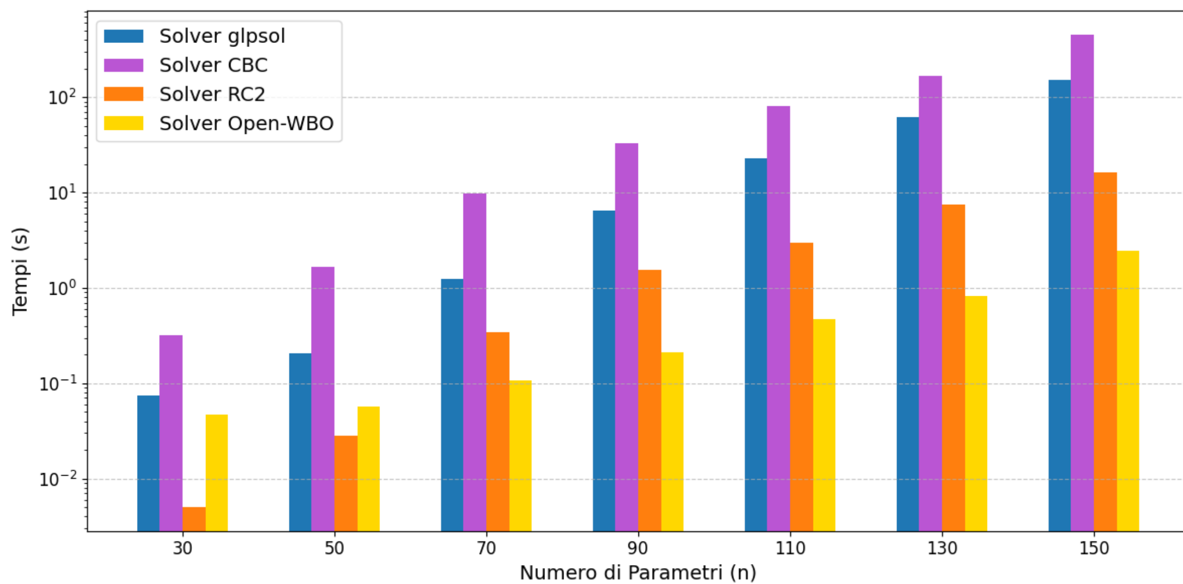

Figura 4.5: Problema dell'Insieme Indipendente

Dal grafico emerge che, per il problema dell'insieme indipendente massimo, i solver per la Programmazione Lineare Intera registrano tempi di

esecuzione notevolmente più elevati rispetto ai solver MaxSAT all'aumentare delle dimensioni del grafo. Questo comportamento si può attribuire alla natura NP-completa del problema, che comporta un aumento esponenziale del tempo di esecuzione per metodi risolutivi esatti come quelli basati su PLI. L'incremento della dimensione del grafo incide direttamente sulla complessità computazionale dei solver PLI, poiché richiedono di esaminare tutte le possibili combinazioni di archi e nodi, o gran parte di esse, per determinare la soluzione ottima. Al contrario, i solver per la soddisfacibilità booleana si distinguono per un comportamento più efficiente in termini di tempo di esecuzione. Grazie al fatto che le variabili sono puramente binarie, i solver MaxSAT possono adottare approcci approssimati che permettono di ridurre notevolmente il tempo di calcolo. Pertanto, questi solver sono particolarmente adatti per affrontare il problema dell'Insieme Indipendente Massimo su grafi di grandi dimensioni, mantenendo tempi di esecuzione contenuti anche all'aumentare di n.

Conclusioni

In questa tesi sono stati analizzati tre noti problemi di Ottimizzazione Combinatoria tramite due approcci risolutivi, Programmazione Lineare Intera e MaxSAT. I problemi considerati sono:

- il problema del cammino minimo su grafo pesato
- il problema dello zaino
- il problema dell'insieme indipendente massimo

L'obiettivo principale è stato quello di valutare le performance di questi due approcci nel risolvere i problemi sopra citati. Questo è stato possibile grazie ad una analisi approfondita dei tempi di esecuzione sui solver `glpsol`, `CBC`, `RC2` e `Open-WBO`. Dalle sperimentazioni condotte emerge che non c'è un solver universalmente migliore di un altro, ma dipende dalla natura e dalla complessità del problema. I solver per la Programmazione Lineare Intera si sono rivelati migliori per vincoli lineari più complessi come il problema dello zaino. D'altra parte i solver MaxSAT hanno dimostrato di essere più efficienti su problemi legati a grafi di grandi dimensioni, come nel problema del cammino minimo e dell'insieme indipendente, grazie alla loro capacità di rappresentare le variabili in modo più compatto.

