UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI SS. MM. FF.

CORSO DI LAUREA IN MATEMATICA

TESI DI LAUREA

Integrazione di Programmazione con Vincoli e Programmazione Matematica per risolvere Problemi di Ottimizzazione Combinatoria

Relatore: Ch.ma Prof.ssa Francesca Rossi

Controrelatore: Dott. Carlo Filippi

Laureanda: Maria Silvia Pini

A.A. 2002-2003

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Negli ultimi anni parecchi ricercatori operanti nei settori della Ricerca Operativa (Operations Research, OR) e della Programmazione con Vincoli (Constraint Programming, CP) hanno esaminato la possibilità di integrare le metodologie dei due campi, per risolvere in modo più efficiente varie classi di problemi di ottimizzazione combinatoria. Tale indagine, tuttora oggetto di studio da entrambe le comunità di OR e AI, ha già ottenuto notevoli successi e si ritiene che possa ancora fornire risultati importanti.

I Problemi di Ottimizzazione riguardano la minimizzazione o massimizzazione di una funzione di più variabili soggette a un insieme di vincoli. Tali problemi sorgono di frequente in diversi settori quali l'industria, la logistica, la finanza, i trasporti ecc. Per risolvere questa tipologia di problematiche fin dagli anni '60 si sono usate tecniche di programmazione lineare e intera. Di particolare rilevanza sono i problemi di ottimizzazione in cui alcune variabili o tutte sono vincolate ad essere intere. Tali problemi, rilevanti per quanto concerne le applicazioni, sono difficili dal punto di vista risolutivo.

Per lungo tempo queste problematiche furono risolte solo dalla Ricerca Operativa, il cui approccio si basa sulla rappresentazione matematica del problema, modellato come programma lineare intero in cui le variabili sono legate da equazioni e disequazioni lineari.

Durante gli anni '80-'90 si è sviluppato un nuovo paradigma di programmazione: la Programmazione con Vincoli, basata su tecniche a cavallo tra AI (Artificial Intelligence), Linguaggi di Programmazione e Programmazione Logica. Questa nuova metodologia, descritta in [7, 8], è stata usata con successo per modellare e risolvere parecchi problemi di ottimizzazione combinatoria, come

ad esempio problemi di scheduling e di planning. Recentemente, si sono sviluppati linguaggi basati sui vincoli che sono indipendenti dalla Programmazione Logica (come ad esempio ILOG Solver [5]).

Il metodo adottato dalla Programmazione con Vincoli, per trattare i problemi di ottimizzazione combinatoria, consiste nel modellare il problema come un insieme di variabili, che prendono i loro valori su domini finiti di interi e sono legate da un insieme di vincoli, che possono essere matematici o globali. Tipico della Programmazione con Vincoli sono gli algoritmi di propagazione di vincoli (Constraint Propagation algorithms) [7, 11], originati nel campo dell'AI, strumenti che riducono efficacemente lo spazio di ricerca, eliminando alcune combinazioni di assegnamenti di valori alle variabili che si dimostrano irrealizzabili. Il successo di CP è dovuto soprattutto agli strumenti che utilizza per modellare e risolvere le problematiche in esame. Tramite questi è possibile esprimere in modo naturale relazioni molto complesse e sviluppare un metodo di ricerca dipendente dal problema. Strumenti di modellizzazione molto rilevanti in CP sono i vincoli globali [12], relazioni logiche tra variabili, che rappresentano astrazioni opportune in grado di catturare sottostrutture interessanti di un problema. Per tali vincoli si sono sviluppati algoritmi di propagazione specifici, che realizzano un'efficace riduzione dei domini, come descritto in [22], deducendo valori irrealizzabili per le variabili rispetto alla struttura che essi rappresentano.

Il successo industriale di CP ha catturato l'attenzione di molti ricercatori di OR che, insieme a studiosi della comunità di AI, hanno esaminato la possibilità di integrare le metodologie dei due campi tramite la Programmazione con Vincoli.

Gli approcci di CP e OR per problemi di ottimizzazione combinatoria sono complementari. La CP si basa sulla propagazione di vincoli per ridurre lo spazio di ricerca, eliminando la possibilità di assegnare combinazioni di valori alle variabili che determinano inconsistenza rispetto ai vincoli. La OR, rilassando alcuni vincoli, definisce un nuovo problema (problema rilassato), che può essere risolto in maniera ottimale. Il valore della soluzione ottima del problema rilassato rappresenta una valutazione ottimistica della soluzione ottima del problema originale e può essere usato per restringere lo spazio di ricerca. La riduzione dello spazio di ricerca è ottenuta in CP ragionando sulla realizzabilità della configurazione di valori, e in OR considerando l'ottimalità.

Da alcuni anni, molti ricercatori di CP e OR hanno cercato di integrare le

metodologie dei due campi, con lo scopo di combinare i vantaggi di CP, tra cui la facilità del modeling e la propagazione di vincoli, con quelli di OR, soprattutto il ragionamento globale sull'ottimalità e il rilassamento, per superare le limitazioni di CP riguardanti lo scarso ragionamento sull'ottimalità e quelle di OR relative all'assenza di vincoli globali e di modelli flessibili. Sono stati già raggiunti risultati molto buoni in questa direzione. Per esempio, inserendo nei vincoli globali algoritmi di filtraggio dei domini basati sui costi, che rimuovono quelle combinazioni di valori che conducono a soluzioni di costo peggiore della miglior soluzione trovata finora, si sono potuti risolvere problemi di un ordine di grandezza più grande di quelli risolvibili con tecniche di puro CP (come illustrato in [3]) e anche introducendo piani di taglio specifici nei vincoli globali si sono ottenuti risultati molto positivi a livello di complessità computazionale (come mostra [4]).

Questa tesi propone una metodologia nuova per integrare le tecniche di CP e OR. L'innovazione rispetto alla ricerca già effettuata consiste nell'utilizzare in CP dei tagli indipendenti dal problema, i tagli di Gomory, che sono strumenti specifici della Ricerca Operativa. In passato, ad esempio in [4, 10], si sono utilizzati tagli dipendenti dal problema, per migliorare i bounds del problema rilassato; in questa tesi si usano tagli (tagli di Gomory) generici, adatti a un qualunque tipo di problema. Lo scopo di questa tesi è di osservare se questi tagli, inseriti nell'ambito della Programmazione con Vincoli, migliorino la performance dei sistemi CP o OR puri nei problemi di ottimizzazione.

Più in dettaglio, il metodo di risoluzione implementato in questa tesi prevede l'esecuzione della seguente sequenza di operazioni:

1. Modellizzazione del problema in CP tramite l'impiego di variabili, domini, vincoli.

2. Propagazione di vincoli : i vincoli sono propagati con l'obiettivo di ridurre i domini delle variabili, tramite l'eliminazione di valori inconsistenti. I principali algoritmi di propagazione usati per i vincoli binari sono quelli della consistenza sugli archi (arc-consistency algorithm) e della consistenza sui limiti (bound consistency algorithm), descritti in [11, 7].

3. Trasformazione in problema ILP : il modello CP, ottenuto dopo la propagazione, viene formulato in modo lineare come descritto in [2], prestando particolare attenzione al cambiamento dei vincoli globali [12].

4. Rilassamento del vincolo di interezza e risoluzione del problema LP rilassato tramite tecniche OR e l'aggiunta dei tagli di Gomory [1].

5. Utilizzazione della soluzione ottima rilassata appena trovata per determinare la strategia di branching da adottare nel lato CP: si sceglie, come prossima variabile su cui attuare il branching, la variabile CP corrispondente a quella che nella soluzione ottima rilassata ha valore frazionario più vicino a 0.5 oppure quella con valore frazionario più grande e si stabilisce di seguire prima la strada che istanzia tale variabile con il valore intero più vicino, oppure con il valore intero più lontano, o con il valore immediatamente più grande o con quello immediatamente più piccolo.

6. Nuova propagazione dei vincoli : il vincolo appena aggiunto alla formulazione CP può determinare un'ulteriore riduzione dei domini delle variabili.

7. Iterazione : il processo di risoluzione riparte dal punto 3.

Questa sequenza di operazioni viene effettuata più volte, fino al raggiungimento di una soluzione realizzabile del problema originale.

In questa tesi è stato implementato un algoritmo ibrido CP/ILP che esegue le operazioni descritte sopra. Per esaminare la comportamento di tale algoritmo, esso è stato confrontato con approcci di sola CP e di sola OR, anch'essi implementati in questa tesi, su una classe di problemi detti di Latin Square. I gradi di libertà , che sono stati valutati nella sperimentazione sono :

- la scelta della classe di problemi (Latin Square Problem);
- la scelta delle istanze di tale classe;
- la scelta riguardante i tagli di Gomory : quanti tagli utilizzare nell'albero di ricerca.

L'indagine sperimentale effettuata sui problemi di *LatinSquare* ha dimostrato che l'algoritmo ibrido realizzato, che utilizza in maniera moderata i tagli di Gomory e scambia continuamente informazioni tra CP e LP è un algoritmo interessante. Esso infatti, anche se in alcuni casi registra tempi superiori a quelli dell'algoritmo CP, riesce a risolvere un numero di istanze maggiore dell'algoritmo CP, ed è quindi essenziale per tali istanze. Dall'analisi sperimentale è emerso che oltre all'algoritmo ibrido, anche l'algoritmo ILP riesce

a risolvere queste istanze con tempi molto simili ai tempi ibridi, tuttavia, dal momento che nella maggior parte dei casi esaminati l'algoritmo ibrido registra tempi decisamente minori di quelli valutati dall'algoritmo ILP, sembra corretto dichiarare che l'algoritmo ibrido è in generale migliore di quello ILP a livello di tempi computazionali. In definitiva, inserire in maniera moderata i tagli di Gomory in ambiente CP è un valido strumento di integrazione delle discipline CP e OR, in quanto permette di determinare soluzioni di istanze, che l'approccio con sola CP non sarebbe in grado di valutare e che l'approccio ILP risolverebbe in tempi più lunghi.

Questa tesi è organizzata come segue. Il Capitolo 2 fornisce il background necessario per la comprensione degli argomenti trattati in questa dissertazione: presenta brevemente i problemi di ottimizzazione combinatoria, i metodi adottati per risolverli e le caratteristiche principali di Programmazione con Vincoli e Programmazione Matematica. Il Capitolo 3 descrive le direzioni più significative di integrazione di CP e OR proposte in letteratura. Il Capitolo 4 illustra in dettaglio gli algoritmi implementati in questa tesi. Il Capitolo 5 presenta ed analizza una vasta gamma di risultati raggiunti con l'indagine sperimentale. Il Capitolo 6, infine, trae alcune considerazioni relative all'efficienza dell'algoritmo ibrido realizzato, paragonando il suo comportamento a quello di algoritmi di puro stile CP o ILP.

Capitolo 2

Nozioni Preliminari

2.1 Problemi di Ottimizzazione Combinatoria

2.1.1 Problemi di Ottimizzazione

Molti problemi, aventi importanti applicazioni pratiche, possono essere rappresentati come problemi di ottimizzazione e descritti come segue:

$$min\{f(x) \mid x \in F \subset R^n\} \tag{2.1}$$

dove $x \in R^n$ è il vettore delle *variabili del problema*, F è la *regione realizzabile* (l'insieme di tutte le soluzioni realizzabili) e $f: F \to R$ è la *funzione obiettivo*. Ogni $x \in F$ è detto *soluzione realizzabile* di (2.1). Se esiste una $x^* \in F$ che soddisfa

$$f(x^*) \leq f(x), \forall\ x \in f$$

allora x^* è detto soluzione ottima (globale) e $f(x^*)$ è chiamato minimo (globale) con riferimento a (2.1).

Equivalentemente, un problema di ottimizzazione può essere definito come segue:

$$\begin{gathered} min\ f(x) \\ subject\ to \\ g_i(x) \geq 0,\ i=1,\ldots,m \\ h_j(x) = 0,\ j=1,\ldots,p \end{gathered}$$

dove $g_i(x)$, $h_j(x)$ sono funzioni $R^n \to R$, che rappresentano i *vincoli* del problema di ottimizzazione.

2.1.2 Problemi di Ottimizzazione combinatoria

Un problema con un numero finito di soluzioni candidate ottime, in cui l'ottimo può essere trovato confrontando un numero finito di soluzioni, è chiamato Problema di Ottimizzazione Combinatoria. Alcune classi importanti di problemi di ottimizzazione combinatoria, considerate nel corso della trattazione, sono le seguenti:

- *Programma Lineare* (Linear Program, LP). Un problema di ottimizzazione combinatoria è un LP se f, g_i e h_j sono funzioni lineari.

- *Problema di Ottimizzazione con Dominio Finito.* Un problema di ottimizzazione combinatoria è problema di ottimizzazione con Dominio Finito se x *intero*, e $x \in [a_i, b_i]$, per $i = 1,...,n$.

- *Programma Lineare Intero* (Integer Linear Program, ILP). Un problema di ottimizzazione combinatoria è un ILP se la funzione obiettivo f è lineare ed esiste esattamente un vincolo non lineare, che impone x *intero*.

I Programmi Lineari, a differenza dei Problemi di Ottimizzazione con Dominio Finito e dei Programmi Lineari Interi, hanno la particolarità di essere facilimente risolvibili da alcuni algoritmi molto noti (ad esempio il metodo del simplesso) [1]. Il modello generale di queste problematiche richiede un insieme di variabili, che rappresentano le entità di base del problema e un insieme di vincoli. Le *variabili*, che possono assumere un insieme di valori discreti, sono legate da un gruppo di relazioni, chiamate *vincoli*, che devono essere soddisfatte per il raggiungimento di una soluzione. I problemi esaminati sono caratterizzati da un numero finito di soluzioni e, per preferirne una, cioè la *soluzione ottima*, rispetto alle altre, esistono degli opportuni criteri di ottimizzazione. Essi sono espressi nella *funzione obiettivo* che associa alle varie soluzioni un valore. Se la funzione obiettivo deve essere minimizzata (per es. minimizzare il costo di un processo produttivo), il problema di ottimizzazione si dice *problema di minimizzazione*, in caso contrario (per es. massimizzare il profitto della produzione) si parla di *problema di massimizzazione.* Poiché i problemi di massimizzazione possono essere convertiti in quelli di minimizzazione cambiando semplicemente il segno della funzione obiettivo, nel seguito considereremo solo problemi del secondo tipo senza perdita di generalità.

2.1.3 Complessità Computazionale

I problemi di ottimizzazione combinatoria sorgono in molte applicazioni della vita reale e generalmente sono difficili da risolvere. La difficoltà teorica di un problema può essere definita usando la nozione di complessità computazionale, che verrà brevemente presentata in questa sezione.

La *taglia n* dell'istanza I di un problema è definita come il numero di simboli necessari per decifrare I in modo compatto. Data un'istanza I del problema di taglia n, si dice che un algoritmo ha una *complessità tempo* di $O(g(n))$, se il tempo più lungo richiesto per eseguire l'algoritmo ha un limite superiore asintotico di $k\ g(n)$, dove k è una costante e $g(n)$ una funzione di n. Un algoritmo, se ha complessità tempo polinomiale, è un algoritmo con *complessità polinomiale*, altrimenti è un algoritmo con *complessità esponenziale*. Sia c il più grande intero dell'istanza di un problema. Un algoritmo ha *complessità pseudo-polinomiale*, se la sua complessità tempo è polinomiale su c e n.

Ogni problema di ottimizzazione combinatoria può essere trasformato in un corrispondente problema decisionale (che cerca le risposte *sí*, *no* riguardo alla realizzabilità). È facile mostrare che un problema decisionale PD *non* è *più difficile* del corrispondente problema di ottimizzazione PO. Se si assume l'ipotesi che $f(x^*)$ possa essere rappresentata con un numero di simboli polinomiali nella taglia del problema, è semplice anche mostrare che un PO *non* è *più difficile* del corrispondente PD.

Un problema decisionale appartiene alla classe $\mathcal{P}$, se un'istanza del problema può essere risolta da un noto algoritmo con complessità polinomiale (per esempio, può essere risolto da una Macchina di Turing limitata polinomialmente). Appartiene, invece, alla classe $\mathcal{NP}$, se un'istanza del problema può essere risolta con un albero decisionale con profondità polinomiale (cioè può essere risolto in tempo polinomiale da una macchina di Turing non deterministica). I problemi della classe $\mathcal{P}$, a differenza di quelli $\mathcal{NP}$, sono *facili*, nel senso che esistono algoritmi polinomiali capaci di risolverli. La classe dei Programmi Lineari è in $\mathcal{P}$, mentre la maggior parte delle altre classi di problemi di ottimizzazione combinatoria sono in $\mathcal{NP}$. Quindi, in generale, per risolvere un problema di ottimizzazione combinatoria, è necessario un algoritmo con complessità esponenziale.

2.2 Tecniche Risolutive

Quando si risolve un problema di ottimizzazione combinatoria, si possono individuare due fasi concettuali:

- una parte di soddisfazione dei vincoli, rivolta alla ricerca di una soluzione *realizzabile*;
- una parte di ottimizzazione, che seleziona tra le soluzioni realizzabili quella che minimizza la funzione obiettivo.

In generale la soluzione di questa classe di problemi richiede l'esplorazione di uno *spazio di ricerca*, che è rappresentato da tutte le possibili combinazioni di assegnamenti di valori alle variabili. Chiaramente alcune parti di questo spazio non sono *realizzabili*, poiché contengono combinazioni di assegnamenti che non soddisfano i vincoli. Per trovare una soluzione realizzabile e ottima si visita lo spazio di ricerca; per testare che la soluzione ottima trovata sia davvero tale, si effettua la *prova di ottimalità*, che si realizza esplorando la rimanente parte dello spazio di ricerca e dimostrando che non esistono soluzioni migliori.

Per evitare di visitare l'intero spazio di ricerca, si definiscono delle tecniche di *pruning* (cioè di riduzione) dello spazio di ricerca. Concettualmente si possono determinare due tipi di pruning: quello che si realizza a partire da un ragionamento di *realizzabilità* e quello che si concentra sulla ricerca dell' *ottimalità*. Il pruning riguardante la realizzabilità, noto come Constraint Propagation (Propagazione di Vincoli) [7, 8], elimina quelle combinazioni di valori che non conducono ad alcuna soluzione realizzabile. Il pruning relativo all'ottimalità, invece, rimuove quelle combinazioni di valori che determinano soluzioni peggiori di quella migliore trovata finora.

2.2.1 Metodi Esatti e Incompleti

Il fatto che la maggior parte dei problemi di ottimizzazione combinatoria sia difficile dal punto di vista risolutivo implica che non esiste un metodo efficace per trovare soluzioni ottime e per provare tale ottimalità. Tuttavia molti problemi di ottimizzazione combinatoria di taglia ragionevole possono essere risolti in modo ottimo in tempo ragionevole. Per altri, invece, ci si limita ad utilizzare metodi *incompleti*, che forniscono buone soluzioni, senza provarne l'ottimalità. I metodi esatti e quelli incompleti possono condividere gli stessi criteri di ricerca.

2.2.2 Piani di Taglio

L'idea di risolvere i programmi lineari interi (o mixed integer) rafforzando il rilassamento iniziale con l'aggiunta di disequazioni, dette *piani di taglio* , è nota in programmazione matematica dal 1958, grazie al lavoro di Gomory [34]. Intuitivamente, dato il programma lineare intero (PLI) P, il metodo dei piani di taglio si basa sulla soluzione *iterativa* di una sequenza di programmi lineari (PL) P_L^i, ognuno dei quali approssima P. Ogni PL P_L^i, che caratterizza questa sequenza, migliora la precedente approssimazione P_L^{i-1}. Il PL iniziale, ottenuto eliminando il vincolo di interezza, è il rilassamento lineare P_L^0 di P; la soluzione ottima di P_L^0, detta x^*, fornisce un lower bound (per problemi di minimizzazione) cx^* per la soluzione ottima di P, in cui solitamente è violato il vincolo dell' interezza. La generazione dei piani di taglio determina alcune *disequazioni lineari*, che rimuovono la soluzione x^*, irrealizzabile per P, senza eliminare alcuna soluzione intera di P. Il problema della ricerca della disequazione $\alpha x \leq \alpha_0$, soddisfatta da tutte le soluzioni realizzabili di P e tale che $\alpha x^* > \alpha_0$, è detto *problema di separazione*. In linea di principio si potrebbero aggiungere iterativamente tutte le disequazioni di questo tipo al *convex hull* dell'insieme delle soluzioni di P, ma non è detto che questa scelta sia vantaggiosa; comunque non esiste un modo efficiente per costruire il convex hull di un problema.

Tagli di Gomory

Il problema generale ILP sembra essere inerentemente difficile. Si sono sviluppati parecchi algoritmi, alcuni dei quali anche molto efficaci su certe classi di programmi, ma non si è trovato nessun algoritmo efficiente in generale, che riuscisse a risolvere problemi grandi, come fa l'algoritmo del simplesso per i problemi LP.

Gli algoritmi generali implementati per la classe ILP cadono in due categorie: *cutting plane algorithms*, che sono derivati dall'algoritmo del simplesso e *enumerative algorithms*, che si basano sull'enumerazione intelligente di tutte le possibili soluzioni. In questa sezione viene descritto un semplice cutting-plane algorithm, che ci servirà per descrivere i concetti di base degli algoritmi basati sui tagli.

Si consideri un ILP nella forma standard

$$\begin{aligned} &min\ cx \\ &Ax = b \\ &x \geq 0,\ \ integer \end{aligned} \tag{2.2}$$

dove A, b and c sono interi. Il problema LP senza il vincolo di interezza, cioè

$$\begin{aligned} &min\ cx \\ &Ax = b \\ &x \geq 0 \end{aligned} \tag{2.3}$$

viene detto il *rilassamento* dell' ILP nell'Eq. 2.2.

Se si risolve il rilassamento di un problema ILP, per esempio con l'algoritmo del simplesso, per ottenere una soluzione di base realizzabile x^*, si scopre che in generale x^* non è intero. È naturale pensare di raggiungere una soluzione per il problema originale, arrotondando le coordinate di x^* agli interi più vicini, ma ciò non sempre funziona, potrebbe infatti non esserci nessun punto realizzabile vicino a x^*.

È conveniente a questo punto segnalare due fatti importanti, per capire l' idea principale dei cutting plane algorithms: *i)* se l'ottimo continuo x^*, la soluzione del LP, è intera, allora risolve il corrispondente ILP; *ii)* altrimenti, il costo ottimo cx^* è un lower bound per il costo dell'ottimo discreto cx^0. Parliamo ora dell'idea principale dei cutting plane algorithms. Se si aggiunge un vincolo ad un ILP, che non esclude punti interi realizzabili, allora la soluzione non cambia. La strategia è quella di aggiungere vincoli lineari a un ILP, uno alla volta, finchè la soluzione del rilassamento LP è intera. Poiché non abbiamo escluso nessun punto intero realizzabile, questa soluzione finale dell'ILP rilassato con i vincoli aggiunti risolverà il problema originale ILP. Questo processo è illustrato in Figura 2.1.

La Figura 2.1(a) mostra l'originale ILP e la soluzione continua ottima x^*. In Fig. 2.1(b) viene aggiunto un vincolo lineare che non esclude nessun punto intero realizzabile, detto *piano di taglio* (o semplicemente *taglio*), che attraversa parte dell'insieme realizzabile. Il nuovo ottimo continuo si sposta nel posizione indicata. La Figura 2.1(c) mostra il risultato ottenuto dall'aggiunta di un altro piano di taglio; questa volta l'effetto è quello di rendere intero l'ottimo continuo, che ora risolve l'originale ILP.

Si prende ora in esame un nuovo metodo algebrico per generare i tagli dovuto a R.E.Gomory [1, 34]. Si supponga di avere un'istanza dell'ILP e di

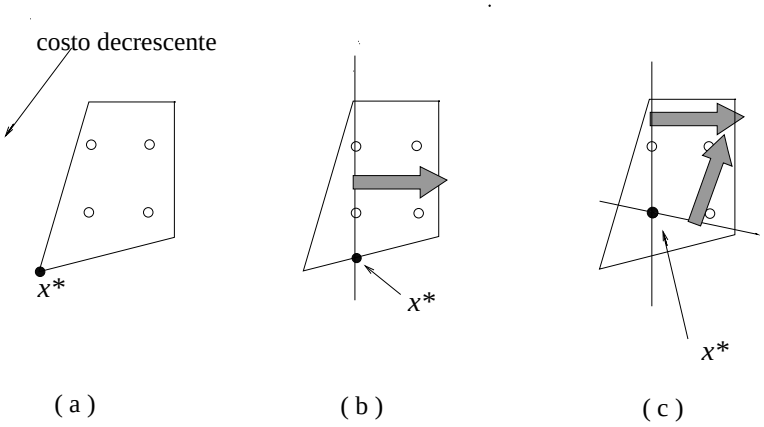

Figura 2.1: Illustrazione di un cutting plane algorithm.

iniziare a risolvere il rilassamento LP con l'algoritmo del simplesso primale, e di ottenere una soluzione ottima continua in base realizzabile con base associata $\mathcal{B}$. Un tipica equazione in un tableau finale è

$$x_{B(i)} + \sum_{j \notin B} y_{ij} x_j = y_{i0} \tag{2.4}$$

per qualche i, $0 \leq i \leq$ m. (Si può considerare $x_{B(0)}$ = -z, dove z è il costo.) È conveniente introdurre la seguente notazione: dato un numero reale y, si indica con $\lfloor$y$\rfloor$ (detta *parte intera di y*) il più grande intero q tale che q $\leq y$. La variabile x nell'Eq. 2.4 è vincolata ad essere non negativa, così si ha che:

$$\sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \leq \sum_{j \notin B} y_{ij} x_j \tag{2.5}$$

Perciò l'Eq. 2.4 diventa

$$x_{B(i)} + \sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \leq y_{i0} \tag{2.6}$$

Nel problema ILP che si sta cercando di risolvere la x è vincolata ad essere intera, e così la parte sinistra dell'Eq. 2.6 è intera. La parte destra può perciò essere sostituita con la sua parte intera senza modificare la relazione, producendo

$$x_{B(i)} + \sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \leq \lfloor y_{i0} \rfloor \tag{2.7}$$

Sottraendo all'Eq. 2.7 l'Eq. 2.4 si ha

$$\sum_{j \notin B} (y_{ij} - \lfloor y_{ij} \rfloor) x_j \geq y_{i0} - \lfloor y_{i0} \rfloor \tag{2.8}$$

Sia

$$f_{ij} = y_{ij} - \lfloor y_{ij} \rfloor \quad i = 0, ..., m \tag{2.9}$$

Il numero f_{ij} è detto *parte frazionaria* di y_{ij} e soddisfa

$$0 \leq f_{ij} < 1 \tag{2.10}$$

Inserendo l'Eq. 2.9 nell'Eq 2.8, si ottiene il vincolo

$$\sum_{j \notin B} f_{ij} \geq f_{i0} \tag{2.11}$$

detto *taglio di Gomory* corrispondente alla *riga i*.

Il progetto ora è quello di aggiungere il taglio di Gomory al tableau iniziale. Per mantenere una soluzione di base, si moltiplica l'Eq. 2.11 con -1 e si aggiunge una variabile di scarto s, ottenendo

$$-\sum_{j \notin B} f_{ij} x_j + s = -f_{i0} \tag{2.12}$$

Dato un tableau con una soluzione di base, che è primale irrealizzabile e duale realizzabile, è più naturale usare l'algoritmo del simplesso duale. Uno o più pivot produrranno un nuovo ottimo continuo o diranno che il primale è irrelizzabile. Quest'ultima possibilità significa che l'originale ILP non ha punti interi realizzabili.

Ecco uno schema dell'intero algoritmo, tratto da [1], comunemente chiamato *algoritmo frazionario duale* , poiché il tableau ha entrate frazionarie e persiste la realizzabilità duale.

```
procedure fractional dual
begin
solve the relaxation of ILP, obtaining optimal solution x*;
feasible := yes
while x* is not integer and feasible = yes  do
  begin
    choose a source row i;
    add the generated Gomory cut and a
```

```
        apply the dual simplex algorithm;
        if  dual is unbounded then feasible := no;
        let x* be the new optimum
      end
    end
```

2.2.3 Metodi di Branch and Bound

Un problema discreto di ottimizzazione combinatoria P può essere modellato da un insieme di variabili $x_1,...,x_n$, un insieme di vincoli su queste variabili $c_1,...,c_m$ e una funzione obiettivo $f(x_1,...,x_n)$. Sia D_i l'insieme dei valori che possono essere assunti dalla variabile x_i, un vincolo c_k può essere definito come un sottinsieme del prodotto cartesiano dei D_i, cioè $c_k \subseteq D_1 \times ....\times D_n$.

Nella struttura del Branch and Bound si usa un albero di ricerca per decomporre ricorsivamente P in sottoproblemi disgiunti più semplici. La decomposizione di un sottoproblema P_k si arresta in due casi: quando si prova che esso è irrealizzabile (perché viola alcuni vincoli, o perché conduce a soluzioni peggiori di quella migliore trovata finora) e quando a tutte le variabili decisionali viene assegnato un valore, e si trova una soluzione realizzabile.

L'organizzazione del Branch and Bound è stata utilizzata su larga scala da entrambe le comunità di AI e OR; il settore OR è interessato soprattutto alle proprietà matematiche dei vari sottoproblemi, mentre la comunità di AI si sofferma sulle decisioni euristiche per la ricerca di soluzioni, concentrandosi prevalentemente sul *branch*.

Nei problemi di ottimizzazione, ogni volta che si trova una soluzione realizzabile, viene aggiunto al problema originale P un ulteriore vincolo, che stabilisce che ogni nuova soluzione deve fornire un valore migliore della funzione obiettivo. Supponiamo che esista un metodo per determinare una valutazione ottimistica del valore della soluzione ottima di un problema P_k, se tale valore è peggiore della miglior soluzione trovata finora, allora non occorre risolvere il problema P_k, poiché sicuramente non porterà ad alcun miglioramento. La valutazione ottimistica del valore della soluzione ottima definisce un *bound* per la funzione obiettivo. In OR, il metodo generalmente usato per determinare tale bound per un dato problema Q, si riduce alla risoluzione di un problema diverso $R(Q)$ (problema rilassato) più facile dal punto di vista risolutivo e tale che ogni soluzione di Q è anche soluzione di $R(Q)$.

Un albero di ricerca può essere esplorato in modi diversi a seconda dell'ordine con cui si considerano i nodi, come illustrato in [1, 10]:

- *Depth First Search* è la più comune procedura di ricerca: ad ogni passo c'è un nodo corrente aperto P_k. Se P_k è un problema irrealizzabile o non ha figli inesplorati, P_k viene chiuso e viene eseguito un *backtrack* che porta la ricerca al nodo padre, che diventa il nuovo nodo corrente aperto. Se P_k è un nodo foglia, cioè se si è trovata una soluzione, la ricerca può continuare (per cercare nuove soluzioni), chiudendo il nodo e facendo backtracking al nodo padre, che diventa il nuovo nodo corrente aperto. Se P_k ha ancora figli inesplorati può ancora essere decomposto, allora il suo primo figlio diventa l'attuale nodo aperto.

- *Breadth First Search*: ad ogni passo i c' è un insieme S^i di nodi correnti aperti, che contiene tutti nodi di profondità i. Nel passo $i+1$ tutti i nodi $P_k \in S^i$, che possono essere ancora decomposti, generano tutti i loro figli, determinando l'insieme S^{i+1}.

- *Best Bound First Search*: ad ogni passo c'è un insieme S di nodi correnti aperti; il nodo avente il bound migliore viene rimosso da S e tutti i suoi figli (se ce ne sono) realizzabili e ancora decomponibili sono inseriti in S.

