Alma Mater Studiorum - Università di Bologna

FACOLTÀ DI INGEGNERIA

Dipartimento di Elettronica, Informatica e Sistemistica

Corso di Laurea in Ingegneria Elettronica

DETERMINAZIONE DEL VINCITORE NELLE ASTE COMBINATORIE: UN APPROCCIO A VINCOLI

Tesi di Laurea di:
ALESSIO GUERRI

Relatore:
Chiar.ma Prof.ssa PAOLA MELLO
Titolare del corso di **Intelligenza Artificiale**

Correlatori:
Chiar.ma Prof.ssa MICHELA MILANO
Dott. Ing. ANDREA ROLI

Anno Accademico 2001-2002

PAROLE CHIAVE

Aste Combinatorie
Limited Discrepancy Search (LDS)
Programmazione a Vincoli
Commercio Elettronico
Intelligenza Artificiale

Desidero ricordare tutti coloro che hanno contribuito alla realizzazione di questa tesi.

Innanzitutto, un ringraziamento particolare a Michela Milano, oltre che per il supporto, l'assistenza e i chiarimenti relativi alla tesi, per l'eccezionale disponibilità dimostrata e per i consigli di carattere organizzativo che mi hanno consentito di arrivare al giorno della laurea con relativa tranquillità.

Ringrazio i numeri 1, 2 e 3 del lab2.

Fabio Bucciarelli, per l'aiuto tecnico e per la pazienza dimostrata nel rimediare ai danni da me causati alle macchine.

Gianluca Tonti e Dario Bottazzi per l'aiuto tecnico e per la piacevole compagnia nei momenti di svago.

Sono inoltre loro grato per avermi eletto numero 4 del laboratorio.

Ringrazio anche chi, facendomi trascorrere dei momenti indimenticabili, ha comunque rubato tempo al mio lavoro.

Mai furto fu più gradito.

Dulcis in fundo, last but not least, non riuscirò mai a ringraziare abbastanza il mitico Q.d.V.d.P.+E., impareggiabile compagnia di tutti questi anni bolognesi, per quello che abbiamo fatto e per quello che, spero, ancora faremo insieme. Senza le feste e le serate trascorse in compagnia non avrei avuto la "benzina" necessaria per arrivare fino in fondo.

A Dario e Loredana,
a Federica,
a Violante e Natalina

Indice

Indice vi

Introduzione viii

1 PROBLEMI DI SODDISFACIMENTO DI VINCOLI 1

1.1 Introduzione 3

1.2 Problemi di Soddisfacimento di Vincoli e Problemi di Ottimizzazione 3

1.3 Albero decisionale e strategie di esplorazione 4

1.3.1 Tecniche di Consistenza e Algoritmi di Propagazione 6

1.4 Funzioni Euristiche per l'ordinamento delle variabili e la selezione dei valori .. 9

2 ALGORITMI DI RICERCA INCOMPLETI 13

2.1 Introduzione 15

2.2 Limited Discrepancy Search 15

2.3 Improved Limited Discrepancy Search 17

2.3.1 Un esempio di applicazione degli algoritmi: il problema della partizione di numeri 19

2.4 Depth-bounded Discrepancy Search 21

2.4.1 Esempio: ricerca euristica con probabilità costante di errore nella scelta 23

2.4.2 Esempio: il problema della partizione dei numeri 27

3 PROGRAMMAZIONE LOGICA A VINCOLI 31

3.1 Introduzione 33

3.2 Limitazioni dei linguaggi di programmazione logica 33

3.3 CLP come metodo per superare i limiti della LP 34

3.3.1 Esempio di funzionamento di una macchina CLP 35

3.4 Vincoli Simbolici 37

3.4.1 Vincoli disgiuntivi 38

4 ASTE COMBINATORIE SU INSIEMI COORDINATI DI ATTIVITÀ 43

4.1 Introduzione 45

4.2 Le aste combinatorie 45

4.3 Linguaggi di offerta 47

4.4 Valutazione di offerte su un insieme coordinato di attività: un esempio 49

5 LIBRERIA ILOG SOLVER 53

5.1 Introduzione 55

5.2 Rappresentazione di un problema 55

5.3 Estrazione di un modello 58

5.4 Risoluzione di un problema estratto 59
5.4.1 Le funzioni IloInstantiate e IloGenerate 60
5.4.2 La funzione IloApply 61

5.5 Definizione di nuove classi di vincoli 62

6 PROGETTI E ALGORITMI PER LA DETERMINAZIONE DEL VINCITORE IN UN'ASTA COMBINATORIA 65

6.1 Introduzione 67

6.2 Generazione di problemi di aste combinatorie 67

6.3 Algoritmi di risoluzione utilizzati da MAGNET 68
6.3.1 Integer Programming 68
6.3.2 Simulated Annealing 70

6.4 Determinazione del vincitore di un'asta combinatoria tramite la programmazione a vincoli 72
6.4.1 Generazione dei problemi 72
6.4.2 Risoluzione dei problemi: variabili utilizzate 75
6.4.3 Risoluzione dei problemi: vincoli e metodi di propagazione dei vincoli 78
6.4.4 Risoluzione dei problemi: ottimizzazione e strategie per la selezione di variabili e valori 84
6.4.5 Risoluzione dei problemi: esplorazione dello spazio di ricerca 87

7 RISULTATI SPERIMENTALI 91

7.1 Introduzione 93

7.2 Risultati ottenuti risolvendo problemi generati da MAGNET 93

7.3 Risultati ottenuti risolvendo problemi generati da CATS 102

7.4 Considerazioni conclusive sui risultati 111

Conclusioni 115

Bibliografia 117

Introduzione

La rapida evoluzione del commercio elettronico permette alle società di ampliare il proprio bacino d'utenza e le proprie potenzialità, eliminando le tipiche barriere che il commercio classico impone. In questo scenario, le aste combinatorie si inseriscono come metodo per superare le limitazioni intrinseche delle aste classiche, in cui è possibile piazzare un bene alla volta, decidendo il vincitore soltanto in base al prezzo offerto.

I beni che è possibile vendere tramite un'asta combinatoria possono essere di varia natura. In questa tesi saranno prese in considerazione aste combinatorie su insiemi coordinati di attività, attività, cioè, che devono essere svolte in accordo con determinati vincoli temporali, sia per quanto riguarda l'esecuzione dell'attività stessa, sia rispettando delle relazioni di precedenza tra due o più di esse.

I vantaggi ottenuti utilizzando tali aste sono indubbi, sia per chi vende sia per chi compra; tuttavia, l'introduzione di vincoli temporali e la possibilità di fare offerte su più oggetti contemporaneamente, rende estremamente più complicata la risoluzione del problema della determinazione del vincitore, intesa come ricerca della combinazione di offerte che riesce ad eseguire tutte le attività al minor prezzo possibile.

Il lavoro svolto nella seguente tesi intende risolvere tale problema tramite un approccio a vincoli, utilizzando algoritmi di ricerca incompleti, per ottenere risultati migliori sia in termini temporali sia in termini di ottimalità rispetto a quelli raggiungibili con altri sistemi esistenti.

Nel primo capitolo verranno introdotte le caratteristiche delle tecniche di soddisfacimento di vincoli, descrivendo gli algoritmi di esplorazione dello spazio di ricerca e i metodi utilizzati per guidare tale ricerca verso strade che più probabilmente portano ad una buona soluzione.

Nel secondo capitolo saranno introdotti i migliori algoritmi di ricerca incompleti reperibili in letteratura. Un algoritmo incompleto è un algoritmo che tenta di risolvere un problema eliminando alcune porzioni dello spazio di ricerca che, verosimilmente, non possono contenere una soluzione.

Nel terzo capitolo sarà introdotta la programmazione logica a vincoli sottolineando come, tramite questo paradigma, vengono superate le limitazioni tipiche della programmazione logica.

Nel quarto capitolo verranno descritte più formalmente le caratteristiche di un'asta combinatoria su insiemi coordinati di attività, specificando, in particolare, tutti i vincoli che dovranno essere soddisfatti da una qualsiasi soluzione.

Nel quinto capitolo sarà introdotta la libreria ILOG Solver, un insieme di metodi e funzioni utili per risolvere problemi di programmazione, allocazione, ottimizzazione, gestione delle risorse e simili, sfruttando la programmazione a vincoli. In particolare l'attenzione verrà focalizzata sulle funzioni effettivamente utilizzate nella tesi.

Nel sesto capitolo sarà descritto nel dettaglio il sistema software progettato e i metodi di generazione di problemi di aste combinatorie. Verrà introdotto MAGNET, un sistema per la generazione e risoluzione dei problemi che la presente tesi intende risolvere.

Nel settimo ed ultimo capitolo saranno presentati e discussi i risultati ottenuti dal sistema progettato. I risultati saranno inoltre paragonati con quelli ottenuti da MAGNET, al fine di verificare se l'approccio a vincoli possa rappresentare una valida alternativa ai metodi di risoluzione esistenti.

CAPITOLO 1

PROBLEMI DI SODDISFACIMENTO DI VINCOLI

1.1 Introduzione

In questo capitolo saranno introdotti i Problemi di Soddisfacimento di Vincoli (Constraint Satisfaction Problems, CSP) e i Problemi di Ottimizzazione Combinatoria (Combinatorial Optimization Problems, COP), che rappresentano i tipici problemi analizzati e risolti dai linguaggi di Programmazione Logica a Vincoli (Constraint Logic Programming, CLP).

1.2 Problemi di Soddisfacimento di Vincoli e Problemi di Ottimizzazione

Un CSP può essere definito su un insieme finito di variabili $(X_1,X_2,...,X_n)$ i cui valori appartengono a domini di definizione $(D_1,D_2,...,D_n)$ che, per questa trattazione, verranno supposti finiti, e su un insieme di vincoli. Un vincolo $c(X_{i1},X_{i2},...,X_{ik})$ tra k variabili è un sottoinsieme del prodotto cartesiano $D_{i1} \times D_{i2} \times ... \times D_{ik}$ che specifica i valori delle variabili tra loro compatibili. Tale sottoinsieme non deve essere esplicitamente definito, ma è rappresentato in termini di relazioni.

Una soluzione ad un CSP prevede l'assegnamento di un valore ad ogni variabile, in modo da soddisfare tutti i vincoli imposti.

Un COP è un CSP cui viene aggiunto un obiettivo. Esso è quindi più formalmente descrivibile come un CSP il cui scopo non sia solo quello di trovare una soluzione ammissibile, bensì la soluzione ottima secondo un certo criterio di valutazione. Dato un algoritmo generale in grado di risolvere qualsiasi CSP, è possibile utilizzarlo per risolvere qualsiasi COP semplicemente aggiungendo all'insieme di variabili, domini e vincoli del CSP, un'ulteriore variabile che rappresenti la funzione obiettivo. Ogni volta che viene trovata una soluzione, il risolutore impone il vincolo che ogni ulteriore soluzione abbia il valore della funzione obiettivo migliore. Quando il procedimento porta ad un fallimento, l'ultima soluzione trovata è senz'altro quella ottima.

La maggior parte di questi problemi sono NP-difficili, sono cioè problemi per i quali non è ancora stata trovata, e probabilmente non esiste, un algoritmo in grado di trovare la soluzione in un tempo polinomiale nella dimensione del problema.

1.3 Albero decisionale e strategie di esplorazione

Qualunque tecnica di risoluzione di problemi NP-difficili fa uso, almeno concettualmente, di un albero decisionale. Per un CSP, l'albero decisionale si può ottenere facendo corrispondere ad ogni livello l'assegnamento di una variabile, secondo un ordine prestabilito, e ad ogni nodo la scelta del valore da assegnare alla variabile corrispondente al livello in cui in nodo stesso si trova. È quindi evidente che ogni foglia dell'albero rappresenta una diversa combinazione di valori assegnati alle variabili. Le foglie associate ad assegnamenti compatibili con tutti i vincoli del problema sono soluzioni del problema stesso. La ricerca di una soluzione è quindi equivalente all'esplorazione dell'albero decisionale associato al problema, al fine di trovare una foglia con un assegnamento compatibile con i vincoli imposti.

Una metodologia di esplorazione dell'albero può essere quella di generare un assegnamento e verificare a posteriori se sia compatibile con i vincoli dati. In caso negativo viene continuata la ricerca in **backtracking**, risalendo l'albero in maniera cronologica fino ad arrivare al primo nodo che contenga strade ancora inesplorate, e ripetendo l'algoritmo su tali strade. Tale algoritmo è chiamato **Generate and Test**. Un altro algoritmo, leggermente migliore di questo, è lo **Standard Backtracking**, in cui ad ogni assegnamento di una variabile viene controllata la compatibilità di tutte le variabili correntemente assegnate con i vincoli, evitando in caso di incompatibilità di percorrere ulteriormente una strada che porterà sicuramente ad un fallimento.

Notando però che, in un problema con v variabili, ognuna delle quali abbia un dominio di cardinalità d, esistono d^v foglie, cioè d^v possibili

assegnamenti, si capisce che utilizzare i vincoli solo a posteriori sia una strada impraticabile per i CSP normalmente analizzati. Infatti per un semplice problema con 5 variabili e domini di cardinalità 10, esistono 10 milioni di permutazioni dei valori, cioè 10 milioni di foglie nell'albero decisionale. Oltretutto, essendo questo valore esponenziale nel numero delle variabili, già un problema con il doppio di variabili porta ad avere 10 miliardi di possibili assegnamenti, con un incremento del tempo di computazione di un fattore 1000.

È però possibile utilizzare i vincoli in maniera più intelligente riducendo a priori l'albero di ricerca. Per esempio, nella successiva figura 1.1 è rappresentato l'albero decisionale di un semplice problema di 3 variabili con domini (0,1) e vincoli $X \neq Y, X \neq Z, Z \geq X$.

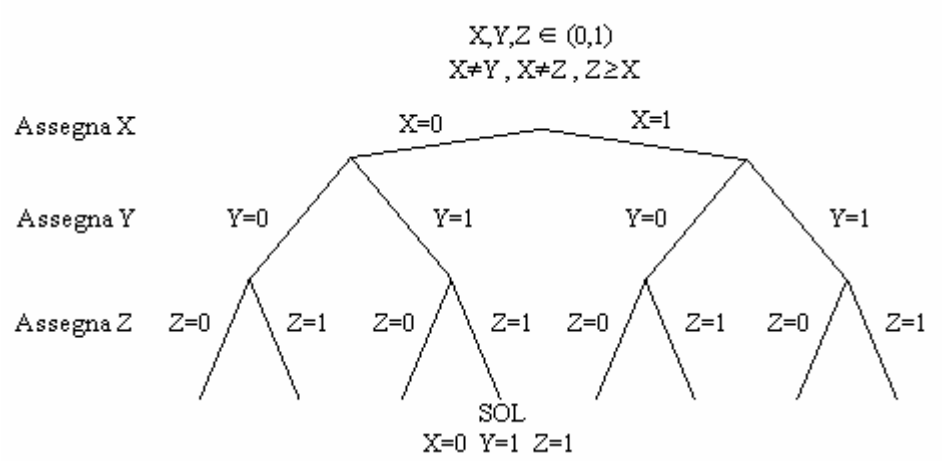

Fig. 1.1: *Esempio di albero decisionale per un problema di tre variabili con domini di cardinalità 2*

Il Generate and Test risale 4 volte l'albero in backtracking prima di trovare la soluzione (0,1,1), mentre lo Standard Backtracking soltanto 2 volte. Utilizzando però i vincoli subito dopo l'assegnamento del valore 0 ad X, si può eliminare a priori il valore 0 sia da Y sia da Z, in quanto incompatibili rispettivamente con il primo e con il secondo vincolo, ed arrivare alla soluzione senza eseguire nessun backtracking.

1.3.1 Tecniche di Consistenza e Algoritmi di Propagazione

Le Tecniche di Consistenza e gli Algoritmi di Propagazione sono metodi che cercano, utilizzando i vincoli in maniera attiva, di prevenire a priori i fallimenti anziché recuperare quelli già avvenuti. Può, infatti, accadere che un assegnamento sbagliato nei primi livelli dell'albero produca come effetto il fallimento della ricerca in quel ramo molto più tardi, quindi molto in profondità nell'albero decisionale. Un algoritmo in grado di riconoscere subito questo tipo di errori evita la generazione inutile di un elevato numero di sottoalberi. Entrambi i metodi si occupano di utilizzare i vincoli il prima possibile per evitare inutili ricerche e, quindi, inutile spreco di tempo e di risorse.

Le Tecniche di Consistenza sono meccanismi per propagare i vincoli prima di cominciare la ricerca, e quindi derivare un problema più semplice, ma equivalente a quello originale completo. Viceversa i secondi si basano sulla propagazione dei vincoli per eliminare, durante la ricerca, quei valori dai domini delle variabili, e quindi quelle porzioni dell'albero decisionale, che porterebbero, associati ai valori già assegnati, ad un sicuro fallimento. Tipicamente in un CSP vengono applicate prima le Tecniche di Consistenza e successivamente gli Algoritmi di Propagazione.

Una volta che non sia più possibile applicare tali metodi, o si è giunti ad una soluzione o è necessario scegliere un valore da assegnare ad una delle variabili ancora libere.

La figura 1.2 esemplifica la differenza tra utilizzo dei vincoli a priori e a posteriori. Utilizzando i vincoli a priori viene evitata la fase di test, riducendo quindi drasticamente il numero di backtracking. Infatti, ogni assegnamento generato dal metodo Generate and Test, ha, perlomeno a priori, la stessa probabilità di essere una soluzione di tutti gli altri assegnamenti; viceversa, l'eliminazione di valori incompatibili dai domini grazie alla propagazione dei vincoli, comporta anche un aumento

della probabilità di successo degli assegnamenti generati utilizzando solo i valori rimanenti.

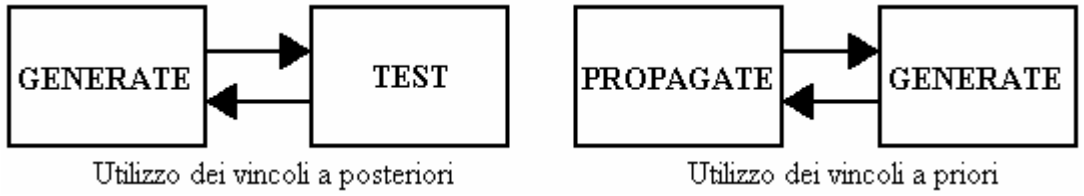

Fig. 1.2: *Utilizzo dei vincoli nell'esplorazione dell'albero decisionale*

Propagare i vincoli è in ogni caso un'operazione complessa che richiede un certo tempo per essere eseguita. Occorre quindi utilizzare l'algoritmo che, nel problema in questione, riesca a fornire il miglior trade-off tra il costo della strategia e i risultati ottenuti, in termini di riduzione di domini.

Tra gli algoritmi di propagazione solitamente implementati troviamo il **Forward Checking** e il **Look Ahead.** Il primo, dopo ogni assegnamento, elimina i valori incompatibili con quello appena istanziato dai domini delle variabili non ancora istanziate. Se ad un certo punto della computazione uno o più domini risultassero vuoti, il meccanismo fallisce subito e prova altri valori per le variabili già istanziate, evitando così di scendere in un ramo dell'albero decisionale che porterebbe sicuramente al fallimento e, di conseguenza, al backtracking. L'algoritmo Look Ahead prevede che ad ogni istanziazione venga eseguito il Forward Checking, e venga inoltre sviluppato il Look Ahead (sguardo in avanti) che controlla l'esistenza, nei domini delle variabili ancora libere, di valori compatibili con i vincoli contenenti solo variabili non istanziate. In altre parole, si controlla se, a causa della riduzione effettuata dal Forward Checking, i valori rimasti nei domini delle variabili libere possano ancora portare, scendendo nell'albero, ad una soluzione, o se siano viceversa destinati ad un sicuro fallimento.

Contrariamente agli algoritmi di propagazione, che propagano i vincoli in seguito ad istanziazioni delle variabili coinvolte nel problema,

le tecniche di consistenza riducono il problema originale eliminando dai domini delle variabili i valori che non possono comparire in alcuna soluzione.

Tutte le tecniche di consistenza sono basate su una rappresentazione del problema come un grafo di vincoli. Gli archi possono essere orientati o no: per esempio il vincolo > è rappresentato da un arco orientato, mentre il vincolo ≠ da uno semplice (non orientato o doppiamente orientato).

Per ogni CSP, esiste un grafo in cui i nodi rappresentano le variabili, e gli archi i vincoli tra le variabili costituenti i nodi che l'arco unisce. I vincoli unari sono rappresentati da archi che iniziano e terminano sullo stesso nodo X_i, mentre i vincoli binari collegano 2 nodi X_i e X_j.

Fig. 1.3: *Rappresentazione di vincoli unari e binari in un grafo di un CSP*

Esistono vari algoritmi che realizzano gradi diversi di consistenza.