Questi risultati forniscono indicazioni utili per la scelta del solver più appropiato in base alle caratteristiche del problema da risolvere. Inoltre

il lavoro svolto mostra nuove prospettive per migliorare le prestazioni dei solver, magari combinando i due approcci risolutivi per ottenere risultati migliori.

Bibliografia

[1] Giancarlo Bigi et al. «Appunti di Ricerca Operativa». In: *SEU-Servizio Editoriale Universitario Pisano* (2003).

[2] Alan Bertossi, Alberto Montresor et al. *Algoritmi e strutture di dati*. Citta'Studi Edizioni (De Agostini), 2010.

[3] Pierre Hansen e Brigitte Jaumard. «Algorithms for the maximum satisfiability problem». In: *Computing* 44.4 (1990), pp. 279–303.

[4] Phil Sung. «Maximum satisfiability». In: *Maximum Satisfiability* (2006), pp. 1–2.

[5] Peter L Hammer e Sergiu Rudeanu. «Pseudo-boolean programming». In: *Operations Research* 17.2 (1969), pp. 233–261.

[6] Marta Grobelna. «Solving Pseudo-Boolean Constraints». Tesi di dott. Bachelor's thesis. RWTH Aachen University, 2017.

[7] Jiongzhi Zheng et al. «Rethinking the Soft Conflict Pseudo Boolean Constraint on MaxSAT Local Search Solvers». In: *arXiv preprint arXiv:2401.10589* (2024).

[8] Università di Roma Dipartimento di Ingegneria informatica Automatica e gestionale. *Modelli di Programmazione Lineare Intera*. Consultato il 13 agosto 2024. URL: `http://www.diag.uniroma1.it/~roma/didattica/RONO/cap8.pdf`.

[9] Università di Roma Dipartimento di Ingegneria informatica Automatica e gestionale. *Metodi generali per la soluzione di problemi di PLI*. Consultato il 27 novembre 2024. URL: `http://www.diag.uniroma1.it/~roma/didattica/RONO/cap10.pdf`.

[10] Università di Torino Prof. Locatelli. *Programmazione lineare intera*. Consultato il 11 agosto 2024. URL: `http://www.di.unito.it/~locatell/didattica/intera.pdf`.

[11] Flavio Pietrelli. «Introduzione al problema dei cammini minimi». In: ().

[12] G. Bongiovanni e T. Calamoneri. *Dispense per i corsi di Informatica Generale e Introduzione agli Algoritmi*. Consultato il 15 settembre 2024. 2013. URL: https://twiki.di.uniroma1.it/pub/Info_gen/Dispense/Capitolo9.pdf.

[13] *MaxSAT evaluation*. Consultato il 13 agosto 2024. URL: https://maxsat-evaluations.github.io.

[14] Fahiem Bacchus et al. «MaxSAT evaluation 2020: solver and benchmark descriptions». In: (2020).

[15] *Documentazione GLPK, GNU Linear Programming Kit*. Consultato il 29 settembre 2024. URL: https://www.gnu.org/software/glpk/.

[16] *CBC User Guide*. Consultato il 15 settembre 2024. URL: https://www.coin-or.org/Cbc/cbcuserguide.html.

[17] *PySAT Documentation*. Consultato il 14 settembre 2024. URL: https://pysathq.github.io/docs/html/api/examples/rc2.html.

[18] Alexey Ignatiev, António Morgado e João Marques-Silva. «RC2: an efficient MaxSAT solver». In: *Journal on Satisfiability, Boolean Modeling and Computation* 11.1 (2019), pp. 53–64.

[19] Alexey Ignatiev, Antonio Morgado e Joao Marques-Silva. «RC2: a python-based MaxSAT solver». In: *MaxSAT Evaluation* 2018 (2018), p. 22.

[20] *Open-WBO Documentation*. Consultato il 3 ottobre 2024. URL: https://github.com/sat-group/open-wbo.

[21] Ugo Dal Lago e Luca Orlandello. «Analisi Sperimentale del Costo Computazionale di Problemi su Grafi Formulati in Programmazione Lineare Intera». In: ().

Ringraziamenti

Desidero ringraziare il Professore Ugo Dal Lago per la competenza con cui mi ha guidata nella stesura di questa tesi e per la costante disponibilità dimostrata durante tutto il percorso.