- *Discrepancy Based Search* è una strategia di ricerca che ha ottenuto risultati molto buoni in parecchi problemi di ottimizzazione combinatoria. In un albero binario la *discrepanza* di un nodo P_k è il numero di volte in cui la ricerca ha scelto il ramo destro, cioè quello esplorato dopo un backtracking, dalla radice fino al nodo P_k. Dato un parametro d, una procedura di ricerca basata sulla discrepanza prima esplora il sottoalbero definito da tutti i nodi con discrepanza minore di d, poi visita il sottoalbero formato dai nodi con una discrepanza compresa tra d e $2d$, ecc. Questa metodologia di ricerca è piuttosto saggia, poiché esplora prima i nodi che seguono l'euristica e poi quelli che la contraddicono, limitando il numero di volte in cui si scelgono i rami destri.

Il metodo di ricerca, comunemente usato dagli studiosi di CP e di OR, che sarà anche quello utilizzato in questa tesi, è il primo citato (*Depth First Search*). Si preferisce adottare questo criterio, perché è quello più comunemente usato ed ha complessità in spazio polinomiale.

La lista dei metodi descritti non è completa, ma comprende i metodi più significativi, che hanno caratterizzato la ricerca in questo campo. Alcune di queste metodologie hanno contribuito notevolmente allo sviluppo di questa tesi. Uno dei pricipali obiettivi di questa dissertazione consiste infatti nell'analizzare il comportamento dei Tagli di Gomory all'interno di un ambiente CP.

2.3 Programmazione con Vincoli

Il linguaggio CP sui Domini Finiti (Finite Domain, FD) [7] è uno strumento efficace per modellare e risolvere i problemi combinatori discreti. Esso configura un problema come un insieme di variabili, legate da un insieme di vincoli matematici o globali, che prendono i loro valori su domini finiti di interi. La ricerca passata ha dimostrato che quest'ultima tipologia di vincoli, opportunamente utilizzata, riesce a migliorare l'efficienza dei risolutori CP. I vincoli globali, infatti, permettono di creare un modello chiaro e conciso di un sottoproblema, che compare nel problema originale e di ridurre efficemente lo spazio di ricerca tramite la *Constraint Propagation* (propagazione di vincoli). Tale procedura rimuove dallo spazio di ricerca alcune combinazioni di assegnamenti variabile-valore che si rivelano irrealizzabili; è un processo iterativo che coinvolge tutti i vincoli del problema e raggiunge un punto fisso quando si realizza un certo livello di consistenza.

Può essere anche definita una funzione obiettivo sulle variabili del problema, tuttavia i risolutori CP non se ne occupano particolarmente, poiché generalmente la riduzione dei domini viene raggiunta ragionando sulla realizzabilità e non sull'ottimalità.

La Programmazione con Vincoli trae origine dalla Soddisfazione di Vincoli (Constraint Satisfaction) [7, 8], che è una teoria sviluppatasi nella comunità dell' AI.

2.3.1 La Soddisfazione di Vincoli

I problemi CSP (Constraint Satisfaction Problem) giocano un ruolo centrale in molti campi dell' Intelligenza Artificiale. Un CSP può essere definito come un insieme di variabili $X_1,...,X_n$, che assumono valori su certi domini finiti $D_1,...,D_n$, e un insieme di vincoli $C_1,...,C_m$, che rappresentano relazioni tra

variabili. Un vincolo $C_j(X_1,...,X_n) \subset D_1 \times .. .. \times D_n$, è una relazione che denota le combinazioni di valori legali per $X_1,...,X_n$, limitando i valori che le variabili possono assumere contemporaneamente. La ricerca sui CSPs riguarda in modo particolare i CSPs binari, dove si considerano solo vincoli binari, cioè vincoli che legano coppie di variabili, come ad esempio $X_i \leq X_j$. Si è cercato poi di estendere i risultati trovati per i vincoli binari ai vincoli che relazionano n variabili.

Ogni problema di vincoli binari può essere rappresentato come un *grafo* in cui i nodi sono le variabili e gli archi sono i vincoli. Ogni nodo ha un dominio associato, che corrisponde al dominio della variabile. Nel seguito ci si riferisce ai nodi con i loro indici i ($i = 1,...,n$), ai domini con D_i e agli archi con il vincolo associato. Se un arco lega le variabili (nodi) i e j, allora il vincolo corrispondente si indica con C_{ij}.

Un CSP può essere risolto usando le tecniche di ricerca con backtracking, che istanziano successivamente le variabili. Se un'istanziazione parziale viola qualche vincolo, allora viene attuato un *backtrack* cronologico, che determina una ricerca (ad esempio di tipo Depth First) dello spazio delle soluzioni. È chiaro che una numerazione completa è inefficiente, perché lo spazio di ricerca cresce esponenzialmente con la dimensione del problema ed è noto che il backtrack cronologico presenta il problema del *trashing*: non si riconosce immediatamente che una soluzione è irrealizzabile, quindi si esplora inutimente tutto il suo sottoalbero, prima di fare backtracking sulla decisione che rappresenta la vera causa dell'irrealizzabilità.

Sono state proposte molte tecniche di propagazione di vincoli per limitare tale problema, che si basano su nozioni di consistenza sui nodi (node consistency), sugli archi (arc consistency), su un cammino (path consistency) e la più generale k-consistency. Nel seguito vengono introdotte le definizioni di consistenza sui nodi e sugli archi, che sono i concetti principalmente usati per propagare i vincoli binari nei risolutori CP .

Definizione 1 (Consistenza sui nodi) *Un nodo i è consistente sui nodi sse per ogni valore $x \in D_i$ vale il vincolo unario $C_i(x)$. Un CSP è consistente sui nodi sse ogni nodo è consistente sui nodi.*

Definizione 2 (Consistenza sugli archi) *Un arco che lega le variabili i e j è consistente sugli archi sse per ogni valore $x \in D_i$ consistente sui nodi, esiste un valore $y \in D_j$ consistente sui nodi tale che valga il vincolo binario $C_{ij}(x,y)$*

e viceversa. Un CSP è consistente sugli archi sse ogni arco è consistente sugli archi.

Una definizione generale di *k*-consistency è stata fornita da Freuder [35]: dati i valori di *k-1* variabili, che soddisfano tutti i vincoli tra queste variabili, allora per ogni *k*-esima variabile esiste un valore nel suo dominio che soddisfa tutti i vincoli tra queste k variabili. Freuder ha inoltre proposto un algoritmo per raggiungere un qualunque grado di consistenza. In base a questa definizione, la consistenza sugli archi corrisponde alla 2-consistency.

Gli algoritmi riguardanti la consistenza sui nodi e sugli archi, descritti più in dettaglio nella sezione 2.3.2, sono frequentemente usati, come tecniche di propagazione di base, nei risolutori CP. Questi algoritmi non risolvono completamente un CSP, ma rimuovono alcune inconsistenze locali, che non appaiono in alcuna soluzione consistente globale. Un CSP può essere consistente sugli archi, ma globalmente irrealizzabile. Vediamo un esempio che illustra chiaramente questo fatto: siano *X, Y, Z* tre variabili aventi tutte lo stesso dominio contenente i valori in $\{a,b\}$ e siano $X \neq Y$, $X \neq Z$ e $Z \neq Y$ i vincoli. Il CSP è consistente sugli archi, poiché per ogni valore nel dominio di ogni variabile ne esiste uno nel dominio di ogni altra variabile che lo soddisfa, tuttavia il CSP è banalmente inconsistente, poiché non ha nessuna soluzione. Notare inoltre che non è 3-*consistent*.

Un risultato generale trovato da Freuder [36] stabilisce che un CSP che coinvolge n variabili può essere risolto completamente (cioè, si può trovare una soluzione senza ricerca), se si realizza un algoritmo di n-consistenza . La complessità di questo algoritmo sarà esponenziale in n. Molti risultati collegano la struttura di un CSP con il livello di consistenza sufficiente per risolverlo. Ad esempio, se il numero massimo di archi che connettono ogni nodo agli altri è k, allora la $(k+1)$-consistenza è sufficiente per risolvere il CSP senza ricerca .

2.3.2 Concetti base di CP (FD)

La Constraint Programming sui Domini Finiti è uno strumento efficace per affrontare problemi (di ottimizzazione) combinatori. In questa sezione vengono presentate alcune nozioni basilari della CP: il modeling, la propagazione di vincoli, la ricerca e l'ottimizzazione.

Modeling

Esiste un *mapping* tra i concetti del CSP introdotti nella sezione precedente e le strutture sintattiche del CP(FD). Nei linguaggi CP(FD) si possono infatti definire variabili che prendono valori su domini finiti e un insieme di vincoli che le connettono. Grazie a tale mapping questi linguaggi ereditano tutti i risultati raggiunti nei CSPs.

Una delle caratteristiche fondamentali dei linguaggi CP(FD) riguarda la facilità con cui è possibile modellare i problemi, che possono essere stabiliti in modo dichiarativo, estesi e modificati; l'espressività e la flessibilità sono sempre state le due qualità predominanti della programmazione con vincoli.

Nei linguaggi CP(FD) le variabili si estendono su domini finiti di interi, che rappresentano i valori che le variabili possono assumere durante la computazione. Per esempio, X_1 :: [1..10] stabilisce che la variabile X_1 può adottare uno dei valori interi compresi tra 1 e 10, mentre X_2 :: [3,5,9] dichiara che la variabile X_2 è 3, o 5 oppure 9. Per quanto riguarda le notazioni, data una variabile X, nel seguito si indicherà il valore minimo (risp. massimo) nel suo dominio con *inf(X)* (risp. *sup(X)*), il suo dominio con *domain(X)* e il suo valore dopo l'istanziazione con *value(X)*. Le variabili sono legate da vincoli matematici, come $X_1 > X_2$, $X_1 < X_2$, $X_1 \geq X_2$, $X_1 \leq X_2$, $X_1 = X_2$, $X_1 \neq X_2$ o globali. Questi ultimi vincoli, che possono essere definiti dall'utente, sono molto potenti all'interno del complesso meccanismo della propagazione. Un tipico vincolo globale è *element(varArray,X, Y)* disponibile in molti risolutori CP(FD) come ECL^{i}PS^e [37] e ILOG Solver [5]. Tale vincolo vale sse, date due variabili X e Y, e un array di variabili *varArray*, Y assume un valore uguale a *varArray[X]*. Un altro vincolo globale largamente utilizzato in CP è *alldifferent*$(X_1, ...X_n)$, che è soddisfatto se le variabili $X_1, ...X_n$ hanno valori diversi.

La struttura di base di un programma che descrive un CSP è la seguente (usiamo una notazione tipica di ILOG Solver, che sarà lo strumento che utilizzeremo nella parte sperimentale di questa tesi):

```
solve(VarArray decisionVarArray){
 makeDecisionVariables(decisionVarArray);
 makeProblemConstraints(decisionVarArray);
 search(decisionVarArray); }
```

Il parametro *decisionVarArray* è un array di variabili, che rappresenta le entità del problema. Nel passo *makeDecisionVariables* si definiscono le variabili con i rispettivi domini, in *makeProblemConstraints* si dichiarano i vincoli del problema. Questi ultimi vengono propagati dal risolutore, che realizza una riduzione dei valori nei domini delle variabili e, o si trova una soluzione (ogni variabile è istanziata con un valore realizzabile), o si individua un fallimento, oppure si inizia il passo etichettato *search*, che sancisce l'implementazione di una strategia di ricerca. L' intero processo viene iterato finchè tutte le variabili sono istanziate, cioè, fino a quando si trova una soluzione.

Per problemi di ottimizzazione, invece, la struttura del programma può essere descritta come segue:

solve(VarArray decisionVarArray, Var obj){
makeDecisionVariables(decisionVarArray, obj);
makeProblemConstraints(decisionVarArray);
minimize(decisionVarArray, obj); }

Per rappresentare la funzione obiettivo si aggiunge il parametro *obj* e il predicato *minimize*, che può essere utilizzato, con riferimento a *obj*, per rintracciare tra le soluzioni realizzabili quella migliore. Il comando *minimize* implementa una forma di Branch and Bound. In generale i linguaggi CP incorporano meccanismi necessari per implementare sia l'albero di ricerca che il Branch and Bound.

Propagazione di vincoli

I problemi di soddisfazione di vincoli (CSPs) potrebbero essere risolti tramite una semplice enumerazione delle soluzioni, tuttavia questo metodo non sarebbe realizzabile nella pratica, poiché lo spazio di ricerca cresce esponenzialmente con la dimensione del problema. Così i CSPs beneficiano della riduzione dello spazio di ricerca realizzata dalla propagazione di vincoli e alcune applicazioni sono risolte in modo più efficiente utilizzando approcci CP, piuttosto che metodi IP.

Durante la computazione i vincoli vengono propagati per ridurre il dominio delle variabili, eliminando i valori inconsistenti; se un dominio diventa vuoto, allora si ha un fallimento e viene attuato un backtrack. L'algoritmo base della

propagazione con vincoli è quello della consistenza sugli archi [16], che sarà descritto successivamente in questa sezione, che elimina quei valori dai domini delle variabili che non soddisfano la condizione illustrata nella definizione della consistenza sugli archi. Esistono anche algoritmi specifici riguardanti i vincoli globali, che sfruttano la semantica del vincolo stesso e sono competitivi in termini di efficienza.

Si consideri, per esempio, il vincolo *allDifferent*(X_1,...,X_n), che vale sse sono assegnati valori diversi a tutte le variabili. Tale vincolo è equivalente a un insieme di *n(n-1)/2* vincoli binari di disuguaglianza, che connettono ogni coppia di valori in (X_1,...,X_n). Esso si può implementare tramite un ragionamento globale sull' insieme delle variabili. Si supponga ad esempio di avere un *allDifferent* tra tre variabili (X_1,X_2,X_3), con domini $D_1 = D_2 = [1,2]$ e $D_3 = [1,2,4]$. Mentre un insieme di disequazioni binarie è consistente sugli archi (se non lo è si modificano i vincoli in modo che lo sia) e quindi non può dedurre l'eliminazione di nessun valore, il vincolo globale può ragionare sulla cardinalità degli insiemi delle variabili e dei valori. Si ha un insieme di variabili (X_1,X_2) con lo stesso dominio $D_1 = D_2 = [1,2]$, aventi la medesima cardinalità uguale 2. I valori *1* e *2* devono essere *risevati* per le variabili X_1 e X_2 (non importa quale dei due valori sarà assegnato alle variabili), quindi non sono più realizzabili per la variabile X_3, che assumerà pertanto il valore *4* senza dover provare i valori 1 e 2.

Una caratteristica interessante della Constraint Programming è l'interazione tra i vincoli, che cooperano tramite le variabili condivise. Un algoritmo di propagazione, che è parte del vincolo stesso, viene implementato quando si verifica un cambiamento nel dominio di una sua variabile. Tale variazione può essere causata, o dall'elimininazione di un valore, o dalla riduzione di un bound del dominio, oppure dall'istanziazione di una variabile (il dominio è ridotto ad un singolo valore). Così, appena un vincolo produce una modifica sul dominio di una variabile X, ecco che sono azionati tutti i vincoli che coinvolgono la X e viene effettata un propagazione sulla base dello stato corrente dei domini delle variabili.

Per capire meglio l'interazione tra i vincoli, si consideri questo esempio. Siano X, Y, Z le variabili tali che X::[*1..5*], Y::[*1..5*] e Z::[*1..5*] e siano $X = Y + 1$, $Y = Z + 1$, $Z = X + 1$ i vincoli. La propagazione di $X = Y + 1$ determina questa riduzione di domini: X::[*2..5*], Y::[*1..4*] e Z::[*1..5*]; il vincolo $X = Y + 1$ non è risolto, quindi potrebbe essere esaminato ancora. La propagazione di

$Y = Z + 1$ riduce i domini a X::[*2..5*], Y::[*2..4*] e Z::[*1..3*]. Il cambiamento del dominio di Y risveglia il primo vincolo, determinando l'eliminazione del valore *2* dal dominio di X. La propagazione di $Z = X + 1$, infine, rimuove tutti i valori dal dominio di X provocando un fallimento. L'ordine con chi vengono considerati i vincoli non condiziona il risultato, ma può influenzare l' esecuzione del processo di propagazione.

L'algoritmo della consistenza sugli archi, precedentemente citato, è uno degli strumenti più diffusi in ambiente CP per realizzare la propagazione dei vincoli. Eccone una breve descrizione tratta da [11].

Arc Consistency

$Arc(V_i, V_j)$ è *arc consistent* se per ogni valore x nel dominio corrente di V_i esiste un valore y nel dominio di V_j tale che $V_i = x$ e $V_j = y$ soddisfano il vincolo binario tra V_i e V_j e viceversa.

Chiaramente, un arco (V_i, V_j) può essere reso consistente semplicemente eliminando quei valori dal dominio di V_i per i quali la condizione sopra non è verificata. L'eliminazione di alcuni valori non rimuove nessuna soluzione dell'originale CSP. I seguenti algoritmi, presi da [16], sfruttano proprio questo fatto.

```
procedure REVISE(V_i, V_j);
DELETE ← false;
for each x ∈ D_i do
        if there is no such v_j ∈ D_j
        such that (x, v_j) is consistent,
        then
                delete x from D_i;
                DELETE ← true;
        endif;
endfor;
return DELETE;
end_REVISE
```

Per rendere ogni arco del *constraint graph* consistente, non è sufficiente eseguire la procedura REVISE per ogni arco solo una volta. Quando REVISE riduce il dominio di qualche variabile V_i, allora ogni arco (V_i, V_j) esaminato precedentemente deve essere analizzato ancora, perché alcuni dei valori del

dominio di V_j potrebbero ora essere incompatibili con i restanti membri del dominio V_i.

Il seguente algoritmo, anch'esso preso da [16], realizza la consistenza sugli archi per l'intero grafo G.

```
procedure AC-1
Q ← {(V_i, V_j) ∈ arcs(G), i ≠ j};
repeat
        CHANGE ← false;
        for each (V_i, V_j) ∈ Q do
            CHANGE ← (REVISE(V_i, V_j) or CHANGE);
        endfor;
until not(CHANGE);
end_AC
```

Il maggior problema riguardante questo algoritmo è che una revisione, che ritorna true per ogni arco in qualche iterazione, obbliga tutti gli archi ad essere riesaminati nella prossima iterazione, anche se poi solo un piccolo numero di essi viene effettivamente modificato. Mackworth [16] ha proposto una variazione (detta AC-3) di questo algoritmo, che elimina l'inconveniente suddetto. Questo algoritmo effettua un'analisi ulteriore solamente di quegli archi che possono essere modificati dalla revisione precedente.

```
procedure AC-3
Q ← {(V_k, V_m) ∈ arcs(G), i ≠ j};
while Q non empty
        select and delete any arc (V_i, V_j) from Q;
        if (REVISE(V_k, V_m)) then
          Q ← ∪{(V_i, V_k) such that (V_i, V_k) ∈ arcs(G), i ≠ k, i ≠ m}
        endif;
endwhile;
end_AC
```

Si supponga che la taglia del dominio di ogni variabile sia d e che il numero totale dei vincoli binari (cioè, il numero degli archi nel constraint graph) sia e. La complessità di un algoritmo di consistenza sugli archi di [16] è $O(ed^3)$, come afferma [17]. Variazioni e miglioramenti a questi algoritmi sono stati apportati

da Mohr e Handerson [18] nel 1986, che hanno presentato un algoritmo con complessità di $O(ed^2)$, da Han [19] nel 1988 e da Chen [20] nel 1991.

Un altro algoritmo largamente usato nella comunità di AI, per realizzare la propagazione dei vincoli, è quello relativo alla consistenza sugli limiti. Arc e node consistency realizzano un buon pruning dei domini nei CSPs binari, ma non funzionano bene se il problema contiene vincoli primitivi con più di due variabili, poiché questi vincoli vengono ignorati durante la verifica della consistenza. Inoltre la modifica dei domini fatta dagli algoritmi di AC può rendere i domini non compatti, cioè non rappresentabili tramite un intervallo di interi, cosa che è generalmente desiderata per motivi di efficienza e risparmio di spazio. Per questo motivo si usa la bound consistency.

Bound Consistency

Un vincolo primitivo aritmetico c è *bound consistent* con dominio D se per ogni variabile $x \in vars(c)$, c'è:

- un assegnamento di numeri *reali*, detti $d_1, d_2, ..., d_k$, per la variabili rimanenti in c, dette $x_1, x_2..., x_k$, tale che $min_D(x_j) \leq d_j \leq max_D(x_j)$ per ogni d_j e

  $$\{x \to min_D(x), x_1 \to d_1, ..., x_k \to d_k\}$$

  è una soluzione di c e

- un altro assegnamento di numeri *reali*, detti $d'_1, d'_2, ..., d'_k$, a $x_1, x_2..., x_k$ tale che $min_D(x_j) \leq d'_j \leq max_D(x_j)$ per ogni d'_j e

  $$\{x \to max_D(x), x_1 \to d'_1, ..., x_k \to d'_k\}$$

  è una soluzione di c.

Un CSP aritmetico, cioè un CSP in cui le variabili prendono valori su un dominio finito di interi e i vincoli primitivi sono aritmetici, con vincolo $c_1 \wedge ... \wedge c_n$ e dominio D è *bound consistent* se ogni vincolo primitivo c_i è *bound consistent* con D per $1 \leq i \leq n$.

Poiché la consistenza sui limiti dipende solo dai limiti superiore e inferiore dei domini delle variabili, quando si testa questo tipo di consistenza, si devono considerare solo i domini che assegnano ranges a ogni variabile.

L'algoritmo, tratto da [7], che trasforma un CSP aritmetico in uno equivalente consistente sui limiti è il seguente:

C is an aritmetic constraint;
C_0 is a set of primitive constraints;
D e D_1 are domains;
$c_1, ..., c_n$ are primitive constraints;
and x is a variable.

bounds_consistent(C, D)
  let C be of the form $c_1 \wedge ... \wedge c_n$
  $C_0 := \{c_1, ..., c_n\}$
  while $C_0 \neq \emptyset$ do
    choose $c \in C_0$
    $C_0 := C_0 \setminus \{c\}$
    $D_1 :=$ bounds_consistent_primitive(c, D)
    if D_1 is a false domain then return D_1 endif
    for $i := 1$ to n do
      if there exists $x \in vars(c_i)$ such that $D_1(x) \neq D(x)$ then
        $C_0 := C_0 \cup \{c_i\}$
      endif
    endfor
    $D := D_1$
  endwhile
  return D

L'algoritmo riceve in input un CSP aritmetico con vincolo C e dominio D e ritorna come output un dominio D_1 tale che il CSP con vincolo C e dominio D_1 è bounds consistent ed equivalente al CSP in input. L'algoritmo utilizza la funzione parametrica bounds_consistent_primitive(c, D), che applica le regole di propagazione per il vincolo primitivo c al dominio D e ritorna il nuovo dominio. L'algoritmo elabora iterativamente l'*insieme attivo* dei vincoli primitivi C_0. Un vincolo primitivo è attivo se può essere bounds consistent con il dominio corrente D. Il ciclo while seleziona ripetutamente un vincolo primitivo c dall'insieme attivo e modifica il dominio D in modo che diventi consistente sui limiti con c. Se questo dà un dominio vuoto l'algoritmo termina, altrimenti i vincoli, che potrebbero non essere più consistenti sui limiti a causa dei cambiamenti fatti a D, vengono aggiunti all'insieme attivo. L'algoritmo termina

quando non ci sono più vincoli primitivi nell'insieme attivo, poiché ciò significa che tutti i vincoli primitivi sono bounds consistent con i domini correnti.

Ricerca

Terminato il processo di propagazione dei vincoli, si possono presentare tre diversi scenari: *i)* un dominio diventa vuoto; *ii)* ogni dominio contiene un solo valore; *iii)* alcuni domini contengono più di un valore. Nel primo caso il problema è irrealizzabile. Nel secondo caso si dà quel singolo valore ad ogni variabile e si controlla se è una soluzione oppure no. Nel terzo caso, poiché la constraint propagation non è completa, per trovare le soluzioni occorre esplorare lo spazio di ricerca tramite l'albero di ricerca (Branch and Bound per i problemi di ottimizzazione), che è solitamente visitato usando una procedura *Depth First*. Nella comunità di CP si sono sviluppate parecchie strategie dipendenti dal problema, ma quella più diffusa è molto semplice: *seleziona una variabile* non istanziata e le assegna un valore tra quelli che appartengono al suo dominio, se il suo dominio è vuoto effettua un *backtrack*. Tale metodologia è molto simile alle strategie di branching usate negli algoritmi di OR, ma ha la differenza che non dipende dal calcolo della soluzione ottima di un problema rilassato (bound). Se, ad un dato nodo, si vuole dividere il problema in sottoproblemi più semplici, si possono utilizzare diversi tipi di vincoli per realizzare il branching. La modalità con cui esplorare l'albero di ricerca è una decisione da non sottovalutare affatto, dal momento che da essa dipende l'efficienza della constraint propagation. Deve essere chiaro che durante il processo di ricerca i vincoli sono presi in grande considerazione e sono propagati, per realizzare un pruning dello spazio di ricerca. Infatti, quando una variabile viene istanziata, vengono azionati tutti i vincoli che la coinvolgono e incomincia un nuovo processo di propagazione. Una caratteristica importante, che i linguaggi CP hanno ereditato dalla comunità di AI, è l'idea di inserire nei metodi risolutivi la conoscenza del problema in questione. Tutti questi linguaggi, avendo predicati di alto livello, consentono di scrivere facilmente le euristiche di ricerca e quindi di sviluppare metodi di branching sofisticati, per realizzare risolutori sempre più efficienti. Alcuni solvers recenti, come ad esempio ILOG Solver, oltre ad incorporare strategie di ricerca standard, offrono all'utente delle primitive, che gli permettono di costruire metodi nuovi per visitare l'albero di ricerca.

Ottimizzazione

In alcune applicazioni non si cerca una soluzione realizzabile, ma una ottima, con riferimento a una certa funzione obiettivo f definita sulle varibili del problema. Per ottenere questa particolare tipologia di soluzione, i sistemi CP sfruttano il meccanismo del Branch and Bound, risolvendo un insieme di problemi decisionali di realizzabilità (cioe, si trova una soluzione se essa esiste), che determinano in successione soluzioni sempre più vantaggiose. In particolare, ogni volta che si scopre una soluzione realizzabile z^* (il cui costo associato è $f(z^*)$), si aggiunge a ogni sottoproblema dell'albero di ricerca rimanente il vincolo $f(z) < f(z^*)$, dove x è il vettore delle variabili del problema. Lo scopo di questo nuovo vincolo, detto *upper bounding constraint*, è di rimuovere quelle porzioni dell'albero di ricerca che non possono portare a soluzioni più convenienti di quella migliore trovata finora. L' approccio appena descritto presenta però due svantaggi: *i)* CP non fa affidamento su algoritmi sofisticati per calcolare il lower bound e l'upper bound della funzione obiettivo, ma trae questi valori dai domini delle variabili; *ii)* il legame tra la funzione obiettivo e le variabili decisionali è, in generale, piuttosto debole e non produce un filtraggio effettivo dei domini delle variabili.

Pertanto recentemente si sono investite parecchie risorse per inserire nei comuni metodi CP tecniche incentrate sul *ragionamento di ottimalità*, che sono tipiche di OR; per esempio, in parecchi algoritmi CP, sono stati utilizzati i bounds derivanti dalla soluzione ottima di un rilassamento linere e in molti vincoli globali si sono incorporate varie tipologie di rilassamento, per raggiungere una forma di pruning basata sui bounds e sull'analisi di sensitività.

2.4 Programmazione Matematica

In questa sezione si descrivono brevemente alcune nozioni di Programmazione Matematica, in particolare, la forma standard adottata per modellare i Programmi Lineari Interi e i metodi implementati per definire e risolvere il rilassamento di un problema. Come tecnica di risoluzione generale si considera il meccanismo del Branch and Bound, che si basa su una procedura di ricerca caratterizzata da due fasi: nella prima si calcola la soluzione ottima di un rilassamento, nella seconda si esegue un passo di branching.

2.4.1 Programmazione Lineare Intera

In un problema ILP si deve ottimizzare una data funzione lineare su un insieme finito di soluzioni. Lo spazio delle soluzioni è determinato da un insieme di vincoli lineari (uguaglianze e disuguaglianze) e dal vincolo di interezza, entrambi definiti sulle variabili del problema. Senza perdita di generalità, si considerano problemi di minimizzazione, allora un ILP si può scrivere nella seguente forma standard:

$$min\{c^T x \mid Ax = b,\ x \in Z_+^n\}$$

dove Z_+^n è l'insieme di n-vettori interi non negativi, $x = \{x_1,...,x_n\}$ è il vettore che rappresenta le variabili del problema, c è un n-vettore, A una matrice m $\times$ n, e b un m-vettore. L'insieme $S = \{x \in Z_+^n \mid Ax = b\}$ è detto *regione realizzabile* e $x \in S$ *soluzione realizzabile.* La funzione $z(ILP) = c^T x$ è detta *funzione obiettivo* e un punto realizzabile $x^* \in S$ tale che $c^T x^* \leq c^T x$ per ogni $x \in S$ è detto *soluzione ottima.*

I problemi di Programmazione (Lineare) Intera sono $\mathcal{NP}$-hard, cioè sono molti difficili computazionalmente. Una metodologia efficace usata per risolvere queste problematiche è quella del Branch and Bound descritta nella sezione 2.2.3, dove i rilassamenti del problema appaiono estremamente validi se si vuole visitare l' albero di ricerca in modo efficiente. Lo scopo di un rilassamento all'interno di una procedura di Branch and Bound è quello di dimostrare che non si deve risolvere un sottoproblema $\mathcal{NP}$-hard, perchè non porterebbe ad una soluzione più conveniente di quella migliore trovata finora. Perciò, per poter realizzare un grande pruning dell'albero, il valore della soluzione ottima del rilassamento dovrebbe essere il più possibile vicino a quello del problema originale.

2.4.2 Rilassamenti

Esistono molti metodi per generare il rilassamento $R(P)$ di un problema P. Questa sezione ne descrive alcuni, segnalando che è possibile applicarli una o più volte durante il processo di risoluzione.

Nello schema del Branch and Bound un rilassamento $R(P)$ è conveniente se è più semplice di P dal punto di vista risolutivo. In generale si realizzano questi rilassamenti per ottenere problemi risolvibili polinomialmente. I Programmi Lineari sono spesso adottati come rilassamenti, perché trovare la soluzione

ottima di un LP ha in media una complessità tempo polinomiale, sebbene l'algoritmo comunemente usato, l'*algoritmo del simplesso*, sia esponenziale nel caso peggiore.

Rilassamento ottenuto tramite l'eliminazione di un vincolo

Il metodo più semplice per creare il rilassamento di un dato problema P, è rimuovere alcuni dei vincoli che lo definiscono. Si consideri per esempio un Traveling Salesman Problem (TSP, Problema del Commesso Viaggiatore) P; esso cerca un cammino di costo minimo che visiti tutti nodi in un grafo esattamente una volta. Togliendo il vincolo che impone che i nodi siano percorsi da un solo cammino, il problema rilassato può essere così definito : $R(P)$ cerca un certo numero di cammini che visitino tutti i nodi in un grafo esattamente una volta con costo minimo. Quest'ultimo problema, detto Assignment Problem (AP, Problema di Assegnamento), rappresenta chiaramente un rilassamento del TSP ed è un po' più semplice da risolvere.

Rilassamento Lineare

Un caso speciale del rilassamente descritto sopra è il *Rilassamento Lineare*. Se il problema originale P può essere descritto da vincoli lineari su variabili intere, la rimozione dei vincoli di integrità per tutte le variabili intere conduce al Rilassamento Lineare. Il problema risultante LP ha la forma:

$$min\{\, c^T x \mid Ax = b,\ x \in R^n_+ \,\}$$

e può essere risolto con tecniche di Programmazione Lineare (per esempio, l'algoritmo del simplesso).