- **Node-Consistency:** consistenza di grado 1**.** Un nodo di un grafo è consistente se, per ogni valore nel dominio, i vincoli unari associati al nodo sono soddisfatti. Tutti i valori che non soddisfano i vincoli possono sicuramente essere eliminati dal dominio. Un grafo viene definito node-consistent se tutti i suoi nodi sono consistenti. Se, per esempio, in un CSP avessimo una variabile X con dominio iniziale [0..10], e il vincolo X>2, per rendere il grafo node-consistent è necessario eliminare i valori [0..2] dal dominio di X.
- **Arc-Consistency:** consistenza di grado 2. La consistenza di grado 2 si ottiene partendo da un grafo node-consistent verificando la consistenza di tutti gli archi. Un arco è consistente se per ogni valore nel dominio di una variabile esiste almeno un valore nel dominio dell'altra variabile che soddisfi i vincoli associati all'arco.

Ovviamente la rimozione di alcuni valori a causa dell'arc-consistency rende necessarie ulteriori verifiche che coinvolgano i vincoli unari sulle due variabili in gioco. Questo procedimento iterativo deve essere ripetuto fino a che la rete non converge ad uno stato stabile arc-consistent.

- **K-Consistency:** consistenza di grado k. Dato un grafo consistente fino al grado k-1, la consistenza di grado k si ottiene scegliendo ogni possibile (k-1)-pla di variabili, consistente per definizione con i vincoli imposti, e cercando un valore per ogni ulteriore variabile del problema che soddisfi i vincoli fra tutte le k variabili prese in considerazione.

Si dimostra che, se un grafo contenente n variabili è k-consistent con k<n, allora per trovare una soluzione è sufficiente una ricerca nello spazio restante. In un grafo di n variabili n-consistent, la soluzione può quindi essere trovata senza eseguire alcuna ricerca, basta avere l'accortezza di scegliere nel giusto ordine le variabili da assegnare. Tuttavia, rendere un grafo n-consistent ha complessità esponenziale in n. Occorre quindi, anche in questo caso, ricercare il valore di k che fornisca il miglior trade-off tra la complessità del processo iterativo di consistenza e la riduzione effettuata sullo spazio di ricerca.

1.4 Funzioni Euristiche per l'ordinamento delle variabili e la selezione dei valori

Una **Funzione Euristica**, applicata ad un nodo n, valuta la distanza, quindi lo sforzo computazionale, che separa il nodo n dalla soluzione. Ordinando le strade aperte dalla più "promettente" alla meno "promettente", cioè da quella che ha una funzione euristica migliore a quella peggiore, è possibile raggiungere una buona soluzione in tempi ragionevoli anche per i problemi più complessi. Ovviamente, affinché un'euristica porti ad una soluzione, occorre che essa sia buona, sia in

grado cioè di ordinare il più esattamente possibile le strade, quindi i rami dell'albero di ricerca, secondo la probabilità che possano portare ad una soluzione, o alla soluzione ottima nel caso di COP.

Le euristiche si possono suddividere in **statiche** e **dinamiche**.

- **Euristiche statiche:** determinano l'ordine in cui l'albero verrà esplorato prima di iniziare la ricerca; tale ordine rimane invariato durante tutto il problema.
- **Euristiche dinamiche:** scelgono la successiva selezione da effettuare ogni volta che una nuova selezione viene richiesta, in base alla nuova situazione che si è venuta a creare nell'esplorazione dell'albero.

Le euristiche dinamiche sono potenzialmente migliori, poiché consentono di risalire in backtracking l'albero un numero minore di volte, ma la determinazione dell'euristica perfetta, vale a dire quella che ad ogni passo sappia scegliere la strada migliore, è un problema che ha, in genere, la stessa complessità dl problema originale. Occorre quindi trovare un giusto compromesso tra attendibilità e complessità della funzione euristica.

Tra i principi di euristica comunemente accettati troviamo **il First-Fail Principle** (FFP), il **Least Constraining Principle** (LCP) e il **Most Constraining Principle** (MCP).

Il FFP consiglia di risolvere prima i sottoproblemi più difficili da risolvere, che hanno quindi una probabilità più elevata di portare a fallimenti.

Il LCP consiglia di fare le scelte meno vincolanti per il sottoproblema rimanente. Il MCP consiglia invece di effettuare le scelte più vincolanti.

In un CSP, le funzioni euristiche si applicano ai criteri di scelta dell'ordinamento delle variabili e all'ordine di selezione dei valori.

Le **euristiche per la selezione della variabile** determinano quale debba essere la successiva variabile da istanziare. Il FFP sceglie la variabile con il dominio di cardinalità minore, mentre il MCP quella legata a più vincoli. Entrambe queste euristiche decidono di istanziare prima le variabili più difficili da assegnare.

Analogamente le **euristiche per la selezione del valore** determinano quale valore assegnare alla variabile selezionata. In genere per questo genere di selezione viene seguito il LCP, viene cioè scelto prima il valore che si ritiene abbia più probabilità di successo.

CAPITOLO 2

ALGORITMI DI RICERCA INCOMPLETI

2.1 Introduzione

Utilizzare l'algoritmo di ricerca più appropriato al problema da risolvere è di fondamentale importanza per ottenere la soluzione in un tempo accettabile. Tuttavia, può accadere che, pur applicando tutte le strategie descritte in precedenza, lo spazio di ricerca associato al problema resti comunque troppo ampio per poter essere esplorato esaustivamente.

Gli **algoritmi incompleti** sono algoritmi che limitano l'azione di ricerca alle regioni dell'albero decisionale che più probabilmente contengono una buona soluzione. Ovviamente la scelta di tali regioni può essere dettata solo da una funzione euristica, che assume quindi un'importanza fondamentale per la riuscita della ricerca.

In questo capitolo saranno descritti alcuni tra i più efficienti algoritmi di ricerca incompleti.

2.2 Limited Discrepancy Search

Tra gli algoritmi incompleti descritti in letteratura, uno dei migliori è senza dubbio la Limited Discrepancy Search (LDS), definito nel '95 da Ginsberg e Harvey[1].

Sebbene per molti problemi una buona euristica riesca a portare direttamente alla soluzione, questo non è necessariamente sempre vero. La LDS indirizza il problema verso un'altra strada quando l'euristica fallisce. L'idea che ne sta alla base è che un'euristica non ottimale potrebbe comunque aver successo, se non fosse per un piccolo numero di "scelte sbagliate" suggerire durante la ricerca. In un albero binario di profondità d, esistono soltanto d strade in cui l'euristica possa consigliare una scelta sbagliata, e soltanto $d\frac{d-1}{2}$ in cui ne possa consigliare 2. Un piccolo numero di scelte errate può essere superato

cercando sistematicamente tutte le strade che differiscono da quella consigliata dall'euristica in un piccolo numero di punti di scelta, detti **discrepanze**. LDS è un algoritmo che esplora in backtracking i nodi dell'albero in ordine crescente di tali discrepanze; numerando con k le iterazioni dell'algoritmo eseguite, alla k-esima ripetizione vengono esplorate tutte le strade con al più k discrepanze, garantendo che se una soluzione esiste viene trovata, mentre, se il problema non ha soluzione, l'algoritmo termina riportando il fallimento, dopo aver eseguito una ricerca completa sull'albero decisionale. La figura 1.4 confronta la ricerca eseguita con la LDS con quella eseguita da un algoritmo completo, nella fattispecie la DFS, che percorre l'albero esplorando tutte le foglie da sinistra verso destra, restituendo poi la foglia che ha fornito la soluzione migliore. I numeri sotto l'albero mostrano l'ordine con cui sono visitate le foglie, mentre i numeri accanto ai rami posti più in basso rappresentano la discrepanza associata alla strada che porta alla foglia.

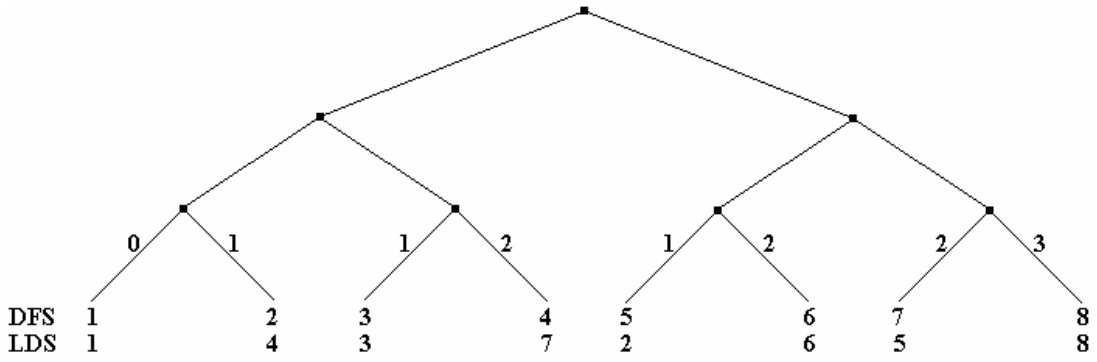

Fig. 2.1: *Ordine in cui sono visitate le foglie di un albero binario completo dagli algoritmi DFS e LDS.*

Dalla figura si può notare la sequenza di esplorazione effettuata dall'algoritmo LDS. Inizialmente viene visitata l'unica foglia con discrepanza 0, di seguito tutte quelle con discrepanza 1, iniziando da quelle che hanno la discrepanza nei rami più alti dell'albero, e così via per discrepanze sempre maggiori.

Generalizzando al caso di albero non binario, una volta che l'euristica abbia ordinato tutti i figli di un nodo, la discrepanza di un figlio è il numero d'ordine tra i suoi fratelli, considerando ovviamente il nodo migliore di ordine 0, quindi con discrepanza 0.

Esperimenti effettuati su vari problemi hanno mostrato che la LDS riesce, nel 95% dei casi, a fornire la soluzione ottima nell'1% del tempo impiegato da un algoritmo completo, tipicamente la DFS.

Per com'è implementato, operando in backtracking l'algoritmo LDS espande alcuni nodi già aperti in una precedente esplorazione. In un albero binario completo di profondità d, il numero di nodi espansi con discrepanza limitata a x è al massimo d^{x+1}. Se d è grande, il costo di ogni singola ripetizione supera la somma dei costi di tutte le iterazioni precedenti. Inoltre la LDS non fa distinzione tra discrepanze presenti in alto nell'albero di ricerca piuttosto che più in profondità, mentre è più probabile che un'euristica possa sbagliare nei primi livelli dell'albero, quando tutte le strade sono ancora aperte.

Sono stati proposti 2 algoritmi che riprendono e migliorano la LDS risolvendo le cause di inefficienza sopra descritte.

2.3 Improved Limited Discrepancy Search

L'Improved LDS (ILDS) è stato proposto, sempre nel '95, da Korf [2].

Uno degli inconvenienti della LDS è che esplora alcune foglie più di una volta; ciò è ovviamente inutile e porta ad un incremento del tempo di computazione. L'ILDS modifica la LDS permettendo di generare, ad ogni iterazione k, soltanto i percorsi con esattamente k discrepanze. Questo può essere ottenuto conoscendo ad ogni passo la profondità della rimanente parte da esplorare, e qualora questa fosse minore o uguale al numero di discrepanze ancora possibili, esplorando solo i rami di destra.

I risultati migliori si ottengono esplorando un albero di profondità uniforme, nel qual caso la complessità asintotica, supponendo l'albero

binario, si riduce dal valore $O\left(\frac{d+2}{2}2^d\right)$, tipico della LDS, al valore $O(2^d)$. Il miglioramento è molto minore in un albero non uniforme, per esempio in un albero cui siano state applicate tecniche di consistenza per fare il pruning a priori delle parti che non possono contenere soluzioni.

Va tuttavia notato che, sebbene l'ILDS esplori molti meno nodi rispetto all'algoritmo originale, alcuni nodi interni vengono comunque esplorati più volte, specialmente quando, non essendo l'albero uniforme, non è possibile sapere a priori quale sia la profondità della parte di albero rimanente.

Compariamo le prestazioni delle due versioni di LDS, per determinare quanto l'ILDS migliori le prestazioni dell'algoritmo originale. Assumendo di eseguire la ricerca su un albero binario completo di profondità d, poiché ad ogni iterazione k l'ILDS genera sono i percorsi con esattamente k discrepanze, ogni foglia è visitata solo una volta. Quindi una foglia il cui percorso contiene k rami destri, viene analizzata alla k-esima iterazione. Dato che l'albero in considerazione ha 2^d foglie, l'ILDS visita 2^d foglie; questo valore rappresenta anche la complessità asintotica dell'algoritmo. È stato infatti dimostrato che l'esplorazione di nodi interni non incide sulla complessità asintotica[2].

Diversamente, per calcolare il numero di foglie generate dalla LDS, occorre contare i percorsi che contengono al massimo k discrepanze. C'è un percorso con 0 discrepanze, quello più a sinistra; ci sono d percorsi con 1 discrepanza, dato che la scelta della strada a destra può avvenire a qualsiasi livello dell'albero; in generale, il numero di percorsi con esattamente k discrepanze è $\binom{d}{k}$, cioè il numero di possibili scelte di k diversi oggetti su un totale di d. In totale occorrono d+1 iterazioni per esplorare completamente un albero di profondità d, poiché il numero di discrepanze può variare tra 0 e d. Il percorso con 0 discrepanze è visitato d+1 volte, una per ogni iterazione; i d percorsi con 1 discrepanza sono

visitati d volte, ad ogni iterazione tranne la prima; in generale, detto x il numero di foglie visitate dalla LDS, si ha

$$x = (d+1)\binom{d}{0} + d\binom{d}{1} + (d-1)\binom{d}{2} + \cdots + 2\binom{d}{d-1} + 1\binom{d}{d}$$

Sviluppando i calcoli si ottiene $x = \frac{d+2}{2}2^d$, che è il valore, già incontrato, della complessità asintotica della LDS.

Le prestazioni dell'ILDS sono quindi migliori di quelle della LDS di un fattore $\frac{d+2}{2}$. Purtroppo, nei casi pratici, questo fattore è destinato ad essere ridotto, principalmente per due motivi. Il primo è che la LDS è pensata per essere eseguita su alberi decisionali molto ampi, nel qual caso possono essere eseguite solo poche iterazioni, riducendo l'impatto che l'inefficienza sopra descritta ha sulla complessità asintotica. Secondariamente, nella maggior parte dei problemi, viene eseguito il pruning, che taglia alcune strade prima del raggiungimento della profondità massima, rendendo quindi l'albero non uniforme; le foglie con profondità minore di quella massima sono generate più volte dall'ILDS. La differenza tra le due complessità asintotiche risulta, di conseguenza, minore del valore teorico trovato.

2.3.1 Un esempio di applicazione degli algoritmi: il problema della partizione di numeri

In [2] vengono paragonati i risultati ottenuti dagli algoritmi DFS, LDS e ILDS applicati al problema della partizione di numeri. Dato un insieme di interi, il problema della partizione dei numeri consiste nel dividere l'insieme in due sottoinsiemi, in modo che le somme dei numeri dei due sottoinsiemi siano le più vicine possibili. Per esempio, dato l'insieme (4,5,6,7,8), la partizione (4,5,6) e (7,8), essendo la somma di entrambi i sottoinsiemi 15, oltre ad essere la soluzione migliore, è anche quella ottima. Il problema della partizione dei numeri è NP-completo[3].

L'algoritmo risolutivo utilizzato è basato su un'euristica di complessità polinomiale nel tempo, proposta da Karmarkar e Karp in [4].

L'esperimento è stato effettuato scegliendo i numeri a caso tra 0 e 10 miliardi, in modo da avere elementi con 10 cifre di precisione. Gli insiemi sono stati generati con cardinalità crescente, da 5 a 100, con incrementi di 5. Per ogni valore della cardinalità sono state effettuate circa 100 differenti prove. La figura 2.2 mostra i nodi mediamente generati prima di trovare la soluzione ottima. L'asse orizzontale rappresenta la cardinalità dell'insieme, in altre parole l'ampiezza del problema, e l'asse verticale il numero di nodi, in scala logaritmica.

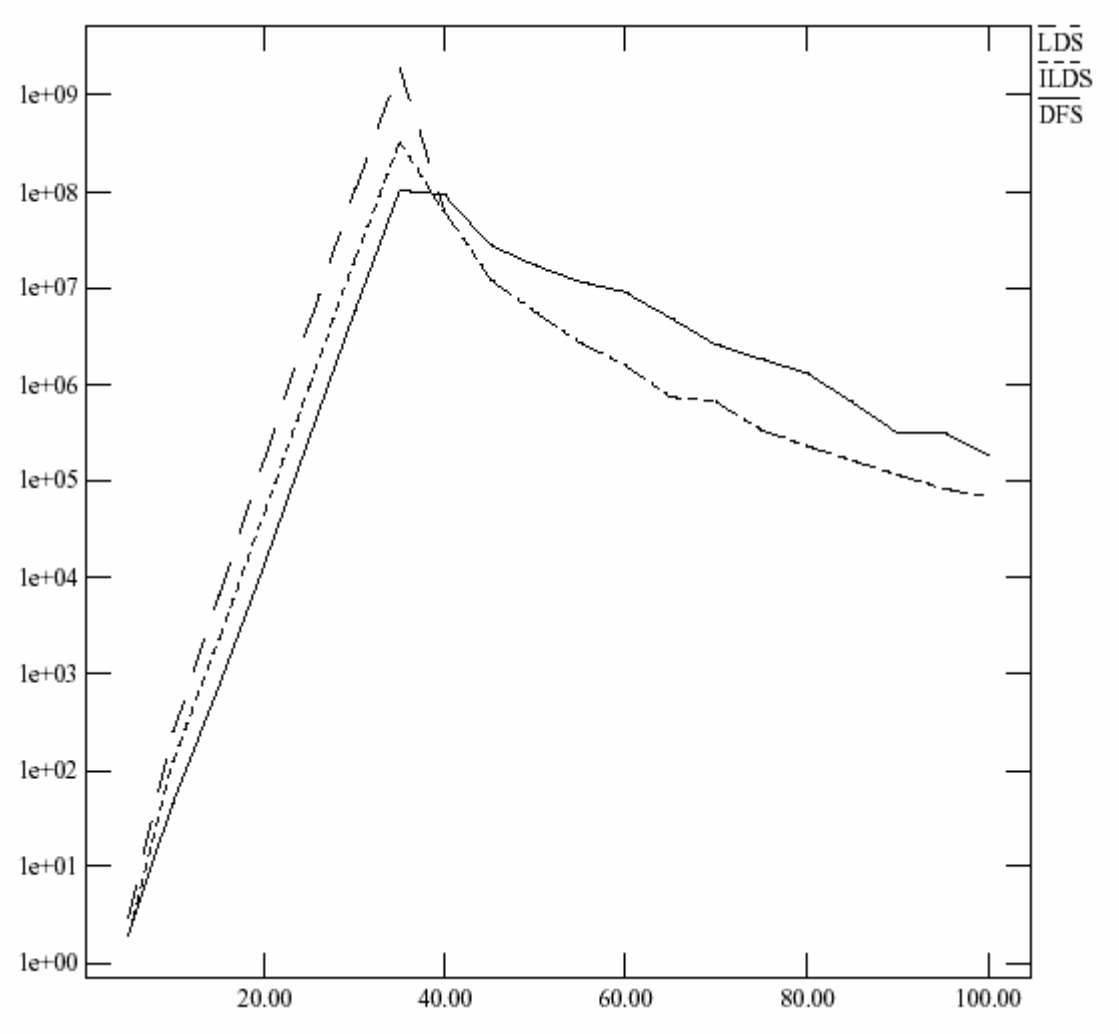

Fig. 2.2: *Nodi generati per risolvere il problema della partizione ottima di numeri di 10 cifre in funzione dell'ampiezza del problema.*

La prima cosa da notare è che, per tutti e tre gli algoritmi, la difficoltà del problema inizialmente cresce al crescere delle dimensioni, raggiunge un massimo, e quindi decresce. Questo accade perché, nella parte sinistra del grafico, la soluzione ottima ha meno probabilità di esistere, e quindi l'albero deve essere esplorato completamente. Più ampio è il problema, più ampio è l'albero generato e di conseguenza maggiore è il numero di nodi visitati. Viceversa, nella parte destra, è più probabile che esista una partizione ottima, e non appena una di esse viene trovata, la ricerca termina. All'aumentare delle dimensioni del problema, aumenta la densità di partizioni perfette, rendendole più facili da trovare, e quindi si riduce il numero di nodi visitati. Il picco occorre quando la probabilità che la partizione perfetta esista vale ½.