Se la soluzione ottima del problema LP, chiamiamola x^*, è intera, allora è soluzione realizzabile e ottima anche per il problema originale P. In generale però x^* viola il vincolo di integrità e contiene valori frazionari. Per alcuni problemi specifici x^* è sempre intera: è il caso, per esempio, dei problemi aventi una matrice A *totalmente unimodulare*, che possono essere risolti in maniera ottima risolvendo semplicemente il corrispondente rilassamento lineare.

Rilassamento Lineare e Piani di Taglio

In molti casi, per ottenere bounds migliori per il problema originale, è conveniente restringere la formulazione del Rilassamento Lineare, generando iterativamente alcuni *piani di taglio*.

Come menzionato nella sezione 2.2.2, dato il problema lineare intero P, il metodo dei piani di taglio si basa sulla soluzione iterativa dei rilassamenti lineari di P, detti P_L. La soluzione ottima x^* di P_L fornisce un lower bound $c^T x^*$ per il valore della soluzione ottima di P, dove in generale vengono violati alcuni vincoli del problema originale. Lo scopo della generazione dei piani di taglio è trovare alcune disequazioni lineari corrispondenti al vincolo violato, che tolgano x^*, irrealizzabile per P, senza rimuovere nessuna soluzione realizzabile di P. Il problema di trovare una disequazione $\alpha x \leq \alpha_0$, soddisfatta da tutte le soluzioni realizzabili di P, e tale che $\alpha x^* > \alpha_0$, è detto *problema di separazione*. Uno potrebbe aggiungere iterativamente tutte le disequazioni di questo tipo al convex hull dell'insieme delle soluzioni, ma non è detto che sia conveniente, poiché il problema di separazione, come quello originale, è $\mathcal{NP}$-hard.

Pertanto, in generale, il metodo usato consiste nell' aggiungere in modo iterativo un sottinsieme S_{cut} di disequazioni, ottenute risolvendo il problema di separazione per alcune *classi* di disuguaglianze, possibilmente tramite euristiche. Ogni aggiunta di piani di taglio (o semplici tagli) dà luogo a un nuovo rilassamento lineare P_{cut}^{new} di P e di solito il valore della soluzione ottima aumenta rispetto a P_L (P_{cut}^{new} è un'approssimazione migliore di P_L).

Quando l'aggiunta dei tagli diventa inefficace (il bound non aumenta più), la generazione dei piani di taglio si ferma e il programma lineare finale rappresenta un rilassamento *stretto* del problema originale P. La generazione iterativa dei piani di taglio nella struttura del Branch and Bound è stata introdotta da Padberg e Rinaldi per il problema TSP [38]. Tale metodologia, nota com il nome di *Branch and Cut*, è stata applicata a una grande varietà di problemi di ottimizzazione combinatoria.

2.5 ILOG Optimization suite

Fondato nel 1987, ILOG [5] sviluppa delle librerie di ottimizzazione e di visualizzazione, che sono utilizzate da parecchie migliaia di utenti in tutte le aree

dell'industria a del commercio. ILOG's optimization suite include i seguenti pacchetti informatici:

- **ILOG Solver**: un motore C++ di Constraint Programming orientato agli oggetti;
- **ILOG CPLEX**: è una libreria C per la Programmazione Lineare e la Programmazione Intera Mista.
- **ILOG Hybrid**: è una libreria C++ che permette di far cooperare gli algoritmi della Programmazione con Vincoli di ILOG Solver e gli algoritmi della Programmazione Intera Mista di ILOG CPLEX.
- **ILOG Scheduler**: una libreria C++ aggiunta a ILOG Solver che fornisce modeling objects e algorimi di scheduling basati sui vincoli per problemi di scheduling e di allocazione delle risorse;
- **ILOG Dispatcher**: una libreria C++ aggiunta a ILOG Solver che offre metodi di Ricerca Locale basati sulla Constraint Programming per problemi di spedizione e di vehicle routing;
- **ILOG Planner**: una libreria C++ aggiunta a ILOG Solver che permette di integrare i metodi della Programmazione Lineare all'interno di quelli della Programmazione con Vincoli;
- **ILOG Configurator**: una libreria C++ aggiunta a ILOG Solver che fornisce modeling objects e algoritmi per risolvere problemi di configurazione, dove vengono prima selezionate e poi assemblate alcune parti di un catalogo nelle quantità desiderate, per incontrare le richieste di connessione e per raggiungere la configurazione cercata.
- **ILOG OPL Studio**: è un ambiente grafico interattivo ed un linguaggio di alto livello che permette di creare rapidamente il prototipo di molte applicazioni di ottimizzazione.
- **ILOG Concert Technology**: è un linguaggio che modella i problemi di ottimizzazione combinatoria fornendo un insieme di oggetti C++, che consentono di separare il modeling dagli algoritmi utilizzati per trovare le soluzioni. Un modello Concert puo' essere *estratto* da tutte le librerie di ottimizzazione che contengono gli algoritmi per risolvere il problema.

Questa tesi, per quanto riguarda l'implementazione degli algoritmi proposti, dipende molto dai prodotti di ottimizzazione forniti da ILOG, e in particolare da ILOG Solver e da CPLEX.

2.5.1 ILOG Solver

ILOG Solver è una libreria C++ per la Programmazione con Vincoli, che propone classi e funzioni che permettono agli utenti di modellare problemi di ottimizzazione e di applicare algoritmi a questi modelli. Include classi predefinite di variabili; classi predefinite di vincoli aritmetici e vincoli globali insieme con un meccanismo per implementare nuovi vincoli; algoritmi di ricerca predefiniti come pure uno strumento per scrivere metodi di ricerca definiti dall'utente.

Le variabili, che sono rappresentate come oggetti con un dominio associato, sono di vari tipi: variabili intere, variabili enumerate (variabili il cui dominio è un insieme finito di indirizzi di oggetti C++), variabili reali, variabili booleane e variabili di insieme (variabili il cui dominio è rappresentato da un insieme di insiemi). Usando il meccanismo C++ del sovraccaricamento (overloading) degli operatori è possibile adoperare tutti i tipi di operatori aritmetici (come +, -, $*$, /) su tali variabili.

I vincoli, che sono oggetti che legano le variabili, sono associati a uno (o più) algoritmi di propagazione. Possono essere *posted*, e in tal caso devono essere soddisfatti, oppure *meta-posted*, cioè trattati come variabili booleane e combinati attraverso operatori aritmetici.

La ricerca delle soluzioni viene tipicamente realizzata mediante l'albero di ricerca, ma è anche possibile implementare degli algoritmi personali adottando le primitive proposte da Solver. L'albero di ricerca viene visitato, per default, con il criterio *Depth First*, ma ancora un volta, grazie ad alcune funzioni predefinite della libreria, è possibile definire criteri propri.

2.5.2 ILOG CPLEX

ILOG CPLEX fornisce librerie di C, C++ e Java utili per risolvere problemi di programmazione lineare. In particolare risolve linearmente problemi di ottimizzazione in cui la funzione obiettivo può essere espressa come una funzione lineare. Le variabili del modello possono essere dichiarate come continue oppure vincolate ad assumere valori interi.

ILOG CPLEX si presenta in tre forme distinte per soddisfare le varie esigenze degli utenti:

- ILOG CPLEX **Interactive Optimizer**: è un programma eseguibile che può leggere un problema in modo interattivo o da files scritti in formati speciali, risolvere il problema e dare la soluzione interattivamente o scriverla su opportuni files di testo.

- **Concert Tecnology**: è un insieme di librerie che offrono modeling facilities che consentono al programmatore di inserire ILOG CPLEX optimizer in applicazioni C++ e Java. Questa è forma di CPLEX utilizzata nella tesi.

- ILOG CPLEX **Callable Library**: è una libreria di C che permette di usare ILOG CPLEX optimizers in applicazioni scritte in C, Visual basic, Fortran o altri linguaggi che chiamano funzioni C.

ILOG CPLEX è principalmente utilizzato per risolvere problemi di ottimizzazione lineari, in cui le variabili della funzione obiettivo sono continue in senso matematico senza gap tra valori reali. Per risolvere tali problemi CPLEX implementa soprattutto ottimizzatori basati sugli algoritmi del simplesso (simplesso primale e duale). CPLEX viene anche utilizzato nella risoluzione di problemi in cui la funzione obiettivo è quadratica (Quadratic Programs, QPs) e in cui alcune variabili o tutte sono vincolate da assumere valori interi (Mixed Integer Programs, MIPs).

2.5.3 ILOG Hybrid

ILOG Hybrid è una libreria C++, che permette di far cooperare la Programmazione con Vincoli e la Programmazione Intera Mista, combinando i vantaggi delle due discpline: il mantenimento di una soluzione ottima rilassata e la generazione di piani di taglio dal lato MIP e la propagazione di vincoli dopo ogni search decision dal lato CP. Tale libreria, che può essere usata per risolvere problemi di resource allocation e production planning, permette di sfruttare facilmente procedure di branch & cut nella ricerca delle soluzioni di tali problemi. ILOG Hybrid, che racchiude ILOG CPLEX in un vincolo globale, permette di risolvere programmi lineari ad ogni nodo di un albero di ricerca di

ILOG Solver. La soluzione ottima rilassata, cioè la soluzione di un programma lineare inteso come rilassamento, aiuta a guidare la ricerca, a restringere i domini delle variabili e a scoprire in anticipo casi di irrealizzabilità.

Capitolo 3

Integrazione di Programmazione con Vincoli e Ricerca Operativa

3.1 Introduzione

Negli ultimi anni l'integrazione delle tecniche di Intelligenza Artificiale e di Ricerca Operativa ha determinato il miglioramento, in termini di efficienza e di ottimalità, delle soluzioni di problemi di ottimizzazione combinatoria molto complessi. Situato alla confluenza dei due campi, Constraint Programming, un paradigma di programmazione che sfrutta tecniche di soddisfazione di vincoli, è stato considerato un ambiente idoneo al raggiungimento di tale integrazione. La seguente trattazione sarà limitata alla *Constraint Programming on Finite Domains* (CP(FD)), poiché essa è particolarmente adatta a trattare problemi di ottimizzazione combinatoria. Una descrizione completa di CP può essere trovata in [7]. Questo capitolo presenta brevemente le principali direzioni dell'integrazione esplorate in letteratura.

Un primo passo verso l'integrazione riguarda il *confronto* degli approcci di CP e OR [13, 14]; si definiscono somiglianze, differenze e proprietà del problema, per poter preferire nelle varie circostanze un approccio all'altro. Per alcuni problemi, come ad esempio il warehouse location problem, descritto in [5, 15], Constraint Programming sembra essere più efficace. I vantaggi principali di CP sono: modelli semplici e concisi, primitive ricche a livello semantico per sviluppare facilmente strategie di ricerca dipendenti dal problema e potenti algoritmi di propagazione per ridurre efficacemente lo spazio di ricerca. Altri problemi, invece, sono risolti più semplicemente usando tecniche di Integer Li-

near Programming (ILP), si vedano ad esempio i problemi di crew scheduling e flow aggregation in [13]. In generale, i problemi *puri* (senza side constraints) sono risolti con tecniche di MP, capaci di sfruttare la loro struttura geometrica, mentre quelli *meno puri*, cioè con side constraints, sono trattati con metodi di CP. Un tipico esempio di questo comportamento è il Traveling Salesman Problem (TSP). Gli approcci di Branch and Cut per il TSP raggiungono risultati migliori degli approcci puri di CP di parecchi ordini di grandezza, ma, non appena si considerano ad esempio i vincoli *Time Windows*, questo divario viene sostanzialmente ridotto (si vedano [3] e [4]).

Oltre ai confronti su differenti applicazioni, alcuni studi individuano direttive che aiutano il designer a scegliere l'approccio migliore per il problema da risolvere. A questo scopo Bockmayer e Kasper hanno proposto uno schema di unificazione, Branch and Infer, descritto in [9], che offre una visione uniforme di ILP e CP(FD), che riguarda sia il modeling che il solving.

Branch and Infer identifica le componenti basilari del modeling e del solving su cui confrontare i concetti di ILP e quelli CP. Dal punto di vista del modeling vengono individuati due tipi di vincoli: *primitivi* e *non primitivi*. I primi sono quelli che vengono trattati facilmente da un risolutore di vincoli, mentre i secondi sono quelli per i quali non esistono metodi efficienti che esaminino in tempo polinomiale la realizzabilità e l'ottimalità. Nei sistemi CP i vincoli primitivi sono i seguenti:

$$Prim \ = \ \{x \leq u, x \geq b, x \neq v, x = y, integer(x)\}$$

dove x e y sono variabili, u, v, b sono vincoli. Tutti gli altri vincoli sono *non primitivi*. In ILP, invece, tutte le equazioni e le disequazioni lineari sono primitive, mentre il vincolo di interezza è non primitivo.

Lo scopo, relativo al solving, di una computazione in un sistema basato sui vincoli è inferire vincoli primitivi da quelli non primitivi. In CP, la propagazione dei vincoli causa la rimozione di valori e il restringimento di bounds (vincoli primitivi per il sistema CP). In ILP, la generazione di piani di taglio inferisce nuove disequazioni lineari (vincoli primitivi per il sistema ILP). Quando non si possono più dedurre vincoli primitivi, sia CP che ILP attuano il branching, per dividere il problema in un insieme di sottoproblemi più semplici.

Per realizzare l'ottimizzazione, CP implementa una semplice forma di Branch and Bound, dove ogni volta che una soluzione è trovata viene imposto un vincolo che stabilisce che le prossime soluzioni devono avere un valore migliore

della funzione obiettivo. In ILP il Branch and Bound si basa sulla soluzione ottima di un rilassamento del sottoproblema corrente. Tale soluzione offre una valutazione ottimistica della miglior soluzione che può essere trovata per quel sottoproblema.

I lavori di unificazione sono così utili, sia per individuare i concetti comuni e le differenze, sia per fornire le basi dell'integrazione, come discusso in [13].

Nelle prossime tre sezioni si discutono in dettaglio le principali direzioni d' integrazione delle tecniche di MP in CP: modeling del problema , algoritmi di filtro, metodi di ricerca.

3.2 Modeling

CP offre linguaggi dichiarativi, flessibili e facilmente estendibili, per esprimere problemi di soddisfazione e di ottimizzazione. I maggiori strumenti industriali di CP introducono i vincoli globali, per una descrizione concisa del problema, ed efficaci algoritmi di propagazione.

D'altra parte, una forte tradizione dipende dai modelli matematici adatti alle tecniche di soluzioni di MP. Modellare problemi attraverso ILP è in generale più complesso e coinvolge un gran numero di variabili e vincoli.

Il primo problema, per integrare le tecniche di MP in CP, consiste nel decidere se far coesistere i differenti modelli, se farli cooperare o se fonderli in unico linguaggio. In ogni caso si dovrebbe definire una mappa tra le due parti.

In letteratura sono state indagate tre direzioni di integrazione:

1. in alcuni approcci, CP è mantenuto come modeling language; sia il modello di MP, che la mappa tra CP e MP sono nascosti all'utente, come illustrato in Fig. 3.1. Un esempio è il lavoro di Refalo [2]. In generale questi approcci, che dipendono dalla trasformazione automatica dei vincoli di CP in programma lineare, hanno solitamente due vantaggi: rendere l'integrazione trasparente all'utente e facilitare la creazione del modeling dei problemi tramite il linguaggio CP;

2. in alcuni approcci il problema può essere stabilito usando sia i modelli di CP che quelli di MP. I due modelli cooperano tramite le variabili condivise nel constraint store, come mostrato in Fig. 3.2. Ovviamente il vantaggio consiste in una maggior flessibilità rispetto all'approccio pre-

cedente, dovuto ad un maggior controllo sulla parte MP; lo svantaggio è che l'utente deve scrivere modelli più complessi per i problemi;

3. alcuni approcci fondono i modelli di CP e MP, proponendo un modello unico in cui viene inserita sia la parte MP che quella CP (si veda la Fig. 3.3). Ancora una volta il modeling del problema risulta più complicato che in metodi di puro CP, ma una visione uniforme del problema potrebbe aiutare a sviluppare soluzioni ibride.

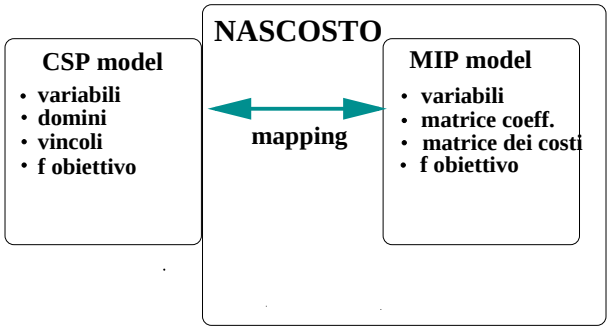

Figura 3.1: Primo metodo di integrazione

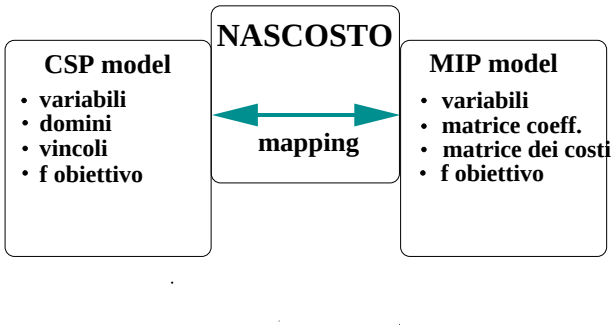

Figura 3.2: Secondo metodo di integrazione

Recentemente si sono sviluppati due nuovi liguaggi *OPL* [6] e *Concert* [5]. In questi linguaggi il modello è indipendente dalla tecnica di soluzione: un problema modellato usando Concert può essere risolto combinando CP e MP in modi arbitrari, nel senso che ogni vincolo può essere mandato ad un risolutore di CP o a uno di MP o ad entrambi (in questo ultimo caso i due solvers comunicano attraverso le variabili condivise), come illustra la Fig. 3.4.

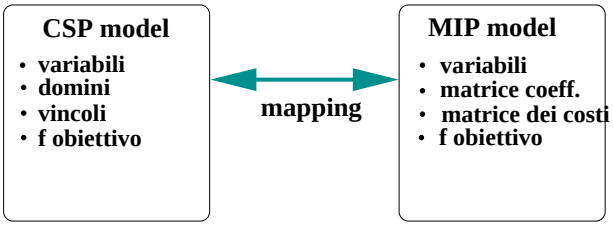

Figura 3.3: Terzo metodo di integrazione

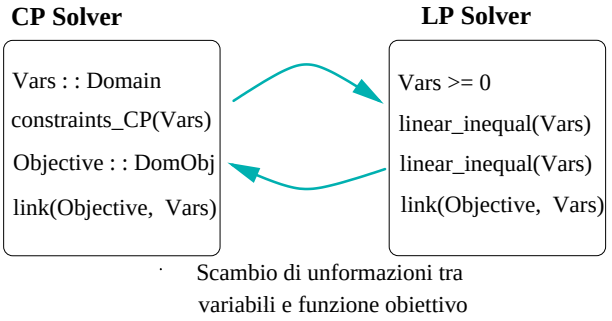

Figura 3.4: Scambio di informazioni tra i solvers CP e LP

Un punto importante sull'integrazione del modello riguarda quale parte del problema modellare usando CP e quale usando MP. Alcuni approcci duplicano le informazioni, modellando una parte del problema nei due modi. Per esempio nella trasformazione automatica del problema, l'intero modello viene duplicato nei due risolutori. Il solver di MP lavora su un insieme di disequazioni lineari, che corrispondono ai vincoli trasformati, mentre il risolutore di CP lavora all'interno di un dato vincolo globale. Infine, alcuni approcci suddividono il problema in sottoparti e modellano ogni parte nel risolutore che meglio le si addice.

Il metodo di integrazione relativo al modeling, che sarà adottato in questa tesi, descritto più in dettaglio nella prossima sezione, sfrutta il lavoro di Refalo [2]: CP è mantenuto come modeling language e, sia il modello di MP che

la mappa tra CP e MP sono nascosti all'utente. Quindi ricade nella prima tipologia di integrazione, descritta in Fig. 3.1.

3.2.1 Formulazione Lineare dei modelli CP

CP offre una gran varietà di *modeling facilities*, come i vincoli globali e quelli logici, che permettono di creare modelli chiari e concisi per esprimere problemi di ottimizzazione combinatoria. I modelli CP contengono informazioni strutturali utili, che permettono di sviluppare algoritmi di riduzione dei domini efficienti per vincoli *high level* e di realizzare strategie di ricerca ad hoc. Implementazioni industriali (per esempio, ILOG Solver [5]) hanno mostrato l'efficacia di CP per la risoluzione di problemi in diverse aree.

Anche le tecniche MIP sono efficienti e largamente utilizzate per risolvere problemi di ottimizzazione combinatoria. Con MIP il modello è limitato a un insieme di vincoli lineari binari interi o di variabili reali. Una formulazione MIP è spesso lontana dal linguaggio naturale e contiene poche informazioni strutturali; l'enfasi riguarda i buoni rilassamenti del problema. I risolutori MIP determinano una soluzione ottima rilassata del rilassamento lineare e generano piani di taglio per rafforzare il rilassamento [4].

Refalo ha proposto un modo per fornire una formulazione MIP dei modelli CP. Nel seguito saranno descritti in modo particolare le riformulazioni del vincolo *alldifferent* [21], dei vincoli sulle occorrenze di valori come *among*. Le formulazioni presentate possono essere viste come un'estensione del lavoro fatto da Wallace sull'autentica trasformazione di un constraint logic program (che include vincoli alldifferent) in modello MIP [24] e del lavoro fatto sulla trasformazione delle formule logiche in modelli CP [25, 26].

La formulazione lineare del vincolo è divisa in due parti: l'insieme dei vincoli lineari che sono *richiesti* per la linearizzazione e l'insieme dei vincoli lineari che sono *ritardati*, come ad esempio i piani di taglio, che sono aggiunti alla formulazione lineare quando sono violati dalla soluzione ottima rilassata.

Un modello CP è composto da variabili e vincoli. I vincoli possono essere (1) vincoli di dominio, (2) vincoli elementari come vincoli aritmetici, (3) vincoli high-level che possono essere logici [27] o globali [22].

Una *riformulazione lineare* di un insieme S di vincoli su un dominio finito di variabili V è un insieme S' di vincoli lineari su un insieme di variabili $V \cup V'$ tale

che entrambi gli insiemi hanno le stesse soluzioni sulle variabili V. Un insieme di vincoli lineari definisce un insieme convesso. Sia Q l'insieme delle soluzioni di un vincolo high level c. La più forte formulazione lineare di c rappresenta il più piccolo insieme convesso contenente le soluzioni di Q. Questo insieme è detto *convex hull* di Q e i vincoli lineari associati rappresentano una formulazione *sharp* (precisa) di Q [25]. Varie formulazioni sharp possono rappresentare lo stesso insieme di soluzioni. Comunque, in molti casi la cardinalità di una formulazione può essere molto grande, allora è preferibile dividere il problema in due insiemi:

$$\mathcal{F}(\mathrm{c}) = \mathcal{L}(\mathrm{c}) \cup \mathcal{D}(\mathrm{c})$$

dove l'insieme $\mathcal{L}(\mathrm{c})$ è l'insieme dei vincoli lineari richiesti nella riformulazione e $\mathcal{D}(\mathrm{c})$ è l'insieme dei vincoli lineari la cui aggiunta può essere ritardata. Da segnalare che nella riformulazione si considerano anche i vincoli di dominio.

Poiché molte condizioni logiche sui vincoli lineari possono essere rappresentate come una disgiunzione di insiemi convessi, la ricerca in questo campo si è focalizzata sulla rappresentazione lineare di queste disgiunzioni. Una formulazione sharp di $D = \{Ax \leq b \vee A'x \leq b'\}$ può avere un numero esponenziale di vincoli, quando viene espressa sulla variabile x. Introducendo due nuovi vettori di variabili x^1 e x^2 e due variabili binarie γ_1 e γ_2, una formulazione sharp di D è data dal sistema

$$\mathcal{L}(D) = \begin{cases} Ax^1 \leq \gamma_1 b \\ Ax^2 \leq \gamma_2 b' \\ x = x^1 + x^2 \\ \gamma_1 + \gamma_2 = 1 \\ \gamma_i \in \{0,1\} \; per \; i \in \{1,2\} \end{cases}$$

Questo risultato fondamentale, noto come *formulazione disgiuntiva*, è stato sviluppato da Balas e Jeroslow [31, 32]. Questa trasformazione sarà applicata in seguito ai vincoli che possono essere definiti come disgiunzioni di insiemi lineari di vincoli.

Una proprietà importante di una formulazione lineare, usata insieme alla ricerca Branch and Bound, è l'*hereditary sharpness*: la formulazione rimane sharp quando le variabili sono fissate da un nodo padre a un nodo figlio dell'albero di ricerca [25]. In CP, quando si considerano i risolutori ibridi, si deve

mantenere la sharpness anche in seguito alla riduzione dei domini. In [28] viene presentato un principio generale, detto *tigth cooperation*, in cui le formulazioni lineari delle strutture di alto livello vengono dinamicamente aggiornate tramite il variable fixing e la generazione di piani di taglio, quando si riducono i domini. Ciò va oltre il classico metodo di cooperazione dei risolutori [29, 30], dove il solver della riduzione dei domini e l'ottimizzatore lineare si scambiano solo i bounds delle variabili.

Lo stesso approccio viene applicato qui. Per ogni vincolo, la formulazione viene aggiornata in modo tale che le modifiche fatte durante la ricerca sul modello CP si riflettano sulla formulazione lineare. Per esempio, si supponga che nella formulazione disgiuntiva sopra l'alternativa $Ax \leq b$ diventi irrealizzabile rispetto al nuovo dominio sulla variabile x ad un nodo dell' albero di ricerca. Fissare la variabile γ_1 a 0 mantiene la sharpness della formulazione rispetto ai nuovi domini. Al contrario, se il dominio di x implica il vincolo $Ax \leq b$, la variabile γ_1 deve essere fissata a 1.

Riformulazione dei Vincoli

Questa sezione presenta la riformulazione dei vincoli di dominio e di alcuni vincoli di alto livello, quali *alldifferent* e *among*.

- Vincolo di Dominio
Una formulazione lineare di un vincolo di dominio $x \in D$ è ottenuta dalla formulazione disgiuntiva di $\bigvee_{a \in D}(x = a)$. Per rendere più semplice la scrittura, si denota la variabile binaria introdotta per l'alternativa $x = a$ con $v_{x=a}$. Si assume $v_{x=a} = 0$ quando $a \notin D$. La formulazione lineare di un vincolo di dominio è questa:

$$\mathcal{L}(x \in D) = \begin{cases} \sum_{a \in D} v_{x=a} = 1 \\ 0 \leq v_{x=a} \leq 1 \; per \; a \in D \\ integer(v_{x=a}) \; per \; a \in D \end{cases}$$

$$\mathcal{D}(x \in D) = \left\{ \; x = \sum_{a \in D} a \times v_{x=a} \right.$$

L'aggiunta dei vincoli che legano le variabili binarie con la variabile originale è ritardata, poiché la formulazione lineare di vincoli di alto livello spesso riguarda solo le variabili binarie. Ritardando questi vincoli si riduce in modo significativo la cardinalità del problema lineare da risolvere.

Gli algoritmi di propagazione di vincoli determinano la riduzione dei domini delle variabili. La formulazione lineare può essere aggiornata quando il dominio D è ridotto ad un nuovo dominio D'. Per mantenere la sharpness della formulazione, le variabili vengono fissate in base ai valori rimossi, secondo queste regole di propagazione:

$$\begin{aligned} \forall \alpha \in D, \alpha \notin D' &\iff v_{x=a} = 0 \\ D' = \{\alpha\} &\iff v_{x=a} = 1 \end{aligned}$$

Al contrario, se alcune variabili della riformulazione vengono fissate con tecniche di programmazione intera, come reduced cost fixing [33], il dominio originale viene aggiornato.

Tutte le formulazioni lineari descritte in questa sezione utilizzano ancora le variabili introdotte per la riformulazione del vincolo di dominio. Perciò queste regole sono sufficienti per preservare la sharpness dei vincoli globali presentati in seguito. Si noti inoltre che, in molti casi, queste regole di aggiornamento sono sufficienti per assicurare che il vincolo ritardato sia soddisfatto, di conseguenza non sarà necessario aggiungerlo.

- Vincolo alldifferent

Il vincolo alldifferent$(x_1, ..., x_n)$ è soddisfatto se le variabili $x_1, ..., x_n$ hanno tutte valori diversi. Una formulazione sharp ben nota di questa vincolo è quella di un bipartite matching [33].

Sia $K = \bigcup_{i=1}^{n} D_i$ l'unione dei domini. La linearizzazione di questo vincolo stabilisce che ogni valore di K può essere dato al massimo una volta a una delle variabili $x_1, ..., x_n$. Si ha dunque

$$\mathcal{L}(alldifferent(x_1, ..., x_n)) = \left\{ \sum_{i=1}^{n} v_{x_i=j} \leq 1,\ j \in K \right\}$$

- Vincolo sulle occorrenze di valori

Molte richieste in problemi pratici limitano il numero di valori che un insieme di variabili può avere in una soluzione. Questo è il caso di *among* [22].

Il vincolo $among(y, [x_1, ..., x_n], [a_1, ..., a_k])$, dove x_i e y sono variabili e a_i sono numeri reali, è soddisfatto se esattamente y variabili tra $x_1, ..., x_n$ assumono valore nell'insieme $\{a_1, ..., a_k\}$. La formulazione di questo vincolo specifica che ogni valore a_i deve essere assunto y volte dalle variabili x_i:

$$\mathcal{L}(among(y, [x_1, ..., x_n], [a_1, ..., a_k])) = \left\{ y = \sum_{i=1}^{n} \sum_{j=1}^{k} v_{x_i=a_j} \right\}$$

3.3 Algoritmi di Filtro

CP sfrutta le tecniche di soddisfazione dei vincoli per risolvere problemi combinatori (di ottimizzazione). La constraint propagation ha molte somiglianze con i problemi di riduzione di Ricerca Operativa. Il problema è considerato come problema di *realizzabilità* e la constraint propagation è usata per dedurre la diminuzione dei domini e bounds più stretti per le variabili; non appena la propagazione raggiunge un punto fisso, un branching suddivide il problema in sottoproblemi indipendenti. Comunque, questo semplice metodo enumerativo non prende in considerazione nessun ragionamento di *ottimalità*.

Nelle prossime due sezioni saranno discusse le possibili integrazioni delle tecniche di MP in CP, sia per quanto riguarda la realizzabilità, che per quanto concerne l'ottimalità.

3.3.1 Indagine sulla realizzabilità

La prima integrazione delle tecniche di MP in CP si può trovare negli algoritmi di propagazione dei vincoli globali. Un esempio ben noto dell'uso dei risultati OR, per sviluppare algoritmi di propagazione efficienti, è l'algoritmo di *Edge Finder* [13].

Gli algoritmi di propagazione possono essere reinterpretati come algoritmi specifici di OR, inseriti in componenti software di CP. Questi algoritmi devono essere *polinomiali* e *incrementali*, due caratteristiche cruciali, affinché un algoritmo sia usato efficientemente in CP. Gli algoritmi di propagazione devono essere veloci, poiché lo stesso algoritmo potrebbe essere chiamato parecchie volte all'interno di uno stesso nodo dell'albero di ricerca prima che venga raggiunto un punto fisso. Comunque la tendenza generale in CP è quella di basarsi sullo schema del branching (usando euristiche dipendenti dal problema), più che sul calcolo ad ogni nodo, molto costoso in termini di tempo. Gli algoritmi oggetto di studio dovrebbero essere incrementali, poichè sono determinati dai cambiamenti nei domini delle variabili loro associate. Invece di risolvere nuovamente l'intero problema di consistenza, un algoritmo incrementale è capace di ristabilire una situazione di consistenza, ragionando solo sui cambiamenti dalla condizione di consistenza.