Si nota che nella parte sinistra dell'albero, cioè fino a quando è necessario esplorare tutto l'albero per ottenere una soluzione, l'algoritmo migliore è la DFS; questo è ovvio poiché, per com'è implementato, esso esplora ogni nodo una volta soltanto. Si nota inoltre che, sebbene resti sempre migliore della LDS, l'ILDS perde in efficacia all'aumentare delle dimensioni del problema, sino ad arrivare ad esserle asintoticamente equivalente. Questo accade perché, in questo particolare problema, viene a mancare il requisito, necessario per l'ottima applicazione dell'algoritmo, che l'albero sia completo. Entrambi gli algoritmi incompleti sono in ogni caso circa un ordine di grandezza migliori della DFS nella parte di problema in cui la soluzione ottima esiste con un'elevata probabilità.

2.4 Depth-bounded Discrepancy Search

La Depth-bounded Discrepancy Search (DDS), proposta nel '97 da Walsh[5], combina insieme la LDS e l'Iterative Deepening Search. Questo secondo algoritmo, definito da Korf nell'85[6], modifica la DFS

permettendo, ad ogni iterazione k, di esplorare l'albero solo fino al k-esimo livello.

L'idea alla base della DDS è che un'euristica tende ad essere meno informata, e quindi a commettere più errori, nei primi livelli dell'albero di ricerca, in pratica quando ancora le strade sono tutte aperte. Per esempio, l'algoritmo di Karmarkar-Karp per risolvere il problema della partizione di un insieme di numeri, descritto nel paragrafo precedente, prende sempre la stessa decisione, che potrebbe anche essere errata, alla radice dell'albero. Viceversa, vicino alle foglie, quando restano 4 o meno numeri nell'insieme, l'euristica fa sempre la scelta migliore.

Dato un tempo limitato per eseguire la ricerca, potrebbe essere conveniente esplorare le discrepanze poste in alto nell'albero prima di quelle più in profondità.

La DDS tende ad esplorare le discrepanze più vicine alla radice per mezzo di un vincolo di profondità incrementale. Le discrepanze oltre tale livello di profondità sono proibite. Alla prima esplorazione la DDS esplora la strada più a sinistra; alla k-esima iterazione vengono esplorati tutti i percorsi in cui le discrepanze occorrono ad una profondità minore di k.

Come l'ILDS, anche la DDS evita di analizzare le foglie già visitate nelle precedenti iterazioni, oltretutto in un modo semplicissimo da implementare. Al livello k-1 della k-esima iterazione, viene scelto solo il ramo destro, in quanto il ramo sinistro è stato necessariamente scelto nelle precedenti iterazioni. Ai livelli inferiori è obbligatorio seguire l'euristica, in altre parole è obbligatorio seguire il percorso più a sinistra.

La ricerca termina quando il valore del vincolo di profondità supera il valore della profondità massima dell'albero. In questo caso l'albero è stato completamente esplorato. Nella figura 2.3 è mostrata la ricerca tramite DDS di un albero completo di profondità 4.

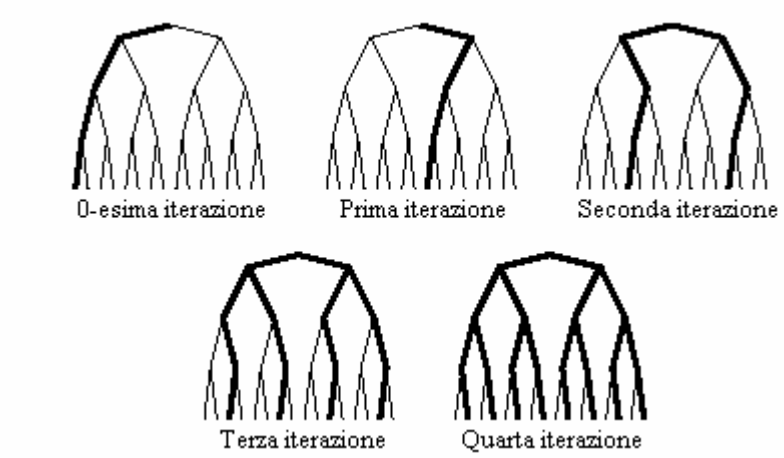

Fig. 2.3: *DDS in un albero binario di profondità 4.*

In un albero profondo d, la complessità asintotica della DDS è 2^d, così come per l'ILDS. Confrontando però il numero di discrepanze visitate dai 2 algoritmi, si nota che, mentre l'ILDS ha esplorato, all'iterazione k, solo percorsi con al massimo k discrepanze, la DDS ha esplorato percorsi che possono contenere fino a $k \cdot \log_2(d)$ discrepanze.

2.4.1 Esempio: ricerca euristica con probabilità costante di errore nella scelta

Per analizzare in maniera teorica gli algoritmi di ricerca basati sulle discrepanze, Ginsberg e Harvey[1] hanno introdotto un semplice modello di ricerca euristica. I nodi dell'albero decisionale sono etichettati come "buoni" o "cattivi". Un nodo buono ha almeno una soluzione nel suo sottoalbero, uno cattivo non ne ha nessuna. La probabilità di errore, m, è la probabilità che un figlio scelto a caso sia cattivo. Si assume che m sia costante in tutto l'albero. La probabilità euristica, p, è la probabilità che, in un nodo buono, l'euristica scelga un figlio buono. Supponendo che il nodo sia buono, se l'euristica sceglie un figlio cattivo allora l'altro figlio è necessariamente buono. Se l'euristica prende le scelte a caso, si ha p=1-m; se sceglie meglio che a caso allora p>1-m; se invece sceglie sempre la strada sbagliata risulta p=1-2m. Si assume che p resti costante in ogni prova. La figura 2.4 elenca tutte le situazioni che si possono

presentare allorché si esplora un nodo buono, con la probabilità con cui ogni situazione occorre.

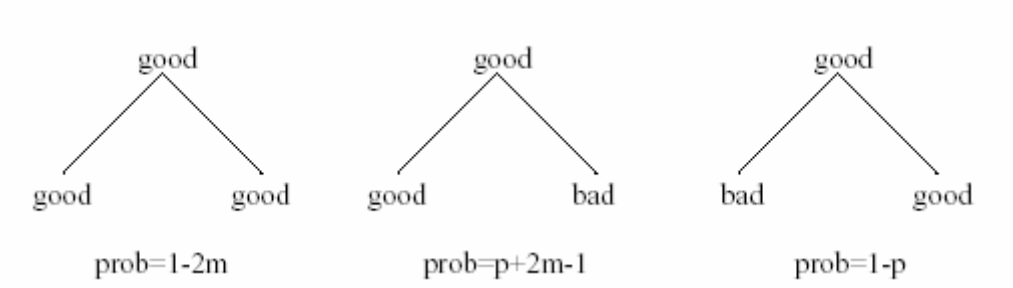

Fig. 2.4: *Diramazioni di un nodo buono. Siccome le situazioni sono disgiunte, la somma delle probabilità vale 1.*

Il confronto tra i tre algoritmi, DFS, LDS e DDS, è stato effettuato applicandoli a 3 diversi problemi. Nel primo viene eseguita l'esplorazione di 100000 alberi diversi, tutti con profondità 30 e m=0,2. Le prove sono state effettuate con p=0,9 e con p=0,95. I risultati ottenuti sono raffigurati nella figura 2.5.

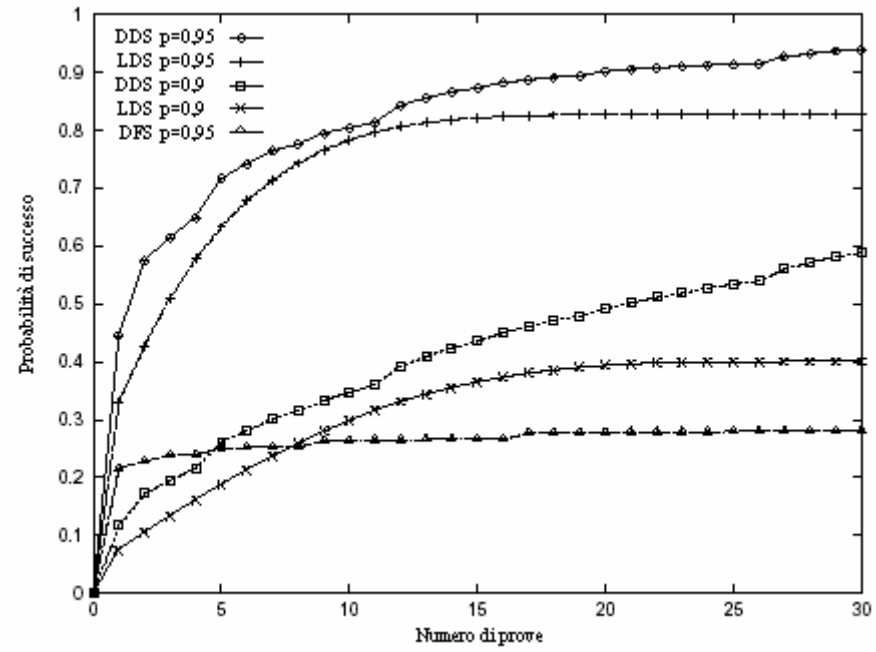

Fig. 2.5: *Probabilità di successo in alberi di profondità 30 con probabilità d'errore m=0,2.*

Tali alberi hanno circa un miliardo di foglie, di cui soltanto un milione sono soluzioni. Con una soluzione ogni 1000, il problema è relativamente facile, ma si riscontrano tuttavia notevoli differenze tra gli algoritmi. Le discontinuità nel gradiente dei grafici per la DDS corrispondono all'incremento del valore del vincolo di profondità. Tali discontinuità non sono presenti nel grafico della LDS perché, essendo per questa analisi (come si vede in ascissa) il massimo numero di percorsi esplorati limitato a 30, non si arriva ad eseguire l'iterazione di livello 2.

Il secondo problema, i cui risultati sono riassunti in figura 2.6, è di più difficile risoluzione. La profondità dell'albero vale ora 100, con m=0,1. Le prove, con p=0,95 e p=0,975, sono ripetute su 10000 alberi diversi. Gli alberi hanno 10^{30} foglie, e soltanto 1 su 38000 è una soluzione.

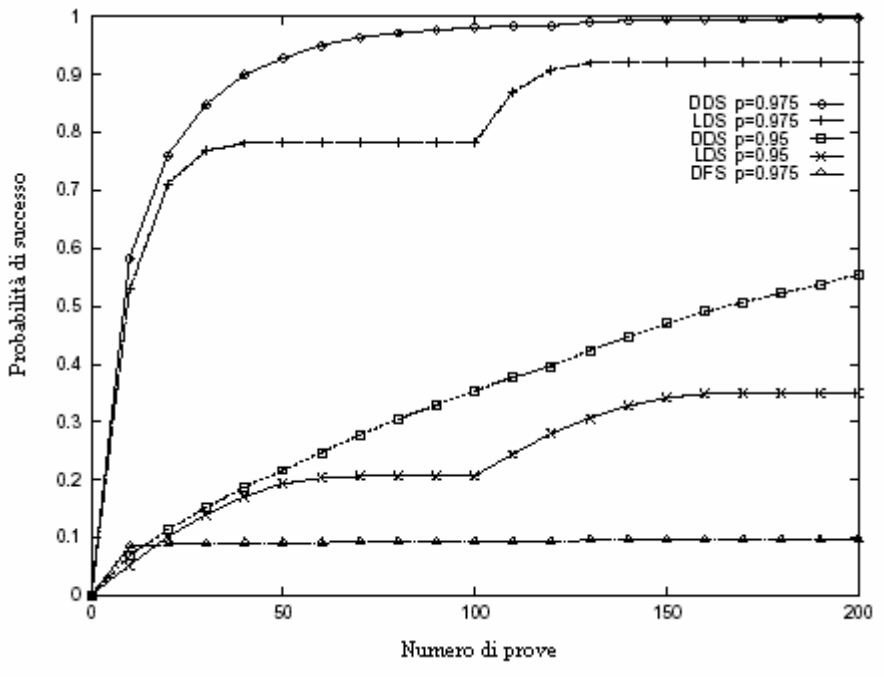

Fig. 2.6: *Probabilità di successo in alberi di profondità 100 con probabilità d'errore m=0,1.*

Il grafico riesce a mostrare fino alla seconda iterazione della LDS. Si nota che, così come per la DDS, anche il grafico della LDS sale velocemente ad ogni nuova iterazione. Anche in questo caso la DDS fornisce i risultati migliori.

Per finire, nel terzo problema, viene eliminato il vincolo che impone la costanza di p. In questo caso viene fatto variare linearmente da 1-m alla radice a 1 alla profondità massima (quindi la probabilità varia tra la scelta a caso alla radice e la scelta sempre esatta nelle foglie). A profondità i, p vale $(1 - m) + \frac{i \cdot m}{100}$. Anche analizzando questo modello, che dovrebbe avvicinarsi più degli altri al reale comportamento delle funzioni euristiche, la DDS fornisce le prestazioni migliori. Gli altri parametri sono identici a quelli del secondo problema. I risultati sono rappresentati nella sottostante figura 2.7.

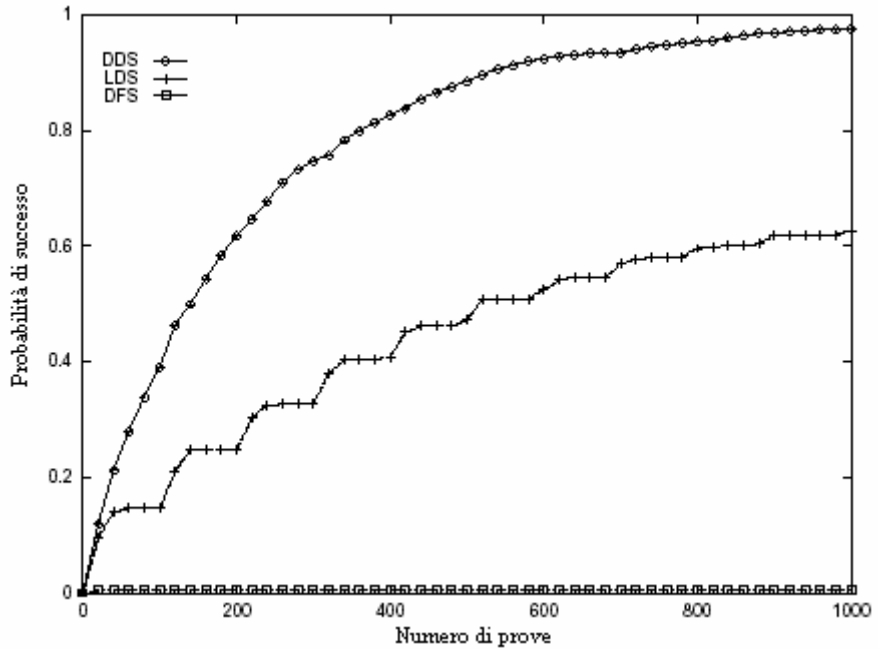

Fig. 2.7: *Probabilità di successo in alberi di profondità 100 con probabilità d'errore m=0,1 e probabilità euristica p linearmente crescente dal valore (1-m) alla radice (scelta a caso) al valore 1 nelle foglie (scelta esatta).*

Tutti i grafici di questo esempio sono stati ricavati da [1], [5] e [7].

2.4.2 Esempio: il problema della partizione dei numeri

Come già detto, sia la non completezza degli alberi decisionali, sia il pruning su di essi effettuato, appiattiscono le differenze tra le prestazioni dei vari algoritmi. Anche la DDS non sfugge a questa legge e, quindi, nei problemi pratici, le prestazioni sono peggiori di quelle riscontrate nel precedente esempio, puramente teorico e costruito ad hoc per l'utilizzo dell'algoritmo.

Come esempio, nella tabella 2.1 sono riportati i rami e i nodi esplorati dai 3 algoritmi, DFS, ILDS e DDS, applicati al problema della partizione di numeri, già descritto nel paragrafo 3. Ogni insieme analizzato, contiene 2n numeri scelti a caso tra 1 e 2^n, essendo n il numero riportato nella prima colonna della tabella.

n	**DFS**		**ILDS**		**DDS**	
	Rami	**Nodi**	**Rami**	**Nodi**	**Rami**	**Nodi**
5	1	7	1	7	1	7
10	1	17	1	17	1	17
15	11	46	7	48	3	81
20	266	565	29	472	29	1007
25	5324	10691	443	5287	354	14553
30	105647	211343	6037	69628	5531	260654

Tab. 2.1: Nodi e rami mediamente esplorati dagli algoritmi DFS, ILDS e DDS per suddividere un insieme di 2n numeri scelti a caso in $(0,2^n]$ utilizzando il metodo proposto da Karmarkar e Karp.

Si nota che, pur riuscendo a raggiungere la soluzione percorrendo il minor numero di rami, la DDS esplora il maggior numero di nodi. Questo accade perché la DDS esplora e scarta molti più nodi degli altri algoritmi nei primi livelli dell'albero, mentre è molto più efficace nei livelli più profondi. Sono però questi i livelli che, essendo più pesanti da gestire,

anche di diversi ordini di grandezza, dominano la complessità computazionale dell'algoritmo. È proprio questa la motivazione che rende vera l'affermazione, fatta nel precedente paragrafo 3, secondo cui l'esplorazione dei nodi interni non incide sulla complessità asintotica di un algoritmo.

CAPITOLO 3

PROGRAMMAZIONE LOGICA A VINCOLI

3.1 Introduzione

I linguaggi di Programmazione Logica a Vincoli (Constraint Logic Programming, CLP), combinano la dichiaratività della Programmazione Logica (Logic Programming, LP) e l'efficienza della Propagazione di Vincoli. Questa nuova classe di linguaggi riesce a superare le limitazioni intrinseche nei linguaggi di LP tipo Prolog.

In questo capitolo verranno descritte queste limitazioni e le tecniche che la CLP adotta per superarle, candidandosi come strumento ideale per la risoluzione di problemi combinatori.

3.2 Limitazioni dei linguaggi di programmazione logica

Gli oggetti che i programmi logici manipolano sono strutture che non vengono interpretate, quindi l'unificazione può essere eseguita con successo sono se gli oggetti sono sintatticamente identici. Per esempio, affermando X=2+3, l'atomo X viene legato alla struttura 2+3 e non al valore 5, quindi un'eventuale query X=5 restituisce un fallimento, in quanto il motore di inferenza non riesce ad unificare 5 con 2+3. Per ovviare a questi problemi, il Prolog definisce il predicato is/2, ma viene così a mancare la purezza logica del linguaggio, sacrificata per renderlo utilizzabile praticamente. Infatti il Prolog risponde affermativamente alle query 5 is 2+3, e A is 2+3 unificando A con 5, ma non riesce a fornire alla query 5 is A+3 la risposta esatta, A=2.

Inoltre, un linguaggio logico elabora le query in ordine testuale, così un goal del tipo X>0,X=2 fallisce perché, non essendo la variabile X istanziata, non può essere risolto il vincolo X>0. Viceversa un goal X=2,X>0 viene eseguito con successo in quanto, al momento dell'analisi del vincolo X>0, la variabile X è stata unificata al valore 2. Analogamente, dato il fatto p(X,X), la query p(Y,Z) ritorna yes, Y=Z; al contrario, dato il fatto p(X,Y):-X>Y, la stessa query p(X,Y) ritorna un

fallimento anziché la risposta naturale yes, if Y>Z. Questo accade perché il predicato =/2, diversamente dal predicato >/2, provoca un'unificazione della variabile ed è quindi sempre risolubile.

Un risolutore in grado di "addormentare" (freeze) i vincoli che non sia in grado di valutare, riprendendoli in considerazione solo quando saranno disponibili le informazioni necessarie a risolverlo correttamente, solleverebbe il programmatore dall'onere di gestire tali situazioni mediante i meta-predicati sopra citati. I vincoli addormentati dovranno essere memorizzati e ripresi in considerazione ad ogni passo della computazione.

3.3 CLP come metodo per superare i limiti della LP

La CLP nasce nel 1987 da studi condotti da Jaffar e Lassez come metodo per superare le principali limitazioni intrinseche nella LP[8].

Sono 2 le prerogative salienti implementate dalla CLP: la capacità di associare ad ogni oggetto la sua semantica, e la possibilità di sfruttare procedure di ricerca nell'albero decisionale più intelligenti.

Associare ad ogni oggetto la sua semantica significa riconoscere in esso una struttura che vada oltre la semplice sintassi, e che possa quindi essere associata alle operazioni primitive che tale struttura supporta. Quindi, per esempio, l'oggetto 5 non è più solo un atomo costante, bensì un numero e come tale supporta tutte le operazioni tipiche dei reali; inoltre l'oggetto 2+3 è visto come l'operazione di somma tra 2 numeri e non come il predicato +/2. È così possibile riconoscere l'uguaglianza semantica tra 5 e 2+3.

Analogamente X>0 viene riconosciuto come una relazione d'ordine tra numeri reali, e non come un generico predicato o test.