Sebbene gli algoritmi incrementali non siano nuovi in OR, non sono ritenuti importanti come in ambiente CP. Una possibile direzione di ricerca per l'integrazione di CP e OR potrebbe essere rappresentata proprio dagli algoritmi

incrementali, se questi fossero studiati più approfonditamente dai ricercatori di OR.

3.3.2 Indagine sull'ottimalità

La principale limitazione dei sistemi CP si manifesta quando si considera una funzione obiettivo in problemi di *ottimizzazione* combinatoria. Esplorando l'albero, non appena si raggiunge una foglia e si trova una nuova soluzione di valore z^*, CP aggiunge semplicemente un nuovo vincolo, che impone che la prossima soluzione deve avere un valore migliore[1] di z^*.

Il nuovo vincolo in generale si propaga poco sulle variabili decisionali del problema. Per esempio, si consideri il Traveling Salesman Problem, che cerca di minimizzare la somma dei costi. Rappresentare il problema con una variabile per nodo, identificare il prossimo nodo nel cammino ottimo, la riduzione dei domini di queste variabili, sono tutti eventi che modificano molto poco i bounds della funzione obiettivo e viceversa. In questi casi, infatti, solamente i nodi profondi possono essere esaminati (quando quasi tutti i rami sono stati creati), producendo così un grande albero di ricerca.

D'altra parte, MP ha sviluppato fin dagli anni '50 tecniche molto efficaci per calcolare lower bounds (per problemi di minimizzazione) sul valore della funzione obiettivo attraverso la soluzione di problemi rilassati. L'integrazione di queste tecniche in CP ha ottenuto ovviamente grande attenzione, e si sono esplorati due diversi livelli di integrazione.

1. Il primo livello di integrazione riguarda la definizione di rilassamenti per il problema completo. Questo corrisponde alla trasformazione del modello di CP in programma lineare, già discusso nella Sezione 3.2. In questo caso abbiamo due solvers separati, che interagiscono attraverso le variabili condivise. Il solver CP sfrutta la soluzione ottima del programma lineare rilassato calcolato da un risolutore di Programmazione Lineare, sia per aggiornare il bound della funzione obiettivo, che per guidare la ricerca. Possono essere usati anche i costi ridotti per scopi di filtraggio, per rimuovere dai domini i valori peggiori di quelli ottimi. D'altra parte il solver LP sfrutta la propagazione CP per fissare o cambiare i bounds delle variabili.

[1] più grande (risp. più piccolo) di z^* per problemi di minimizzazione (risp. massimizzazione)

2. Il secondo livello di integrazione riguarda il calcolo dei bounds rispetto ai vincoli globali. Questi vincoli modellano parti ben strutturate dell'intero problema, così il rilassamento su queste parti può essere definito e risolto efficacemente. Le soluzioni di questi rilassamenti, che possono essere calcolati da algoritmi specifici, forniscono dei bounds validi e l'analisi di sensitività (per esempio LP reduced costs) per l'intero problema. Ovviamente, la qualità di questi bounds può non essere molto elevata, poiché essi si riferiscono a una parte specifica dell'intero problema. Comunque, il loro uso in congiunzione con la propagazione porta ad interessanti risultati all'interno di una struttura CP (si veda l'articolo [3], per avere maggiori dettagli sulla propagazione basata sui costi).

Entrambi i metodi possono trarre beneficio dalle tecniche di generazione dei *piani di taglio*. Quando l'intero problema è modellato come un LP, l'aggiunta iterativa dei piani di taglio restringe il bound. La generazione ripetitiva dei tagli è molto utile soprattutto quando il modello LP richiede un numero esponenziale di vincoli (si esamini, ad esempio, il classico rilassamento lineare del TSP con i sub-tour elimination constraints in [4]).

L'integrazione dei tagli in CP è stata esaminata a lungo ed è ancora oggetto di studio per molti ricercatori, che vogliono mantenere una formulazione *stretta* del problema durante la ricerca. Recentemente, Ottosson e Refalo [9, 2] hanno realizzato lavori interessanti in questo campo, mantenendo una formulazione stretta del problema tramite una generazione locale di piani di taglio validi.

L'integrazione suddetta non è l'unica esaminata, si è anche cercato di chiarire la relazione esistente tra piani di taglio e propagazione, per riconoscere somiglianze, differenze e possibili metodi di cooperazione. Comunque, la maggior parte dei metodi proposti ha due obiettivi: il raggiungimento di equazioni valide a partire dalla propagazione dei vincoli e la creazione di potenti meccanismi di inferenza attraverso i piani di taglio (a questo proposito si veda il Tutorial di Hooker [14] e la tesi di Ottosson [9]).

3.4 Ricerca : l'albero

Questa sezione descrive l'integrazione dei metodi di CP e OR rispetto alle strategie di ricerca. Sia CP che ILP usano il criterio del Branch and Bound; in questo contesto la comunità di CP/AI ha trazionalmente prestato particolare

attenzione ai problemi dipendenti dalle euristiche e ha recentemente proposto metodi di esplorazione dell'albero innovativi e molto promettenti. Dall'altra parte, la comunità di MP è solita usare l'informazione della soluzione ottima del rilassamento, per guidare la ricerca. Un altro metodo, che ha prodotto risultati interessanti sull'integrazione dei due campi, è quello della Local Search (ricerca locale).

In questa sezione si analizza solo il primo metodo citato, quello relativo all'albero di ricerca. Per quanto riguarda il criterio della Ricerca Locale, è possibile trovare una descrizione dettagliata nella tesi di Focacci [10].

Un metodo basto sull'albero di ricerca è individuato da due caratteristiche importanti: lo schema di branching, che definisce la forma dell'albero, e il metodo di esplorazione, che determina il sottoalbero da esplorare. Focalizzando l'attenzione sugli algoritmi costruttivi, nel seguito saranno descritti brevemente gli schemi di branching più usati.

Algoritmi Costruttivi

Un algoritmo di ricerca globale produce una soluzione prendendo decisioni e facendo backtracking in caso di fallimento. Le decisioni prese in un ramo rappresentano un vincolo da aggiungere al problema. Alcuni schemi generali di branching selezionano una variabile dal modello (per esempio, quella con il più piccolo numero di valori nel dominio) e la istanziano: in questi casi generali è difficile interpretare lo stato del sistema prima di raggiungere una soluzione. La situazione è differente per gli schemi di branching, che sono specifici del problema, dove le decisioni ad ogni punto di scelta costruiscono una piccola parte della soluzione finale. Per esempio, per i problemi di scheduling, i ranking algorithms costruiscono il programma di una macchina in ordine cronologico, decidendo quale compito deve essere eseguito per primo, quale per secondo, ecc.; per i problemi di timetabling, gli algoritmi di assegnamento decidono i compiti di una persona (o di un gruppo di persone) per uno slot di tempo. Questi algoritmi di ricerca globali sono detti *algoritmi di ricerca costruttivi*: i loro stati potrebbero essere infatti interpretati come rilevanti soluzioni parziali (schedulazione a tempo breve per la progettazione solo di un sottinsieme di lavori oppure tabelle di tempo solo per un sottinsieme di persone) ed è facile valutare un bound della funzione obiettivo, aggiugendo il

contributo delle decisioni passate a una valutazione dell'impatto che ci sarà in seguito alle decisioni prese.

La ricerca in un algoritmo costruttivo è guidata dall'euristica: ad ogni punto di scelta viene esaminata una funzione h per tutte le possibili opzioni e ogni scelta viene valutata in base ai valori crescenti di h; si considera preferibile la decisione che minimizza h. Sebbene l'euristica usata sia sempre dipendente dal problema, si possono tuttavia individuare alcuni principi generici.

Euristica basata sul First-Fail

L'euristica di tipo First-Fail, come prima cosa, si concentra sulla risoluzione della parte più difficile del problema. Per esempio, in problemi di scheduling, se esiste una risorsa che rappresenta un elemento bloccante del problema, è saggio schedulare prima quella e solo successivamente le altre. Molto spesso, in un problema CSP, l'euristica First-Fail consiste nello scegliere la variabile con il dominio più piccolo e istanziarla ad un valore. Qualche volta consiste anche nello scegliere la variabile coinvolta nel maggior numero di vincoli. Questa tipologia di euristica si focalizza solo sulla realizzabilità di un problema, trascurando il discorso dell'ottimalità.

Euristica basata sul Rilassamento

La strategia di ricerca tipicamente adottata in ILP considera la soluzione ottima di un problema rilassato; se essa soddisfa tutti i vincoli del problema originale, allora è una soluzione di quest'ultimo, altrimenti viola qualche vincolo. Un'euristica frequentemente utilizzata fa il branching sul vincolo violato e lo rafforza. Per esempio, quando si usa il rilassamento lineare, il branching prende solitamente una variabile x_i con valore non intero nella soluzione ottima rilassata ($\neg\text{integer}(x_i^*)$) e impone $x_i \geq \lceil x_i^* \rceil \; OR \; x_i \leq \lfloor x_i^* \rfloor$. Il vantaggio è che in ogni ramo generato uno dei vincoli violati dal nodo padre è rimosso. La soluzione ottima rilassata può anche essere utilizzata come *suggerimento*. Nel precedente esempio, si supponga che $x_i \in \{0, 1\}$ e che $x_i^* = 0.1$; è saggio fare il branching su x_i provando ad assegnargli prima il valore 0 (il più vicino al valore della soluzione ottima rilassata) e, in caso di backtracking, il valore 1. Alternativamente, si può usare l'analisi di sensitività di LP, per selezionare il valore che genera il più piccolo incremento del lower bound.

Euristica basata sul Regret

Una strategia di ricerca di CP, che prende in considerazione la funzione obiettivo è l'euristica del *massimo rincrescimento*. Il rincrescimento di una variabile x può essere definito come il costo aggiuntivo da pagare oltre all'ottimo, se a x *non* è assegnato il suo valore ottimo. Chiaramente il rincrescimento può essere calcolato solo se si conoscono tutte le soluzioni delle problema, tuttavia anche un'approssimazione del rincrescimento può essere un'informazione utile per definire un'euristica. L'euristica in esame propone di scegliere per prima la variabile con il massimo rincrescimento, così da minimizzare il rischio di pagare un costo alto, se l'assegnamento migliore diventasse irrealizzabile in seguito a decisioni euristiche sbagliate.

Esempio: si considerino due variabili x_1, x_2, con domini correnti $D_1 = D_2 = [0, 1]$. Si supponga che assegnare x_1 a 0 costi 10, mentre assegnarlo a 1 costi 100, assegnare x_2 a 0 costi 20, mentre assegnarlo a 1 costi 40. Il rincrescimento di x_1 può essere calcolato (valutato euristicamente) come $100 - 10 = 90$, mentre il rincrescimento di x_2 come $40 - 20 = 20$. L'euristica del massimo rincrescimento suggerisce di assegnare prima x_1, perché se il suo miglior valore (0) diventasse irrealizzabile, la funzione obiettivo aumenterebbe di 90.

3.5 Conclusioni

Questa breve esposizione è stata intrapresa per individuare le principali direzioni dell'integrazione delle tecniche di MP e di CP(FD). Le procedure di MP possono essere utilizzate per accrescere l'efficienza dei metodi CP. Comunque, nonostante anche MP possa trarre beneficio da questa integrazione, sono ancora pochi gli studiosi impegnati in questa direzione. Dal punto di vista del solving, in ambiente MP si sono sfruttate le soluzioni realizzabili dei problemi trovate con CP, come punto di partenza per l'algoritmo del simplesso. Per quanto riguarda il modeling, CP offre un aiuto notevole al settore MP, ancora troppo legato alla tradizione.

Il campo dell'integrazione di CP e MP è piuttosto giovane e molte direzioni di ricerca sono ancora aperte. Molte tecniche di MP devono ancora essere esplorate per una loro possibile integrazione in ambiente CP . Allo stesso modo, per molti criteri di CP non è stato ancora esaminato il loro potenziale impiego

in ambito MP. Da un punto di vista metodologico, una questione ancora aperta è quella di determinare quale tecnica o quale integrazione di tecniche sia più idonea per una data applicazione.

In questa tesi adotteremo il primo dei metodi di integrazione di CP e OR citati nella sezione 3.2. Utilizzeremo in ambiente CP i tagli di Gomory [1], strumenti specifici della Ricerca Operativa e valuteremo se questa modalità di integrazione migliori la performance dei sistemi CP o OR puri nei problemi di ottimizzazione.

Capitolo 4

Gli algoritmi

In questa sezione vengono descritti gli algoritmi implementati nella tesi, e in particolare quello realizzato per indagare se sia conveniente o meno introdurre in ambiente CP i tagli di Gomory, tagli che hanno la peculiarità di non essere specifici per il problema.

In passato si è cercato di integrare le due discipline CP e OR utilizzando dei tagli specifici per il problema e si sono raggiunti risultati interessanti. In questa tesi si cerca di valutare se sia possibile generalizzare il risultato della ricerca precedente, in modo da avere un buon metodo generale per trattare una qualunque classe di problemi.

Per valutare l'efficienza della metodologia di integrazione attuata utilizzando i tagli di Gomory in ambiente CP si sono implementati due algorimi che rispettivamente sfruttano tecniche di sola CP e di sola OR. L'algoritmo CP è un algoritmo tipico della Programmazione con Vincoli che si basa sulla ricerca con backtracking. L'algoritmo ILP è un algoritmo tipico di branch & cut.

L'algoritmo ibrido, che è l'elemento portante di questa tesi, prova ad esaminare una strategia di soluzione che integra CP e OR, risolvendo ad ogni nodo dell'albero di ricerca CP il rilassamento del corrispondente problema MIP e aggiungendo in maniera moderata i tagli di Gomory per rafforzare il rilassamento. La soluzione ottima rilassata risultante viene utilizzata per guidare la ricerca con backtrack che si sviluppa in ambiente CP.

Gli algoritmi realizzati nella tesi hanno la peculiarità di essere algoritmi generali che possono essere applicati a una qualunque classe di problemi. In questo capitolo tali algoritmi vengono pertanto presentati facendo riferimento ad una classe generica di problemi.

4.1 Gli algoritmi CP e ILP

Gli algoritmi CP e ILP implementati presentano alcune differenze sostanziali che riguardano la modellizzazione del problema tramite variabili, vincoli e funzione obiettivo e la strategia di risoluzione adottata.

4.1.1 L'algoritmo CP

L' algoritmo realizzato dal lato CP è caratterizzato dalla seguente sequenza di operazioni.

1. Modellizzazione del problema tramite variabili e vincoli : essa si basa sulla formulazione CSP della classe di problemi in esame.

2. Propagazione iniziale dei vincoli : essa rimuove tutti i valori dai domini delle variabili che non prenderanno parte a nessuna soluzione e riduce lo spazio di ricerca da esaminare.

3. Applicazione della strategia di soluzione all'albero di ricerca risultante dal punto 2. : è una ricerca in profondità, in cui si prova come prima cosa ad assegnare alla variabile con dominio di taglia minima il primo valore del suo dominio.

4. Nuova propagazione dei vincoli : essa rimuove dai domini *correnti* tutti i valori che violano i vincoli. Da notare che questa propagazione non rimuove i valori dai domini effettivi delle variabili ma solo da quelli correnti.

5. Se il valore scelto per la variabile considerata determina un fallimento, tale valore viene eliminato dal suo dominio effettivo, viene fatto un backtrack nell'albero di ricerca e si prova ad assegnarle l'elemento più piccolo del suo nuovo dominio.

6. Si continuano a ripetere le operazioni descritte ai punti 3, 4 e 5, finchè si trova una soluzione realizzabile, cioè una combinazione di assegnamenti di valori alle variabili che soddisfano i vincoli, a quel punto l'algoritmo termina.

4.1.2 L'algoritmo ILP

L'algoritmo realizzato in ambiente ILP, un algoritmo di tipo *branch & cut*, è determinato della seguente sequenza di operazioni.

1. Modellizzazione del problema tramite variabili, vincoli e funzione obiettivo : essa si basa sulla formulazione ILP della classe di problemi in esame.

2. Risoluzione di una serie di sottoproblemi continui : per diriger in modo efficiente questi sottoproblemi si costruisce un albero in cui ogni sottoproblema è un nodo. La radice dell'albero è il *rilassamento continuo* del problema intero originario.

3. Se la soluzione del rilassamento ha una o più variabili frazionarie, si cerca di aggiungere al problema dei tagli, cioè dei vincoli che eliminano le parti della regione realizzabile del rilassamento che contengono soluzioni frazionarie.

4. Se la soluzione del rilassamento ha ancora una o più variabili intere con valore frazionario, dopo aver applicato i tagli, si effettua il branching su una variabile frazionaria per generare due nuovi sottoproblemi, ognuno dei quali ha bounds più restrittivi sulla variabile del branching, in particolare, nel caso di variabili binarie, un nodo fisserà la variabile a 0, l'altro a 1. La variabile scelta non soddisfa a nessun ordine di priorità automatico.

5. I sottoproblemi risultanti possono avere come risultato una soluzione tutta intera, una soluzione realizzabile o un'altra soluzione frazionaria. Se si è in quest'ultimo caso viene ripetuto il processo descritto ai punti 3 e 4. Durante la ricerca si seleziona come prossimo nodo da esaminare quello con la migliore funzione obiettivo. Ogni volta che l'algoritmo trova una soluzione intera, rende quella soluzione la soluzione *corrente* e quel nodo il nodo *corrente* ed effettua il pruning nell'albero di tutti quei sottoproblemi con funzione obiettivo peggiore di quella corrente.

4.2 L'algoritmo ibrido CP/ILP

Il metodo di risoluzione ibrido implementato in questa tesi prevede l'esecuzione della seguente sequenza di operazioni.

1. Modellizzazione del problema in CP tramite l'impiego di variabili, domini, vincoli (vedi punto 1. dell'algoritmo CP).

2. Propagazione di vincoli : i vincoli sono propagati con l'obiettivo di ridurre i domini delle variabili, tramite l'eliminazione di valori inconsistenti. I principali algoritmi di propagazione usati per i vincoli binari sono quelli della consistenza sugli archi (arc-consistency algorithm) e della consistenza sui limiti (bound consistency algorithm), descritti in [11, 7].

3. Trasformazione in problema MIP : il modello CP, ottenuto dopo la propagazione, viene formulato in modo lineare come descritto in [2], prestando particolare attenzione al cambiamento dei vincoli globali [12] (vedi punto 1. dell'algoritmo ILP).

4. Rilassamento del vincolo di interezza e risoluzione del problema MIP rilassato tramite tecniche OR e l'aggiunta dei tagli di Gomory [1] : l'algoritmo usato per risolvere il rilassamento del problema alla radice dell'albero di ricerca è il Simplesso Primale, quello utilizzato per risolvere il problema ai nodi è il Simplesso Duale (quest'ultima scelta, consigliata in [41], è la più appropriata per trattare frequenti cambiamenti di bounds e aggiunte di vincoli); i tagli di Gomory sono aggiunti alla formulazione lineare per migliorare i bounds del problema rilassato. Il risultato conseguito in questo passo è costituito dalla soluzione ottima e dal valore ottimo z_{lb} del problema rilassato.

5. Utilizzazione della soluzione ottima rilassata appena trovata per determinare la strategia di branching da adottare nel lato CP: si sceglie, come prossima variabile su cui attuare il branching, la variabile CP corrispondente a quella che nella soluzione ottima rilassata ha valore frazionario più vicino a 0.5 oppure quella con valore frazionario più grande e si stabilisce di seguire prima la strada che istanzia tale variabile con il valore intero più vicino, oppure con il valore intero più lontano, o con il valore immediatamente più grande o con quello immediatamente più piccolo.

6. Nuova propagazione dei vincoli : il vincolo appena aggiunto alla formulazione CP può determinare un'ulteriore riduzione dei domini delle variabili.

7. Iterazione : il processo di risoluzione riparte dal punto 3.

Questa sequenza di operazioni viene effettuata più volte, fino al raggiungimento di una soluzione realizzabile del problema originale.

Capitolo 5

Risultati Sperimentali

5.1 Il problema

Il problema esaminato in questa tesi riguarda il completamento di problemi di Latin Square parziali. La struttura di tale problema è simile a quella trovata in molti problemi della vita reale come scheduling, timetabling, error correcting codes e wavelength routing in fiber optics networks [39].

Un *latin square parziale* (partial latin square, PLS) di ordine n è un array $n \times n$ in cui ogni cella è o vuota o contiene esattamente uno dei "colori" $1, ..., n$ e ogni "colore" compare al massimo una volta in ogni riga o colonna. Un *latin square* (latin square, LS) di ordine n è un PLS di ordine n senza celle vuote. Un latin square parziale si dice *completabile* se si possono colorare le sue celle vuote per ottenere un latin square.

Questa tesi esamina il problema di trovare un' estensione di un latin square parziale con il massimo numero di celle colorate, applicando diverse strategie di ricerca, in particolare, un approccio basato sulla Constraint Satisfaction (CSP), un approccio incentrato sulla Linear Programming e un approccio ibrido CSP/Linear Programming.

Dettagli implementativi dell'algoritmo CP

La modellizzazione e la risoluzione sono state realizzate tramite l'impiego di ILOG Solver [5].

La modellizzazione del problema tramite variabili e vincoli si basa sulla seguente formulazione CSP [39], dove n è la dimensione del latin square parziale

da completare e $alldiff$ è un vincolo che stabilisce che tutte le variabili presenti in esso devono avere valori diversi [21] :

$$x_{i,j} \in \{1, .., n\} \ \ \forall i, j$$
$$x_{i,j} = k \ \ \forall i, j \, tale \, che \, PLS_{i,j} = k$$
$$alldiff\,(x_{i,1}, x_{i,2}, ..., x_{i,n}) \ \ \forall i = 1, 2, ..., n$$
$$alldiff\,(x_{1,j}, x_{2,j}, ..., x_{n,j}) \ \ \forall j = 1, 2, ..., n$$

La modellizzazione CP è caratterizzata dalla definizione di una matrice di variabili *square* di dimensione $n \times n$ che inizializzo di volta in volta con le matrici generate random e da una serie di vincoli che rappresentano l'$alldiff$ per righe e per colonne.

La risoluzione viene realizzata dichiarando un oggetto $CPsolver$ di tipo $IloSolver$ e chiamando $CPsolver.solve(goal)$. Il $goal$ è un oggetto di tipo $IloGoal$ che definisce la strategia di ricerca che si vuole che ILOG Solver utilizzi nel processo di ricerca di una soluzione. Una volta risolto il problema, cioè dopo aver trovato un assegnamento di valori alle variabili che soddisfano i vincoli, appendo nel file dei risultati $risultati\_CP.txt$ le informazioni relative al tempo impiegato per la ricerca, al numero di fails, alla percentuale di successi per quel problema. Dato un certo n e un certo k, per avere risultati più veritieri, risolvo un certo numero ($NumRuns$) di problemi che ricevono come input una matrice generata in modo random con quegli stessi n e k, e poi considero la media dei risultati ottenuti.

Dettagli implementativi dell' algoritmo ILP

La modellizzazione e la risoluzione sono state realizzate tramite l'impiego di ILOG CPLEX [5].

La modellizzazione del problema tramite variabili, vincoli e funzione obiettivo si basa sulla seguente formulazione ILP del problema come programma intero [40], dove n è la dimensione del latin square parziale da completare :

$$max \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} x_{i,j,k}$$
$$subject \, to$$
$$\sum_{i=1}^{n} x_{i,j,k} \leq 1 \ \ \forall j, k$$
$$\sum_{j=1}^{n} x_{i,j,k} \leq 1 \ \ \forall i, k$$
$$\sum_{k=1}^{n} x_{i,j,k} \leq 1 \ \ \forall i, j$$
$$x_{i,j,k} = k \ \ \forall i, j, k \, tale \, che \, PLS_{i,j} = k$$

$$x_{i,j} \in \{0,1\} \ \ \forall i,j,k$$
$$i,j,k = 1,2,...,n$$

La modellizzazione Iè caratterizzata dalla definizione di un array tridimensionale di variabili $matBin$ di dimensione $n \times n \times n$ che inizializzo di volta in volta con le matrici generate random mat_parz così: se $mat\_parz[i][j] = k$ allora impongo il vincolo $matBin[i][j][k-1]$ deve essere $= 1$ (ricordo che k=1,...,n). Nel modello IP sono inoltre definiti opportuni vincoli che traducono nel linguaggio specifico di ILOG le disequazioni descritte nella formulazione IP definita sopra. In tale modello è anche prevista una funzione obiettivo che si deve massimizzare.

La risoluzione, che utilizza l'algoritmo del Simplesso Primale per risolvere i continui rilassamenti dei vincoli di interezza, viene realizzata dichiarando un oggetto $IPsolver$ di tipo $IloCplex$ e chiamando $IPsolver.solve()$. Come algoritmo di risoluzione viene adottato l'algoritmo del simplesso primale. Una volta risolto il problema appendo nel file dei risultati $risultati\_LP.txt$ le informazioni relative al tempo impiegato per la ricerca e alla percentuale di successi per quel problema. Come nel caso CP, dato un certo n e un certo k, risolvo un certo numero ($NumRuns$) di problemi che ricevono come input una matrice generata in modo random con quegli stessi n e k e poi considero la media dei risultati ottenuti.

Dettagli implementativi dell'algoritmo CP/ILP

Le novità rispetto agli algoritmi precedentemente illustrati risiedono nella modellizzazione del problema e nella sua risoluzione effettuata ora grazie all'impiego di ILOG Hybrid [41], che permette di far cooperare ILOG Solver e ILOG CPLEX [5].

La modellizzazione è caratterizzata dalla definizione di un unico oggetto *env* di tipo $IloEnv$, un unico modello $IloModel$ *model*, una sola istanza di $IloSolver$, che ho chiamato *solver* ed un'istanza del vincolo $IloLinConstraint$ che ho definito in questo modo $lc(solver)$. Quest'ultimo vincolo, che è uno degli elementi portanti di ILOG Hybrid, permette la comunicazione tra ILOG Solver e ILOG CPLEX [5], estraendo dal modello creato solo i vincoli lineari e la funzione obiettivo lineare, risolvendo il rilassamento lineare e informando l'istanza di IloSolver delle bound modifications realizzate; è come un vincolo globale che racchiude al suo interno ILOG CPLEX. All'unico modello definito

vengono poi aggiunte le variabili CP, una matrice 2-dimensionale di variabili intere chiamata *square*, le variabili ILP, una matrice 3-dimensionale di variabili binarie chiamata $matBin$, i vincoli CP di $Alldiff$ su *square*, i vincoli su $matBin$ e la funzione obiettivo ILP. Tali vincoli sono descritti nelle formulazioni ILP e CP del *latin square's problem* riportate precedentemente in questa sezione.

Per realizzare il mapping tra le variabili CP e le variabili ILP si sono utilizzati dei vincoli di tipo $IloIfThen$ che si comportano nel modo seguente:

- Se durante la ricerca si trova $square[i][j] = k$ e $matBin[i][j][k] \neq 1$ allora viene aggiunto il vincolo $matBin[i][j][k] = 1$;
- Se si trova $square[i][j] \neq k$ e $matBin[i][j][k] = 1$ allora viene aggiunto il vincolo $matBin[i][j][k] \neq 1$;
- Se si trova $matBin[i][j][k] = 1$ e $square[i][j] \neq k$ allora viene aggiunto il vincolo $square[i][j] = k$;
- Se si trova $matBin[i][j][k] \neq 1$ e $square[i][j] = k$ allora viene aggiunto il vincolo $square[i][j] \neq k$;

Dettagli implementativi della generazione delle istanze

Ogni algoritmo realizzato in questa tesi prevede la generazione di istanze in modo random. Una volta che si stabilisce la dimensione n dell'istanza del PLS da esaminare e il numero di elementi k da inserire, viene creato un PLS della dimensione voluta n e con k elementi, scelti in modo random tra i valori interi tra 1,...,n, inseriti in posizioni scelte random.

In particolare, è stata realizzata una funzione che riceve come input tre interi n, k, i e genera dei files chiamati $filen\_k\_i.dat$ (cioè per esempio $file3\_2\_1.dat$ se n=3, k=2, i=1) ciascuno dei quali contiene una matrice $n \times n$ con k valori inseriti. Tali valori, scelti in modo random tra gli interi da 1 a n, sono collocati in posizioni della matrice anch'esse determinate in modo random. La scelta del posto in modo random nella matrice viene effettuata facendo attenzione che tale postazione non sia stata occupata precedentemente da un altro valore, altrimenti il numero di celle riempite risulterebbe minore di k. La matrice così generata viene poi scritta nel file corrispondente nel formato richiesto da ILOG (per esempio, [[3, 2], [1, 5]]). La funzione $FunzGeneratore$ prende n e

k dalle righe del file $file\_NK.dat$ in cui sono stati scritti in ogni riga gli n e k che si vogliono utilizzare per generare le matrici random, e i dall'indice del ciclo for del $main()$ in cui viene chiamata la funzione in esame. Una volta generati i files $filen\_k\_i.dat$ i loro nomi sono scritti sulle righe di un opportuno file $nomi.dat$. Per scegliere il file contenente la matrice che voglio usare per inizializzare la matrice delle variabili, seleziono via via le righe di $nomi.dat$.

5.2 Latin Square Parziali random

Per poter disporre facilmente dei latin square parziali da cui partire, è stata implementata un'apposita procedura, che genera in modo random matrici quadrate 2-dimensionali della dimensione voluta n e con il numero desiderato k di elementi inseriti. La generazione delle matrici in questa prima fase viene realizzata in maniera puramente random: si sceglie random la posizione della matrice in cui inserire i valori e tali valori sono scelti in modo random tra gli interi compresi tra 1 e n. Questo tipo di generazione delle matrici di input può determinare problemi di latin square a priori irrealizzabili. Per questo motivo si è deciso di realizzare anche una procedura di generazione delle matrici random controllata: viene ancora selezionata in modo random la posizione della matrice in cui inserire i valori e tali valori sono ancora scelti random tra gli interi tra 1 e n, ma in modo che soddisfino i vincoli dei problemi di latin square parziali. I risultati ottenuti con questa seconda procedura sono riportati nella prossima sezione, in questa sezione vengono presentati i risultati ottenuti adottando la prima tipologia di generazione delle matrici di input, cioè quella puramente random.

Per avere risultati più veritieri si sono risolti 10 problemi con gli stessi n e k e si sono considerati i valori medi delle varie grandezze esaminate.

Prima di analizzare i risultati sperimentali è importante segnalare che i problemi che si stanno esaminando possono richiedere tempi di soluzione molto diversi, anche se la dimensione delle matrici generate random prese da input e il numero di valori in esse inseriti sono gli stessi, cioè anche se n e k sono gli stessi, e l'algoritmo adottato è il medesimo. In [39] si descrive questa peculiarità, specificando che le istanze di latin square parziali *bilanciate*, cioè le istanze in cui i valori inseriti random sono distribuiti in maniera uniforme, sono in molti

casi non risolvibili in tempi ragionevoli da nessuna strategia realizzata fin a questo momento. È chiaro pertanto che anche i latin square parziali generati random utilizzati in questa tesi possono contenere questa particolarità. Un esempio di tempi sperimentali molto diversi, relativi a istanze dello stesso tipo, viene illustrato nelle Figure 5.1 e 5.2. In tali figure vengono presentate due istanze di ordine 15 con 3 elementi inseriti, che richiedono tempi che differiscono addirittura di 6 ordini di grandezza, anche se sono valutati eseguendo lo stesso algoritmo CP.

I tempi computazionali valutati nella sperimentazione sono i tempi impiegati per effettuare la ricerca. Per valutarli si è utilizzato un timer, che si azionava prima di iniziare la ricerca e si bloccava non appena veniva trovata una soluzione o veniva determinato un fallimento. Tali tempi sono stati valutati con un AMD Athlon(tm) 1250 MHz.