Queste prerogative, unite al fatto di poter implementare tecniche di Propagazione di Vincoli e di Consistenza, consentono, al momento dell'elaborazione di un vincolo che il Prolog non sarebbe in grado di risolvere, sia di ridurre i domini delle variabili in gioco finché è

possibile, sia di mantenere attivo il vincolo e di tornare a valutarlo ad ogni passo computazionale. Esiste una struttura, detta constraint store, dove i vincoli addormentati vengono memorizzati.

Formalizzando, la CLP si compone di 4 passi fondamentali:

- **Risoluzione:** selezione di un atomo dal risolvente e unificazione, eventualmente seguita dall'aggiunta al constraint store dei vincoli scaturiti dall'unificazione.
- **Constraining:** selezione di un vincolo dal risolvente e aggiunta del vincolo stesso al constraint store.
- **Propagazione:** trasformazione del constraint store e dei domini associati alle variabili vincolate.
- **Consistenza:** verifica della possibilità di soddisfacimento del constraint store.

Il Prolog esegue soltanto il passo di risoluzione tramite una successione di selezione di atomi e unificazioni.

3.3.1 Esempio di funzionamento di una macchina CLP

Consideriamo le inferenze

r(X,Y) :- p(X), X>Y, q(Y).

p(Z) :- Z=3.

q(K) :- K>5.

q(K) :- K>2.

e il goal **?- r(A,B).** Nella tabella che segue viene presentata la risoluzione di questo problema, mostrando riga per riga il passo fondamentale eseguito, il risolvente corrente e il contenuto del constraint store.

Passo fondamentale	**Risolvente**	**Constraint store**
Risoluzione	**R(A,B)**	∅
(unificazione)	p(X'), X'>Y', q(Y')	**A=X', B=Y'**

Propagazione	(idem)	(idem)
Consistenza	(idem)	(idem)
Risoluzione	**p(X')**, X'>Y', q(Y')	(idem)
(unificazione)	Z'=3, X'>Y', q(Y')	A=X', B=Y', **X'=Z'**
Propagazione	(idem)	A=X'=Z', B=Y'
Consistenza	(idem)	(idem)
Constraining	**Z'=3**, X'>Y', q(Y') X'>Y', q(Y')	A=X'=Z', B=Y' A=X'=Z', B=Y', **Z'=3**
Propagazione	(idem)	A=X'=Z'=3, B=Y'
Consistenza	(idem)	(idem)
Constraining	**X'>Y'**, q(Y') q(Y')	A=X'=Z'=3, B=Y' A=X'=Z'=3,B=Y',**X'>Y'**
Propagazione	(idem)	A=X'=Z'=3>Y', B=Y'
Consistenza	(idem)	(idem)
(unificazione)	K'>5	A=X'=Z'=3>Y',B=Y',**Y'=K'**
Propagazione	(idem)	A=X'=Z'=3>Y',B=Y'=K'
Consistenza	(idem)	(idem)
Constraining	**K'>5** ∅	A=X'=Z'=3>Y',B=Y'=K' A=X'=Z'=3>Y',B=Y'=K',K'=5
Propagazione	(idem)	A=X'=Z'=3>Y',B=Y'=K'>5
Consistenza	**Fallimento**	**Inconsistente**
(unificazione)	**q(Y')** K'>2	A=X'=Z'=3>Y', B=Y' A=X'=Z'=3>Y', B=Y', Y'=K'
Propagazione	(idem)	A=X'=Z'=3>Y',B=Y'=K'
Consistenza	(idem)	(idem)
Constraining	**K'>2** ∅	A=X'=Z'=3>Y',B=Y'=K' A=X'=Z'=3>Y',B=Y'=K',K'>2
Propagazione	(idem)	A=X'=Z'=3>Y',B=Y'=K'>2
Consistenza **Successo**	(idem)	(idem) **A=3, 2<B<3**

Tab. 3.1: Macchina CLP: passi fondamentali

3.4 Vincoli Simbolici

Oltre alla possibilità di definire vincoli matematici in grado di essere propagati intelligentemente durante la ricerca, Beldiceanu e Contejean [9] hanno introdotto nella CLP il concetto di vincoli simbolici. Questi vincoli hanno la stessa sintassi dei predicati Prolog ma, a differenza di questi ultimi, assumono un significato semantico che guida la propagazione dei vincoli stessi. Essi sono quindi in grado sia di incapsulare un metodo di propagazione globale ed efficiente, sia di fornire una formulazione concisa del vincolo da utilizzare.

Analizzando, per esempio, il vincolo predefinito alldifferent/1, che accetta come argomento una lista di variabili, si nota che, avendo affermato alldifferent([X_1,X_2,X_3,X_4]), se le variabili X_1, X_2 e X_3 hanno lo stesso dominio [1,2,3] e la variabile X_4 ha dominio [1,2,3,4], è possibile inferire che X_4 debba necessariamente valere 4. Questo accade perché esiste un insieme di variabili di cardinalità 3 che hanno il medesimo dominio anch'esso di cardinalità 3, quindi tali 3 valori devono essere assunti dalle 3 variabili; la propagazione del vincolo di disuguaglianza elimina i valori 1, 2 e 3 dal dominio della variabile X_4, che rimane con un solo valore, appunto 4. Questo assegnamento, che può essere generato dal Prolog solo in seguito all'esplorazione dell'albero di ricerca, è invece automaticamente effettuato dalla macchina CLP grazie alle informazioni semantiche incapsulate nel predicato alldifferent/1.

Esistono altri vincoli simbolici predefiniti, tra cui **cumulative/4** e **element/3**:

- **Cumulative/4,** i cui primi tre argomenti sono liste di numeri di eguale cardinalità c, e rappresentano rispettivamente l'istante d'inizio, la durata e le risorse utilizzate di c differenti attività. Il quarto argomento indica il limite della disponibilità delle risorse, e può anche variare nel tempo. La figura 2.1 rappresenta graficamente il vincolo **cumulative([1,2,4],[4,2,3],[1,2,2],3)**. L'attività 1, che inizia all'istante 1 e dura 4 unità di tempo, occupa 1 risorsa; l'attività 2, che inizia

all'istante 2 e ha durata 2, e l'attività 3, che inizia al tempo 4 e dura 3 unità di tempo, impiegano entrambe 2 risorse. È così verificato il vincolo di non occupare più di 3 risorse contemporaneamente.

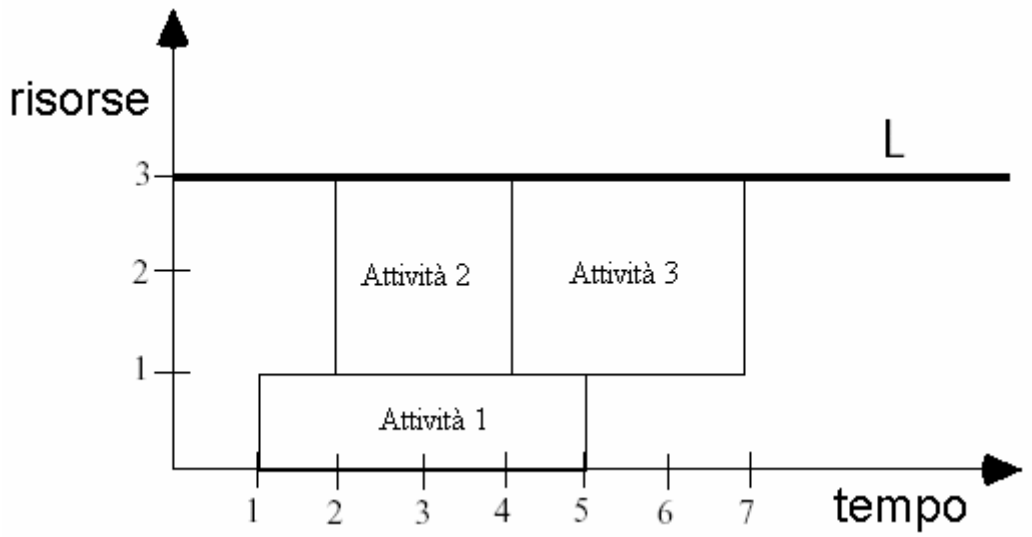

Fig. 3.1: *Rappresentazione grafica del vincolo simbolico cumulative/3*

- **element/3,** i cui argomenti sono rispettivamente un numero N, una lista di variabili, e il valore che l'N-esimo elemento della lista deve avere. L'analogia con il predicato Prolog arg/3 è soltanto apparente, poiché, ad un'eventuale affermazione element(N,[X_1,X_2,...,X_n],c), il motore CLP inferisce (N=1 and X_1=c) or(N=2 and X_2=c)or...or(N=n and X_n=c), mentre arg(N,f(X_1,X_2,...,X_n),c) ritorna fallimento.

3.4.1 Vincoli disgiuntivi

In un problema di allocazione di risorse a più attività, può accadere che l'esistenza di alcuni vincoli dipenda dall'ordine in cui le attività vengono eseguite. Per esempio, se un oratore deve tenere 2 conferenze nello stesso giorno, chiamate StartCn e TimeCn le variabili che rappresentano rispettivamente l'istante di inizio e la durata dell'n-esima conferenza, sussiste il vincolo **StartC1 + TimeC1 <= StartC2** se viene tenuta prima la conferenza C1, mentre deve essere imposto che sia **StartC2 + TimeC2 <= StartC1** nel caso contrario.

In CLP è possibile mettere in OR questi due vincoli. Vengono quindi generati 2 problemi **indipendenti**, dove le scelte effettuate per un problema non hanno effetto sull'altro. Purtroppo i vincoli OR aumentano esponenzialmente la complessità computazionale del problema; infatti, N vincoli in OR fra di loro generano 2^N sotto-problemi indipendenti.

Sono stati proposti 3 metodi che, avendo come obiettivo il contenimento dell'aumento della complessità, gestiscono tali vincoli disgiuntivi:

- **Disgiunzione costruttiva [10]:** l'idea è quella di aggiungere al constraint store solo i vincoli che sono implicati da tutte le parti della disgiunzione. Per esempio, nel problema **X::[5..10], Y::[7..11], Z::[1..20], (Z=X OR Z=Y)**, il primo vincolo ridurrebbe il dominio di Z a [5..10], mentre il secondo a [7..11]. Risultato della disgiunzione costruttiva è la riduzione del dominio di Z a [5..11].
- **Operatore di cardinalità:** viene introdotto un nuovo vincolo simbolico **#/3**, detto appunto operatore di cardinalità. Il terzo argomento è una lista di vincoli, mentre i primi due argomenti sono numeri che rappresentano rispettivamente il numero minimo e massimo di vincoli soddisfatti tra quelli nella lista.

  Con questo metodo il problema delle conferenze sarebbe modellato nel modo seguente:

  #(1,1,[StartC1 + TimeC1 <= StartC2, StartC2 + TimeC2 <= StartC1])

  significando che tra i vincoli uno e soltanto uno dovrà risultare soddisfatto.
- **Meta-constraint:** ad ogni vincolo viene associata una variabile booleana B. Se B=1 (true) il vincolo è verificato, se B=0 (false) non è verificato. La notazione **c⇔B** associa ad un vincolo c la variabile booleana B. Con i meta-constraint, la disgiunzione dell'esempio può essere modellata come segue:

  B1::[0,1], B2::[0,1]

  StartC1 + TimeC1 <= StartC2 ⇔ B1,

StartC2 + TimeC2 <= StartC1 ⇔ B2,

B1 + B2 = 1.

Anche in questo caso l'informazione aggiunta è che esattamente un vincolo dovrà essere soddisfatto.

Alcuni linguaggi di CLP, come per esempio ECLiPSe, offrono ulteriormente la possibilità di controllare i meccanismi di propagazione dei vincoli sia definendo nuovi vincoli simbolici, sia gestendo i meccanismi di propagazione di quelli predefiniti.

4.1 Introduzione

La rapida espansione del commercio elettronico sta permettendo alle aziende, o più in generale a chi ne fa uso, di ampliare il numero di clienti e fornitori, riducendo allo stesso tempo costi e rischi inevitabilmente connessi con qualsiasi transazione economica.

Tipicamente, tramite un'asta, un venditore cerca di piazzare un bene al miglior offerente; qualora il bene stesso possa essere suddiviso in più parti distinte vendibili singolarmente e, quindi, non necessariamente allo stesso compratore, non è più possibile ottenere dei risultati economicamente convenienti con una semplice asta.

In quest'ottica, le **aste combinatorie** si pongono come metodo per superare le limitazioni intrinseche nel corrente sistema di compravendita, basato sui singoli prodotti o su cataloghi a prezzo fisso, in cui si tiene conto soltanto del costo, che, invece, è semplicemente una delle molte variabili presenti nelle relazioni compratore-fornitore.

4.2 Le aste combinatorie

Nello svolgimento di un'asta su una molteplicità di oggetti, può essere necessario avere la possibilità di fare offerte su combinazioni degli stessi, così come considerare il tempo, al pari del costo, una variabile determinante nella ricerca dell'offerta migliore. Un'asta che permetta tutto questo è chiamata **combinatoria**.

Il sistema delle aste combinatorie è stato progettato per permettere le trattative di compravendita di entità (siano essi lavori, servizi, insiemi di prodotti ecc.) tra una popolazione di agenti tra loro indipendenti. In questo lavoro, saranno presi in considerazione insiemi **coordinati** di attività, cioè insiemi di attività cui sono associati vincoli temporali, quali per esempio il tempo di inizio o la durata, oppure vincoli di precedenza.

Un’asta combinatoria in cui chi bandisce debba acquistare dei beni da dei venditori, anziché venderli a dei compratori, prende il nome di asta combinatoria **inversa**.

I vantaggi di questo tipo di aste, dovendo sia vendere sia comprare, sono enormi. Supponendo che chi indice l’asta debba vendere delle attività, egli può a priori definire i vincoli cui tali attività devono soddisfare. Il suo scopo è quello di vendere tutti gli oggetti messi all’asta, ottenendo il miglior guadagno possibile, sia in termini di maggior ricavo economico, sia di minor tempo impiegato dai compratori a terminare le attività.

Viceversa, chi compra può fare diverse offerte. In letteratura tali offerte sono definite tramite una coppia (S,p), dove S è un insieme che contiene una o più tra le attività messe all’asta, per ognuna delle quali viene specificata una finestra temporale entro cui si intende dare inizio all’attività, e la durata prevista; p è il prezzo che si intende pagare per ottenere gli oggetti contenuti in S. Intento di chi compra è di ottenere il maggior numero di oggetti, in pratica di vincere una coppia (S,p) con S grande.

Si suppone che un’offerta possa essere o rifiutata o accettata in toto, in altre parole che non sia possibile ottenere solamente alcuni degli oggetti contenuti in S. Questo consente al compratore di ottenere solo ciò di cui ha effettivamente bisogno. Per esempio, in un’asta per vendere delle tratte radio, se un compratore ha bisogno di una tratta tra due luoghi A e B, può comprare due tratte, rispettivamente tra A e C, e tra C e B, mentre è completamente inutile che ottenga solo la tratta tra A e C. Tramite un’asta combinatoria, tale volontà è semplicemente espressa facendo un’offerta sull’insieme che contiene entrambe le tratte. È così garantito che o si perde l’asta, o si ottiene esattamente quello che si desidera.

È evidente che le potenzialità delle aste combinatorie, rispetto a quelle classiche, sono molto maggiori e più raffinate, riuscendo ad ottimizzare i guadagni e a ridurre i rischi per entrambe le parti.

Purtroppo, utilizzare le aste combinatorie introduce delle complessità aggiuntive di cui occorre tenere conto. Primariamente, essendo associati ad ogni attività una finestra temporale e dei vincoli di precedenza, ogni algoritmo di selezione delle offerte deve assicurare la realizzabilità temporale delle offerte accettate. Inoltre, dato che chi vende intende vendere tutti gli oggetti messi all'asta, ovviamente ognuno ad un solo compratore, occorre garantire la copertura di tutte le attività. Infine, ritardi e fallimenti che possono avvenire durante l'esecuzione delle attività programmate, possono avere effetti devastanti sul raggiungimento dell'obiettivo finale. Lo spazio di ricerca associato ad un'asta combinatoria è quindi tipicamente molto ampio e complesso da analizzare, in quanto esistono molti vincoli tra le variabili in gioco. Il problema della determinazione del vincitore di un'asta combinatoria su un insieme coordinato di attività è NP-completo.

4.3 Linguaggi di offerta

Si è visto nel paragrafo precedente che il problema della determinazione del vincitore in un'asta combinatoria è NP-completo. Per risolverlo possono quindi essere utilizzati gli algoritmi introdotti nei precedenti capitoli. Se, come avviene nei casi reali, a causa dell'elevato numero delle attività e delle offerte, il problema genera uno spazio di ricerca molto ampio, la scelta dell'algoritmo più adeguato ha un forte impatto sul tempo di risoluzione del problema. È necessario definire un modello cui applicare i vari algoritmi per testarne l'efficacia. In letteratura sono stati proposti vari modelli, sia per la selezione delle variabili e dei valori, sia per la generazione di offerte. Questi ultimi, definiti da Nisan[11], sono chiamati **Linguaggi di offerta**.

Prima di passare alla loro descrizione, occorre introdurre i concetti di **complementarità** e **sostituzione**. Per un offerente j, due insiemi disgiunti S e T si dicono complementari se $v_j(S \cup T) > v_j(S) + v_j(T)$, dove v_j è una funzione che restituisce la valutazione che l'offerente j dà

all'insieme argomento della funzione stessa. Viceversa due insiemi si dicono sostituti se $v_j(S \cup T) < v_j(S) + v_j(T)$.

Nisan introduce 3 tipologie basilari di offerta: **Atomiche**, per cui ogni offerente può presentare una sola offerta, considerata come coppia (S,p); **OR**, per cui possono essere proposte più offerte atomiche, sapendo che ogni offerente può ottenere un numero qualsiasi di coppie; **XOR**, per cui possono ancora essere proposte più offerte atomiche, ma il numero di coppie ottenibili dal singolo offerente è al massimo uno.

Si dimostra che le offerte OR possono rappresentare tutte e sole le offerte che non hanno sostituti; si dimostra inoltre che le XOR possono esprimere tutti i tipi di offerte. Componendo questi 3 linguaggi si possono ottenere, per esempio, le offerte **OR-of-XOR**, intese come offerte OR su gruppi di offerte XOR, il suo analogo **XOR-of-OR**, o il più generale **OR/XOR** in cui si possono combinare i due tipi di offerte.

Vengono proposti anche degli algoritmi per assegnare un valore al prezzo, cioè alla variabile p della coppia (S,p). Una prima distinzione è tra modelli simmetrici e asimmetrici. Nei primi, tutti gli oggetti messi all'asta sono considerati identici, almeno dal punto di vista del compratore, perciò il valore di p dipende solo dalla dimensione dell'insieme S. Viceversa, una valutazione asimmetrica distingue gli oggetti secondo una certa logica, e il prezzo è assegnato in accordo con tale distinzione. Numerosi sono i modelli di valutazione simmetrica proposti da Nisan:

- **Valutazione additiva**: ad ogni sottoinsieme è associato un valore pari alla sua cardinalità. Questo tipo di valutazione non ha né sostituti, né complementari. Utilizzando il metodo di offerta OR, una valutazione additiva su m oggetti ha dimensione m, ma questo valore sale a 2^m se si utilizza il metodo XOR.
- **Valutazione di singolo oggetto**: ogni insieme non vuoto è valutato 1, in quanto l'offerente desidera ogni singolo oggetto, e solo quello. In questo caso, tutti gli oggetti sono tra di loro sostituti.

- **Valutazione K-budget:** ogni sottoinsieme ha un valore pari al numero minore tra la sua cardinalità e il numero K, che definisce il budget massimo a disposizione per l'asta.
- **Valutazione simmetrica generica:** l'offerente definisce una serie di valori p_1, p_2, …, p_n. Il prezzo p_i specifica quanto si intende pagare per l'i-esimo oggetto vinto. Risulta $v(S) = \sum_{i=1}^{|S|} p_i$.
- **Valutazione simmetrica decrescente:** definita come la valutazione generica, ma con i valori decrescenti. Questa valutazione cattura il caso reale in cui ogni elemento addizionale è valutato meno del precedente. Una valutazione del genere, su un insieme di m oggetti, può essere espressa tramite offerte OR-of-XOR di dimensione m^2.

Per quanto riguarda le valutazioni asimmetriche, interessanti sono la **valutazione monocromatica** e la **valutazione uno-per-tipo**. La prima suddivide gli oggetti in due sottoinsiemi; ad ogni elemento del primo è attribuito un colore, diciamo rosso, mentre ad ogni elemento del secondo ne è attribuito un altro, diciamo blu. La valutazione di un insieme S, che contenga k elementi rossi ed h blu, è uguale al massimo tra k ed h. L'idea è di dare valore solo ad elementi appartenenti allo stesso gruppo.

La seconda, la valutazione uno-per-tipo, suddivide invece gli oggetti in coppie, valutando k+h ogni sottoinsieme che contenga k coppie e h oggetti singoli. In questo caso l'idea è di voler ottenere almeno un elemento per ogni coppia, e il valore p esprime, appunto, il numero di coppie di cui è stato ottenuto almeno un oggetto.