5.2.1 Risultati dell'algoritmo CP

In questa sezione vengono presentati i risultati medi, relativi a tempi, percentuali di successi e fails, raggiunti dall'algoritmo CP. Tali risultati si riferiscono a sperimentazioni realizzate adottando le euristiche illustrate nella sezione 4.1.1. In particolare, è stata scelta come strategia di ricerca quella di tipo Depth First, come variabile su cui attuare il branching, quella con dominio più piccolo e come prossimo valore per istanziare tale variabile il più piccolo valore nel suo dominio. È importante segnalare che le euristiche adottate sono state decise, dopo aver realizzato varie prove sperimentali, e aver valutato come scelte migliori quelle che fornivano tempi computazionali minori.

Il grafico in figura 5.3 illustra la percentuale di problemi PLS con soluzione con $n = 10$ e k che varia in [0,100] e il tempo, normalizzato nell'intervallo [0,1], impiegato dall'algoritmo CP per risolvere tali probemi. Osservando tale grafico si nota che il picco della search si registra circa quando k=10, che per k <10 i problemi hanno quasi tutti soluzione e sono facili da risolvere e che per k >10 sono pochi i problemi con soluzione ma è semplice scoprire che non esiste soluzione.

In Tabella 5.1 sono riportate le medie di tempi e fails registrati al variare di n e k per l'algoritmo CP. I k esaminati sono $k = 0$, $n^2/4$, $n^2/2$, $3n^2/4$; il numero di esecuzioni che si è scelto di realizzare con gli stessi n e k è 10.

file30_7_0.dat

[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, **8**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **28**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[**7**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **10**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **24**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **12**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **26**, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0,0]]

Tempo = 5e-06 sec (Esiste soluzione)

Figura 5.1: Esempio di latin square parziale generato random con n = 30 e k= 7. Il tempo è quello valutato dall'algoritmo CP.

file30_7_1.dat

[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **3**, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **10**, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, **26**, 0, 0, 0, 0, 0, 0, 0, 0, **2**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **28**],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, **4**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **13**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

Tempo = 5.51 sec (Non esiste soluzione)

Figura 5.2: Altro esempio di latin square parziale con $n = 30$ e k= 7. Il tempo è quello valutato dall'algoritmo CP.

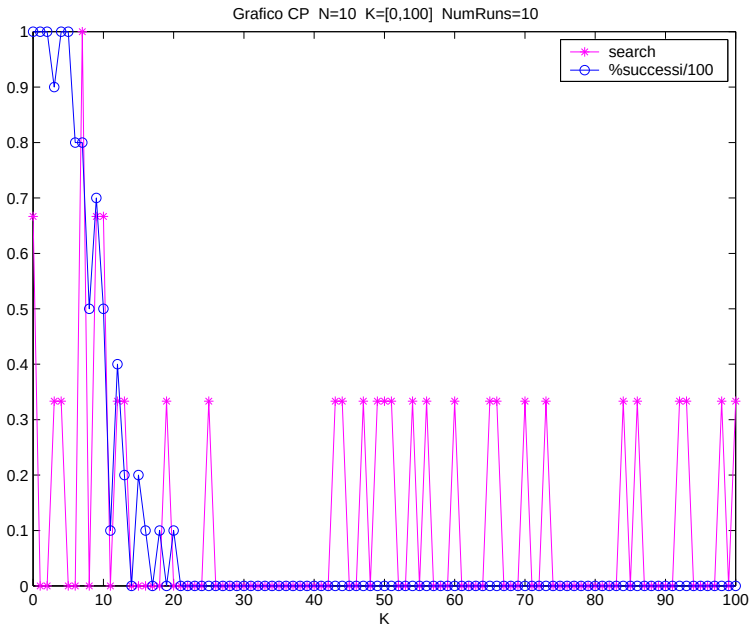

Figura 5.3: Grafico CP: search e percentuale di successi, search = tempo normalizzato tra 0 e 1.

Se si analizzano i dati presenti in tale tabella si nota che per tutte le istanze considerate CP è molto veloce.

5.2.2 Risultati dell'algoritmo ILP

I risultati medi raggiunti dall'algoritmo ILP, relativi a tempi e percentuali di successi, presentati in questa sezione, si basano sulle euristiche citate nella sezione 4.1.2. Le scelte effettuate e, in particolare quella del simplesso primale come algoritmo per risolvere i continui rilassamenti del problema, non sono casuali, ma discendono da un'attenta analisi dei tempi computazionali trovati effettuando parecchie prove sperimentali con vari algoritmi.

Il grafico in figura 5.4, che è il corrispondente dal lato ILP del grafico CP di figura 5.3, illustra la percentuale di problemi PLS con soluzione con $n = 10$ e k che varia in [0,100] e il tempo, normalizzato nell'intervallo [0,1], impiegato dall'algoritmo ILP per risolvere tali probemi. Se si confronta questo grafico con quello CP in figura 5.3, si nota che l'algoritmo ILP risolve più velocemente di CP i problemi con k >10 dove quasi tutti i problemi sono irrisolvibili, mentre ci impiega un tempo maggiore per problemi con k <10 dove quasi tutti i problemi

n	k	$\%Succ$	$Secs\ CP$	$Fails$
5	0	100%	0	0
5	6	10%	0	0.9
5	12	0%	0	1
5	18	0%	0	1
10	0	100%	1e-07	1
10	25	0%	0	1
10	50	0%	0	1
10	75	0%	0	1
15	0	100%	5e-07	14
15	56	0%	0	1
15	112	0%	0	1
15	168	0%	1e-07	1
17	0	100%	9e-07	93
17	72	0%	0	1
17	144	0%	0	1
17	216	0%	2e-07	1
19	0	100%	1.2e-06	37
19	90	0%	1e-07	1
19	180	0%	0	1
19	270	0%	1e-07	1
20	0	100%	3.6e-06	645
20	100	0%	0	1
20	200	0%	0	1
20	300	0%	1e-07	1

Tabella 5.1: Tempi, fails medi e percentuale di successi dell'algoritmo CP. ($Secs\ CP$ - tempo in secondi impegato dall'algoritmo CP; $Fails$ - numero di fallimenti; $\%Succ$ - percentuale di problemi con soluzione.)

hanno soluzione. Il picco della search, come dal lato CP, viene registrato per k vicino a 10.

In Tabella 5.2 sono riportate le medie di tempi e fails registrati al variare di n, k e dei tagli di Gomory applicati. I k esaminati sono $k = 0$, $n^2/4$, $n^2/2$, $3n^2/4$, le scelte riguardanti i tagli di Gomory sono se applicarli quando risulta conveniente nel corso della ricerca o se non applicarli mai. Il numero di esecuzioni che si è scelto di realizzare con gli stessi n e k è 10. Se si confrontano i dati riportati in tabella 5.2 relativi all'algoritmo ILP con quelli CP presenti in tabella 5.1 si scopre che l'algoritmo ILP impiega tempi notevolmente maggiori di quelli dell'algoritmo CP nelle istanze considerate. In particolare, nelle istanze con k=0 l'algoritmo ILP impiega tempi di quattro ordini di grandezza più grandi di quelli dell'algoritmo CP.

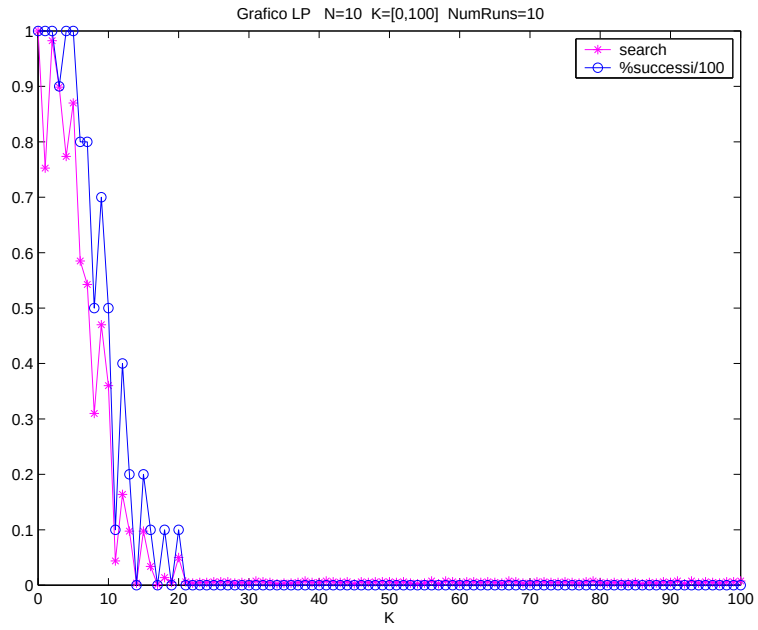

Figura 5.4: Grafico ILP: search e percentuale di successi, search = tempo normalizzato tra 0 e 1.

Per aver un'idea di come variano i tempi, se si aggiungono o no i tagli di Gomory nell'algoritmo ILP , si puo' osservare la Tabella 5.3, in cui sono riportati i tempi computazionali misurati eseguendo l'algoritmo ILP in tre casi: nel primo ($ILP(-1)$) si sceglieva di non applicare nessun taglio di Gomory, nel secondo ($ILP(0)$) si applicavano solo se era conveniente. Osservando i risultati ottenuti, appare evidente che in questo caso l'applicazione dei tagli di Gomory, anche se realizza il pruning dell'albero, peggiora i tempi computazionali,

n	k	$\%Succ$	$Secs\ ILP\ (0)$	$Secs\ ILP\ (-1)$
5	0	100%	2.3e-06	2e-06
5	6	10%	2e-07	2e-07
5	12	0%	1e-07	2e-07
5	18	0%	1e-07	1e-07
10	0	100%	7.8e-05	3.2e-05
10	25	0%	3e-07	4e-07
10	50	0%	4e-07	4e-07
10	75	0%	3e-07	4e-07
15	0	100%	0.00176	0.00358
15	56	0%	1.5e-06	1.6e-06
15	112	0%	1.6e-06	1.4e-06
15	168	0%	1.1e-06	1.4e-06
17	0	100%	0.00775	0.00823
17	72	0%	2.1e-06	2e-06
17	144	0%	2e-06	2e-06
17	216	0%	2e-06	2e-06
19	0	100%	0.0051	0.00376
19	90	0%	3e-06	3e-06
19	180	0%	2.8e-06	3e-06
19	270	0%	3e-06	2.9e-06
20	0	100%	0.06676	0.05901
20	100	0%	3.4e-06	3.5e-06
20	200	0%	3.2e-06	3.6e-06
20	300	0%	3.2e-06	3.3e-06

Tabella 5.2: Tempi medi e percentuale di successi dell'algoritmo ILP. ($Secs\ ILP\ (0)$ - tempo in secondi valutato dall'algoritmo ILP se si applicano i tagli di Gomory solo quando serve nella ricerca; $Secs\ ILP\ (-1)$ - tempo in secondi valutato dall'algoritmo ILP se non si applicano mai i tagli di Gomory; $\%Succ$ - percentuale di problemi con soluzione.)

n	k	$\%Succ$	$Secs\ ILP$ (-1)	$Secs\ ILP$ (0)	$Secs\ ILP$ (1)
10	2	100%	2.8e-05	6.4e-05	0.00033
10	5	90%	2.9e-05	5.7e-05	0.00029
10	7	70%	1.9e-05	4.1e-05	0.00017
15	3	100%	0.00098	0.00115	0.01011
15	7	90%	0.00069	0.00091	0.00846
15	11	60 %	0.00031	0.00064	0.00461

Tabella 5.3: Tempi medi e percentuale di successi dell'algoritmo ILP aggiungendo o no i tagli di Gomory al rilassamento lineare. ($\%Succ$ - percentuale di problemi con soluzione, $Secs\ ILP$ (-1) - tempo in secondi se non si applicano tagli di Gomory; $Secs\ ILP$ (0) - tempo in secondi se si applicano tagli di Gomory solo se risulta conveniente; $Secs\ ILP$ (1) - tempo in secondi se si applicano tagli di Gomory in maniera moderata.)

quindi qui non è conveniente usarli.

5.2.3 Risultati dell'algoritmo ibrido

Le prove sperimentali realizzate con l'algoritmo ibrido si differenziano per :

- la scelta della variabile su cui realizzare il branching, a partire dalla soluzione ottima rilassata : la variabile con valore frazionario più vicino a 0.5 ($MostNotInteger$), o la variabile con valore frazionario più grande ($MostNearOne$);
- la scelta della strategia di branching da adottare, cioè come definire il ramo dell'albero di ricerca da seguire per primo : istanzio la variabile scelta o con il valore intero più vicino al suo valore nella soluzione ottima rilassata ($BranchCloserFirst$), o con il valore intero più lontano ($BranchFartherFirst$), o con il valore intero immediatamente più grande ($BranchUpFirst$), o con il valore intero immediatamente più piccolo($BranchDownFirst$);
- la scelta relativa al numero di volte in cui realizzare lo scambio di informazioni tra il lato CP e il lato ILP , cioè quanto spesso risolvere il

rilassamento lineare del problema : ad ogni nodo ($cont$=0), o solo dopo un certo numero di nodi ($cont$=k, per k=1,2,..., dove k è il numero di nodi ogni quanto applicare il rilassamento);

- la scelta relativa ai tagli di Gomory : non applicarli mai nel corso della ricerca ($Gomory(-1)$), applicarli solo se è conveniente ($Gomory(0)$), aggiungerli sempre ma in maniera moderata ($Gomory(1)$).

I risultati medi raggiunti dall'algoritmo ibrido, relativi a tempi e percentuali di successi, presentati in questa sezione, adottano le euristiche citate nella sezione 4.1.2. Le scelte effettuate e in particolare quella del Simplesso Primale come algoritmo per risolvere il rilassamento del problema alla radice e quella del Simplesso Duale per risolvere il rilassamento nei nodi dell'albero di ricerca, non sono casuali, ma discendono da un'attenta analisi dei tempi computazionali trovati effettuando varie prove sperimentali con combinazioni di diversi algoritmi: Simplesso Primale e Simplesso Primale, Simplesso Duale e Simplesso Duale, Barrier Algorithm e Simplesso Duale, Barrier e Simplesso Primale. La coppia di algoritmi scelti, il Simplesso Primale e il Simplesso Duale, è quella che nelle istanze considerate determina tempi minori.

In generale, le scelte effettuate nelle prove sperimentali, che si andranno ad analizzare, sono state dettate da valutazioni preliminari di tempi. Si sono scelte quelle strategie che, dopo una prima analisi sperimentale su diverse istanze, si sono rivelate le migliori dal punto di vista dei tempi computazionali.

I risultati trovati eseguendo l'algoritmo ibrido su matrici con gli stessi n e k di quelle indagate dal lato CP e dal lato ILP sono riportati nel grafico di figura 5.5 e in Tabella 5.4. Il grafico in figura 5.5 illustra la percentuale di problemi PLS con soluzione con $n = 10$ e k che varia in [0,100] e il tempo, normalizzato nell'intervallo [0,1], impiegato dall'algoritmo ibrido per risolvere tali probemi. Se si confronta questo grafico con quello ILP in figura 5.4, si nota che, come nel caso ILP, l'algoritmo ibrido impiega tempi piccoli nei problemi con $k >10$ dove quasi tutti i problemi sono irrisolvibili e tempi maggiori per problemi con $k <10$ dove quasi tutti i problemi hanno soluzione. I tempi impiegati dall'algoritmo ibrido soprattuto per risolvere le istanze con k=0 sono peggiori di quelli CP, ma migliori di quelli ILP. Il picco della search, come osservato nei grafici CP e ILP rispettivamente in figura 5.3 e 5.4 viene registrato per k vicino a 10.

I dati presenti in Tabella 5.4 si riferiscono a sperimentazioni in cui si è deciso di effettuare il rilassamento lineare ad ogni nodo dell'albero di ricerca e di applicare in maniera moderata i tagli di Gomory. Esaminando i risultati ibridi in Tabella 5.4 è possibile osservare che i tempi registrati dall'algoritmo ibrido sono decisamente migliori di quelli ILP riportati in Tabella 5.2. Tuttavia, per le istanze con nessun elemento inserito, l'algoritmo CP risolve il problema più velocemente dell'algoritmo ibrido.

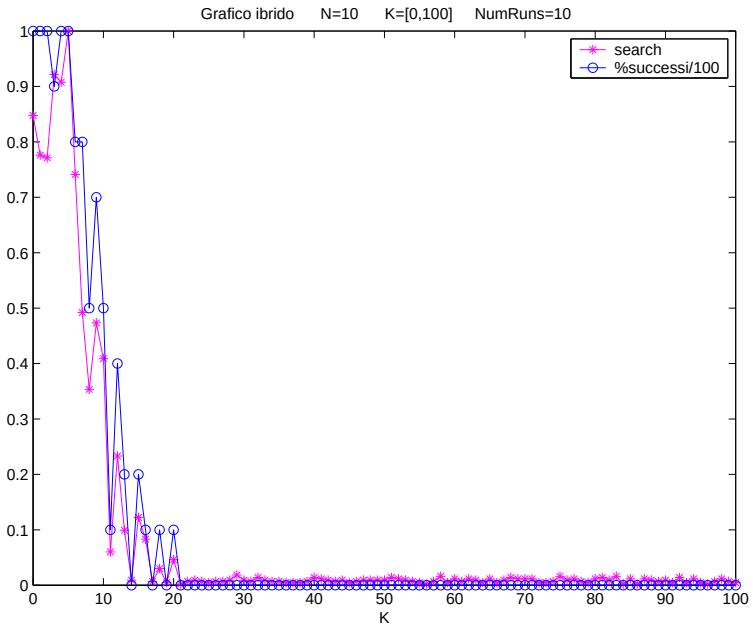

Figura 5.5: Grafico Ibrido: search e percentuale di successi, search = tempo normalizzato tra 0 e 1. *Scelte adottate* : MostNotInteger, BranchCloserFirst, Gomory(1), cont=0

Nel corso della trattazione vengono riportati alcuni risultati trovati eseguendo l'algoritmo ibrido a partire da diverse istanze e adottando varie tipologie di ricerca.

In Tabella 5.5, per esempio, vengono presentati i tempi impiegati dall'algoritmo ibrido per risolvere varie istanze del problema in esame adottando varie strategie di branching. In entrambi i casi si sceglie come variabile quella che nella soluzione ottima rilassata ha valore frazionario più vicino a 0.5 (euristica che chiameremo $MostNotInteger$), ma in ciascun caso si adotta una strategia diversa per decidere quale strada del branching seguire per prima. Nel primo caso illustrato in Tabella 5.5 si prova ad istanziare la variabile

n	k	$\%Succ$	$Secs\ H$	$Fails$
5	0	100%	3e-06	0
5	6	10%	2e-07	0.9
5	12	0%	0	1
5	18	0%	2e-07	1
10	0	100%	3.7e-05	0
10	25	0%	4e-07	1
10	50	0%	4e-07	1
10	75	0%	2e-07	1
15	0	100%	0.00085	0
15	56	0%	1.1e-06	1
15	112	0%	1.2e-06	1
15	168	0%	1.4e-06	1
17	0	100%	0.00122	0
17	72	0%	2e-06	1
17	144	0%	1.8e-06	1
17	216	0%	1.7e-06	1
19	0	100%	0.00393	0
19	90	0%	3.4e-06	1
19	180	0%	2.8e-06	1
19	270	0%	2.8e-06	1
20	0	100%	0.02592	0
20	100	0%	3.2e-06	1
20	200	0%	3.1e-06	1
20	300	0%	3.3e-06	1

Tabella 5.4: Tempi, fails medi e percentuale di successi dell'algoritmo Ibrido. ($Secs\ H$ - tempo in secondi impiegato dall'algoritmo ibrido; $Fails$ - numero di fallimenti; $\%Succ$ - percentuale di problemi con soluzione.) *Scelte adottate*: MostNotInteger, BranchCloserFirst, Gomory(1), cont=0.

scelta con il valore intero più vicino al valore frazionario della soluzione ottima ($BranchCloserFirst$), nel secondo caso con il valore intero più lontano ($BranchFarther\ First$), nel terzo caso con il valore intero immediatamente più grande ($BrachUpFirst$) e nel quarto caso con il valore intero immediatamente più piccolo ($BrachDownFirst$). In queste prove sperimentali, come anche nelle seguenti, si è scelto di usare la la strategia $MostNotInteger$ e non quella MostNearOne, perché nelle istanze esaminate dava tempi decisamente migliori. Da una prima analisi dei dati riportati in Tabella 5.5 emerge che la strategia $BranchCloserFirst$è la migliore, poiché nelle istanze di un certo spessore, cioè in quelle di ordine alto presenta tempi computazionali decisamente minori.

Nuove esecuzioni degli algoritmi CP, ILP e ibrido hanno determinato i tempi raffigurati in Tabella 5.6, dove vengono considerate istanze aventi n = 30, 35, 40 e $k = n^2/4,\ n^2/2,\ 3n^2/4$ e alcune istanze con k=0. Si nota che soprattutto per le istanze con $k = 0$, cioè istanze non vincolate a priori, l'algoritmo CP è più conveniente.

Sono state realizzate parecchie indagini e sono state apportate diverse modifiche all'algoritmo ibrido, per ottenere un miglioramento dei tempi computazionali. A questo proposito si è cercato di cambiare l'algoritmo ibrido in modo tale che il rilassamento ILP non venisse effettuato ad ogni nodo, ma solo ogni tanto. La Tabella 5.8 mostra i risultati ottenuti in ambiente ibrido, al variare del numero di nodi dopo cui decido di fare il rilassamento. Se si analizzano i valori presenti in tale Tabella, dove si sono scelti come valori di k $=n/4,\ n/2,\ 3n/4$, si nota che i tempi tendono ad aumentare al crescere di *cont*. La strategia ibrida con *cont*=0 sembra tuttavia essere la migliore. Per le istanze di ordine n=15 e 17 si registra un tempio medio che decresce all'aumentare del numero di elementi inseriti, mentre per l'istanza con n=20 si registra un tempo che cresce all'aumentare del numero di elementi inseriti. Per avere una visione generale dei risultati ottenuti, in Tabella 5.9 sono riportati i risultati migliori dell'algoritmo ibrido, quelli valutati con *cont*=0 e dei due algoritmi CP e ILP considerati singolarmente. Si nota che i risultati ibridi trovati per n=15, 17 effettuando il rilassamento lineare ad ogni nodo sono migliori in generale dei corrispondenti tempi trovati in ambiente ILP, anche se non vincono il confronto con i tempi CP. Un comportamento del tutto differente dell'algoritmo CP e dell'algoritmo ibrido è illustrato Tabella 5.7 dove vengono presentate istanze con n=20 e k=10, 100 in cui ibrido trova velocemente la soluzione, mentre

n	k	$\%Succ$	$Secs\ BC$	$Secs\ BF$	$Secs\ BU$	$Secs\ BD$
5	0	100%	3e-06	1e-07	1.7e-06	2.2e-06
5	6	10%	2e-07	2e-07	1e-07	2e-07
5	12	0%	0	0	1e-07	1e-07
5	18	0%	2e-07	1e-07	1e-07	1e-07
10	0	100%	3.7e-05	3.89e-05	3.84e-05	3.37e-05
10	25	0%	4e-07	3e-07	4e-07	5e-07
10	50	0%	4e-07	3e-07	4e-07	3e-07
10	75	0%	2e-07	3e-07	4e-07	3e-07
15	0	100%	0.00085	0.00068	0.00047	0.00077
15	56	0%	1.1e-06	1.3e-06	1.1e-06	1e-06
15	112	0%	1.2e-06	1.4e-06	1.4e-06	1.6e-06
15	168	0%	1.4e-06	1.6e-06	1.3e-06	1.5e-06
17	0	100%	0.00122	0.00345	0.00122	0.03171
17	72	0%	2e-06	2e-06	1.6e-06	1.6e-06
17	144	0%	1.8e-06	1.7e-06	1.9e-06	2e-06
17	216	0%	1.7e-06	2.2e-06	2.3e-06	1.8e-06
19	0	100%	0.00393	0.00545	0.01411	0.02721
19	90	0%	3e-06	3.1e-06	2.8e-06	2.9e-06
19	180	0%	2.8e-06	2.9e-06	3e-06	2.8e-06
19	270	0%	2.8e-06	2.6e-06	3.1e-06	3e-06
20	0	100%	0.02592	0.04634	0.02925	0.03759
20	100	0%	3.2e-06	3.1e-06	3.3e-06	3.2e-06
20	200	0%	3.1e-06	2.9e-06	3.5e-06	3.1e-06
20	300	0%	3.3e-06	3.7e-06	3.6e-06	3.6e-06

Tabella 5.5: Tempi medi e percentuale di successi dell'algoritmo ibrido. ($\%Succ$ - percentuale di problemi con soluzione, $Secs\ BC$ - tempo in secondi adottando la strategia BranchCloserFirst; $Secs\ BF$ - tempo in secondi adottando la strategia BranchFartherFirst; $Secs\ BU$ - tempo in secondi adottando la strategia BranchUpFirst; $Secs\ BD$ - tempo in secondi adottando la strategia BranchDownFirst.) *Euristica adottata*: scelgo come variabile quella che nella soluzione ottima rilassata ha valore frazionario più vicino a 0.5.

n	k	$\%Succ$	$Secs\ CP$	$Secs\ ILP$	$Secs\ H$
30	0	100%	5e-06	N.A.	N.A.
30	225	0%	2e-07	1.39e-05	1.25e-05
30	450	0%	1e-07	1.36e-05	1.29e-05
30	675	0%	2e-07	1.36e-05	1.12e-05
35	306	0%	1e-07	2.46e-05	2e-05
35	612	0%	2e-07	2.34e-05	2.13e-05
35	918	0%	1e-07	2.32e-05	2.15e-05
40	0	100%	0.01643	N.A.	N.A.
40	400	0%	0	3.88e-05	3.27e-05
40	800	0%	1e-07	3.78e-05	3.16e-05
40	1200	0%	1e-07	3.74e-05	3.34e-05

Tabella 5.6: Tempi medi dell'algoritmo CP e dell'algoritmo ibrido a confronto. (N.A. - l'algoritmo non ha dato risultati dopo 1 ora) *Scelte adottate* : per l'algoritmo ibrido si sono sono scelte le strategie MostNotInteger, BranchCloserFirst, cont=0, Gomory(1); per l'agoritmo ILP si lascia decidere a CPLEX se applicare i tagli nel corso della ricerca.

n	k	$\%Succ$	$Secs\ CP$	$Secs\ ILP$	$Secs\ H$
10	26	100%	N.A.	1.78e-05	4.78e-05
20	0	100%	2.9e-06	0.06676	0.02592
20	10	90%	N.A.	0.01322	0.01353
20	100	100%	N.A.	0.00252	0.00554

Tabella 5.7: Tempi medi dell'algoritmo CP , dell'algoritmo ILP e dell'algoritmo ibrido a confronto. (N.A. - l'algoritmo non ha dato risultati dopo 1 ora) *Scelte adottate* : per l'algoritmo ibrido si sono sono scelte le strategie MostNotInteger, BranchCloserFirst, cont=0, Gomory(1); per l'agoritmo ILP si sceglie di applicare i tagli di Gomory solo se risulta conveniente nel corso della ricerca.

n	k	$\%Succ$	$cont = 0$	$cont = 1$	$cont = 5$
15	3	100%	0.00068	0.00129	0.00206
15	7	60%	0.00038	0.00054	0.00092
15	11	40%	0.00029	0.00024	0.00086
17	4	100%	0.00139	0.00035	0.00602
17	8	60%	0.00093	0.00222	0.00336
17	12	70%	0.00129	0.00247	0.00393
20	5	90%	0.01391	0.03037	0.06408
20	10	80%	0.01366	0.02795	0.05694
20	15	70%	0.0094	0.02063	0.03689

Tabella 5.8: Tempi medi e percentuale di successi dell'algoritmo ibrido adottando una doppia strategia di ricerca, facendo variare il numero di volte in cui di realizza il rilassamento lineare del problema. (*cont* - numero di nodi dopo cui viene risolto il rilassamento linare nell'algoritmo ibrido. *Scelte adottate*: MostNotInteger, BranchCloserFirst e Gomory(1))

l'algoritmo CP non ritorna alcuna soluzione nell'arco di tempo stabilito.

5.3 Latin Square Parziali random controllati

In questa sezione si esaminano alcune istanze precedentemente illustrate, cioè alcune istanze aventi gli stessi n e k indagati prima, utilizzando gli stessi algoritmi CP, ILP, ibrido con le stesse euristiche di prima, ma adottando un un nuovo metodo di generazione delle matrici di input. I valori da inserire in posizioni casuali della matrice sono ancora scelti in modo random, ma tra i valori scelti random si scartano quelli che violano a priori i vincoli del problema. Per esempio, se all'inizio viene collocato in modo casuale sulla riga i o sulla colonna j il valore 4, allora i prossimi elementi scelti random che andranno a riempire la riga i o la colonna j della matrice di input dovranno essere diversi da 4; si continuano a generare valori random finchè se ne trova uno diverso da 4 da inserire nella riga i o nella colonna j. Le matrici generate in questo modo hanno il vantaggio di non determinare subito un fallimento. I risultati presentati in questa sezione differiscono un po' rispetto a quelli della sezione precedente, soprattutto per quanto riguarda le percentuali di successi, cioè la percentuale

n	k	$\%Succ$	$Secs\ CP$	$SecsILP$	$SecsH$
15	3	100%	4e-07	0.00102	0.00068
15	7	60%	1.3e-06	0.00065	0.00038
15	11	40%	2e-07	0.00037	0.00029
17	4	100%	7e-06	0.00225	0.00139
17	8	60%	3e-06	0.00179	0.00093
17	12	70%	2.6e-05	0.00139	0.00129
20	5	90%	1e-06	0.01336	0.01391
20	10	80%	N.A.	0.00975	0.01366
20	15	70%	7e-06	0.00901	0.0094

Tabella 5.9: Tempi medi degli algoritmi CP, ILP, ibrido a confronto. *Euristiche adottate*: dal lato solo CP scelgo la variabile con dominio più piccolo e provo ad istanziarla con il primo valore nel suo dominio; dal lato ILP lascio decidere a CPLEX che tagli applicare, dal lato ibrido utilizzo MostNotInteger, BranchCloserFirst, cont=0, Gomory(1))

di problemi con soluzione. In questa seconda serie di risultati compare una percentuale di successi decisamente maggiore in generale rispetto alla sezione precedente e di conseguenza anche i tempi misurati sono spesso superiori rispetto a prima. È chiaro che ricevere come PLS (partial latin square) un problema a priori irrealizzabile facilita notevolmente il lavoro degli algoritmi, che sono in grado di risolvere quasi istantaneamente il problema segnalando un fallimento. Diverso invece è il caso in cui la maggior parte dei problemi da considerare, non rivelandosi a priori irrealizzabile, necessita una ricerca vera e propria nell'albero delle soluzioni.

5.3.1 Risultati dell'algoritmo CP

In questa sezione vengono presentati i risultati medi, relativi a tempi, percentuali di successi e fails, raggiunti dall'algoritmo CP, adottando il nuovo metodo di generazione delle matrici di input. Le percentuali di successi registrate e di conseguenza i tempi riscontrati sono in generale superiori a quelli presentati nella sezione precedente. Il grafico in Figura 5.6 illustra la percentuale di problemi PLS con soluzione con $n = 10$ e k che varia in [0,65] e il tempo impiegato dall'algoritmo CP per risolvere tali probemi. Osservando tale grafico si nota

che l'algoritmo CP impiega un tempo molto basso quasi ovunque, tranne che nella regione compresa tra k=20 e k=30, dove cioè le istanze presentano una percentuale di riempimento pari al 20% e al 30%. In particolare si registra un tempo pittosto elevato rispetto alla media per k=22 e si nota che per k=26 l'algoritmo CP non è in grado di risolvere il problema nell'arco di tempo stabilito. Il grafico di Figura 5.6 mostra inoltre che il passaggio tra problemi con soluzioni e problemi senza soluzione avviene quando la matrice di input ha circa il 45% degli elementi inseriti. L'irrisolvibilità di questa istanza non si presentava nella prima serie di risultati realizzati, perché in corrispondenza di tale valore di k la percentuale di successi era quasi nulla e quindi era facile determinare il fallimento.

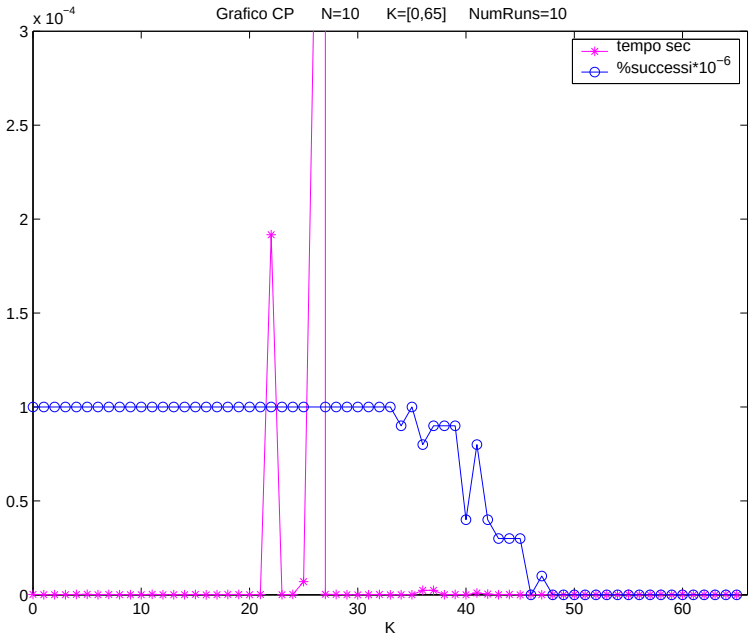

Figura 5.6: Grafico CP: tempo e percentuale di successi.

In Tabella 5.10 sono riportate le medie di tempi e fails registrati al variare di n e k per l'algoritmo CP, a partire da alcune delle istanze esaminate nella Tabella 5.1 della sezione precedente. Il numero di esecuzioni che si è scelto di realizzare con gli stessi n e k è 10. Se si analizzano i dati presenti in Tabella 5.10 si nota che per tutte le istanze considerate CP è molto veloce, eccetto per n=17 e k=72 dove si registra un tempo superiore agli altri tempi presenti in tabella e in particolare per n=17 e k=144, che l'algoritmo CP non è in grado di risolvere nel tempo stabilito. Queste ultime istanze registrano un

comportamento decisamente differente da quello illustrato in Tabella 5.1, dove queste particolari istanze presentavano una percentuale di successi pari a 0%.

5.3.2 Risultati dell'algoritmo ILP

I risultati medi raggiunti dall'algoritmo ILP, relativi a tempi e percentuali di successi, presentati in questa sezione, si basano sulle euristiche citate nella sezione 4.1.2. La novità anche qui riguarda il metodo di generazione dei PLS che ora non possono essere a priori irrealizzabili. In questa sezione quando si parla di successi si fa riferimento a quei problemi che hanno come valore della funzione obiettivo il suo massimo valore possibile. Tale decisione è stata presa per avere una corrispondenza diretta con i problemi CP che, per essere realizzabili, devono avere tutte le celle riempite.

Il grafico in Figura 5.7, che è il corrispondente dal lato ILP del grafico CP di Figura 5.6, illustra la percentuale di problemi PLS con soluzione con $n = 10$ e k che varia in [0,65] e il tempo impiegato dall'algoritmo ILP per risolvere tali probemi. La differenza rispetto al grafico ILP della sezione precedente in Figura 5.4 riguarda come nel caso CP la percentuale di problemi con soluzione che ora è decisamente maggiore. Se si confronta il grafico di Figura 5.7 con quello CP in Figura 5.6, si nota che l'algoritmo ILP registra tempi maggiori di quelli dell'algoritmo CP soprattutto per k piccoli, compresi tra 0 e 20. Tuttavia si nota un tempo migliore per n=22 e soprattutto per n=26 che l'algoritmo ILP, a differenza di quello CP è in grado di risolvere e lo fa anche molto velocemente. L'algoritmo ILP risolve quasi istantaneamente i problemi con percentuale di successi vicino a zero, dove cioè l'algoritmo ILP trova un valore della funzione obiettivo che non è il massimo possibile. Nel grafico ILP di Figura 5.7 si registra un picco dei tempi attorno a k=34, quindi l'algoritmo ILP impiega un tempo maggiore per risolvere problemi di latin square parziali con un 40% di preassegnamenti.

In Tabella 5.11 sono riportate le medie di tempi e fails registrati al variare di n, k e dei tagli di Gomory applicati. I k esaminati sono gli stessi di quelli della sezione precedente presentati nella tabella 5.2; le differenze riguardano i tempi valutati in alcune delle istanze che prima avevano una percentuale nulla di successi. Il numero di esecuzioni che si è scelto di realizzare con gli stessi n e k è 10. Se si confrontano i dati riportati in tabella 5.11 relativi all'algoritmo ILP con quelli CP presenti in tabella 5.10 si scopre che l'algoritmo

n	k	$\%Succ$	$Secs\ CP$	$Fails$
5	0	100%	0	0
5	6	100%	0	0
5	12	10%	1e-07	1
5	18	0%	0	1
10	0	100%	1e-07	1
10	25	100%	2e-07	0
10	50	0%	0	1
10	75	0%	1e-07	1
15	0	100%	5e-07	14
15	56	100%	3e-07	8
15	112	20%	2.5e-05	4979
15	168	0%	1e-07	1
17	0	100%	9e-07	93
17	72	100%	0.00024	59064
17	144	N.A.	N.A.	N.A.
17	216	0%	2e-07	1
19	0	100%	1.2e-06	37
19	90	100%	2e-06	306
20	0	100%	3.6e-06	645
20	100	N.A.	N.A.	N.A.
20	200	N.A.	N.A.	N.A.
20	300	0%	2e-07	1

Tabella 5.10: Tempi, fails medi e percentuale di successi dell'algoritmo CP. ($Secs\ CP$ - tempo in secondi impegato dall'algoritmo CP; $Fails$ - numero di fallimenti; $\%Succ$ - percentuale di problemi con soluzione; N.A. - l 'algoritmo non ha dato risultati dopo 1 ora)

ILP impiega tempi notevolmente maggiori di quelli dell'algoritmo CP nelle istanze con k piccoli, in particolare nelle istanze con k=0, l'algoritmo impiega tempi di quattro ordini di grandezza più grandi di quelli dell'algoritmo CP. Mentre si nota che i tempi ILP sono migliori di quelli CP, quando il numero di elementi inseriti nella matrice di input è abbastanza grande.

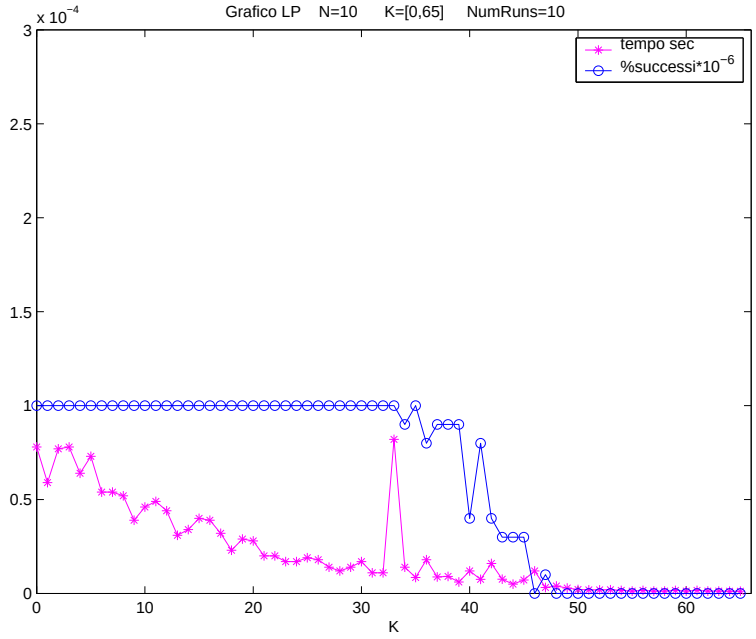

Figura 5.7: Grafico ILP: tempo e percentuale di successi.

Per aver un'idea di come variano i tempi, se si aggiungono o no i tagli di Gomory nell'algoritmo ILP, si puo' osservare la Tabella 5.12, in cui sono riportati i tempi computazionali misurati eseguendo l'algoritmo ILP in tre casi: nel primo ($ILP(-1)$) si sceglieva di non applicare nessun taglio di Gomory, nel secondo ($ILP(0)$) si applicavano solo se era conveniente e nel terzo ($ILP(1)$) si decideva di applicarne sempre alcuni in maniera moderata. Osservando i risultati ottenuti, appare evidente che, come già osservato nella Tabella 5.3 della sezione precedente, l'applicazione dei tagli di Gomory, anche se realizza il pruning dell'albero, peggiora i tempi computazionali, quindi qui non è conveniente usarli.

5.3.3 Risultati dell'algoritmo ibrido

I risultati medi raggiunti dall'algoritmo ibrido, relativi a tempi e percentuali di successi, presentati in questa sezione, adottano le euristiche citate nella sezione

n	k	$\%Succ$	$Secs\ ILP$
5	0	100%	2.3e-06
5	6	100%	1.2e-06
5	12	0%	8e-07
5	18	0%	7e-07
10	0	100%	7.8e-05
10	25	100%	1.74e-05
10	50	0%	8.3e-06
10	75	0%	1.2e-06
15	0	100%	0.00176
15	56	100%	0.00039
15	112	20%	0.00025
15	168	0%	3e-06
17	0	100%	0.00775
17	72	100%	0.00090
17	144	60%	0.00023
17	216	0%	4e-06
19	0	100%	0.0051
19	90	100%	0.00181
19	180	0%	0.00062
19	270	0%	6.1e-06
20	0	100%	0.06676
20	100	100%	0.00267
20	200	80%	0.00067
20	300	0%	7.9e-06

Tabella 5.11: Tempi medi e percentuale di successi dell'algoritmo ILP. ($Secs\ ILP$ - tempo in secondi valutato dall'algoritmo ILP se si applicano i tagli di Gomory solo quando serve nella ricerca.)

n	k	$\%Succ$	$Secs\ ILP$ (-1)	$Secs\ ILP$ (0)	$Secs\ ILP$ (1)
10	2	100%	2.6e-05	6.94e-05	0.0002
10	5	100%	2.5e-05	6.37e-05	0.00035
10	7	100%	3.1e-05	5.25e-05	0.00046
15	3	100%	0.00041	0.00114	0.00737
15	7	100%	0.00044	0.00113	0.00953
15	11	100 %	0.00092	0.0011	0.00966

Tabella 5.12: Tempi medi e percentuale di successi dell'algoritmo ILP aggiungendo o no i tagli di Gomory al rilassamento lineare. ($Secs\ ILP$ (-1) - tempo in secondi se non si applicano tagli di Gomory; $Secs\ ILP$ (0) - tempo in secondi se si applicano tagli di Gomory solo se risulta conveniente; $Secs\ ILP$ (1) - tempo in secondi se si applicano tagli di Gomory in maniera moderata.)

4.1.2 e utilizzanndo il nuovo metodo di generazione delle matrici di input.

Il grafico in Figura 5.8 illustra la percentuale di problemi PLS con soluzione con $n = 10$ e k che varia in [0,65] e il tempo impiegato dall'algoritmo ibrido per risolvere tali probemi. Osservando il grafico si riscontra un tempo superiore a quello dell'algoritmo CP quasi ovunque, eccetto per k=22, dove l'algoritmo ibrido registra un tempo decisamente minore e soprattutto per k=26 che l'algoritmo ibrido è in grado di risolvere molto rapidamente. Si nota un picco per k=40 dove si assiste al passaggio dai problemi con soluzione ai problemi senza soluzione. Se si confronta il grafico di Figura 5.7 con quello ibrido in Figura 5.8 si scopre che l'algoritmo ibrido è decisamente migliore dell'algoritmo ILP a livello di tempi computazionali per $k < 20$ e in particolare per $k = 33$. Per $k > 45$, quando la percentuale di problemi con soluzione è zero, i tempi dell'algoritmo ibrido sono pressochè nulli. Se si confronta questo grafico con quello ILP in figura 5.4, si nota che, come nel caso ILP, l'algoritmo ibrido impiega tempi piccoli nei problemi con $k > 10$ dove quasi tutti i problemi sono irrisolvibili e tempi maggiori per problemi con $k < 10$ dove quasi tutti i problemi hanno soluzione. I tempi impiegati dall'algoritmo ibrido soprattuto per risolvere le istanze con $k = 0$ sono peggiori di quelli CP, ma migliori di quelli ILP. Il grafico in Figura 5.9 illustra in modo compatto l'andamento dei tempi dei tre algoritmi CP, ILP e ibrido.

I dati presenti in Tabella 5.13 si riferiscono a sperimentazioni in cui si è deciso di effettuare il rilassamento lineare ad ogni nodo dell'albero di ricerca e

di applicare in maniera moderata i tagli di Gomory. Le istanze esaminate sono alcune di quelle presentate nella Tabella 5.4 riportata nella sezione precedente. La differenza tra i risultati in Tabella 5.13 e quelli della Tabella 5.4 riguarda in modo particolare i tempi registrati per quelle istanze, che prima presentavano una percentuale di successi pari a zero, e che ora hanno una percentuale di successi del 100%. Esaminando i risultati ibridi in Tabella 5.13 è possibile osservare che i tempi registrati dall'algoritmo ibrido sono decisamente migliori per k =0 e per k piccolo in generale di quelli ILP riportati in Tabella 5.11. Tuttavia, per le istanze con nessun elemento inserito, l'algoritmo CP risolve il problema più velocemente dell'algoritmo ibrido. Il grafico in Figura 5.9 illustra in modo compatto l'andamento dei tempi dei tre algoritmi CP, ILP e ibrido per PLS con $n = 10$ e k che varia nell'intervallo [0,65].

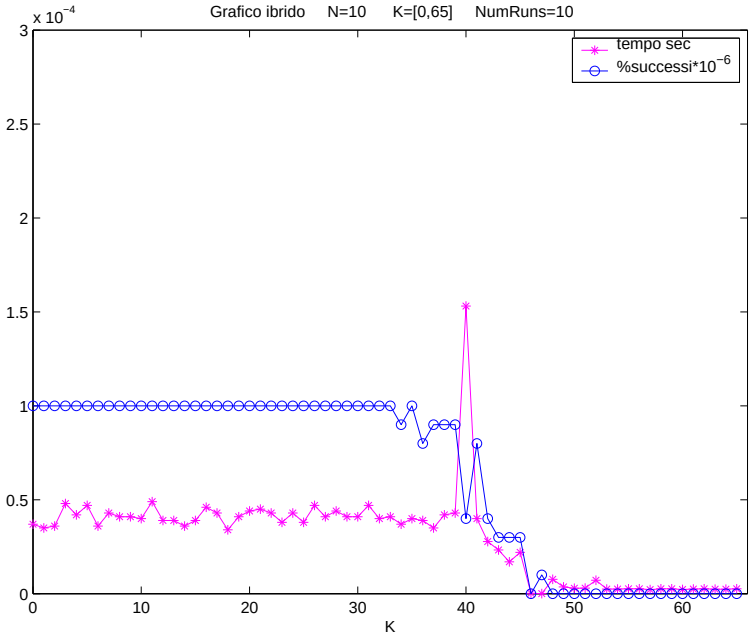

Figura 5.8: Grafico Ibrido: tempo e percentuale di successi. *Scelte adottate* : MostNotInteger, BranchCloserFirst, Gomory(1), cont=0

In Tabella 5.14 vengono presentati i tempi impiegati dall'algoritmo ibrido per risolvere varie istanze del problema in esame adottando varie strategie di branching. Le istanze riporate in Tabella 5.14 sono alcune di quelle presentate nella Tabella 5.5 a cui ne vengono aggiunte di nuove. In entrambi i casi si sceglie come variabile quella che nella soluzione ottima rilassata ha valore frazionario più vicino a 0.5 (euristica che chiameremo $MostNotInteger$),

n	k	$\%Succ$	$Secs\ H$	$Fails$
5	0	100%	3e-06	0
5	6	100%	1.2e-06	0
5	12	0%	2e-07	1
5	18	0%	3e-07	1
10	0	100%	3.7e-05	0
10	25	100%	3.7e-05	1
10	50	0%	2.6e-06	1
10	75	0%	3e-06	1
15	0	100%	0.00085	0
15	56	0%	0.00065	1
17	0	100%	0.00122	0
17	72	100%	0.00173	1
19	0	100%	0.00393	0
19	90	100%	0.00337	0
20	0	100%	0.02592	0
20	100	100%	0.00554	0
20	200	80%	0.00389	0
20	300	0%	0.00011	1

Tabella 5.13: Tempi, fails medi e percentuale di successi dell'algoritmo Ibrido. ($Secs\ H$ - tempo in secondi impiegato dall'algoritmo ibrido; $Fails$ - numero di fallimenti; $\%Succ$ - percentuale di problemi con soluzione.) *Scelte adottate*: MostNotInteger, BranchCloserFirst, Gomory(1), cont=0.

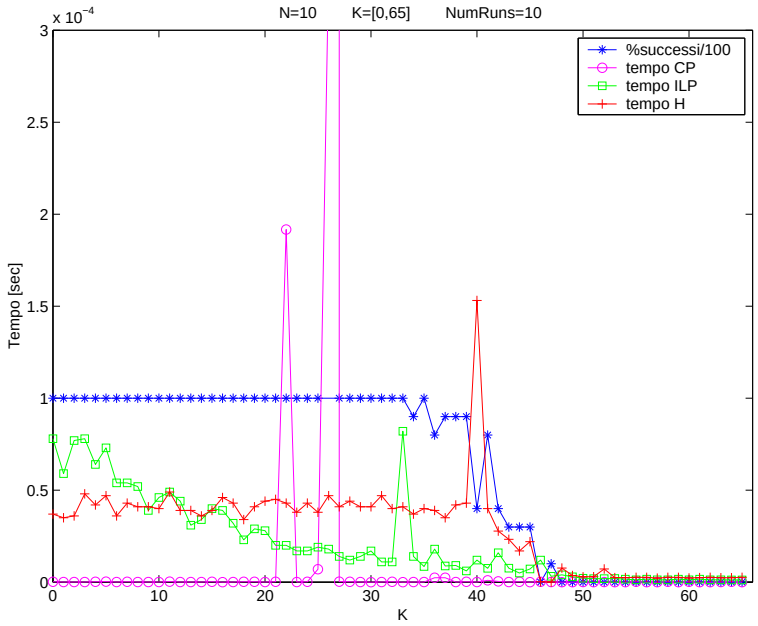

Figura 5.9: Grafici CP, ILP, ibrido: tempi a confronto. *Scelte adottate* : MostNotInteger, BranchCloserFirst, Gomory(1), cont=0

ma in ciascun caso si adotta una strategia diversa per decidere quale strada del branching seguire per prima. Nel primo caso illustrato in Tabella 5.14 si prova ad istanziare la variabile scelta con il valore intero più vicino al valore frazionario della soluzione ottima ($BranchCloserFirst$), nel secondo caso con il valore intero più lontano ($BranchFartherFirst$), nel terzo caso con il valore intero immediatamente più grande ($BrachUpFirst$) e nel quarto caso con il valore intero immediatamente più piccolo ($BrachDownFirst$). In queste prove sperimentali, come anche nelle seguenti, si è scelto di usare la la strategia $MostNotInteger$ e non quella MostNearOne, perché nelle istanze esaminate dava tempi decisamente migliori. Da una prima analisi dei dati riportati in Tabella 5.14 emerge, come riscontrato in Tabella 5.5, che la strategia $BranchCloserFirst$è la migliore, poiché nelle istanze di un certo spessore, cioè in quelle di ordine alto presenta tempi computazionali decisamente minori.

La Tabella 5.15 mostra i risultati ottenuti in ambiente ibrido, al variare del numero di nodi dopo cui decido di fare il rilassamento. Se si analizzano i valori presenti in tale Tabella, dove si sono scelti come valori di $k = n/4,\ n/2,\ 3n/4$, si nota che i tempi tendono ad aumentare al crescere di *cont*, come già si era osservato nella Tabella 5.8. La strategia ibrida con *cont*=0 sembra tuttavia

n	k	$\%Succ$	$Secs\ BC$	$Secs\ BF$	$Secs\ BU$	$Secs\ BD$
5	0	100%	3e-06	1e-07	1.7e-06	2.2e-06
10	0	100%	3.7e-05	3.89e-05	3.84e-05	3.37e-05
15	0	100%	0.00085	0.00068	0.00047	0.00077
15	16	100%	0.00057	0.00071	0.00059	0.00069
17	0	100%	0.00122	0.00345	0.00122	0.0317
17	32	100%	0.00172	0.00158	0.00149	0.0176
19	0	100%	0.00393	0.00545	0.01411	0.02721
19	40	100%	0.00354	0.00387	0.00391	0.0041
20	0	100%	0.02592	0.04634	0.02925	0.03759

Tabella 5.14: Tempi medi e percentuale di successi dell'algoritmo ibrido. ($\%Succ$ - percentuale di problemi con soluzione, $Secs\ BC$ - tempo in secondi adottando la strategia BranchCloserFirst; $Secs\ BF$ - tempo in secondi adottando la strategia BranchFartherFirst; $Secs\ BU$ - tempo in secondi adottando la strategia BranchUpFirst; $Secs\ BD$ - tempo in secondi adottando la strategia BranchDownFirst.) *Euristica adottata*: scelgo come variabile quella che nella soluzione ottima rilassata ha valore frazionario più vicino a 0.5.

n	k	$\%Succ$	$cont = 0$	$cont = 1$	$cont = 5$
15	3	100%	0.00066	0.00119	0.00262
15	7	100%	0.00074	0.00098	0.00305
15	11	100%	0.00060	0.00091	0.00243
17	4	100%	0.00219	0.00264	0.00677
17	8	100%	0.00183	0.00280	0.00861
17	12	100%	0.00177	0.00271	0.00510

Tabella 5.15: Tempi medi e percentuale di successi dell'algoritmo ibrido adottando una doppia strategia di ricerca, facendo variare il numero di volte in cui di realizza il rilassamento lineare del problema. (*cont* - numero di nodi dopo cui viene risolto il rilassamento linare nell'algoritmo ibrido. *Scelte adottate*: MostNotInteger, BranchCloserFirst e Gomory(1))

essere la migliore. La differenza tra la nuova Tabella 5.15 e quella 5.8 riguarda solo i tempi misurati, che ora risultano leggermente superiori a prima poiché la percentuale di problemi con soluzioni è notevolmente maggiore rispetto a prima. Per le istanze di ordine n=15 e 17 si registra un tempo medio che decresce all'aumentare del numero di elementi inseriti, mentre per l'istanza con n=20 si registra un tempo che cresce all'aumentare del numero di elementi inseriti. Per avere una visione generale dei risultati ottenuti, in Tabella 5.16 sono riportati i risultati migliori dell'algoritmo ibrido, quelli valutati con *cont*=0 e dei due algoritmi CP e ILP considerati singolarmente. Si nota che i risultati ibridi trovati per n=15, 17 effettuando il rilassamento lineare ad ogni nodo sono migliori in generale dei corrispondenti tempi trovati in ambiente ILP, anche se non vincono il confronto con i tempi CP. I risultati presentati in Tabella 5.7, che segnalano la vittoria dell'algoritmo ibrido su CP non cambiano, se si utilizza il nuovo metodo di generazione delle matrici, perchè in tali istanze si sono registrate generalmente percentuali di successi pari al 100%. Tali risultati vengono riproposti ora in Tabella 5.17, dove vengono aggiunti nuovi valori trovati nel corso della sperimentazione. Se si analizza tale tabella si scopre che l'algoritmo ibrido è un valido strumento di integrazione della discipline CP e OR, perchè trova la soluzione dove CP non è capace di trovarla e riesce a vincere il confronto con l'approccio solo OR quando il numero di elementi del latin square di partenza è piuttosto piccolo.

n	k	$\%Succ$	$Secs\ CP$	$Secs\ ILP$	$SecsH$
15	3	100%	5e-07	0.00113	0.00066
15	7	100%	6e-07	0.00106	0.00074
15	11	100%	1.2e-06	0.00102	0.0006
17	4	100%	8e-07	0.00294	0.00219
17	8	100%	7e-07	0.00235	0.00018
17	12	100%	8e-07	0.00218	0.00177

Tabella 5.16: Tempi medi degli algoritmi CP, ILP, ibrido a confronto. *Euristiche adottate*: dal lato solo CP scelgo la variabile con dominio più piccolo e provo ad istanziarla con il primo valore nel suo dominio; dal lato ILP lascio decidere a CPLEX che tagli applicare , dal lato ibrido utilizzo MostNotInteger, BranchCloserFirst, cont=0, Gomory(1))

n	k	$\%Succ$	$Secs\ CP$	$Secs\ ILP$	$Secs\ H$
10	22	100%	0.00019	2.7e-05	4.3e-05
10	26	100%	N.A.	1.78e-05	4.7e-05
11	0	100%	3e-07	0.000127	6.9e-05
11	5	100%	3e-07	0.000138	7.5e-05
11	36	100%	N.A.	2.7e-05	8.2e-05
11	40	100%	0.02178	4.1e-05	8.5e-05
12	0	100%	0	0.00028	0.0001
12	49	100%	N.A.	4.5e-05	0.00015
15	67	100%	N.A.	0.00034	0.00066
15	78	100%	N.A.	0.0003	0.00065
15	90	100%	N.A.	0.00019	0.00068
20	0	100%	2.9e-06	0.06676	0.02592
20	10	100%	N.A.	0.01422	0.01453
20	100	100%	N.A.	0.00252	0.00554

Tabella 5.17: Tempi medi dell'algoritmo CP , dell'algoritmo ILP e dell'algoritmo ibrido a confronto. (N.A. - l'algoritmo non ha dato risultati dopo 1 ora) *Scelte adottate* : per l'algoritmo ibrido si sono sono scelte le strategie MostNotInteger, BranchCloserFirst, cont=0, Gomory(1); per l'agoritmo ILP si sceglie di applicare i tagli di Gomory solo se risulta conveniente nel corso della ricerca.

Capitolo 6

Conclusioni

In questa tesi, dopo aver analizzato attentamente le metodologie specifiche della Programmazione con Vincoli e della Ricerca Operativa, si sono indagate le tecniche di integrazione di CP e OR fino ad ora realizzate. Si sono esaminati in maniera approfondita i due settori CP e OR, per capire quali fossero i vantaggi e gli svantaggi del'uno e dell'altro. Quindi si è cercato di valutare una metodologia che permettesse di integrare i benefici dei due settori.

Per determinare una buona tecnica di integrazione si è partiti esaminando la ricerca fino ad ora sviluppata in questo senso e le direzioni principali scelte per far cooperare CP e OR. Una delle modalità di integrazione già indagata dalle comunità di Ricerca Operativa e Programmazione con Vincoli, che ha fornito risultati interessanti, è quella che integra CP e OR inserendo in ambiente CP dei tagli specifici della classe di problemi in esame.

In questa tesi, partendo dalla consapevolezza che i tagli specifici dei problemi, inseriti in ambiente CP, migliorano la performance della sola Programmazione con Vincoli, si è cercato di veder se si poteva generalizzare tale metodologia, utilizzando tagli generali, per poter così ottenere un metodo che andasse bene per una qualunque classe di problemi, non solo per un problema specifico, come era stato fatto in passato. In questa tesi come tagli generali si sono utilizzati i tagli di Gomory.

Il metodo implementato in questa tesi prevede di modellare il problema in ambiente CP, sfruttando i vantaggi forniti dai vincoli globali e di realizzare una prima propagazione di vincoli per eliminare dai domini delle variabili quei valori che sicuramente non parteciperanno ad alcuna soluzione. Il passaggio dal lato CP al lato OR avviene trasformando il problema ottenuto dopo la prima

propagazione in problema MIP. Una volta rappresentato il problema in versione MIP si sfruttano i vantaggi di OR che consistono nella possibilità di risolvere il rilassamento del problema ad ogni nodo e di aggiungere i tagli di Gomory per rafforzare tale rilassamento. La soluzione ottima rilassata risultante viene poi usata per guidare la ricerca con backtrack che si decide di adottare dal lato CP per trovare la soluzione della classe di problemi in esame. Ad ogni nodo dell'albero di ricerca, che si percorre in ambiente CP utilizzando una ricerca in profondità con backtracking, si chiede aiuto a OR nel modo sopra descritto, per avere informazioni utili sulla strada più promettente da seguire nella ricerca.

Per valutare l'efficacia della metodologia appena descritta si sono realizzati altri due metodi che risolvessero la classi di problemi in esame utilizzando solo tecniche di CP e solo tecniche di OR. L'indagine sperimentale è stata effettuata sui problemi di *LatinSquare*. Su questa classe di problemi l'algoritmo ibrido realizzato, che utilizza in maniera moderata i tagli di Gomory e scambia continuamente informazioni tra CP e ILP è risultato un buon algoritmo. Esso infatti, anche se in alcuni casi registra tempi superiori a quelli dell'algoritmo CP, riesce a risolvere un numero di istanze maggiore dell'algoritmo CP, è infatti in grado di risolvere istanze che l'algoritmo CP non è in grado di risolvere nell'intervallo di tempo stabilito. Dall'analisi sperimentale è emerso che oltre all'algoritmo ibrido, anche l'algoritmo ILP riesce a risolvere queste istanze con tempi molto simili ai tempi ibridi, tuttavia, dal momento che nella maggior parte dei casi esaminati l'algoritmo ibrido registra tempi decisamente minori di quelli valutati dall'algoritmo ILP, sembra corretto dichiarare che l'algoritmo ibrido è in generale migliore di quello ILP a livello di tempi computazionali. In definitiva, inserire in maniera moderata i tagli di Gomory in ambiente CP è un valido strumento di integrazione delle discipline CP e OR, in quanto permette di determinare soluzioni di istanze, che l'approccio con sola CP non sarebbe in grado di valutare.

Ringraziamenti

Desidero ringraziare i miei genitori, mio fratello e il mio fidanzato, che mi hanno sempre aiutata e sostenuta nelle mie scelte. Ringrazio la prof. Francesca Rossi, che ha guidato i miei studi in questi ultimi mesi ed è sempre stata molto disponibile a ricevermi e a rispondere alle mie domande di chiarimento. Vorrei ringraziare inoltre Alessio Guerri, che mi ha aiutato a risolvere parecchi problemi inerenti ad ILOG e la prof. Michela Milano che mi ha illustrato come avviene l'integrazione tra CP e OR. Ringrazio in modo particolare tutti gli amici che hanno allietato le mie giornate di studio.