4.4 Valutazione di offerte su un insieme coordinato di attività: un esempio

Si supponga di avere un lavoro da effettuare che richieda l'esecuzione di più attività in maniera coordinata. Le risorse necessarie ad eseguire tali attività devono essere acquistate da fornitori. Lo scopo è quello di terminare il lavoro in un limitato periodo di tempo, minimizzando la

spesa. Per i fornitori, lo scopo è massimizzare il valore delle risorse che possiedono. La risoluzione di un tale problema deve non solo fornire un piano temporalmente fattibile, ma deve anche assicurare che, tra tutti i piani possibili, quello restituito necessiti della spesa minore.

Supponiamo di ricevere le offerte descritte nella successiva tabella.

Offerte	Costo	Attività	Minimo inizio	Massimo inizio	Durata
B1	200	T1	15	30	110
		T3	125	140	85
		T5	220	230	130
B2	290	T2	100	160	180
		T6	340	360	60
B3	160	T4	90	130	60
		T5	150	190	140
B4	20	T4	100	150	110
B5	250	T2	110	150	170
		T4	100	130	80
		T6	320	350	170
B6	180	T4	130	160	100

Tab. 4.1: Esempio di offerte su un insieme coordinato di attività.

Le 6 attività devono soddisfare i vincoli di precedenza descritti dalla successiva figura 4.1.

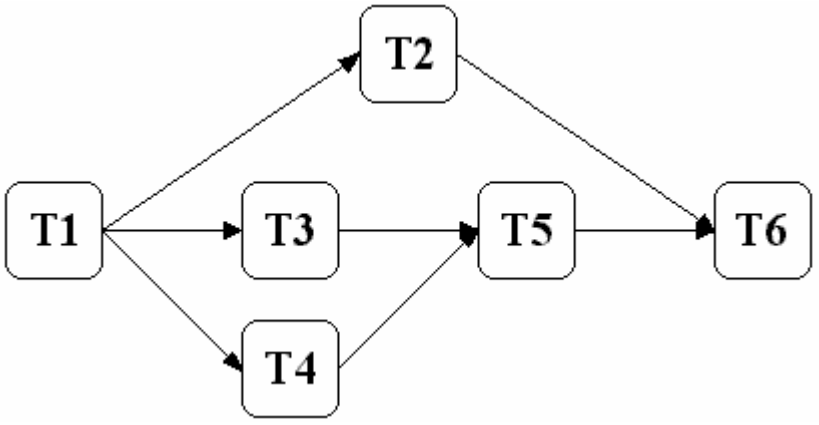

Fig. 4.1: *Esempio di attività coordinate tramite vincoli di precedenza.*

Si supponga infine che il lavoro debba terminare entro l'instante 600.

Alcune osservazioni sono immediatamente evidenti:

- L'offerta B1 deve essere necessariamente accettata, in quanto è l'unica che offre le attività T1 e T3.
- Le offerte B1 e B3 non possono far parte della stessa soluzione, perché entrambe richiedono l'attività T5. Ogni attività deve esse allocata esattamente ad un'offerta.
- Le offerte B1 e B3 non possono essere entrambe vincenti, perché il vincolo di precedenza tra T3 e T5 non potrebbe essere soddisfatto. Infatti, l'offerta B1 non riesce a terminare l'attività T3 prima dell'istante 210, mentre B2 non può iniziare T5 dopo l'istante 190.
- Le offerte B1 e B4 non possono essere entrambe accettate. La motivazione è che, sebbene le relazioni di precedenza tra T1 e T4, e tra T4 e T5 possano essere individualmente soddisfatte, quando le due offerte vengono combinate insieme, la soddisfazione del primo vincolo rende impossibile la soddisfazione anche il secondo. Infatti B4 non può terminare T4 prima dell'istante 235, ma B1 non può iniziare T5 più tardi dell'istante 230.

Analizzando i valori in tabella si evince che, scegliendo le offerte B1, B2 e B6, si ottiene una soluzione. Il piano ottenuto inizia all'istante 15, termina all'istante 420, e costa 670. Continuando però l'analisi dei valori, si trova una seconda soluzione che contiene le offerte B1 e B5. Anche in questo caso l'istante iniziale è 15, ma l'istante finale è 520 e il costo 450. Sebbene la seconda soluzione abbia una durata temporale maggiore, essendo l'obiettivo primario quello di minimizzare il costo, questa soluzione è da preferirsi alla precedente.

Il precedente esempio è ripreso da [12].

CAPITOLO 4

ASTE COMBINATORIE SU INSIEMI COORDINATI DI ATTIVITÀ

CAPITOLO 5

LIBRERIA ILOG SOLVER

5.1 Introduzione

ILOG Solver è un prodotto di ILOG, una compagnia francese tra le più importanti al mondo nel settore della produzione di componenti software avanzati.

ILOG Solver è una libreria object oriented C++ utile per risolvere problemi di programmazione, allocazione, ottimizzazione, gestione e simili, sfruttando la programmazione a vincoli.

In questo capitolo sarà descritto il funzionamento di ILOG Solver, descrivendo nel dettaglio le classi più rilevanti utilizzate nella presente tesi. Per convenzione tutti i nomi delle classi di ILOG Solver iniziano con *Ilo* o *Ilc*.

La versione analizzata è ILOG Solver 5.1.

5.2 Rappresentazione di un problema

La rappresentazione di un problema consiste nella dichiarazione delle incognite e dei vincoli contenuti nel problema stesso. Ogni rappresentazione è specifica del dominio del problema considerato, ed è quindi necessario utilizzare un linguaggio molto espressivo per superare tale specificità. Solver sfrutta l'orientamento agli oggetti per rendere questa attività il più semplice possibile, in quanto fornisce classi per rappresentare tutti gli oggetti che possono far parte di un CSP, quali variabili, vincoli, domini ecc.

Solver lavora con ILOG Concert Technology, una libreria di classi e funzioni utili per la definizione degli oggetti sopra citati.

La prima cosa da fare per descrivere un problema, è creare un ambiente di lavoro in cui costruire il modello del problema da risolvere. Concert Technology definisce 2 classi, **IloEnv** e **IloModel**, preposte a ciò. Un'istanza di IloEnv rappresenta un ambiente e si occupa di gestire i problemi di modellazione interna, quali l'output, la memoria, la terminazione corretta degli algoritmi di ricerca. Un'istanza di IloModel

rappresenta un modello, cioè un contenitore di oggetti, in questo caso variabili e vincoli.

Per rappresentare le variabili numeriche esiste la classe **IloNumVar**. Ogni istanza di tale classe incapsula le specifiche di una variabile numerica, il cui tipo (intera o reale) e dominio, cioè i valori minimo e massimo che può assumere, sono definiti al momento della creazione. Dalla versione 5.1 di Solver sono state introdotte due classi, **IloIntVar** e **IloFloatVar**, che distinguono a priori le variabili intere da quelle reali.

Per molti tipi basilari, Concert definisce le corrispondenti classi array, chiamate *Array* postfisso al nome dell'oggetto di cui è composto. **IloNumVarArray** è quindi la classe che rappresenta un array di variabili numeriche. Al momento della creazione di un array, è sia possibile dichiarare il numero di variabili che lo comporranno, nonché tipo e dominio uguali per tutte, sia creare un array vuoto ed aggiungervi poi le variabili una ad una.

I vincoli sono definiti dalla classe **IloConstraint**, da cui ereditano numerose sottoclassi che specificano diversi tipi di vincoli.

- Gli operatori relazionali =, !=, <, >, >=, <= sono sovraccaricati e, se applicati a due argomenti che rappresentano numeri, variabili numeriche o espressioni, restituiscono un oggetto di tipo IloConstraint pronto per essere aggiunto ad un modello. Per esempio l'istruzione *model.add(X<Y+Z)* aggiunge, tramite il metodo *add* della classe IloModel, ad un modello chiamato *model* il vincolo che il valore assunto da una variabile, denotata da X, debba essere inferiore alla somma di quelli assunti dalle variabili denotate con Y e Z.
- **IloDistribute** è una classe che definisce un vincolo di conteggio. Al momento della creazione di un'istanza riceve, tra gli altri, tre parametri chiamati cards, vars e values, essendo i primi due array di variabili e il terzo un array di valori. Il vincolo forza le variabili contenute nell'array cards a valere il numero di occorrenze, nell'array vars, dei valori contenuti in values. Più precisamente, per ogni i, cards[i] è uguale al numero di volte che le variabili contenute in vars

assumono il valore values[i]. Questo vincolo può servire sia per contare il numero di occorrenze di un valore, sia per forzare le variabili di un array ad assumere dei valori, in modo che solo un limitato numero di esse possa assumere possa assumere un certo valore. Per esempio, se occorresse risolvere il problema di colorare 5 oggetti con 3 colori, in modo che al massimo 3 oggetti siano del primo colore, esattamente tre del secondo e al massimo uno del terzo, gli array da fornire al vincolo IloDistribute sarebbero i seguenti:

cards = [[0,3][3][0,1]]

values = [C_1,C_2,C_3]

vars = [O_1,O_2,O_3,O_4,O_5]

avendo denotato con O_n l'n-esima variabile e con C_n l'n-esimo colore. Ovviamente, il dominio delle variabili contenute in vars è inizialmente values.

Sebbene Solver includa una vasta gamma di vincoli predefiniti, offre anche la possibilità di definirne di nuovi. All'interno di questi oggetti, è possibile implementare dei metodi che riducano, secondo una strategia ottimizzata per il problema in questione, i domini delle variabili vincolate. La creazione di queste classi non è banale, in quanto occorre sia progettare i metodi che propagano i vincoli, sia gestire lo spazio di memoria, tipicamente un heap, in cui le propagazioni sono memorizzate per poter poi essere recuperare in backtracking. Concert viene in aiuto del programmatore definendo la macro ILOCPCONTRAINT, che si occupa di creare il codice che consenta di predisporre il sistema ad una adeguata gestione della memoria e dei domini delle variabili.

Esiste infine una tassonomia di classi, che ereditano da **IloObjective**, che denotano un obiettivo che deve essere raggiunto dalla ricerca. Tra le altre, **IloMinimize** ha come obiettivo quello di minimizzare la variabile o l'espressione passata al suo costruttore al momento della creazione.

Anche un oggetto di questo tipo può essere aggiunto ad un modello.

5.3 Estrazione di un modello

Una volta aggiunti ad un modello tutti gli oggetti necessari, si può passare alla fase di risoluzione del problema. Per fare questo è necessario creare un'istanza di **IloSolver**, che rappresenta un risolutore, responsabile della determinazione di una soluzione per il problema espresso nel modello. Le operazioni basilari che compie un'istanza di IloSolver sono "estrarre" un modello tramite il metodo **extract** e risolverlo usando il metodo **solve**.

Tutti gli oggetti che possono essere inseriti in un modello (variabili, vincoli, obiettivi ecc.), ereditano da una classe base comune **IloExtractable**, che rende possibile la loro estrazione al momento della creazione di un risolutore, cioè di un'istanza di IloSolver. Durante l'estrazione, il risolutore analizza tutti gli elementi del modello e crea, per ognuno di essi, i corrispondenti oggetti computazionali. I nomi di tali oggetti solitamente iniziano con *Ilc*, quindi, per esempio, da un IloIntVar deriva un oggetto estratto istanza di **IlcIntVar**. Gli oggetti così creati contengono le stesse informazioni dell'oggetto modellante, ma in una forma adatta ad essere gestita dall'algoritmo di ricerca che sarà successivamente utilizzato. Mentre gli oggetti di un modello sono allocati in un'area di memoria che viene reclamata solo al termine della sessione di lavoro, gli oggetti estratti sono memorizzati nell'heap, dove è possibile inserire reversibilmente informazioni aggiuntive, che verranno poi liberate quando la risoluzione ritorna, in backtracking, ad un punto precedente all'inserimento dell'informazione stessa in memoria.

Sebbene possano esserci delle eccezioni, in generale si può affermare che ad ogni oggetto IloExtractable presente in un modello, corrisponde un oggetto nel risolutore.

5.4 Risoluzione di un problema estratto

Una volta portata a termine l'estrazione, è possibile passare alla fase di risoluzione vera e propria. La risoluzione di un problema consiste nella selezione di un valore nel dominio di ogni variabile in modo che tutti i vincoli siano soddisfatti. Inoltre, Solver può essere usato per trovare una soluzione che ottimizzi un dato criterio. Per problemi semplici, gli algoritmi generici predefiniti nella libreria Solver sono sufficienti. Per problemi più complicati, è possibile sia definire nuovi algoritmi di ricerca, sia modificare quelli predefiniti per adeguarli alle necessità del problema in questione.

Un particolare tipo di oggetto, chiamato **IloGoal**, può essere aggiunto ad un modello per specificare in che modo l'algoritmo di ricerca debba essere implementato. Esistono numerose classi che ereditano da IloGoal, ognuna delle quali specifica un diverso metodo di ricerca. Infatti, diversamente dagli algoritmi predefiniti forniti da Solver, queste istanze implementano algoritmi in cui l'esatta sequenza di operazioni da eseguire non è nota a priori, ma deve essere calcolata ad ogni passo computazionale.

Al contrario di tutti gli oggetti descritti fino ad ora, gli oggetti di tipo goal non vengono estratti: tramite un meccanismo simile a quello dell'estrazione, viene creata un'istanza di IlcGoal, ma solo per un uso interno del risolutore. La differenza principale tra le due tipologie di estrazione risiede nel fatto che l'estrazione di un oggetto IloExtractable provoca anche l'estrazione di tutti gli eventuali oggetti dello stesso tipo che esso contiene, cosa che l'estrazione di un oggetto IloGoal non fa. Per esempio, estraendo il vincolo (X<Y), dove X e Y denotano variabili numeriche intere, vengono creati sia il corrispondente IlcConstraint, sia i due IlcIntVar che rappresentano X e Y, se non erano già stati creati da una precedente estrazione. Viceversa, se X e Y appaiono all'interno di un goal, devono essere aggiunti esplicitamente al modello per essere estratti e non incorrere in un errore a tempo di esecuzione.

5.4.1 Le funzioni IloInstantiate e IloGenerate

Queste funzioni creano e ritornano un goal. La funzione **IloInstantiate** accetta tra i suoi argomenti una variabile e implementa un criterio di selezione del valore sulla variabile stessa. Qualora quest'ultima fosse già stata legata ad un valore, questo goal non fa niente e restituisce un successo, altrimenti predispone un punto di scelta e altera il dominio della variabile tentando ricorsivamente di assegnarle un valore ammissibile. Il modo di scegliere i valori all'interno del dominio può essere specificato dal programmatore.

La funzione **IloGenerate** serve per assegnare i valori ad un insieme di variabili. Accetta un array di variabili e, facoltativamente, due criteri: il primo per la selezione della variabile a cui assegnare il valore, il secondo per la selezione del valore da assegnare alla variabile scelta. IloGenerate tenta di assegnare un valore ad ogni variabile dell'array. Per fare questo chiama la funzione IloInstantiate su ogni variabile, specificando il criterio di selezione dei valori. L'ordine in cui le variabili sono considerate è deciso per mezzo del primo criterio.

Esistono anche le funzioni **IloBestInstantiate** e **IloBestGenerate**. La differenza tra il goal generato dalla funzione IloInstantiate e quello generato da IloBestInstantiate, se applicate agli stessi argomenti, risiede nel numero dei valori scelti. Infatti, dopo aver ordinato i valori secondo il criterio prescelto, il primo goal li prova tutti fino a che non ottiene una soluzione, mentre il secondo goal prova solo il primo valore e, se dovesse fallire, restituisce subito un insuccesso. Analogamente, IloGenerate sceglie la variabile cui assegnare un valore e, se possibile, lo assegna, mentre IloBestGenerate sceglie una variabile, prova il primo valore e, se non ha successo, elimina tale valore dal dominio e applica nuovamente il criterio di scelta della variabile.

Appare evidente che il secondo metodo è più oneroso dal punto di vista computazionale, applicando una funzione di selezione dinamica che comporta, come descritto nel primo capitolo, un maggior impegno di

risorse per valutare ad ogni passo la strada da seguire. Occorre quindi verificare attentamente, caso per caso, che il tempo risparmiato per aver seguito un'euristica migliore sia maggiore di quello impegnato dall'euristica stessa per calcolare i percorsi nell'albero di ricerca.

Sia i criteri per la selezione delle variabili, sia quelli per la selezione dei valori, possono essere personalizzati dall'utente. Esistono comunque alcuni criteri predefiniti, come per esempio la funzione per la selezione della variabile **IlcChooseMinSizeInt**, che ordina le variabili in ordine crescente della cardinalità del dominio.

È possibile anche applicare la funzione IloGenerate ad un array senza indicare nessun criterio. In questo caso le variabili vengono scelte in funzione dell'indice nell'array, e i valori del dominio dal minore al maggiore.

5.4.2 La funzione IloApply

Questa funzione riceve come argomenti un goal e un algoritmo di ricerca e restituisce un nuovo goal. La differenza tra il goal in ingresso e quello in uscita risiede nel modo di esplorare l'albero decisionale. Infatti, mentre per risolvere il goal iniziale il risolutore avrebbe esplorato lo spazio associato al problema seguendo la DFS, l'esplorazione del goal fornito dalla **IloApply** avviene secondo il criterio passato come argomento. Solver implementa delle funzioni che rappresentano gli algoritmi di ricerca più utilizzati, tra cui quelli descritti nei capitoli precedenti. Più precisamente, ogni funzione predefinita ritorna un'istanza di **IloNodeEvaluator**, una classe che descrive un metodo di ordinamento, quindi di selezione, dei nodi dell'albero.

Solver implementa gli algoritmi BFS, DDS, DFS, IDFS e LDS. È tuttavia possibile, qualora se ne presenti la necessità, modificare queste procedure o implementarne di nuove.

5.5 Definizione di nuove classi di vincoli

Nonostante la presenza di numerosi vincoli predefiniti, talvolta può essere necessario definire nuovi vincoli, tipicamente per poter effettuare delle modifiche ai domini delle variabili in gioco in maniera più efficace, racchiudendo così all'interno di un singolo metodo tutte le azioni che devono essere eseguite al presentarsi di un determinato evento. Un vincolo, quindi la sua propagazione, deve essere associato alla variazione di uno o più domini delle variabili presenti nel modello estratto. Tutti gli oggetti creati dall'estrazione di una variabile possiedono tre metodi, tramite i quali è possibile associare a tre diversi eventi dei vincoli. Questi metodi, che accettano come argomento il vincolo, sono:

- **whenValue**, che propaga il vincolo ogni volta che un valore viene assegnato alla variabile;
- **whenRange**, che propaga il vincolo ogni volta che i limiti inferiore e superiore del dominio della variabile vengono modificati;
- **whenDomain**, che propaga il vincolo ogni volta che il dominio della variabile viene modificato.

La creazione di un nuovo vincolo consiste semplicemente nel ridefinire due metodi virtuali, *propagate* e *post*, della classe astratta **IlcConstraintI**, da cui ereditano tutti gli oggetti che definiscono dei vincoli. Il primo metodo implementa le invarianti di vincolo, ossia le azioni da eseguire ogni volta che il vincolo viene invocato; il secondo definisce l'evento cui associare il vincolo, cioè l'esecuzione del primo metodo. Tramite delle macro, l'incombenza dell'allocazione e gestione della memoria necessaria alla propagazione dei vincoli, è demandata al preprocessore del linguaggio.

Dato che la propagazione di un vincolo può eventualmente consistere nella modifica di domini di altre variabili, deve essere possibile accedere direttamente alle variabili estratte. Questo è necessario perché, una volta estratto il modello, qualsiasi modifica al modello stesso viene vista dal

risolutore solo in conseguenza di una nuova estrazione, cioè all'inizio di una nuova ricerca. Per accedere alle variabili estratte, la classe IloSolver fornisce una serie di metodi, tutti con prefisso *get*, che restituiscono un riferimento all'oggetto estratto associato all'oggetto modellante passato come argomento. Ogni variabile estratta possiede infine dei metodi, con prefisso *set*, per modificarne il dominio.

CAPITOLO 6

PROGETTI E ALGORITMI PER LA DETERMINAZIONE DEL VINCITORE IN UN'ASTA COMBINATORIA

6.1 Introduzione

In questo capitolo sarà descritto il sistema software creato per determinare, tramite la programmazione a vincoli, il vincitore di un'asta combinatoria. Questo progetto utilizza le librerie ILOG descritte nel precedente capitolo, ed è quindi implementato utilizzando il linguaggio C++.

Prima di analizzare nel dettaglio le scelte progettuali e gli algoritmi utilizzati, saranno presentati i metodi per generare dei problemi standard di aste combinatorie. Sarà inoltre descritta un'ulteriore applicazione capace di risolvere tali problemi e confrontata con quella presentata nella tesi.