Appendice

L'algoritmo CP

```
#include <ilsolver/ilosolver.h>
#include <ilcplex/ilocplex.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<fstream.h>
#include<sstream>
#include<iostream>
#include<vector>
ILOSTLBEGIN

typedef IloArray<IloNumArray> IloNumArray2;
typedef IloArray<IloIntVarArray> IloIntVarArray2;
typedef IloArray< IloArray<IloIntVarArray> > IloIntVarArray3;
typedef IloArray<IloIntArray> IloIntArray2;
typedef IloArray< IloArray<IloIntArray> > IloIntArray3;

//--------------------------------------------------------
//         FUNZIONE GENERATORE random controllata:
//    genara le matrici random non irrealizzabili a priori
//--------------------------------------------------------
void FunzGeneratore(IloInt n, IloInt k, IloInt i1)
{
  IloInt i,j,n_k,pos_i,pos_j,val;
  IloInt ** Mat;
  Mat= new (IloInt *)[n];
  for(i=0;i<n;i++)
    Mat[i]=new IloInt[n];
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      Mat[i][j]=0;
```

```
  n_k=k;
  IloBool trovato =false;
  IloInt t=0;
  while(n_k>0)
  {
      pos_i=(rand()%n);
      pos_j=(rand()%n);
      if(Mat[pos_i][pos_j]==0)
      {
        val=(rand()%n)+1;
          t=0; trovato =false;
          while(t<n && !trovato)
          {
      if(t != pos_j && Mat[pos_i][t] == val) trovato =true;
           if(t != pos_i && Mat[t][pos_j] == val) trovato =true;
          t++;
        }
          if(!trovato)
        {
         Mat[pos_i][pos_j] = val;
         n_k = n_k - 1;
        }
       }
   }
  string x="data_ok/file", y=".dat", w="_";
  string z ;
  std::ostringstream s,s1,s3;
  if (s << n)
    if(s1 << k )
      if (s3 << i1)
{
  z=x+s.str()+w+s1.str()+w+s3.str()+y;
  ofstream  fout(z.c_str(), ios::out);
  fout<<"[";
  for(i=0;i<n;i++)
    {
      fout<<"[";
      for(j=0;j<n-1;j++)
     fout<<Mat[i][j]<<", ";
        if(i != n-1) fout<<Mat[i][n-1]<<"],"<<endl<<" ";
            else  fout<<Mat[n-1][n-1]<<"]]"<<endl;
    }
}
```

```
  for(i=0;i<n;i++)
    delete Mat[i];
}
//FINE FUNZIONE GENERATORE

/*
//-------------------------------------------------------
//         FUNZIONE GENERATORE random pura
//-------------------------------------------------------
void FunzGeneratore(IloInt n, IloInt k, IloInt i1)
{
  IloInt i,j,n_k,pos_i,pos_j,val;
  IloInt ** Mat;
  Mat= new (IloInt *)[n];
  for(i=0;i<n;i++)
    Mat[i]=new IloInt[n];
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      Mat[i][j]=0;
  n_k=k;
  while(n_k>0)
    {
      pos_i=(rand()%n);
      pos_j=(rand()%n);
      if(Mat[pos_i][pos_j]==0)
      {
        val=(rand()%n)+1;
        Mat[pos_i][pos_j] = val;
        n_k = n_k - 1;
       }
     }
  string x="data_ok/file", y=".dat", w="_";
  string z ;
  std::ostringstream s,s1,s3;
  if (s << n)
    if(s1 << k )
      if (s3 << i1)
{
  z=x+s.str()+w+s1.str()+w+s3.str()+y;
  ofstream  fout(z.c_str(), ios::out);
  fout<<"[";
  for(i=0;i<n;i++)
    {
```

```
      fout<<"[";
      for(j=0;j<n-1;j++)
    fout<<Mat[i][j]<<", ";
        if(i != n-1) fout<<Mat[i][n-1]<<"],"<<endl<<" ";
        else fout<<Mat[n-1][n-1]<<"]]"<<endl;
    }
}
  for(i=0;i<n;i++)
    delete Mat[i];
}
//FINE FUNZIONE GENERATORE Random pura
*/

ILCGOAL1(IlcGenerateMio, IlcIntVarArray, vars)
{
  IlcInt index=IlcChooseMinSizeInt(vars);
  //IlcInt index=IlcChooseMinMaxInt(vars);
  //IlcInt index=IlcChooseMaxMinInt(vars);
  if(index == -1) return 0;
  return IlcAnd(IlcInstantiate(vars[index]),this);
}

ILOCPGOALWRAPPER1(IloGenerateMio, solver, IloIntVarArray, vars)
{
  return IlcGenerateMio(solver, solver.getIntVarArray(vars));
}

//--------------------------------
//    CLASS FILE_ERROR
//--------------------------------
class FileError : public IloException {
public:
  FileError() : IloException("Cannot open data file") {}
};

//--------------------------------
//           MAIN
//--------------------------------
int main(){
  IloEnv envCP;
  try
    {
      IloInt i, j,Nmio, km;
```

```
     const char* fileName;
     fileName= "../../cplex81/prova/data_ok/file_NK.dat";
     ifstream file(fileName);
     if ( !file )
     throw FileError();
     IloIntArray2 mat_dat(envCP);
     file >> mat_dat;
     IloInt N_mat;
     N_mat=mat_dat.getSize();
     envCP.out() <<"MATRICE IN FILE_NK.DAT, dim = "<<N_mat<<endl;
     envCP.out() <<mat_dat<<endl;
     IloInt n_k,p,t, n_ok, k_ok;
     string x="data_ok/file", y=".dat", w="_";
     string z ;
     std::ostringstream s,s1,s4;
     string* arr_nomi;
     arr_nomi = new (string)[N_mat];
     ofstream  fout4("nomi.dat", ios::out);
     IloInt i1;
     for(i1=0;i1<N_mat;i1++)
          {
    n_ok=mat_dat[i1][0];
    k_ok=mat_dat[i1][1];
    if(s << n_ok)
      if(s1 <<k_ok )
        if(s4 << i1 )
{
        z=x+s.str()+w+s1.str()+w+s4.str()+y;
        arr_nomi[i1]=z.c_str();
        fout4<<arr_nomi[i1]<<endl;
        s.str(""); s1.str("");  s4.str("");
}
     FunzGeneratore(n_ok, k_ok, i1);
    }
     const char* fileName5;
     fileName5= "../../cplex81/prova/nomi.dat";
     ifstream file5(fileName5);
     if ( !file5 )
     throw FileError();
     string x_io;
     vector<string> a;
     while (!file5.eof())
      {
```

```
  file5 >> x_io;
  if (!file5.eof())
  a.push_back(x_io);
}
      string x3="../../cplex81/prova/";
      string z2 ;
      std::ostringstream s2;

      //ARRAY UTILI PER MEMORIZZARE I RISULTATI
      IloInt ki = 0;
      IloInt NumRuns=10;
      IloInt n_m= N_mat/NumRuns;
      IloInt pm=0;  //posizione negli array delle medie
      IloNumArray tempi_m(envCP,n_m);  //array che contengono le medie
      IloNumArray percCP_m(envCP,n_m);
      IloNumArray fails_m(envCP,n_m);
      IloNumArray punti_m(envCP,n_m);
      IloNumArray mem_m(envCP,n_m);
      IloInt i2,i3;
      ofstream  fout9("risultati_CP.txt", ios:: app);
      fout9<<"\n\n************************\n";
      fout9<<"  risultati_CP.txt     \n";
      fout9<<"****************************\n\n";
      fout9<<"NumRuns = "<<NumRuns<<endl<<endl;
      for(ki=0;ki<a.size();ki++)        //PRIMO FOR
    {
     fout9<<"--------------------------------\n";
     fout9<<"N = "<<mat_dat[ki][0]
          <<"   K = "<<mat_dat[ki][1]<<endl;
     fout9<<"--------------------------------\n\n";
     IloInt pos=0;
     IloEnv envF;
     IloNumArray tempi(envF, NumRuns);
     IloNumArray fails(envF, NumRuns);
     IloNumArray punti(envF, NumRuns);
     IloNumArray mem(envF, NumRuns);
     IloIntArray succCP(envF, NumRuns);
     IloIntArray succIP(envF, NumRuns);
     IloNumArray tempiIP(envF, NumRuns);
     for(i=ki; i<ki+NumRuns;i++)   //SECONDO FOR
     {
      if (s2 << a[i])          //IF S2<= a[i]
    {
```

```
  IloEnv env;
  z2=x3+s2.str();
  ifstream  file7(z2.c_str());
  cout<<"\n stampo z2:"<<z2<<endl;
  IloIntArray2 mat_parz(env);
  file7 >> mat_parz;
  IloInt N;
  N=mat_parz.getSize();
  //---------------------------
  //       MODELLO CP
  //---------------------------
  IloModel CPmodel(env);

  //-----VARIABILI
  IloIntVarArray squareArr(env,N*N,1,N);
  IloArray<IloIntVarArray> square(env,N);
  for(i3=0;i3<N;i3++)
  {
   square[i3]=IloIntVarArray(env,N);
   for(j=0;j<N;j++)
   {
    IloIntVar k = squareArr[N*i3+j];
    square[i3][j]=k;
  }
  }
  //-----VINCOLI
//VINCOLO DI INIZIALIZZAZIONE
 for(i3=0;i3<N;i3++)
 {
   for(j=0;j<N;j++)
 if( mat_parz[i3][j] != 0 )
  CPmodel.add( square[i3][j]==mat_parz[i3][j] );
 }
//VINCOLI ALLDIFFERENT PER RIGHE E COLONNE
IloIntVarArray row(env,N,1,N);
IloIntVarArray col(env,N,1,N);
for(i1=0;i1<N;i1++)  //costruisco le righe e le colonne
 {
   for(j=0;j<N;j++)
  {
    row[j]=square[i1][j];
    col[j]=square[j][i1];
   }
```

```
    CPmodel.add(IloAllDiff(env,row));
    CPmodel.add(IloAllDiff(env,col));
  }
     //-------------------------------
     //        RISOLUTORE CP
     //-------------------------------
IloSolver CPsolver(env);
CPsolver.extract(CPmodel);
IloTimer CPtimer(env);
IloNum tm;
//SCELGO LA VARIABILE CON IL DOMINIO DI TAGLIA MINIMA
IloGoal goal = IloGenerate(env, squareArr, IloChooseMinSizeInt);
IloGoal goalIP=IloGenerateMio(env,squareArr);
IloGoal goal2IP=IloLimitSearch(env,goalIP, IloTimeLimit(env,3600));
cout<<"\nLimite di tempo scelto : 3600\n";
IloNum start_time=CPsolver.getTime();
CPtimer.start();
           if(CPsolver.solve(goal2IP))
  {
    tm=CPtimer.getTime();
    cout<<"\nCPtimer.getTime()  = "<<CPtimer.getTime();
    CPtimer.stop();
    tempi[pos]=tm;
    fails[pos]=CPsolver.getNumberOfFails();
    punti[pos]=CPsolver.getNumberOfChoicePoints();
    mem[pos]=CPsolver.getMemoryUsage();
    succCP[pos]=1;
    pos=pos+1;
   /*  // STAMPO LA MATRICE SOLUZIONE
  for (i2 = 0; i2 < N; i2++)
  {
   for (j = 0; j < N; j++)
   CPsolver.out() << " " << CPsolver.getValue(square[i2][j]);
   CPsolver.out() << endl;
  }
  CPsolver.out() << endl;
   */
  }
  else
  {
    tm=CPtimer.getTime();
    CPtimer.stop();
    cout<<" CPtimer.getTime  = (tm) no = "<<tm<<endl;
```

```
    if(tm >= 0.3600)
  {
   cout<<"\n    Fuori tempo!!! Non ho trovato sol.   \n"<<endl;
   tempi[pos]= -999999 ;
   succCP[pos]= -999999 ;
   fails[pos]=CPsolver.getNumberOfFails();
   punti[pos]=CPsolver.getNumberOfChoicePoints();
   mem[pos]=CPsolver.getMemoryUsage();
   pos=pos+1;
   i = ki + NumRuns -1;
   // vado all'istanza dopo con una nuova coppia di n e k
  }
    else
  {
   tempi[pos]=tm;
   CPsolver.out() << "\nNO SOLUTION\n " << endl;
   fails[pos]=CPsolver.getNumberOfFails();
   punti[pos]=CPsolver.getNumberOfChoicePoints();
   mem[pos]=CPsolver.getMemoryUsage();
   succCP[pos]=0;
   pos=pos+1;
  }
  }
 CPsolver.printInformation();
 env.end();
 s2.str("");
  } //FINE IF  s2<<a[i]
   }//FINE DEL SECONDO FOR : QUELLO PER i=ki

  //---- CALCOLO LE MEDIE E LE INSERISCO NEGLI ARRAY DELLE MEDIE
IloNum media_t=0, media_f=0, media_p=0, media_m=0 ;
IloInt NumRuns_ok=0;
for(km=0;km<NumRuns;km++)
  {
    NumRuns_ok=NumRuns_ok+ succCP[km];
    media_t=media_t + tempi[km];
    media_f=media_f + fails[km];
    media_p=media_p + punti[km];
    media_m= media_m + mem[km];
  }
percCP_m[pm]= NumRuns_ok * 100/ NumRuns;
tempi_m[pm]=media_t / NumRuns;
fails_m[pm]=media_f / NumRuns;
```

```
punti_m[pm]=media_p / NumRuns;
mem_m[pm]=media_m / NumRuns;

//APPENDO I RISULTATI AL FILE risultati_CP.txt
fout9<<"   ARRAY CON I VALORI ORIGINALI ";
fout9<<"\n Array dei tempi : \n";
fout9<<"[";
for(i=0;i<NumRuns;i++)
  {if (i == NumRuns-1) fout9<<tempi[i]<<"]\n";
  else  fout9<<tempi[i]<<", ";
  }
fout9<<"\n Array dei successi :\n";
fout9<<"[";
for(i=0;i<NumRuns;i++)
  {if (i == NumRuns-1) fout9<<succCP[i]<<"]\n";
  else  fout9<<succCP[i]<<", ";
  }
fout9<<"\n Array dei fails :\n";
fout9<<"[";
for(i=0;i<NumRuns;i++)
  {if (i == NumRuns-1) fout9<<fails[i]<<"]\n";
  else  fout9<<fails[i]<<", ";
  }
fout9<<"\n Array dei punti di scelta :\n";
fout9<<"[";
for(i=0;i<NumRuns;i++)
  {if (i == NumRuns-1) fout9<<punti[i]<<"]\n";
  else  fout9<<punti[i]<<", ";
  }
fout9<<"\n Array della memoria usata :\n";
fout9<<"[";
for(i=0;i<NumRuns;i++)
  {if (i == NumRuns-1) fout9<<mem[i]<<"]\n";
  else  fout9<<mem[i]<<", ";
  }
fout9<<"\n-----VALORI MEDI------";
fout9<<"\n Media dei tempi            : ";
fout9<<tempi_m[pm];
fout9<<"\n Percentuale dei successi   : ";
fout9<<percCP_m[pm];
fout9<<"\n Media dei fails            : ";
fout9<<fails_m[pm];
fout9<<"\n Media dei punti di scelta  : ";
```

```
fout9<<punti_m[pm];
fout9<<"\n Media della memoria usata  : ";
fout9<<mem_m[pm];
fout9<<"\n---------------------\n\n";
pm=pm+1;
cout<<"----pm"<<pm<<endl;
ki=ki+NumRuns-1;
envF.end();
}  //FINE DEL PRIMO FOR
    fout9<<"\n\n----------------------------\n";
    fout9<<"  ARRAY  CON TUTTI I VALORI MEDI  \n";
    fout9<<"--------------------------------\n";
    fout9<<"\n Array delle medie dei tempi : \n";
    fout9<<"[";
    for(i=0;i<n_m;i++)
   { if (i == n_m-1) fout9<<tempi_m[i]<<"]\n";
     else  fout9<<tempi_m[i]<<", ";
   }
    fout9<<"\n Array delle percentuali di successi :\n";
    fout9<<"[";
    for(i=0;i<n_m;i++)
    { if (i == n_m-1) fout9<<percCP_m[i]<<"]\n";
    else  fout9<<percCP_m[i]<<", ";
  }
    fout9<<"\n Array delle medie dei fails :\n";
    fout9<<"[";
    for(i=0;i<n_m;i++)
  { if (i == n_m-1) fout9<<fails_m[i]<<"]\n";
   else  fout9<<fails_m[i]<<", ";
  }
    fout9<<"\n Array delle medie de punti di scelta :\n";
    fout9<<"[";
    for(i=0;i<n_m;i++)
  { if (i == n_m-1) fout9<<punti_m[i]<<"]\n";
    else  fout9<<punti_m[i]<<", ";
  }
    fout9<<"\n Array delle medie della memoria usata :\n";
    fout9<<"[";
    for(i=0;i<n_m;i++)
  { if (i == n_m-1) fout9<<mem_m[i]<<"]\n";
     else  fout9<<mem_m[i]<<", ";
   }
    fout9<<"\n----------------------------\n";
```

```
 }
 catch (IloException& ex)
   { cout << "Error: " << ex << endl;   }
 envCP.end();
 return 0;
} //Fine
```

L'algoritmo ILP

```
#include <ilsolver/ilosolver.h>
#include <ilcplex/ilocplex.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<fstream.h>
#include<sstream>
#include<iostream>
#include<vector>
ILOSTLBEGIN

typedef IloArray<IloNumArray> IloNumArray2;
typedef IloArray<IloIntVarArray> IloIntVarArray2;
typedef IloArray< IloArray<IloIntVarArray> > IloIntVarArray3;
typedef IloArray<IloIntArray> IloIntArray2;
typedef IloArray< IloArray<IloIntArray> > IloIntArray3;

//--------------------------------------------------------
//         FUNZIONE GENERATORE random controllata
//--------------------------------------------------------
void FunzGeneratore(IloInt n, IloInt k, IloInt i1)
{
  IloInt i,j,n_k,pos_i,pos_j,val;
  IloInt ** Mat;
  Mat= new (IloInt *)[n];
  for(i=0;i<n;i++)
    Mat[i]=new IloInt[n];
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      Mat[i][j]=0;
  n_k=k;
  IloBool trovato =false;
  IloInt t=0;
```

```
  while(n_k>0)
  {
      pos_i=(rand()%n);
      pos_j=(rand()%n);
      if(Mat[pos_i][pos_j]==0)
      {
  val=(rand()%n)+1;
          t=0; trovato =false;
          while(t<n && !trovato)
          {
      if(t != pos_j && Mat[pos_i][t] == val) trovato =true;
           if(t != pos_i && Mat[t][pos_j] == val) trovato =true;
          t++;
   }
          if(!trovato)
        {
         Mat[pos_i][pos_j] = val;
         n_k = n_k - 1;
        }
       }
   }
  string x="data_ok/file", y=".dat", w="_";
  string z ;
  std::ostringstream s,s1,s3;
  if (s << n)
    if(s1 << k )
      if (s3 << i1)
{
  z=x+s.str()+w+s1.str()+w+s3.str()+y;
  ofstream  fout(z.c_str(), ios::out);
  fout<<"[";
  for(i=0;i<n;i++)
    {
      fout<<"[";
      for(j=0;j<n-1;j++)
      fout<<Mat[i][j]<<", ";
              if(i != n-1) fout<<Mat[i][n-1]<<"],"<<endl<<" ";
               else  fout<<Mat[n-1][n-1]<<"]]"<<endl;
    }
}
  for(i=0;i<n;i++)
    delete Mat[i];
}
```

```
//FINE FUNZIONE GENERATORE

/*
//-------------------------------------------------------
//         FUNZIONE GENERATORE random pura
//-------------------------------------------------------
void FunzGeneratore(IloInt n, IloInt k, IloInt i1)
{
  IloInt i,j,n_k,pos_i,pos_j,val;
  IloInt ** Mat;
  Mat= new (IloInt *)[n];
  for(i=0;i<n;i++)
    Mat[i]=new IloInt[n];
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      Mat[i][j]=0;
  n_k=k;
  while(n_k>0)
    {
      pos_i=(rand()%n);
      pos_j=(rand()%n);
      if(Mat[pos_i][pos_j]==0)
      {
        val=(rand()%n)+1;
        Mat[pos_i][pos_j] = val;
        n_k = n_k - 1;
       }
     }
  string x="data_ok/file", y=".dat", w="_";
  string z ;
  std::ostringstream s,s1,s3;
  if (s << n)
    if(s1 << k )
      if (s3 << i1)
{
  z=x+s.str()+w+s1.str()+w+s3.str()+y;
  ofstream  fout(z.c_str(), ios::out);
  fout<<"[";
  for(i=0;i<n;i++)
    {
      fout<<"[";
      for(j=0;j<n-1;j++)
              fout<<Mat[i][j]<<", ";
```

```
            if(i != n-1) fout<<Mat[i][n-1]<<"],"<<endl<<" ";
            else fout<<Mat[n-1][n-1]<<"]]"<<endl;
    }
}
  for(i=0;i<n;i++)
    delete Mat[i];
}
//FINE FUNZIONE GENERATORE Random pura
*/
//--------------------
//  CLASS FILE_ERROR
//--------------------
class FileError : public IloException {
public:
  FileError() : IloException("Cannot open data file") {}
};

//------------------------
//       MAIN
//------------------------
int main(){
  IloEnv envIP;
  try {
    IloInt i, j,Nmio, km;
    const char* fileName;
    fileName= "../../cplex81/prova/data_ok/file_NK.dat";
    ifstream file(fileName);
    if ( !file )
      throw FileError();
    IloIntArray2 mat_dat(envIP);
    file >> mat_dat;
    IloInt N_mat;
    N_mat=mat_dat.getSize();
    envIP.out() <<"MATRICE IN FILE_NK.DAT   di dimensione :"
                <<N_mat<<endl;
    envIP.out() <<mat_dat<<endl;
    IloInt n_k,p,t, n_ok, k_ok;
    string x="data_ok/file", y=".dat", w="_";
    string z ;
    std::ostringstream s,s1,s4;
    string* arr_nomi;  //ok
    arr_nomi = new (string)[N_mat];
    ofstream  fout4("nomi.dat", ios::out);
```

```
    IloInt i1;
    for(i1=0;i1<N_mat;i1++)
      {
       n_ok=mat_dat[i1][0];
k_ok=mat_dat[i1][1];
        if (s << n_ok)
   if(s1 <<k_ok )
      if(s4 << i1 )
      { z=x+s.str()+w+s1.str()+w+s4.str()+y;
      arr_nomi[i1]=z.c_str();
      fout4<<arr_nomi[i1]<<endl;
      s.str(""); s1.str("");  s4.str("");
      }
        FunzGeneratore(n_ok, k_ok, i1);
      }
    const char* fileName5;
    fileName5= "../../cplex81/prova/nomi.dat";
    ifstream file5(fileName5);
    if ( !file5 )
      throw FileError();
    string x_io;
    vector<string> a;
    while (!file5.eof())
      {  file5 >> x_io;
         if (!file5.eof())
    a.push_back(x_io);
    }
    string x3="../../cplex81/prova/";
    string z2 ;
    std::ostringstream s2;
    IloInt ki = 0;
    IloInt NumRuns=10;
    cout<<"\nNumRuns  : "<<NumRuns<<endl;
    IloInt n_m= N_mat/NumRuns;
    cout<<"Numero degli el degli array medi :"<<n_m<<endl;
    IloInt pm=0;  //posizione negli array delle medie
    cout<<"----pm :"<<pm<<endl;
    //array che contengono le medie
    IloNumArray tempiIP_m(envIP,n_m);
    IloNumArray percIP_m(envIP,n_m);
    IloInt i2,i3;
    ofstream  fout8("risultati_LP.txt", ios:: app);
    fout8<<"\n\n*********************\n";
```

```
 fout8<<" risultati_LP.txt    \n";
 fout8<<"************************\n\n";
 fout8<<"NumRuns = "<<NumRuns<<endl<<endl;
 for(ki=0;ki<a.size();ki++)         //FOR GRANDE
 {
fout8<<"----------------------\n";
fout8<<"N = "<<mat_dat[ki][0]
     <<"   K = "<<mat_dat[ki][1]<<endl;
fout8<<"------------------------\n\n";
IloInt pos=0;
IloEnv env;
IloNumArray tempi(env, NumRuns);
IloNumArray fails(env, NumRuns);
IloNumArray punti(env, NumRuns);
IloNumArray mem(env, NumRuns);
IloIntArray succCP(env, NumRuns);
IloIntArray succIP(env, NumRuns);
IloNumArray tempiIP(env, NumRuns);
for(i=ki; i<ki+NumRuns;i++)   //SECONDO FOR
{
if (s2 << a[i])                //IF S2<= a[i]
 {
z2=x3+s2.str();
ifstream  file7(z2.c_str());
cout<<"\n stampo z2:"<<z2<<endl;
IloIntArray2 mat_parz(env);
file7 >> mat_parz;
IloInt N;
N=mat_parz.getSize();
//-----------------------------------
//        MODELLO ILP
//-----------------------------------
IloModel IPmodel(env);

//VARIABILI
IloIntVarArray matBinArr(env,N*N*N,0,1);
IloArray<IloIntVarArray2> matBin(env,N);
for(i3=0;i3<N;i3++)
 {
  matBin[i3]=IloIntVarArray2(env,N);
  for(j=0;j<N;j++)
  {
 matBin[i3][j]=IloIntVarArray(env,N);
```

```
   for(t=0;t<N;t++)
    {
     IloIntVar k_io = matBinArr[N*N*i3 + j*N +t];
     matBin[i3][j][t]=k_io;
    }
     }
    }
  //VINCOLO DI INIZIALIZZAZIONE matrice 3-dim
  for(i3=0;i3<N;i3++)
   { for(j=0;j<N;j++)
     for(t=1;t<=N;t++)
       { if( mat_parz[i3][j] == t  )
        IPmodel.add( matBin[i3][j][t-1] == 1 );
       }
   }
  //VINCOLI ALLDIFFERENT PER RIGHE E COLONNE
  IloIntVarArray row1(env,N,0,1);
  IloIntVarArray row2(env,N,0,1);
  IloIntVarArray col1(env,N,0,1);
  for(i1=0;i1<N;i1++)
   {
     for(j=0;j<N;j++)
       {
for(t=1;t<N+1;t++)
  { row2[t-1]=matBin[i1][j][t-1];
  row1[t-1]=matBin[i1][t-1][j];
  col1[t-1]=matBin[t-1][i1][j];
  }
IPmodel.add(IloSum(row1) <= 1);
IPmodel.add(IloSum(row2) <= 1);
IPmodel.add(IloSum(col1) <= 1);
       }
   }
  IloIntVar tot_colors(env,1,N*N);
  IPmodel.add(tot_colors == IloSum(matBinArr));
  IloObjective obj=IloMaximize(env,tot_colors);
  IPmodel.add(obj);
  //----------------------------------------
  //            RISOLUTORE ILP
  //----------------------------------------
  IloCplex IPsolver(env);
  IPsolver.extract(IPmodel);
          IPsolver.setParam(IloCplex:: RootAlg, IloCplex:: Primal);
```

```
//TAGLI DI GOMORY
//IPsolver.setParam(IloCplex::FracCuts, -1);
IPsolver.setParam(IloCplex::FracCuts, 0);
//  IPsolver.setParam(IloCplex::FracCuts, 1);
//  IPsolver.setParam(IloCplex::FracCuts, 2);
IPsolver.setParam(IloCplex::TiLim, 3600);
IPsolver.setParam(IloCplex::MIPDisplay,2);
IPsolver.setParam(IloCplex::MIPInterval,10);
                cout<<"*****getParam(IloCplex::TiLim) : "
   <<IPsolver.getParam(IloCplex::TiLim)<<endl;
cout<<"*****getParam(IloCplex::MIPDisplay) : "
   <<IPsolver.getParam(IloCplex::MIPDisplay)<<endl;
cout<<"*****getParam(IloCplex::FracCuts) : "
   <<IPsolver.getParam(IloCplex::FracCuts)<<endl;
IloNum tmIP;
IloTimer IPtimer(env);
cout<<"*** Ricerca iniziata \n";
IPtimer.start();
if( IPsolver.solve() )
 {
   tmIP=IPtimer.getTime();
   IPtimer.stop();
   cout<<"tmIP SI = IPtimer.getTime()  = "<<tmIP<<endl;
   if(tmIP >= 0.36)
     {
   cout<<"\n    Fuori tempo!!! Non ho trovato sol. \n"<<endl;
   tempiIP[pos]=  -999999 ;
   succIP[pos]=  -999999 ;
   pos=pos+1;   cout<<"\n---pos (no) Limit time :"<<pos<<endl;
   i = ki + NumRuns -1;
     }
   else
     {
 tempiIP[pos]=tmIP;
 succIP[pos]=1;
 pos=pos+1;
 cout<<"\n---pos (si') :"<<pos<<endl;
 cout << "*****stato della soluzione ="<<IPsolver.getStatus()<<endl;
 cout<<"\n*****valore della f obiettivo :"<<IPsolver.getObjValue()<<endl;
/*
   cout<<"\n*****Matrice binaria ottimizzata 3-DIM:\n\n";
  for (i2 = 0; i2 < N; i2++)
 {   for (j = 0; j < N; j++)
```

```
    {
      if(j>0) cout<<"*****";
      for (t = 0; t <N ; t++)
       IPsolver.out()<<" "<<IPsolver.getValue(matBin[i2][j][t]);
  }
     IPsolver.out() << endl;
  }
cout<<"\n stampo la matrice ottima 2-DIM :\n";
IloInt cont=0;
for (i2 = 0; i2 < N; i2++)
  { for (j = 0; j < N; j++)
    {
      cont=0;
      for (t = 1; t <N+1 ; t++)
       {
       IloNum n=IPsolver.getValue(matBin[i2][j][t-1]);
       if(n == 1)
        {
         IPsolver.out()<<" "<<t<<" ";
         cont++;
         }
       }
      if( cont == 0) cout<<"   ";
    }
   IPsolver.out() << endl;
  }
*/
      }
  }  //fine if solo dell' IP .solve()
else
  {
    tmIP=IPtimer.getTime();
    IPtimer.stop();
    cout<<"tmIP NO = IPtimer.getTime()  = "<<tmIP<<endl;
    IPsolver.out() << "\nNo solution\n " << endl;
    if(tmIP >= 0.36)
      {
     cout<<"\n    Fuori tempo!!! Non ho trovato sol.   \n"<<endl;
     tempiIP[pos]=  -999999 ;
     succIP[pos]=  -999999 ;
     pos=pos+1;   cout<<"\n---pos (no) Limit time :"<<pos<<endl;
     i = ki + NumRuns -1;
      }
```

```
  else
    {
  IPsolver.out() << "\nNO SOLUTION\n " << endl;
  tempiIP[pos]=tmIP;
  succIP[pos]=0;
  pos=pos+1;   cout<<"\n---pos (no) :"<<pos<<endl;
    }
}
s2.str("");
  }  //FINE IF  s2<<a[i]
 }//FINE DEL SECONDO FOR : QUELLO PER i=ki
 //---------------------------------
// STAMPO GLI ARRAY CON TUTTI I DATI
//---------------------------------
cout<<"\nARRAY CON I VALORI ORIGINALI\n";
env.out() <<"Array dei tempi IP :\n"<<tempiIP<<endl;
env.out() <<"Array dei successi IP :\n"<<succIP<<endl;
// CALCOLO LE MEDIE E LE INSERISCO NEGLI ARRAY DELLE MEDIE
IloNum media_t=0;
IloInt NumRuns_ok=0;
for(km=0;km<NumRuns;km++)
{
  NumRuns_ok=NumRuns_ok+ succIP[km];
  media_t=media_t + tempiIP[km];
}
cout<<"-----NumRuns_ok :"<< NumRuns_ok<<endl;
cout<<"--NumRuns :" <<NumRuns<<endl;
percIP_m[pm]= NumRuns_ok * 100/ NumRuns; //calcolo la percentuali
tempiIP_m[pm]=media_t / NumRuns;
cout<<"\n-----VALORI MEDI------  \n";
env.out() <<"Media dei tempi   :"<<tempiIP_m[pm]<<endl;
env.out() <<"Percentuale di successi   :"<<percIP_m[pm]<<endl;
cout<<"-------------------------\n";
//APPENDO I RISULTATI AL FILE risultati_LP.txt
fout8<<"--------- ARRAY CON I VALORI ORIGINALI -------";
fout8<<"\n Array dei tempi : \n";
fout8<<"[";
  for(i=0;i<NumRuns;i++)
  {  if (i == NumRuns-1) fout8<<tempiIP[i]<<"]\n";
  else  fout8<<tempiIP[i]<<", ";
  }
fout8<<"\n Array dei successi :\n";
fout8<<"[";
```

```
    for(i=0;i<NumRuns;i++)
    {if (i == NumRuns-1) fout8<<succIP[i]<<"]\n";
    else  fout8<<succIP[i]<<", ";
    }
  fout8<<"\n---------- VALORI MEDI-----------------";
  fout8<<"\n Media dei tempi            : ";
  fout8<<tempiIP_m[pm];
  fout8<<"\n Percentuali dei successi   : ";
  fout8<<percIP_m[pm];
  fout8<<"\n-------------------------------------\n\n";
  pm=pm+1;
  cout<<"----pm"<<pm<<endl;
  ki=ki+NumRuns-1;
  env.end();
      } //FINE DEL FOR GRANDE
      //APPENDO I RISULTATI
      fout8<<"\n\n---------------------------------\n";
      fout8<<"   ARRAY  CON TUTTI I VALORI MEDI \n";
      fout8<<"---------------------------------\n";
      fout8<<"\n Array delle medie dei tempi : \n";
      fout8<<"[";
     for(i=0;i<n_m;i++)
    { if (i == n_m-1) fout8<<tempiIP_m[i]<<"]\n";
     else  fout8<<tempiIP_m[i]<<", ";
      }
      fout8<<"\n Array delle percentuali di successi :\n";
      fout8<<"[";
    for(i=0;i<n_m;i++)
    { if (i == n_m-1) fout8<<percIP_m[i]<<"]\n";
     else  fout8<<percIP_m[i]<<", ";
      }
      fout8<<"\n-----------------------------\n";
 } //parentesi di fine del try
 catch (IloException& ex)
 { cout << "Error: " << ex << endl; }
 envIP.end();
 return 0;
}      //Fine
```