6.2 Generazione di problemi di aste combinatorie

In letteratura, il problema della determinazione del vincitore nelle aste combinatorie è già stato analizzato e risolto con vari algoritmi. Tra i tanti sistemi che generano istanze di aste, per questa tesi ne sono stati presi in considerazione 2, **MAGNET**[13] e **CATS**[14].

MAGNET (**Multi-AGent NEgotation Testbed**) è un sistema creato nell'Università del Minnesota per generare e risolvere problemi complessi di gestione e programmazione, utilizzando un innovativo approccio ad agenti. MAGNET è risultato un prodotto molto utile per questa tesi sia perché rende possibile modificare le caratteristiche dei problemi generati tramite alcuni parametri, sia, soprattutto, perché il sistema risolve i problemi generati. Accade spesso, infatti, che per paragonare i risultati ottenuti da diversi algoritmi eseguiti su macchine differenti, applicati allo stesso problema, occorra anche paragonare le prestazioni delle macchine utilizzate. In questo caso, invece, è possibile eseguire le prove sui medesimi problemi e sulla stessa macchina, rendendo quindi immediatamente paragonabili i risultati ottenuti.

CATS (**Combinatorial Auction Test Suite**) è un generatore di modelli di offerta realistici. È utile poiché consente di generare un'amplissima gamma di problemi. Purtroppo, a differenza di MAGNET, non risolve i problemi generati e, inoltre, non associa alcuna informazione temporale agli oggetti messi all'asta. Questo problema è stato superato intervenendo sul sorgente di MAGNET (scritto in Java e gentilmente concesso da **J. Collins** e **M. Gini**[1]) e prelevando le informazioni temporali relative alle attività generate. I problemi creati con CATS sono quindi, in realtà, ottenuti associando alle attività contenute nelle offerte generate da CATS, i parametri e i vincoli temporali estrapolati da MAGNET. La principale motivazione per cui sono stati presi in considerazione anche i problemi generati con CATS è che MAGNET fornisce offerte composte da pochi oggetti, tipicamente 1 o 2; questo è troppo limitativo per la verifica dell'algoritmo progettato. In fase di testing del sistema è stata quindi trovata soluzione sia agli stessi problemi generati e risolti da MAGNET, paragonandone poi i risultati, sia a problemi più articolati e particolareggiati generati con CATS.

6.3 Algoritmi di risoluzione utilizzati da MAGNET

MAGNET implementa, per risolvere i problemi, due motori di ricerca basati su due diversi modelli, una versione rivista e migliorata della **Integer Programming** (IP), e la **Simulated Annealing** (SA).

6.3.1 Integer Programming

IP[15] si basa sull'utilizzo di vincoli lineari e di variabili che possono assumere soltanto valori interi. In questo caso, le variabili possono valere soltanto 0 o 1, e possono quindi essere anche viste come variabili booleane, considerando falso il valore 0 e vero il valore 1. Ad ogni offerta i è associata una variabile di questo tipo x_i. Se x_i vale 0 l'offerta

[1] Department of Computer Science and Engineering, University of Minnesota

è rifiutata, altrimenti è accettata. Ogni vincolo associato al problema sarà espresso tramite queste variabili.

Il problema in questione può essere caratterizzato da 3 insiemi: l'insieme delle offerte ricevute, l'insieme delle attività messe all'asta e, per ogni elemento di questo insieme, l'insieme delle attività che lo devono precedere temporalmente. In maniera più formale, un problema consiste in un insieme ***S*** di attività s_j, j=1..m; per ogni s_j, un insieme $P_j = \{s_{j'} \mid s_{j'} \prec s_j\}$; un insieme ***B*** di offerte b_i, i=1..n. Ogni offerta b_i specifica un insieme di attività S_i, un prezzo p_i e, per ogni attività s_j inclusa in S_i, una tripla di valori $\left(d_j^i, e_j^i, l_j^i\right)$ che rappresentano rispettivamente la durata e il minimo e il massimo istante iniziale offerti.

I vincoli sono espressi come segue:

- Minimizzazione del costo totale:

$$\sum_{i=1}^{n} p_i x_i$$

- Copertura: ogni attività s_j deve essere comprata una sola volta

$$\forall j = 1..m \quad \sum_{i \mid sj \in Si} x_i = 1$$

- Consistenza locale: ogni attività s_j deve poter cominciare dopo il minimo istante di terminazione possibile di ogni suo predecessore $s_{j'}$

$$\forall j = 1\ldots m, \quad \forall i \mid s_j \in \mathbf{S}_i, \quad \forall i' \mid s_{j'} \in (\mathbf{S}_{i'} \cap \mathbf{P}_j),$$

$$x_i f_j^i \geq x_{i'}(e_{j'}^{i'} + d_{j'}^{i'}) - 10^{12}(1 - x_i)$$

dove l'ultimo termine serve per rendere soddisfatto il vincolo quando l'offerta è rifiutata (x_i=0).

- Consistenza globale: si combinano insieme i vincoli di consistenza locale; infatti, come esemplificato nel quarto capitolo, tre attività che a due a due rispettano i vincoli temporali, possono non rispettarli più se combinate assieme.

Questa formulazione del problema è corretta, ma MAGNET la migliora basandosi su alcune osservazioni:

- Se un'attività è contenuta in una sola offerta, tale offerta deve necessariamente far parte della soluzione. Inoltre, ogni altra offerta con cui essa entri in conflitto può essere scartata a priori.
- I vincoli di consistenza locale possono essere semplificati calcolandoli in fase di preprocessing. In altre parole, se esiste un vincolo del tipo $x_i f_j^i \geq x_{i'}\left(e_{j'}^{i'} + d_{j'}^{i'}\right) - 10^{12}\left(1 - x_i\right)$, si calcola $f_j^i - \left(e_{j'}^{i'} + d_{j'}^{i'}\right)$. Soltanto nel caso in cui il risultato sia negativo, si aggiunge il vincolo $x_i + x_{i'} \leq 1$. In questo modo x_i e $x_{i'}$ non possono far parte della stessa soluzione. Una semplificazione simile può essere applicata ai vincoli di consistenza globale, ottenendo delle formule che assicurano che, in un insieme di *n* offerte, al più *n-1* possano far parte della soluzione.
- Se all'interno di una stessa offerta due o più attività non rispettano i vincoli di consistenza, tale offerta non potrà far parte di nessuna soluzione; si aggiunge quindi il vincolo $x_i + x_i \leq 1$.

6.3.2 Simulated Annealing

SA[16] inserisce i nodi dell'albero di ricerca in una lista, di lunghezza massima prestabilita, ordinata secondo un certo algoritmo. Le caratteristiche di SA sono una temperatura di annealing T, che è periodicamente ridotta di un fattore ε, e una procedura stocastica di selezione del nodo. Se si suppone che ad ogni nodo dell'albero di ricerca siano associate 2 liste, una lista tabù che contiene tutte le offerte che non possono far parte delle eventuali soluzioni posizionate sotto il nodo, e una lista delle espansioni tentate che contiene le offerte già analizzate, l'algoritmo può essere descritto a grandi linee come segue:

1. **Inizializzazione della ricerca**:
   1.1 **Pre-Processing delle offerte**: per ogni attività generare una lista delle offerte che la includono e il prezzo offerto medio.
   1.2 **Copertura**: se un'attività non ha alcuna offerta, **uscire**.

1.3 **Singola offerta**: se un'attività ha una singola offerta, allora questa deve far parte di ogni soluzione. L'offerta può contenere una o più attività. Creare il nodo, o i nodi, che mappano l'offerta, calcolarne il valore e aggiungerli alla coda.

1.4 **Inizializzazione della coda**: se non esistono singletons creare un nodo mappando ogni attività a nessuna offerta, e aggiungerlo alla coda.

1.5 **Inizializzazione della temperatura di annealing**.

2. **while not (**timeout **and** soluzione_migliorata) **do**:

2.1 **Selezione di un nodo N per l'espansione**: selezionare un numero random $R=V_{min} - T \ln(1-r)(V_{max}-V_{min})$, dove:

-r è un numero casuale uniformemente distribuito tra 0 e 1,

-T è la corrente temperatura di annealing, e

-V_i è il valore del nodo i-esimo nella coda. I nodi sono ordinati per valori crescenti, quindi V_{min} è il valore del primo nodo.

Scegliere come nodo N l'ultimo nodo della coda per cui $V_N \leq R$.

2.2 **Selezione di un'offerta B**:

Scartare tutte le offerte che appaiono nella lista tabù di N.

Scartare tutte le offerte che sono già state usate per espandere N.

Scegliere un'offerta secondo la corrente politica di selezione.

2.3 **Espansione di N con l'offerta B, per generare il nodo N'**:

Per ogni attività contenuta nell'offerta B che è già stata associata ad un offerta B', rimuovere B'.

Se B' è un singleton, **abbandonare l'espansione**.

Aggiungere B alla lista delle espansioni tentate associata ad N.

Copiare la lista tabù di N in quella di N' e aggiungere B in cima.

Troncare la lista tabù di N' alla dimensione massima.

2.4 **Valutazione del nodo N'**:

$V_N=Cost_N+Feas_N+Cov_N$, dove:

-$Cost_N$ è la somma dei costi delle offerte (alle attività non allocate è assegnato il valore medio),

-$Feas_N$ è il valore pesato del costo previsto per tornare indietro da un fallimento, e

-Cov_N è il valore medio pesato del numero di attività da allocare.

2.5 **Aggiornamento delle informazioni sulla migliore offerta**.

2.6 **Modifica del valore della temperatura di annealing T**.

3. **Restituzione del miglior nodo trovato**.

Per quanto riguarda i criteri di selezione delle offerte, utilizzati al punto 2.2, MAGNET ne considera 4: uno che sceglie a caso e 3 che cercano di scegliere l'offerta che, verosimilmente, porti ad un miglioramento rispettivamente di copertura, possibilità di fattibilità e costo.

6.4 Determinazione del vincitore di un'asta combinatoria tramite la programmazione a vincoli

Il programma implementato in questa tesi si compone principalmente di 2 parti: una prima parte in cui si definiscono i dati del problema da risolvere, e una seconda parte in cui il problema viene analizzato e risolto dal modello creato utilizzando i vincoli e i metodi ILOG.

6.4.1 Generazione dei problemi

Per generare un problema, come descritto nel paragrafo precedente, sono stati utilizzati 2 diversi sistemi. Il codice che si occupa di generare i problemi è diverso a seconda che si utilizzi MAGNET o CATS.

MAGNET, in origine, memorizza tutti i dati del problema e tutte le operazioni eseguite dall'algoritmo risolutivo in un unico file di testo, rendendo quindi particolarmente complicata l'estrazione dei soli dati utili. Il sorgente di MAGNET è stato quindi modificato aggiungendo delle istruzioni per memorizzare in un file a se stante i dati necessari

alla completa specificazione del problema, in pratica offerte e vincoli di precedenza. MAGNET non permette una buona specifica del problema, infatti gli unici parametri che è in grado di ricevere sono il numero di offerte e di attività. È in realtà anche possibile diversificare attività ed offerte mediante dei file di configurazione. Si possono descrivere più tipologie di attività, specificandone durata media, costo medio e percentuale approssimata di presenza nelle offerte. Per quanto concerne i parametri delle offerte, può essere specificata la dimensione media, come numero di attività contenute, e la deviazione rispetto a tale valore medio. Purtroppo tali parametri incidono poco sull'algoritmo di generazione, e il risultato è che tutti i problemi si assomigliano, in particolare il numero di attività per offerta è tipicamente 1, sporadicamente 2 e molto raramente un numero superiore.

CATS, invece, è implementato in C++, ed è quindi stato immediato inserire il codice che genera il problema, contenuto nel file *Scheduling.cpp*, all'interno del sistema. Diversamente da MAGNET, CATS permette di specificare con più precisione la forma dell'asta. Principalmente, il numero medio di attività contenute in un'offerta e la rispettiva deviazione sono parametri molto vincolanti all'interno del generatore. CATS consente di specificare un gran numero di parametri; agendo su di essi è possibile creare problemi che rappresentino molte situazioni reali, non solo di aste combinatorie ma anche, per esempio, di allocazione di tratte audio, di utilizzo di appezzamenti di terreno adiacenti, di problemi di routing, ecc. Anche utilizzando CATS, è tuttavia necessario generare un problema con MAGNET da cui prelevare le finestre temporali, le durate e i vincoli di precedenza da associare alle attività create da CATS. Il meccanismo utilizzato prevede di generare, tramite MAGNET, un problema con lo stesso numero di attività contenute in quello generato da CATS, ma con un numero di offerte molto maggiore, tipicamente un ordine di grandezza. Per ogni attività, verranno ricavate da tutte le offerte MAGNET che la contengono, le informazioni temporali. Queste informazioni saranno poi ordinatamente

assegnate alle corrispondenti attività CATS. Il maggior numero di offerte di MAGNET rispetto a CATS è dovuto al fatto che ogni offerta MAGNET contiene poche attività. Se la somma delle attività di MAGNET fosse minore della somma di quelle di CATS, occorrerebbe associare a più attività le stesse informazioni, diminuendo quindi la casualità del problema generato.

I dati generati dai 2 sistemi sono infine inseriti in due array e passati come argomenti al risolutore, contenuto in *Solver.cpp*. Ogni elemento del primo array è una struttura, chiamata *Bid*, che contiene le informazioni sulle offerte, nome, costo, numero di attività e, per ogni attività, una ulteriore struttura, *Task*, con nome, minimo e massimo istante di inizio, durata.

```
typedef struct Task {IloInt name;
                     IloInt eStart;
                     IloInt lStart;
                     IloInt duration;
                    }Task;

typedef struct Bid {char name[7];
                    IloInt price;
                    IloInt tasks_n;
                    Task tasks[MAX_TASKS_PER_BID];
                   }Bid;
```

Si può notare che tutte le variabili sono intere. Questo non è un limite poiché, qualora un problema presenti dei valori frazionari, è sufficiente moltiplicarli tutti per un numero che li renda interi, e dividere poi il risultato per lo stesso numero. L'utilizzo di numeri interi semplifica molto la risoluzione del problema; questo è motivato dal fatto che ILOG crea dei domini discreti per le variabili intere e continui per quelle reali, frazionarie incluse. Al momento della propagazione dei vincoli, è molto più semplice eliminare un elemento da un insieme discreto piuttosto che un intervallo di valori da un insieme continuo, soprattutto considerando

che occorre tener traccia di tutte le operazioni eseguite sui domini, per tornare alla situazione precedente in fase di backtracking.

Il secondo array passato come argomento al risolutore contiene i vincoli di precedenza. È strutturato come un'array di array di interi. Ogni array di interi (*array[n]*) si riferisce ad un'attività, e gli elementi di tale array (*array[n][i]*) indicano le attività che la devono precedere temporalmente. Essendo una struttura creata staticamente, per riconoscere la fine di un array l'ultimo elemento è sempre il numero –1.

6.4.2 Risoluzione dei problemi: variabili utilizzate

I dati del problema sono inseriti in 6 array di variabili ILOG:

- IloIntVarArray **Start**; il dominio iniziale dell'n-esima variabile dell'array contiene tutti i possibili istanti iniziali dell'n-esima attività. Questi valori sono ottenuti facendo l'unione, per ogni attività, di tutte le finestre temporali di possibile inizio fornite da ogni offerta che contenga l'attività stessa.
- IloIntVarArray **Duration**; analogamente a Start, ogni variabile contiene tutti i possibili valori della durata di ogni singola attività.
- IloIntVarArray **Bids**; il dominio iniziale dell'n-esima variabile contiene i valori di tutte le offerte che comprendono l'n-esima attività.
- IloIntVarArray **Tasks_per_bid**; l'n-esima variabile, che rappresenta l'n-esima offerta, vale il numero di attività che ogni offerta può ottenere. In accordo con le ipotesi di progetto, secondo cui ogni attività può essere o rifiutata o accettata in toto, tale valore può essere o 0 o il numero di attività contenute nell'offerta. I domini iniziali di ogni variabile contengono quindi questi due valori solamente.
- IloIntVarArray **Bid_Cost**; analogamente a Tasks_per_bid, può valere o 0 o il prezzo dell'offerta.

- IloIntVar **Cost**; variabile che rappresenta il costo totale dell'asta. Inizialmente il dominio va da 0 al prodotto tra il numero di offerte e il costo massimo di ognuna, definito da MAX_PRICE.

I primi tre array hanno una dimensione pari al numero di attività, mentre gli altri due al numero di offerte.

Oltre a questi array di variabili, appartengono al sistema anche 3 array di insiemi di valori, che non saranno modificati dal risolutore, ma che contengono i valori iniziali dei primi 3 array di variabili precedentemente descritti. Questi array, istanze di IloNumSetVarArray, sono rispettivamente Possibile_start, Possibile_duration e Possible_bids.

Per poter accedere direttamente alla rappresentazione interna delle variabili in fase di propagazione dei vincoli, dopo aver inizializzato gli array, si esegue esplicitamente l'estrazione degli oggetti sopra descritti. Dato che ILOG non permette di inizializzare le variabili con domini discontinui, i domini delle variabili risultato dell'estrazione andranno dal minimo al massimo valore che la variabile può avere, inclusi quindi tutti gli eventuali valori interni a cui essa non può essere legata. Inoltre, la funzione di selezione ordinerà le variabili in funzione della cardinalità del dominio; la presenza di valori impossibili da assegnare inficerebbe quindi l'efficacia dell'algoritmo. Occorre perciò eliminare staticamente, cioè prima di iniziare la ricerca con il metodo newSearch dell'istanza di IloSolver creata, tali valori dai domini. Per far questo è sufficiente eliminare tutti i valori contenuti nelle variabili estratte e non contenuti nelle corrispondenti istanze di IloNumSetVarArray. Ancora più semplice è eliminare i valori impossibili da Tasks_per_bid e Bid_Cost; vanno eliminati tutti i valori tranne 0 e, rispettivamente, il numero di attività e il costo dell'offerta. Nel seguito viene riportato il codice che implementa i concetti e le metodologie introdotti.

```
IloIntVarArray Start(env,tasks_n,0,MAX_START);
IloNumSet *Initial_Start=new IloNumSet[tasks_n];
```

```
for (IloInt t=0;t<tasks_n;t++)
  {IloInt* S_array=new IloInt[bids_n*
                              (MAX_START-MIN_START)];
   IloInt* S_values=new IloInt[bids_n*
                               (MAX_START-MIN_START)];
   IloInt S_cont=0;
   IloInt i;
   for (IloInt b=0;b<bids_n;b++)
      if (((i=contains_task(bids[b],t))!=-1)&&
                        (i<bids[b].tasks_n))
        {IloInt min=bids[b].tasks[i].eStart;
         IloInt max=bids[b].tasks[i].lStart;
         for (IloInt s=min;s<=max;s++)
            S_array[S_cont++]=s;
        }
   qsort(S_array,S_cont,sizeof(IloInt *),compare);
   IloInt S_pos=0;
   for (IloInt i=0;i<S_cont-1;i++)
      if (S_array[i]!=S_array[i+1])
         S_values[S_pos++]=S_array[i];
   S_values[S_pos]=S_array[S_cont-1];
   Start[t]=IloIntVar(env,S_values[0],S_values[S_pos]);
   IloIntArray S_temp(env,0);
   for (IloInt i=0;i<=S_pos;i++) S_temp.add(S_values[i]);
   Initial_Start[t]=IloNumSet(env,S_temp);
   free(S_array);
   free(S_values);
  }

IlcIntVarArray IlcStart=solver.getIntVarArray(Start);
for (IloInt t=0;t<tasks_n;t++)
   for (IloInt i=0;i<MAX_START;i++)
      if (!Initial_Start[t].contains(i))
         IlcStart[t].removeValue(i);
```

S_array contiene tutti i valori, eventualmente anche ripetuti, dei possibili istanti iniziali. *S_values* contiene gli stessi valori senza ripetizione e ordinati dal minore al maggiore. Per ordinare il vettore

S_array si utilizza la funzione predefinita *qsort*; l'argomento *compare* è una funzione che, chiamata con 2 argomenti numerici, restituisce un valore positivo, nullo o negativo, a seconda che il primo argomento sia rispettivamente maggiore, uguale o minore del secondo.

```
int compare(const void *arg1,const void *arg2)
   {return ((*(IloInt *)arg1)-(*(IloInt *)arg2));}
```

Ad ogni iterazione t, gli elementi di *S_array* vengono aggiunti al dominio di *Initial_Start[t]*. Infine vengono eliminati dal dominio di *IlcStart[t]*, che inizialmente contiene tutti i numeri compresi tra il minimo e il massimo, i valori che non compaiono nell'insieme *Initial_Start[t]*. Analogamente sono ridotti i domini di *Duration* e *Bids*.

Per ridurre i domini di *Bid_Cost* e di *Tasks_per_bid* si eliminano tutti i valori tranne 0 e il costo, o il numero di offerte. Non occorre creare gli insiemi che memorizzino i valori iniziali.

```
IloIntVarArray Bid_Cost(env,bids_n,0,MAX_PRICE);
IlcBid_Cost=solver.getIntVarArray(Bid_Cost);

for (b=0;b<bids_n;b++)
   for (IloInt val=1;val<=MAX_PRICE;val++)
      if (val!=bids[b].price)
        IlcBid_Cost[b].removeValue(val);
```

Durante la riduzione delle variabili *Bids*, qualora il dominio si riduca ad un solo valore, sono anche istanziate le corrispondenti variabili *Bid_Cost* e *Tasks_per_bid*; siamo infatti in presenza di un singleton, quindi l'offerta associata deve necessariamente far parte della soluzione.

6.4.3 Risoluzione dei problemi: vincoli e metodi di propagazione dei vincoli

I vincoli da aggiungere al modello riguardano principalmente i valori contenuti nei domini delle variabili e le relazioni temporali tra le

attività. I vincoli sui domini sono automaticamente soddisfatti al momento della creazione delle variabili come descritto in precedenza. I vincoli temporali vanno invece esplicitamente aggiunti al modello.

```
for (t=0;t<tasks_n;t++)
  {IloInt j=-1;
   while (pred_array[t][++j]!=-1)
      model.add(Start[t]>=Start[pred_array[t][j]]+
                Duration[pred_array[t][j]]);
  }
```

pred_array è la struttura che contiene tutti i vincoli di precedenza tra le attività.

Un altro importante vincolo del problema è che le offerte possono essere accettate soltanto nella loro totalità. Questo, come già detto, significa che la variabile Tasks_per_bid[n] può valere solo 0 o il numero di attività contenute nell'n-esima offerta. Per trasformare questi vincoli in una forma elaborabile da ILOG, è sufficiente aggiungere al modello il vincolo IloDistribute chiamato sugli array Tasks_per_bid e Bids.

```
model.add(IloDistribute(env,Tasks_per_bid,Bids));
```

Il vincolo impone che il numero di occorrenze di un certo valore *n* nell'array Bids sia uguale al valore dell'*n*-esima variabile dell'array Tasks_per_bid. Essendo l'offerta che ottiene l'*n*-esima attività rappresentata dal valore dell'*n*-esima variabile di Bids, appare evidente come l'utilizzo della funzione IloDistribute colga esattamente il significato del vincolo originario del problema.

Nonostante ILOG implementi ottimi algoritmi di esplorazione, questi possono essere migliorati aggiungendo al modello dei metodi di propagazione dei vincoli ottimizzati per il caso in oggetto. In particolare, ogni volta che il dominio di una variabile Bids si riduce ad un solo valore, sia in seguito ad un assegnamento sia in seguito alla propagazione di altri vincoli, viene chiamato il metodo *propagate* di un

vincolo istanza di IlcConstraint, legato alla variabile istanziata tramite il metodo *whenValue*. Questo vincolo modifica i domini di tutte le variabili che, pur essendo logicamente connesse con la variabile appena istanziata, non subiscono alcuna variazione in seguito all'evento, non esistendo nessun vincolo ILOG tra questi oggetti.

Sfruttando le macro predefinite la ILOG, la generazione di un nuovo vincolo è estremamente semplice. Occorre soltanto implementare due metodi, *post* e *propagate*, chiamare la macro per creare l'oggetto, sottoclasse di IlcConstraint, che racchiuda le specifiche del nuovo vincolo, e infine aggiungerla, utilizzando il metodo *add*, al risolutore.

Il metodo post associa ad ogni variabile, o array di variabili, il vincolo da eseguire all'occorrenza di un determinato evento, in questo caso la riduzione del dominio ad un singolo valore.

```
void IlcWhenValue_Bid_ConstraintI::post()
  {IloSolver solver=getSolver();
   for (IlcInt i=0;i<IlcBids.getSize();i++)
     IlcBids[i].whenValue(Constraint_name(solver,this,i));
  }
```

Constraint_name è il nome passato alla macro per identificare il vincolo.

La funzione propagate, che implementa il metodo di propagazione dei vincoli, inizia recuperando le informazioni sulla variabile legata e il rispettivo valore.

```
void IlcWhenValue_Bid_ConstraintI::propagate(IlcInt index)
  {IloSolver solver=getSolver();
   IlcInt value=IlcBids[index].getValue();
   IlcInt num=0;
   while (bids[value].tasks[num++].name!=index);
      num--;
```

Nel codice *index* riferisce la variabile istanziata e *value* il valore assegnatole. In altre parole, *index* è l'attività e *value* l'offerta cui è stata assegnata la sua l'esecuzione.

Avendo specificato quale offerente eseguirà un'attività, è possibile ridurre anche i domini di Start e Duration associati a tale attività, riducendoli ai soli valori permessi dall'offerta *value*. *num* è l'indice dell'attività *index* all'interno dell'offerta *value*.

```
IlcIntSet possible_start(solver,
            bids[value].tasks[num].eStart,
            bids[value].tasks[num].lStart,IlcTrue);
IlcStart[index].setDomain(possible_start);
IlcDuration[index].setValue(bids[value].tasks[num].
                            duration);
```

Viene creata una variabile temporanea che rappresenta la finestra temporale di possibile inizio dell'attività specificata dall'offerta. A tale finestra è posto uguale il dominio della variabile Start dell'attività; eventuali valori contenuti nella finestra, ma già eliminati dal dominio a causa di propagazioni precedenti, ne restano comunque esclusi. Analogamente viene ridotto il dominio di Duration.

L'offerta è vincente, quindi le due variabili Tasks_per_bid e Bid_cost ad essa associate varranno rispettivamente il numero di attività richieste e il prezzo proposto dall'offerente. Inoltre tutte le altre attività dovranno essere assegnate a questa offerta, e i loro domini di Start e Duration modificati di conseguenza.

```
IlcTasks_per_bid[value].setValue(bids[value].tasks_n);
IlcBid_Cost[value].setValue(bids[value].price);

IlcIntSet sold_tasks(solver,0,tasks_n-1,IlcFalse);
  for (IlcInt t=0;t<bids[value].tasks_n;t++)
    {sold_tasks.add(bids[value].tasks[t].name);
     if (t==num) continue;
     IlcBids[bids[value].tasks[t].name].setValue(value);
```

```
    possible_start=IlcIntSet(solver,
                bids[value].tasks[t].eStart,
                bids[value].tasks[t].lStart,IlcTrue);
    IlcStart[bids[value].tasks[t].name].
                setDomain(possible_start);

    IlcDuration[bids[value].tasks[t].name].
                setValue(bids[value].tasks[t].duration);
}
```

Ogni offerta che contenga perlomeno un'attività già assegnata ad un'altra offerta, può essere eliminata dalla soluzione. Tuttavia, per poterlo fare, occorre eseguire due cicli innestati, uno sulle offerte e uno sulle attività che contengono; se, inoltre, si volessero eliminare dai domini temporali delle attività i valori consentiti soltanto da offerte ormai eliminate, occorrerebbe eseguire due ulteriori cicli innestati, uno sulle attività dell'offerta eliminata e uno su tutte le altre offerte, per verificare se i valori temporali associati all'attività dalla prima offerta sono forniti anche da qualche altra, ed in caso contrario eliminarli. Si può notare come il carico computazionale introdotto da questa analisi aumenti notevolmente la complessità dell'algoritmo.

Fortunatamente, il vincolo aggiunto tramite IloDistribute, crea una struttura a grafo internamente al risolutore che ottimizza la propagazione sopra descritta. Infatti, inizialmente, ogni dominio di Tasks_per_bid contiene 0 e un numero *n*, che è anche uguale al numero di occorrenze di *n* nei domini delle variabili Bids. Al momento in cui ad una variabile Bids viene assegnato un valore, tutti gli altri valori del dominio vengono eliminati; a questo punto il numero di occorrenze dei valori eliminati negli altri domini sarà al più *n-1*. Considerando che il dominio di Tasks_per_bid conteneva soltanto 0 ed *n*, risultando impossibile il secondo valore, la variabile viene legata a 0 e tutte le occorrenze di *n* nei domini di Bids eliminate.

Questo metodo non elimina i valori ormai divenuti impossibili dai domini delle variabili *Start* e *Duration*, tuttavia sperimentalmente si ricava che l'utilizzo di *IloDistribute* fornisce risultati mediamente 2 volte migliori rispetto ad un algoritmo che propaghi i vincoli tramite la *propagate* sopra descritta. Questo può essere spiegato anche considerando che gli algoritmi di selezione di variabili e valori non tengono in considerazione i valori dei domini di *Start* e *Duration*, quindi la loro riduzione non ha alcun effetto diretto sulla velocità di esecuzione di tali algoritmi. Oltretutto, a ulteriore conferma di ciò, quando il risolutore fornisce una soluzione, alla variabile Start non ha attribuito un solo valore, bensì un intervallo all'interno del quale è possibile iniziare ad eseguire l'attività. Le soluzioni restituite all'utente non contengono questa finestra ma solo il suo valore minimo.

Per ovviare a questo problema, un metodo sarebbe stato quello modificare il modello utilizzando 2 variabili per l'istante di inizio dell'attività, *EarlyStart* e *LateStart*, una che contiene tutti i possibili minimi istanti iniziali e l'altra i massimi. Tale soluzione ha tuttavia 2 problemi: innanzitutto il modello conterrebbe un'array di variabili in più, quindi, se *n* è il numero delle attività, *n* vincoli in più; inoltre, anche in questo caso ogni valore contenuto nei domini delle variabili può essere offerto più di una volta, quindi, nel momento in cui un'offerta viene rifiutata, occorrerebbe comunque eseguire una ricerca con due cicli innestati per eliminare eventuali valori impossibili contenuti nei domini di queste variabili. Introducendo 2 variabili *Start* aumenterebbero quindi i vincoli senza che questo apporti significative migliorie al modello. In definitiva, al momento in cui il risolutore restituisce una finestra temporale come insieme di tutti i possibili valori che una variabile *Start* può assumere, il minimo e il massimo istante iniziale possono semplicemente essere dedotti da tale finestra.

È noto che esistono degli eventi trigger analoghi alla whenValue, cioè i whenDomain e whenRange. Le verifiche sperimentali hanno tuttavia dimostrato che, utilizzando questi vincoli, dato che la propagazione dei

vincoli provocata da whenValue modifica molti altri domini, si ha un peggioramento delle prestazioni. Questo è dovuto al fatto che la propagazione dovuta a whenValue modifica i domini di *Start, Duration* e *Bids* di tutte le attività contenute nell'offerta dichiarata vincente. Ad ogni modifica di tali domini, corrisponderebbe un'attivazione del codice associato alla *whenDomain*, che verrebbe quindi eseguito molte volte per ogni evento *whenValue*, spesso senza riuscire ad effettuare riduzioni significative, comportando un aumento del tempo di computazione dovuto ai continui accessi agli spazi di memoria che allocano l'heap delle informazioni di backtracking e lo stack di attivazione delle procedure di riduzione dei domini.

In generale, si può affermare che l'algoritmo migliore è quello che propaga i vincoli in presenza di un assegnamento di un'attività ad un'offerta; in questo caso si eseguono solo ulteriori assegnamenti di valori ad altre variabili logicamente connesse con l'offerta considerata. Viceversa, la propagazione di un vincolo in seguito all'eliminazione di un'offerta, comporta l'esecuzione di codice con dei cicli innestati, quindi un aumento troppo marcato della complessità computazionale.

6.4.4 Risoluzione dei problemi: ottimizzazione e strategie per la selezione di variabili e valori

Obiettivo per la determinazione della soluzione ottima è minimizzare il costo dell'asta; per ottenerlo occorre aggiungere al risolutore l'obiettivo di rendere minima la variabile Cost, precedentemente vincolata a valere la somma dei valori delle variabili Bid_Cost.

```
model.add(Cost==IloSum(Bid_Cost));
model.add(IloMinimize(env,Cost));
```

Il goal da passare all'istanza di IloSolver è composto da 2 differenti goal in *and* tra loro. Il primo goal specifica i metodi di selezione delle variabili e dei rispettivi valori, mentre il secondo richiede di assegnare

sempre un valore alle variabili Bid_Cost, per verificare ad ogni passo risolutivo le modifiche subite da Cost, ed eventualmente recedere in backtracking. Ovviamente, qualora il dominio di una variabile Bid_Cost contiene ancora entrambi i valori, il risolutore, per trovare il costo totale, li considera entrambi per calcolare i nuovi limiti del dominio della variabile Cost; se il limite inferiore, cioè il valore ottenuto ponendo a 0 tutti i prezzi delle offerte non ancora elaborate, è comunque maggiore del costo di una soluzione già trovata, è inutile proseguire la ricerca in quel ramo dell'albero.

```
IlcIntSelect selector=SelectCheaper(solver);
IloGoal gol = IloGenerate(env,Bids,IlcChooseMinSizeInt,
              selector) && IloGenerate(env,Bid_Cost);
```

Per la selezione della variabile è stato scelto il metodo predefinito IlcChooseMinSizeInt, che ordina le variabili Bids in funzione della cardinalità del loro dominio, dalla minore alla maggiore. Si è scelto quindi di tentare di dare un valore per prime alle variabili più difficili da istanziare. Per la selezione del valore da assegnare alle variabili contenute in Bids, è stato creato un nuovo metodo, chiamato *SelectCheaper*, che ordina le offerte dalla più economica alla più costosa. Questa è la soluzione più efficace in quanto, essendo l'obiettivo quello di minimizzare il costo, appare evidente cercare la soluzione come combinazione di offerte economiche. Ad ogni tentativo di assegnamento di un valore, viene chiamato dal risolutore il metodo *select* della classe IlcIntSelect, che restituisce l'offerta migliore, quella più economica, tra quelle contenute nel dominio della variabile Bids[n]. Qualora tale valore non porti ad una soluzione, esso viene dinamicamente eliminato dal dominio e il risolutore procede quindi ad una nuova chiamata di *select*.

```
class IlcIntSelectCheaperI: public IlcIntSelectI
  {public:
```

```
    IlcIntSelectCheaperI() {}
    virtual IlcInt select(IlcIntVar var);
   };

IlcInt IlcIntSelectCheaperI::select(IlcIntVar var)
   {IloInt min=MAX_PRICE;
    IloInt val;
    for (IlcIntExpIterator it(var);it.ok();++it)
       if (bids[*it].price<min)
         {min=bids[*it].price;
          val=*it;
         }
    return val;
   }

IlcIntSelect SelectCheaper(IloSolver s)
   {return new (s.getHeap()) IlcIntSelectCheaperI();}
```

È stata utilizzata la funzione IloGenerate, invece di IloBestGenerate, perché, una volta selezionata la variabile cui assegnare un valore, è probabile che il primo valore che riesca a portare ad una soluzione non sia il migliore secondo l'euristica, a causa dell'elevato numero di vincoli imposti sul problema. In pratica, utilizzare la IloBestGenerate, e quindi IloBestInstantiate, porterebbe ad implementare una funzione euristica che tende troppo a minimizzare il costo lasciando in secondo piano la propagazione efficiente dei vincoli, il che porterebbe ad un maggior numero di backtracking.

Questo è stato anche verificato sperimentalmente notando che la IloBestGenerate fornisce dei risultati peggiori o, per problemi di semplice soluzione, identici, rispetto alla IloGenerate.

L'euristica scelta potrebbe non essere la migliore; è infatti probabile che una euristica **problem dependent**, cioè costruita appositamente per il problema in oggetto, riesca ad essere più efficace, guidando la ricerca verso risultati migliori, sia in termini di costo sia di tempo impegnato. Questa analisi può essere oggetto di un lavoro futuro.

6.4.5 Risoluzione dei problemi: esplorazione dello spazio di ricerca

Il risolutore è stato implementato in maniera da poter esplorare l'albero di ricerca associato al problema utilizzando 2 diversi algoritmi: DFS e LDS. Per applicare la DFS è sufficiente passare all'istanza di IloSolver il goal descritto nel sottoparagrafo precedente, viceversa per applicare la LDS occorre chiamare la funzione IloApply su tale goal, specificando il tipo di ricerca che si desidera implementare e, facoltativamente, nel caso della LDS lo step e il valore massimo delle discrepanze consentito, MaxDiscrepancy. Lo step è un valore intero che specifica il numero di discrepanze massime consentite per ogni iterazione della ricerca: alla k-esima iterazione, si esplorano le strade con un numero di discrepanze compreso tra *k*step* e *(k+1)*step –1*. Tutte le soluzioni con un numero di discrepanze superiore a MaxDiscrepancy vengono comunque ignorate.

Dalla analisi dell'algoritmo si nota che, in realtà, ILOG implementa una sorta di Improved LDS (ILDS), algoritmo descritto nel capitolo 2; infatti, l'ILDS esplora, ad ogni iterazione k, tutte le strade con esattamente k discrepanze. Se lo step fosse posto uguale ad 1, avremmo esattamente l'ILDS. Di default ILOG imposta questo valore a 4, mentre nel progetto in oggetto è possibile modificarlo. Si è infatti notato che, per problemi più ampi, un piccolo valore di step orienta l'esplorazione verso strade che difficilmente portano ad una soluzione, comportando un inutile spreco di tempo. Aumentando tale valore, assieme a queste strade, ne vengono esplorate altre con un maggior numero di discrepanze aumentando decisamente la probabilità di trovare subito una buona soluzione.

```
IloGoal gol = IloGenerate(env,Bids,IlcChooseMinSizeInt,
              selector) && IloGenerate(env,Bid_Cost);
IloGoal goal=IloApply(env,gol,IloLDSEvaluator(env,s));
```

IloLDSEvaluator è la funzione che descrive il funzionamento dell'algoritmo LDS: s è il valore dello step, specificato dall'utente o, eventualmente, ricavato in funzione dell'ampiezza del problema; si è infatti notato che, mediamente, si ottengono i migliori risultati ponendo il valore di step appena minore del numero di attività. In tal caso si riesce quasi sempre a trovare la soluzione migliore direttamente alla prima iterazione.

Ogni volta che il risolutore trova una soluzione, quest'ultima viene restituita e la ricerca si interrompe. Il sistema scrive tale soluzione sia sullo schermo sia in un file, dopodiché chiama il metodo *next* del risolutore, che consente di proseguire l'esplorazione dal punto in cui si era interrotta, alla ricerca di una soluzione migliore, che incrementi cioè l'obiettivo secondo la politica e lo step scelti. Nel caso in analisi la politica è, come già scritto, minimizzare il costo e lo step, essendo il prezzo una variabile intera, è posto uguale ad 1.

```
model.add(IloMinimize(env,Cost));
solver.setOptimizationStep(1);

solver.newSearch(goal);
while (solver.next())
  { //Scrittura a video dei risultati ottenuti
    //Scrittura su file dei risultati ottenuti
  }
```

All'interno del ciclo *while*, le informazioni sono ricavate direttamente da solver, che implementa una serie di metodi *get* per ottenere i valori o i domini delle variabili contenute e altre informazioni sulla ricerca, come il tempo utilizzato, i nodi esplorati, i fallimenti e quindi i backtracking effettuati, le variabili e i vincoli considerati.

CAPITOLO 7

RISULTATI SPERIMENTALI

7.1 Introduzione

In questo capitolo saranno esposti i risultati ottenuti dalle prove sperimentali effettuate. In particolare, saranno presentate le soluzioni ai problemi generati con MAGNET ottenute con il metodo proposto in questa tesi, dopodiché esse saranno paragonate a quelle fornite da MAGNET stesso; saranno inoltre risolti problemi generati con CATS.

Tutte le prove sono state effettuate sulla stessa macchina, con processore Intel Pentium III 800MHz EB, 256Mb di memoria e Microsoft Windows NT 4.0 come sistema operativo.

7.2 Risultati ottenuti risolvendo problemi generati da MAGNET

I problemi generati da MAGNET sono molto simili tra loro. Gli unici parametri che sono davvero tenuti di conto dal generatore sono il numero di offerte e di attività. Tramite un file di configurazione è possibile differenziare leggermente il tipo di attività messe all'asta, specificando per ognuna valori medi di durata, costo e percentuale di occorrenza nelle offerte. Tramite un altro file, possono essere inseriti i parametri caratteristici dell'algoritmo utilizzato, cioè la temperatura di annealing T e il fattore di riduzione ε; per la Simulated Annealing i valori ottimi, impostati dal progettista, sono T=0,35 ed ε=0,997.