L'algoritmo ibrido CP/ILP

```
#include <ilhybrid/ilohybrid.h>
```

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<fstream.h>
#include<sstream>
#include<iostream>
#include<vector>
ILOSTLBEGIN

typedef IloArray<IloNumArray> IloNumArray2;
typedef IloArray<IloIntVarArray> IloIntVarArray2;
typedef IloArray< IloArray<IloIntVarArray> > IloIntVarArray3;
typedef IloArray<IloNumVarArray> IloNumVarArray2;
typedef IloArray< IloArray<IloNumVarArray> > IloNumVarArray3;
typedef IloArray<IloIntArray> IloIntArray2;
typedef IloArray< IloArray<IloIntArray> > IloIntArray3;
typedef IloArray<IlcIntVarArray> IlcIntVarArray2;
typedef IloArray< IloArray<IlcIntVarArray> > IlcIntVarArray3;

//-------------------------------------------------------
//         FUNZIONE GENERATORE random controllata
//-------------------------------------------------------
void FunzGeneratore(IloInt n, IloInt k, IloInt i1)
{
  IloInt i,j,n_k,pos_i,pos_j,val;
  IloInt ** Mat;
  Mat= new (IloInt *)[n];
  for(i=0;i<n;i++)
    Mat[i]=new IloInt[n];
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      Mat[i][j]=0;
  n_k=k;
  IloBool trovato =false;
  IloInt t=0;
  while(n_k>0)
  {
      pos_i=(rand()%n);
      pos_j=(rand()%n);
      if(Mat[pos_i][pos_j]==0)
      {
        val=(rand()%n)+1;
          t=0; trovato =false;
```

```
          while(t<n && !trovato)
          {
          if(t != pos_j && Mat[pos_i][t] == val) trovato =true;
           if(t != pos_i && Mat[t][pos_j] == val) trovato =true;
          t++;
        }
          if(!trovato)
        {
         Mat[pos_i][pos_j] = val;
         n_k = n_k - 1;
        }
       }
   }
  string x="data_ok/file", y=".dat", w="_";
  string z ;
  std::ostringstream s,s1,s3;
  if (s << n)
    if(s1 << k )
      if (s3 << i1)
{
  z=x+s.str()+w+s1.str()+w+s3.str()+y;
  ofstream  fout(z.c_str(), ios::out);
  fout<<"[";
  for(i=0;i<n;i++)
    {
      fout<<"[";
      for(j=0;j<n-1;j++)
      fout<<Mat[i][j]<<", ";
              if(i != n-1) fout<<Mat[i][n-1]<<"],"<<endl<<" ";
               else  fout<<Mat[n-1][n-1]<<"]]"<<endl;
    }
}
  for(i=0;i<n;i++)
    delete Mat[i];
}
//FINE FUNZIONE GENERATORE
/*
//---------------------------------------------------------
//         FUNZIONE GENERATORE random pura
//---------------------------------------------------------
void FunzGeneratore(IloInt n, IloInt k, IloInt i1)
{
  IloInt i,j,n_k,pos_i,pos_j,val;
```

```
  IloInt ** Mat;
  Mat= new (IloInt *)[n];
  for(i=0;i<n;i++)
    Mat[i]=new IloInt[n];
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      Mat[i][j]=0;
  n_k=k;
  while(n_k>0)
    {
      pos_i=(rand()%n);
      pos_j=(rand()%n);
      if(Mat[pos_i][pos_j]==0)
      {
        val=(rand()%n)+1;
        Mat[pos_i][pos_j] = val;
        n_k = n_k - 1;
       }
     }
  string x="data_ok/file", y=".dat", w="_";
  string z ;
  std::ostringstream s,s1,s3;
  if (s << n)
    if(s1 << k )
      if (s3 << i1)
{
  z=x+s.str()+w+s1.str()+w+s3.str()+y;
  ofstream  fout(z.c_str(), ios::out);
  fout<<"[";
  for(i=0;i<n;i++)
    {
      fout<<"[";
      for(j=0;j<n-1;j++)
              fout<<Mat[i][j]<<", ";
              if(i != n-1) fout<<Mat[i][n-1]<<"],"<<endl<<" ";
              else fout<<Mat[n-1][n-1]<<"]]"<<endl;
    }
}
  for(i=0;i<n;i++)
    delete Mat[i];
}
//FINE FUNZIONE GENERATORE Random pura
*/
```

```
////////////////////////////////////////////////
//            PROGRAMMING THE SEARCH
////////////////////////////////////////////////
IlcInt cont=5, contS=1;
IlcInt N;
//------------------------------------------------
//    ILO_SELECT_VAR     (MostNearOne)
//------------------------------------------------
ILOSELECTVAR2(MostNearOne, IloNumVar, x,
              IloLinConstraint, lc){
  if (lc.distToInt(lc.getValue(x)) <= lc.getFeasTolerance())
    return -IloInfinity;
  else
    return lc.frac(lc.getValue(x));
}
//------------------------------------------------
//   ILO_SELECT_VAR    (MostNotInteger)
//------------------------------------------------
ILOSELECTVAR2(MostNotInteger, IloNumVar, x,
              IloLinConstraint, lc){
  IloNum dist = lc.distToInt(lc.getValue(x));
  if (dist < 1e-4)
    return -IloInfinity;
  else
    return dist;
}
//------------------------------------------------
//    ILC_GOAL  ( SOLO 1 GOAL )
//------------------------------------------------
ILCGOAL3(MixedGenerateIlc, IloLinConstraint, lc,
         IlcIntVarArray, y,IloNumVarArray, x )
{
  IlcInt indexLP;
   indexLP = MostNotInteger(x, lc);
  //indexLP = MostNearOne(x,lc);
  if (indexLP == -1)
    {
      lc.fixObjToValue();
      lc.fixAllVarsToValue();
      return 0;
    }
  return IlcAnd(IlcBranchCloserFirst(x[indexLP], lc), this);
  //return IlcAnd(IlcBranchFartherFirst(x[indexLP], lc), this);
```

```
  //return IlcAnd(IlcBranchDownFirst(x[indexLP], lc), this);
  //return IlcAnd(IlcBranchUpFirst(x[indexLP], lc), this);
  //return IlcAnd(IlcInstantiate(y[indexCP]), this);
}
/*
//-------------------------------------------------
//                ILC_GOAL  (DOPPIO)
//-------------------------------------------------
ILCGOAL3(MixedGenerateIlc, IloLinConstraint, lc,
         IlcIntVarArray, y, IloNumVarArray, x){
     IlcInt index;
     IloSolver solver=getSolver();
     //CASO CP
     if(cont != 0) //di sicuro dopo la prima scelta
     {
       index = IlcChooseMinSizeInt(y);
       lc.setSynchronization(IloLinConstraint::SolverOnly);
       cont--;
       if(cont==0)
lc.setSynchronization(IloLinConstraint::LinConstraintAndSolver);
       if(index == -1)
       { cout<<"\nSoluzione CP trovata\n";
         return 0;
       }
       return IlcAnd(IlcInstantiate(y[index]), this);
     }
    //---CASO CP-LP  (cont=0)
    else
     {
       index = MostNotInteger(x, lc);
       // index=MostNearOne(x,lc);
       lc.setSynchronization(IloLinConstraint::SolverOnly);
       //cont = 5;
       cont = 1;     //modificare cont secondo le esigenze
       if (index == -1)
          // cioe' ho trovato una soluzione intera
      {
         lc.fixObjToValue();
         lc.fixAllVarsToValue();
         return 0;
         }
       return IlcAnd(IlcBranchCloserFirst(x[index], lc), this);
     } //fine else
```

```
}//fine goal doppio
*/
//-----------------------------------------------
//    ILC_GOAL_CP_WRAPPER
//-----------------------------------------------
ILOCPGOALWRAPPER3(MixedGenerate, solver, IloLinConstraint,
                  lc,IloIntVarArray, y,IloNumVarArray, x)
{
  return MixedGenerateIlc(solver, lc,solver.getIntVarArray(y), x);
}
////////////////////  FINE SEARCH ////////////////

//-----------------------------------------------
//           CLASS FILE_ERROR
//-----------------------------------------------
class FileError : public IloException {
public:
  FileError() : IloException("Cannot open data file") {}
};
//-----------------------------------------------
//             MAIN
//-----------------------------------------------
int main()
{
  IloEnv envCP;
  try
    {
      IloInt i, j, Nmio, km, n_k, p, t, t1;
      IloInt n_ok,k_ok;
      IloInt i1, i2, i3, ki = 0, NumRuns = 1, N_mat;
      cout<<"\n     risultati_ibridi.txt   \n";
      const char* fileName;
      fileName= "../../cplex81/prova/data_ok/file_NK.dat";
      ifstream file(fileName);
      if ( !file )
        throw FileError();
      IloIntArray2 mat_dat(envCP);
      file >> mat_dat;
      N_mat=mat_dat.getSize();
      envCP.out() <<"MATRICE IN FILE_NK.DAT   di dimensione :"<<N_mat<<endl;
      envCP.out() <<mat_dat<<endl;
      string x="data_ok/file", y=".dat", w="_";
```

```
 string z ;
 std::ostringstream s,s1,s4;
 string* arr_nomi;  //ok
 arr_nomi = new (string)[N_mat];
 ofstream  fout4("nomi.dat", ios::out);
 for(i1=0;i1<N_mat;i1++)
  {
 n_ok=mat_dat[i1][0];
 k_ok=mat_dat[i1][1];
 if (s << n_ok)
   if(s1 <<k_ok )
      if(s4 << i1 )
  { z=x+s.str()+w+s1.str()+w+s4.str()+y;
  arr_nomi[i1]=z.c_str();
  fout4<<arr_nomi[i1]<<endl;
  s.str(""); s1.str("");  s4.str("");
  }
 FunzGeneratore(n_ok, k_ok, i1);
  }
 const char* fileName5;
 fileName5= "../../cplex81/prova/nomi.dat";
 ifstream file5(fileName5);
 if ( !file5 )
  throw FileError();
 string x_io;
 vector<string> a;
 while (!file5.eof())
{
 file5 >> x_io;
 if (!file5.eof())
  a.push_back(x_io);
}
 string x3="../../cplex81/prova/";
 string z2 ;
 std::ostringstream s2;
 NumRuns=10;
 cout<<"\nNumRuns  : "<<NumRuns<<endl;
 IloInt n_m= N_mat/NumRuns;
 cout<<"Numero di el degli array medi :"<<n_m<<endl;
 IloInt pm=0;  //posizione negli array delle medie
 cout<<"----pm :"<<pm<<endl;
 IloNumArray tempi_m(envCP,n_m);
 IloNumArray percCP_m(envCP,n_m);
```

```
    IloNumArray fails_m(envCP,n_m);
    IloNumArray punti_m(envCP,n_m);
    IloNumArray mem_m(envCP,n_m);
    ofstream  fout9("risultati_ibridi.txt", ios:: app);
    fout9<<"\n\n*****************************\n";
    fout9<<"         risultati_ibridi.txt          \n";
    fout9<<"*********************************\n\n";
    fout9<<"NumRuns = "<<NumRuns<<endl<<endl;
       //PRIMO FOR
    for(ki=0;ki<a.size();ki++)
{
fout9<<"-----------------------------------\n";
fout9<<"  N = "<<mat_dat[ki][0]
     <<"   K = "<<mat_dat[ki][1]<<endl;
fout9<<"-----------------------------------\n\n";
IloInt pos=0;
IloEnv envF;
IloNumArray tempi(envF, NumRuns);
IloNumArray fails(envF, NumRuns);
IloNumArray punti(envF, NumRuns);
IloNumArray mem(envF, NumRuns);
IloIntArray succCP(envF, NumRuns);
IloIntArray succIP(envF, NumRuns);
IloNumArray tempiIP(envF, NumRuns);
//SECONDO FOR
for(i=ki; i<ki+NumRuns;i++)
  { //IF S2<= a[i]
   if (s2 << a[i])
  {
IloEnv env;
cont=0;
z2=x3+s2.str();
ifstream  file7(z2.c_str());
cout<<"\n stampo z2:"<<z2<<endl;
IloIntArray2 mat_parz(env);
file7 >> mat_parz;
//IloInt N;
N=mat_parz.getSize();
//-----------------------------------
//         MODELLO IBRIDO
//-----------------------------------
IloModel model(env);
IloModel modelIP(env);
```

```
//VARIABILI CP
IloIntVarArray squareArr(env,N*N,1,N);
IloArray<IloIntVarArray> square(env,N);
for(i3=0;i3<N;i3++)  //metto le variabili in un array 2-dim
  {
    square[i3]=IloIntVarArray(env,N);
    for(j=0;j<N;j++)
  {
  IloIntVar k = squareArr[N*i3+j];
  square[i3][j]=k;
  }
  }
//VARIABILI IP
//metodo per dichiarare una matrice 3-dim e allocare memoria ok
IloIntVarArray matBinArr(env,N*N*N,0,1);
IloArray<IloIntVarArray2> matBin(env,N);
for(i3=0;i3<N;i3++)
  {
    matBin[i3]=IloIntVarArray2(env,N);
    for(j=0;j<N;j++)
  {
    matBin[i3][j]=IloIntVarArray(env,N);
    for(t=0;t<N;t++)
    {
    IloIntVar k_io = matBinArr[N*N*i3 + j*N + t];
    //in matBinArr ho l'array di dimensione N*N*N
    matBin[i3][j][t]=k_io;
    }
  }
  }
IloIntVar tot_colors(env,1,N*N);
IloObjective obj = IloMaximize(env,tot_colors);

//VINCOLI CP
//---VINCOLO DI INIZIALIZZAZIONE dI SQUARE con mat_parz presa da input
for(i3=0;i3<N;i3++)
  {
    for(j=0;j<N;j++)
  if( mat_parz[i3][j] != 0 )
    model.add( square[i3][j] == mat_parz[i3][j] );
  }
  //---VINCOLI ALLDIFFERENT PER RIGHE E COLONNE
```

```
IloIntVarArray row(env,N,1,N);
IloIntVarArray col(env,N,1,N);
for(i1=0;i1<N;i1++)
  {
    for(j=0;j<N;j++)
    { row[j]=square[i1][j];
   col[j]=square[j][i1];
   }
    model.add(IloAllDiff(env,row));
    model.add(IloAllDiff(env,col));
  }
//VINCOLI IP
//---VINCOLO DI INIZIALIZZAZIONE matBin con mat_parz
for(i3=0;i3<N;i3++)
  { for(j=0;j<N;j++)
    { for(t=1;t<=N;t++)
    {
       if( mat_parz[i3][j] == t  )
        modelIP.add( matBin[i3][j][t-1] == 1 );
     }
    }
  }
//----VINCOLI ALLDIFFERENT PER RIGHE E COLONNE
IloIntVarArray row1(env,N,0,1);
IloIntVarArray row2(env,N,0,1);
IloIntVarArray col1(env,N,0,1);
for(i1=0;i1<N;i1++)
  {
    for(j=0;j<N;j++)
  {
   for(t=1;t<N+1;t++)
    {
     row2[t-1]=matBin[i1][j][t-1];
     row1[t-1]=matBin[i1][t-1][j];
     col1[t-1]=matBin[t-1][i1][j];
    }
   modelIP.add(IloSum(row1) <= 1);
   modelIP.add(IloSum(row2) <= 1);
   modelIP.add(IloSum(col1) <= 1);
 }
  }
modelIP.add(tot_colors == IloSum(matBinArr));
modelIP.add(obj);
```

```
//-----VINCOLI DI LINK
for(i3=0;i3<N;i3++)
  {
   for(j=0;j<N;j++)
 {
   for(t=0;t<N;t++)
  {
    model.add(IloIfThen(env,square[i3][j] == t+1 && matBin[i3][j][t] != 1,
                            matBin[i3][j][t] == 1));
    model.add(IloIfThen(env,square[i3][j] != t+1 && matBin[i3][j][t] != 0,
               matBin[i3][j][t] == 0));
    model.add(IloIfThen(env,matBin[i3][j][t] == 1 && square[i3][j] != t+1,
               square[i3][j] == t+1));
    model.add(IloIfThen(env,matBin[i3][j][t] == 0 && square[i3][j] == t+1,
               square[i3][j] != t+1));
  }
 }
  }
model.add(modelIP);
//-----------------------------------
//       RISOLUTORE IBRIDO
//-----------------------------------
IloSolver solver(env);
//ILOLINCONSTRAINT
IloLinConstraint lc(solver);
//SINCRONIZZAZIONE
lc.setSynchronization(IloLinConstraint::LinConstraintAndSolver);
cout<<"\nSincronizzazione (PRIMA DEL .SOLVE)  =   "
    <<lc.getSynchronization()<<endl;
solver.extract(model);
//----ALGORITMO CPLEX
lc.setRootAlgorithm(IloCplex::Primal);
// lc.setRootAlgorithm(IloCplex::Barrier);
lc.setNodeAlgorithm(IloCplex::Dual);
//----TAGLI DI GOMORY
// lc.setParam(IloCplex::FracCuts, -1);
//lc.setParam(IloCplex::FracCuts, 0);
lc.setParam(IloCplex::FracCuts, 1);
// lc.setParam(IloCplex::FracCuts, 2);
if (contS==1)
  {
    fout9<<"\n******lc.getParam(IloCplex::FracCuts) :  "
 <<lc.getParam(IloCplex::FracCuts)<<endl;
```

```
    cout<<"\n******lc.getParam(IloCplex::FracCuts) : "<<
  lc.getParam(IloCplex::FracCuts)<<endl;
    fout9<<"******lc.getParam(IloCplex::MipEmphasis ): "
 <<lc.getParam(IloCplex::MIPEmphasis)<<endl;
    cout<<"******lc.getParam(IloCplex::MipEmphasis) : "
 <<lc.getParam(IloCplex::MIPEmphasis)<<endl;
    fout9<<"******lc.getParam(IloCplex::TiLim) : "
 <<lc.getParam(IloCplex::TiLim)<<endl;
    cout<<"******lc.getParam(IloCplex::TiLim) : "
 <<lc.getParam(IloCplex::TiLim)<<endl;
  }
contS=2;
solver.startNewSearch(MixedGenerate(env,lc,squareArr,matBinArr));
//---TIMER
IloTimer CPtimer(env);
IloNum tm;
                CPtimer.start();
if( solver.next())
  {
    tm=CPtimer.getTime();
    CPtimer.stop();
    cout<<"tm SI=Htimer.getTime()  = "<<tm<<endl;
    tempi[pos]=tm;
    fails[pos]=solver.getNumberOfFails();
    punti[pos]=solver.getNumberOfChoicePoints();
    mem[pos]=solver.getMemoryUsage();
    succCP[pos]=1;
    pos=pos+1;
    cout<<"\n---pos (si‘):"<<pos<<endl;
    /*
    // STAMPO LA MATRICE SOLUZIONE
    for (i2 = 0; i2 < N; i2++)
    {
    for (j = 0; j < N; j++)
    solver.out() << " " << solver.getValue(square[i2][j]);
    solver.out() << endl;
    }
    solver.out() << endl;
    */
 }// fine if del CP.solve
else
  {
    tm=CPtimer.getTime();
```

```
      CPtimer.stop();
      cout<<"tm NO=Htimer.getTime()  = "<<tm<<endl;
      tempi[pos]=tm;
      solver.out() << "\nNO SOLUTION\n " << endl;
      fails[pos]=solver.getNumberOfFails();
      punti[pos]=solver.getNumberOfChoicePoints();
      mem[pos]=solver.getMemoryUsage();
      succCP[pos]=0;
      pos=pos+1;   cout<<"\n---pos (no) :"<<pos<<endl;
    }
  lc.printInformation();
  solver.printInformation();
  solver.end();
  env.end();
  s2.str("");
} //FINE IF  s2<<a[i]
    }  //FINE SECONDO FOR

        // CALCOLO LE MEDIE E LE INSERISCO NEGLI ARRAY DELLE MEDIE
  IloNum media_t=0, media_f=0, media_p=0, media_m=0 ;
  IloInt NumRuns_ok=0;
  for(km=0;km<NumRuns;km++)
    {
      NumRuns_ok=NumRuns_ok+ succCP[km];
      media_t=media_t + tempi[km];
      media_f=media_f + fails[km];
      media_p=media_p + punti[km];
      media_m= media_m + mem[km];
    }
  percCP_m[pm]= NumRuns_ok * 100/ NumRuns;
  tempi_m[pm]=media_t / NumRuns;
  fails_m[pm]=media_f / NumRuns;
  punti_m[pm]=media_p / NumRuns;
  mem_m[pm]=media_m / NumRuns;
   //APPENDO I RISULTATI AL FILE dei risultati
  fout9<<"------ ARRAY CON I VALORI ORIGINALI -----";
  fout9<<"\n Array dei tempi : \n";
  fout9<<"[";
  for(i=0;i<NumRuns;i++)
    {if (i == NumRuns-1) fout9<<tempi[i]<<"]\n";
    else  fout9<<tempi[i]<<", ";
    }
  fout9<<"\n Array dei successi :\n";
```

```
  fout9<<"[";
  for(i=0;i<NumRuns;i++)
    {if (i == NumRuns-1) fout9<<succCP[i]<<"]\n";
    else  fout9<<succCP[i]<<", ";
    }
  fout9<<"\n Array dei fails :\n";
  fout9<<"[";
  for(i=0;i<NumRuns;i++)
    {if (i == NumRuns-1) fout9<<fails[i]<<"]\n";
    else  fout9<<fails[i]<<", ";
    }
  fout9<<"\n Array dei punti di scelta :\n";
  fout9<<"[";
  for(i=0;i<NumRuns;i++)
    {if (i == NumRuns-1) fout9<<punti[i]<<"]\n";
    else  fout9<<punti[i]<<", ";
    }
  fout9<<"\n Array della memoria usata :\n";
  fout9<<"[";
  for(i=0;i<NumRuns;i++)
    {if (i == NumRuns-1) fout9<<mem[i]<<"]\n";
    else  fout9<<mem[i]<<", ";
    }
  fout9<<"\n------ VALORI MEDI-----------";
  fout9<<"\n Media dei tempi            : ";
  fout9<<tempi_m[pm];
  fout9<<"\n Percentuale dei successi   : ";
  fout9<<percCP_m[pm];
  fout9<<"\n Media dei fails            : ";
  fout9<<fails_m[pm];
  fout9<<"\n Media dei punti di scelta  : ";
  fout9<<punti_m[pm];
  fout9<<"\n Media della memoria usata  : ";
  fout9<<mem_m[pm];
  fout9<<"\n----------------------------\n\n";
  //aggiorno pm e ki del PRIMO FOR
  pm=pm+1;
  cout<<"----pm"<<pm<<endl;
  ki=ki+NumRuns-1;
  envF.end();
}  // fine del ciclo for grande con indice ki
       //FINE DEL PRIMO FOR
       fout9<<"\n\n-------------------------------\n";
```

```
        fout9<<"  ARRAY  CON TUTTI I VALORI MEDI\n";
        fout9<<"---------------------------------\n";
        fout9<<"\n Array delle medie dei tempi : \n";
        fout9<<"[";
        for(i=0;i<n_m;i++)
      { if (i == n_m-1) fout9<<tempi_m[i]<<"]\n";
       else  fout9<<tempi_m[i]<<", ";
      }
        fout9<<"\n Array delle percentuali di successi :\n";
        fout9<<"[";
        for(i=0;i<n_m;i++)
      { if (i == n_m-1) fout9<<percCP_m[i]<<"]\n";
       else  fout9<<percCP_m[i]<<", ";
      }
        fout9<<"\n Array delle medie dei fails :\n";
        fout9<<"[";
        for(i=0;i<n_m;i++)
      { if (i == n_m-1) fout9<<fails_m[i]<<"]\n";
       else  fout9<<fails_m[i]<<", ";
      }
        fout9<<"\n Array delle medie de punti di scelta :\n";
        fout9<<"[";
        for(i=0;i<n_m;i++)
      { if (i == n_m-1) fout9<<punti_m[i]<<"]\n";
       else  fout9<<punti_m[i]<<", ";
      }
        fout9<<"\n Array delle medie della memoria usata :\n";
        fout9<<"[";
        for(i=0;i<n_m;i++)
      { if (i == n_m-1) fout9<<mem_m[i]<<"]\n";
        else  fout9<<mem_m[i]<<", ";
      }
        fout9<<"\n---------------------------------\n";
    }//parentesi di fine del try
  catch (IloException& ex)
    {cout << "Error: " << ex << endl; }
  envCP.end();
  return 0;
} //Fine
```

Bibliografia

[1] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, pages 326–333. Prentice-Hall, 1982.

[2] P. Refalo. Linear Formulation of Constraint Programming Models and Hybrid Solver. *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP'00*, pages 369–383. Springer Verlag, LNCS, 2000.

[3] F. Focacci, A. Lodi and M. Milano. Cost-based domain filtering. *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming, CP'99*, pages 189–203. Springer Verlag, LNCS, 1999.

[4] F. Focacci, A. Lodi and M. Milano. Cutting Planes in Constraint Programming: an hybrid approach. *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP'00*, pages 187–201. Springer Verlag, LNCS, 2000.

[5] ILOG Solver 5.3 User's Manual, ILOG Planner User's Manual, CPLEX 8.1 User's Manual, 2002.

[6] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

[7] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[8] K.R. Apt. *Principles of Constraint Programming*, CUP, 2002.

[9] G. Ottosson. *Integration of Constraint Programming and Integer Programming for Combinatorial Optimization*. PhD Thesis, Uppsala University, 2000.

[10] F. Focacci. *Solving Combinatorial Optimization Problems in Constraint Program-ming.* PhD Thesis, Università di Ferrara, 2000.

[11] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[12] M. Milano, G. Ottosson, P. Refalo and E.E. Thorsteinsson. The Role of Integer Programming Tecniques in Constraint-Programming's Global Constraint. *Annales of Mathematics and Artificial Intelligence, special Issue on Large Scale Combinatorial and Optimization and Constraints*, 2002.

[13] M. Milano. *Integration of OR and AI constraint-based techniques for combinatorial optimization.* Tutorial IJCAI 2001, Seattle.
http://www-lia.deis.unibo.it/Staff/MichelaMilano/tutorial1IJCAI 2001.pdf.

[14] J. Hooker. *Constraint Programming and Integer Programming.* Tutorial CP02.
http://ba.gsia.cmu.edu/jnh/cornell.ppt.

[15] I.J. Lustig and J.F. Puget. Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming. *INTERFACE 31*, pages 29–53, 2001.

[16] A.K. Mackworth. *Consistency in Networks of Relations.* Artificial Intelligence 8(1): 99–118, 1977.

[17] A.K. Mackworth and E. Freuder. *The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems.* Artificial Intelligence 25: 65–74, 1985.

[18] R. Mackworth and T.C. Henderson. *Arc and Path Consistency Revisited.* Artificial Intelligence 28: 225–233, 1986.

[19] C.C. Han and C.H. Lee. *Comments on Mohr and Henderson's Path Consistency Algorithm.* Artificial Intelligence 36: 125–130, 1988.

[20] Y. Chen. Improving Han and Lee's Path Consistency Algorithm. *Proceedings of the 3rd International Conference on Tools for AI*, pages 346–350. IEEE Computer Society Press, 1991.

[21] J.C. Regin. A filtering algorithm for constraints of difference in csps. *Proceedings of AAAI-94*, pages 362–367, 1994.

[22] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathl. Comput. Modelling*, 20(12), 1994.

[23] P. van Hentenryck. *Constraint satisfaction in Logic Programming.* MIT Press, 1989.

[24] R. Rodosek, M. Wallace and M.T. Hajian. A new approach to integrate mixed integer programming with CLP. *Proceedings of the CP'96 Workshop on Constraint Programming Applications*, 1996.

[25] H.P. Williams. *Model building in Mathematical Programming.* Wiley, 1999.

[26] G. Mitra, C. Lucas, S. Moody and E. Hadjiconstantinou. Tools for reformulating logical forms into zero-one mixed integer programs. *European Journal of Operational Research*, (72):262–276, 1994.

[27] P. van Hentenryck, H. Simonis and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3):113–159, 1992.

[28] P. Refalo. Tight Cooperation and its applications in piecewise linear optimization. *Proceedings of the 5th International Conference CP 99*, pages 369–383. Springer Verlag, LNCS, 1999.

[29] B. De Backer and H. Beringer. Cooperative solvers and global constraints: The case of linear arithmetic constraints. *Proceedings of the Post Conference Workshop on Constraints, Databases and Logic Programming, IPPS'95*, 1995.

[30] M. Rueher and C. Solnon. Concurrent cooperating solvers over the reals. *Reliable Computing*, 3(3):325–333, 1997.

[31] E. Balas. Disjunctive programming and a hierarchy of relaxations for discrete optimization problems. *SIAM Journal Alg. Disc. Meth.*, 6(3):466–486, 1985.

[32] R. Jeroslow. Logic based decision support: Mixed integer model formulation. *Annals of Discrete Mathematics*, (40), 1989.

[33] L.A. Wolsey. Integer Programming. *Annals of Discrete Mathematics.* Wiley, 1998.

[34] R.E. Gomory. Outline of an Algorithm for Integer Solution to Linear Programs, *Bulletin Amer. Math. Soc.* 64, 5, 1958.

[35] E. Freuder. Complexity of K-Tree Structured Constraint Satisfaction Problems. *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 4–9, 1990.

[36] E. Freuder. Partial Constraint Satisfaction. *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 278–283, 1989.

[37] IC-Parc. *ECL^iPS^e User Manual Release 5.3*, 2001.

[38] M. Padberg and G. Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut, *Oper. Res. Lett.* 6, 1-7, 1987.

[39] C. Gomes and B. Selman. Problem Structure in the Presence of Perturbations. *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 221–227. AAAI Press, 1997.

[40] S.R. Kumar, A. Russel and R. Sundaram. Approximating latin square extensions. *Algorithmica*, 24:128–138, 1999.

[41] ILOG Hybrid Cooperating Optimizers 1.3.1 User's Guide and Reference, 2002.

[42] C. Gomes and D. Shmoys. Completing Quasigroups or Latin Square: A structured Graph Coloring Problem. *Proc. Computational Symposium on Graph Coloring and Generalizations*, 2002.