Sono stati considerate varie tipologie di aste, per ognuna delle quali sono stati creati un numero di problemi tale da poter considerare attendibili i valori medi dei risultati. Dapprima, è stato risolto un problema relativamente semplice: un'asta su 5 attività, su cui sono state fornite una media di 15 offerte. Sono stati paragonati i risultati forniti dai due algoritmi implementati da MAGNET, Integer Programming (IP) e Simulated Annealing (SA), a quelli ottenuti dal progetto di tesi, applicando i due algoritmi Depth First Search (DFS) e Limited Discrepancy Search(LDS). La successiva figura 7.1 e la tabella 7.1

riportano il tempo medio, in millisecondi, impiegato per fornire la soluzione ottima. La tabella, inoltre, confronta il numero di fallimenti subiti e il numero di nodi dell'albero esplorati dagli algoritmi DFS e LDS. Per quanto riguarda i parametri della LDS, lo step delle discrepanze è stato posto uguale a 4. In generale, per tutti gli altri problemi illustrati nel presente capitolo, tale valore sarà posto uguale al numero di attività messe all'asta diminuito di 1.

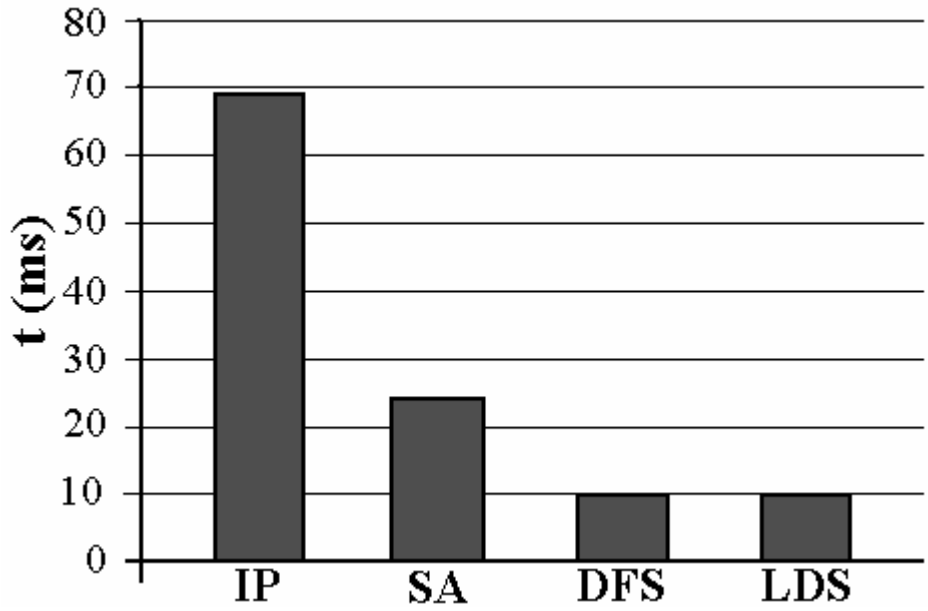

Fig. 7.1: *Tempo di ricerca per un problema con 5 attività e 15 offerte.*

Soluzione ottima	Tempo di ricerca (ms)				Fallimenti		Nodi visitati	
	IP	SA	DFS	LDS	DFS	LDS	DFS	LDS
9508	70	30	30	30	19	17	23	22
11384	60	50	10	10	2	2	18	18
8496	60	40	30	30	16	3	20	19
9622	71	30	10	10	6	8	21	21
10920	141	311	10	10	1	1	16	16
12319	60	10	10	10	3	3	18	18
12979	70	30	11	10	6	6	20	20
10333	90	40	10	10	3	8	21	29
10311	70	10	10	10	4	7	15	17
6624	80	10	10	10	2	3	13	12

Tab. 7.1: Tempo di ricerca, numero di fallimenti e nodi visitati per un problema con 5 attività e 15 offerte.

Tutti gli algoritmi riescono, ovviamente, a trovare la soluzione ottima, ma, nonostante la semplicità del problema, si noti come gli algoritmi DFS e LDS riescano a fornire tale soluzione in un tempo notevolmente minore rispetto agli altri due. La motivazione di questo risultato non è attribuibile soltanto all'algoritmo di esplorazione, ma anche, e soprattutto, all'efficiente propagazione dei vincoli effettuata dai metodi *propagate* e *IloDistribute* descritti nel precedente paragrafo. I valori dei fallimenti e dei nodi visitati sono piccoli, e simili per i due algoritmi; il problema è semplice e l'euristica porta subito alla soluzione ottima.

Analogo è lo scenario se il problema si complica. La figura 7.2 mostra i risultati ottenuti risolvendo problemi rispettivamente con 10 attività e circa 35 offerte e con 10 attività e 100 offerte. La cosa che subito salta all'occhio è la mancanza del risultato della IP nel secondo grafico. Questo algoritmo è infatti riuscito a fornire una soluzione soltanto nel 10% dei casi, in un tempo mediamente 100 volte maggiore di quello impiegato da LDS. Occorre specificare che, mentre nel primo problema SA ha fornito il risultato ottimo circa nel 60% dei casi, nel secondo non è mai riuscito a raggiungere lo stesso risultato fornito da LDS e DFS. Questo aspetto sarà comunque approfondito successivamente.

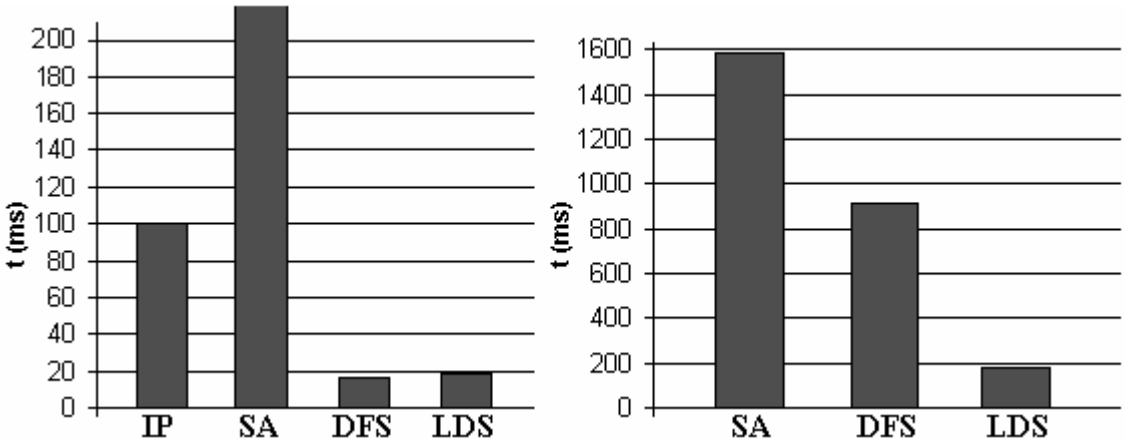

Fig. 7.2: *Tempo di ricerca per due problemi con 10 attività e (sinistra) 35 offerte o (destra) 100 offerte.*

La differenza relativa tra SA e LDS diminuisce, all'aumentare della difficoltà del problema, perché SA interrompe la ricerca dopo un tempo che non è direttamente in relazione alla complessità del problema stesso,

ma della temperatura di annealing e del fattore ε di riduzione. I risultati forniti nel secondo caso sono percentualmente peggiori, se paragonati a quelli di LDS. Il primo problema, avendo poche offerte, ha anche poche soluzioni, e per trovarle occorre percorrere gran parte dell'albero di ricerca; viceversa nel secondo, esistendo verosimilmente un maggior numero di soluzioni, la prima soluzione è trovata molto presto, e questo consente di eliminare molti rami. È per questo che nel primo grafico i valori di DFS e LDS sono molto simili, mentre nel secondo LDS impiega meno del 20% del tempo di DFS. Addirittura, nel primo grafico DFS si comporta meglio perché, se l'albero viene esplorato per intero, quest'algoritmo espande meno nodi interni di LDS.

La tabella 7.2 mostra come, anche per il problema con 35 offerte, DFS e LDS visitino lo stesso numero di nodi e subiscano gli stessi fallimenti nella maggior parte dei casi. La tabella discrimina i casi in cui SA riesce o meno a trovare la soluzione ottima. IP, per come è implementato, se termina con successo restituisce la soluzione ottima, poiché effettua una visita completa dell'albero di ricerca.

Soluzione ottima	**Soluzione SA ottima**	**Tempo di ricerca (ms)**				**Fallimenti**		**Nodi visitati**	
		IP	**SA**	**DFS**	**LDS**	**DFS**	**LDS**	**DFS**	**LDS**
17653	17653	160	2100	10	10	8	8	35	35
19115	22927 (83%)	140	1812	60	40	59	44	68	58
15318	15318	160	1532	20	10	5	5	34	34
17297	17297	90	831	10	10	4	4	36	36
15317	16532 (92%)	80	1933	10	10	0	0	31	31
19758	19758	100	1582	421	40	220	27	227	75
15865	17005 (93%)	150	1352	10	10	4	4	45	45
17795	18059 (98%)	100	2085	20	20	7	7	42	42
16492	16492	281	1993	10	10	4	4	37	37
15172	15172	100	1091	20	20	6	6	37	37

Tab. 7.2: Tempo di ricerca, numero di fallimenti e nodi visitati per un problema con 10 attività e 35 offerte.

Diversamente, dalla successiva tabella 7.3 si vede che LDS subisce molti meno fallimenti di DFS, e riesce quindi a trovare la soluzione ottima visitando molti meno nodi. Per problemi più difficili si può apprezzare l'efficacia dell'algoritmo LDS e della funzione euristica

utilizzata. La tabella mostra anche il costo delle soluzioni fornite da SA e la relativa percentuale rispetto all'ottimo.

Soluzione ottima (LDS e DFS)	Soluzione migliore SA	Tempo di ricerca			Fallimenti		Nodi visitati	
		SA	DFS	LDS	DFS	LDS	DFS	LDS
13815	19063 (72%)	2093	20	30	4	4	114	114
14107	16746 (84%)	4186	12458	490	99300	2543	99312	3481
17519	20643 (85%)	1131	6609	70	57486	398	57497	620
14088	18037 (78%)	3265	761	51	16414	230	16424	455
19468	22521 (86%)	1873	1372	40	11119	138	11181	277
16065	20862 (77%)	2173	10	10	1	1	101	101
15274	17994 (85%)	3245	341	30	2148	76	2159	210
12106	14031 (86%)	3175	20	10	20	17	119	117

Tab. 7.3: Tempo di ricerca, numero di fallimenti e nodi visitati per un problema con 10 attività e 100 offerte.

L'ultimo tipo di problemi generato è di difficile soluzione a causa nell'ampiezza dello spazio di ricerca. Esso contiene 20 attività e circa 400 offerte. Il grafico in figura 7.3 mostra in ordinata la percentuale di ottimalità della soluzione, e in ascissa la percentuale di volte in cui tale valore occorre. Esclusi alcuni problemi particolarmente fortunati, non è stato in generale possibile determinare la soluzione ottima; è stato quindi preso come riferimento il miglior risultato fornito dagli algoritmi.

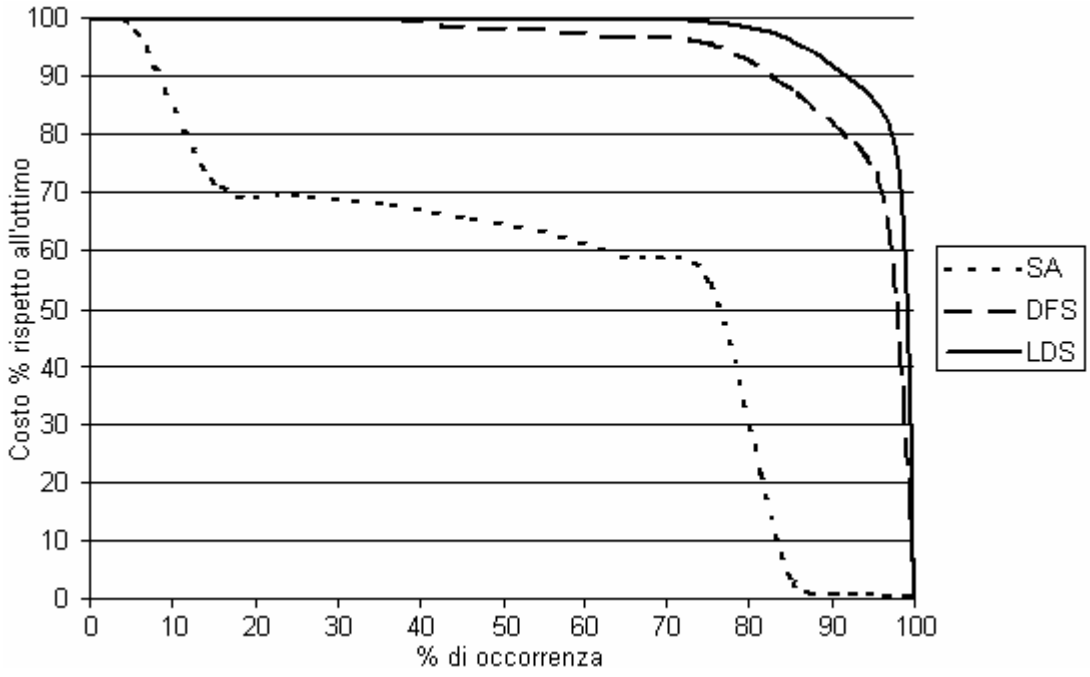

***Fig. 7.3:** Percentuale di ottimalità del risultato per un problema con 20 attività e 400 offerte.*

Gli algoritmi LDS e DFS sono stati interrotti dopo 15 minuti; il risultato migliore è comunque stato tipicamente fornito nei primi 5 minuti di ricerca. Anche in questo caso LDS fornisce il risultato migliore nella maggioranza dei casi. SA, invece, termina la ricerca quando, a fronte di una riduzione della temperatura di annealing, non viene migliorata l'ultima soluzione trovata. I risultati sono forniti in un tempo relativamente breve, ma la soluzione è decisamente peggiore di quelle fornite dal progetto di tesi. La tabella 7.4 riassume i risultati trovati.

Soluzione migliore	**Soluzione SA**	**Soluzione DFS**	**Soluzione LDS**	**Tempo di ricerca (ms)**		
				SA	**DFS**	**LDS**
19490	35487 (55%)	22197 (88%)	19490	10875	30	110
26859	37394 (72%)	27797 (97%)	26859	11256	30	40
23887	36302 (66%)	23897 (99%)	23887	10164	718814	60
28036	No soluzione	29357 (96%)	28036	No	30	320061
24467	24467	24852 (98%)	24467	8062	30	40
26833	38729 (69%)	35861 (75%)	26833	6960	40	9193
25997	41007 (63%)	26588 (98%)	25997	7871	1272	739534
24789	44529 (56%)	24789	25733 (96%)	9393	476295	113273
24718	36283 (68%)	24718	24866 (99%)	3695	30	57753

Tab. 7.4: Soluzione migliore e tempo di ricerca per un problema con 20 attività e 400 offerte.

In accordo con la figura 7.3, dalla tabella si ricava che LDS fornisce il risultato ottimo nella maggior parte dei casi. Diversamente dai problemi analizzati in precedenza, il tempo impiegato per trovare tale soluzione varia molto in funzione del problema e può raggiungere valori anche alcuni ordini di grandezza più elevati di DFS. Sebbene la soluzione fornita sia comunque migliore, ciò rappresenta un'inefficienza dell'algoritmo. La funzione euristica non tiene conto delle particolarità del problema, e si nota infatti che, mentre per alcune istanze i risultati sono ottimi, per altre sono molto peggiori di DFS. Se l'euristica fosse in grado di adattarsi al problema analizzato, rendendosi quindi **problem dependent**, i risultati potrebbero essere decisamente migliori. Questo può essere oggetto di un lavoro futuro teso a migliorare l'efficienza del sistema.

Nella tabella 7.5 sono rapportati il numero di fallimenti e di nodi esplorati da LDS e DFS. Ovviamente i valori sono proporzionali al tempo di ricerca, quindi si riscontra lo stesso comportamento, dipendente dall'istanza analizzata, descritto e commentato nella precedente tabella.

Numero di fallimenti		**Numero di nodi visitati**	
DFS	**LDS**	**DFS**	**LDS**
30	454	403	909
33	29	412	409
2503281	67	2503300	550
39	1115686	412	1457830
10	10	418	418
64	28991	465	40534
4157	1327661	4177	1963834
2095816	249953	2095836	420177
28	152541	422	248636

Tab. 7.5: Numero di fallimenti e di nodi visitati per un problema con 20 attività e 400 offerte.

Per gli stessi problemi è stato anche analizzato il comportamento nel tempo; oltre al risultato ottimo, è interessante confrontare i tempi con cui le altre soluzioni sono fornite, in particolare quanto tempo intercorre tra l'inizio della ricerca e la prima soluzione trovata.

Purtroppo in tale analisi non possono essere paragonati i risultati del progetto di tesi con quelli di MAGNET, poiché quest'ultimo fornisce soltanto la soluzione finale e non quelle intermedie. Viceversa, grazie al metodo IloSolver.next(), che ad ogni soluzione interrompe la ricerca, è semplicissimo visualizzare le soluzioni intermedie di LDS e DFS.

Il primo problema, 5 attività e 15 offerte, è talmente semplice che, nella maggior parte dei casi, la prima soluzione trovata è anche quella ottima.

Analogamente, anche al problema con 10 attività e 35 offerte, viene trovata come prima soluzione quella ottima nell'80% dei casi.

Viceversa, per gli altri due problemi è interessante verificare l'andamento nel tempo delle soluzioni fornite.

La successiva figura 7.4 grafica le soluzioni nel tempo ottenute da LDS e DFS su problemi di 10 attività e 100 offerte. L'ascissa

rappresenta il tempo in millisecondi e l'ordinata il costo della soluzione in percentuale rispetto al costo della soluzione ottima.

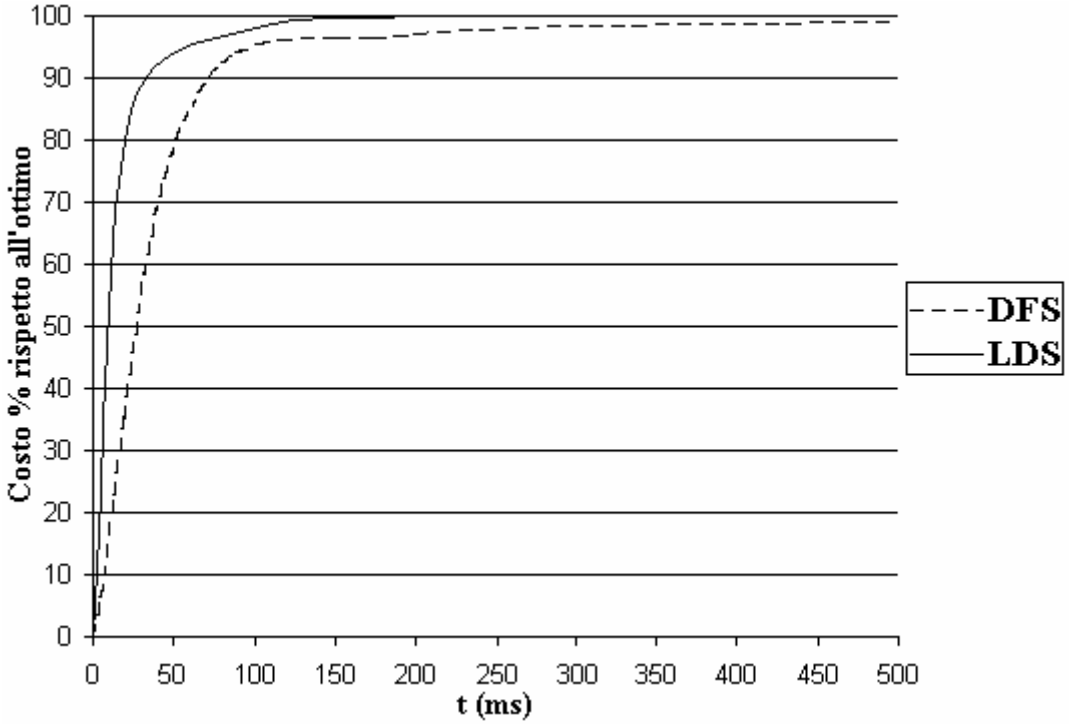

Fig. 7.4: *Comportamento anytime per problemi con 10 attività e 100 offerte.*

Si nota che LDS è sempre migliore di DFS, sia perché riesce a trovare per primo la prima soluzione, sia perché, in ogni istante, la soluzione correntemente fornita è migliore.

La figura 7.5 si riferisce invece ai problemi con 20 attività e 400 offerte. Non essendo, nella maggior parte dei casi, possibile terminare l'algoritmo stabilendo la soluzione ottima, sia il tempo in ascissa sia il costo in ordinata sono in questo caso espressi in percentuale. Per il costo il termine di riferimento è quello della miglior soluzione fornita da LDS, per il tempo l'istante in cui tale soluzione viene trovata. Ovviamente, essendo tutti i valori rapportati alla miglior soluzione fornita da LDS, il grafico relativo a tale algoritmo termina nel punto (100,100). Questo non vuole tuttavia significare che la soluzione trovata sia necessariamente quella ottima.

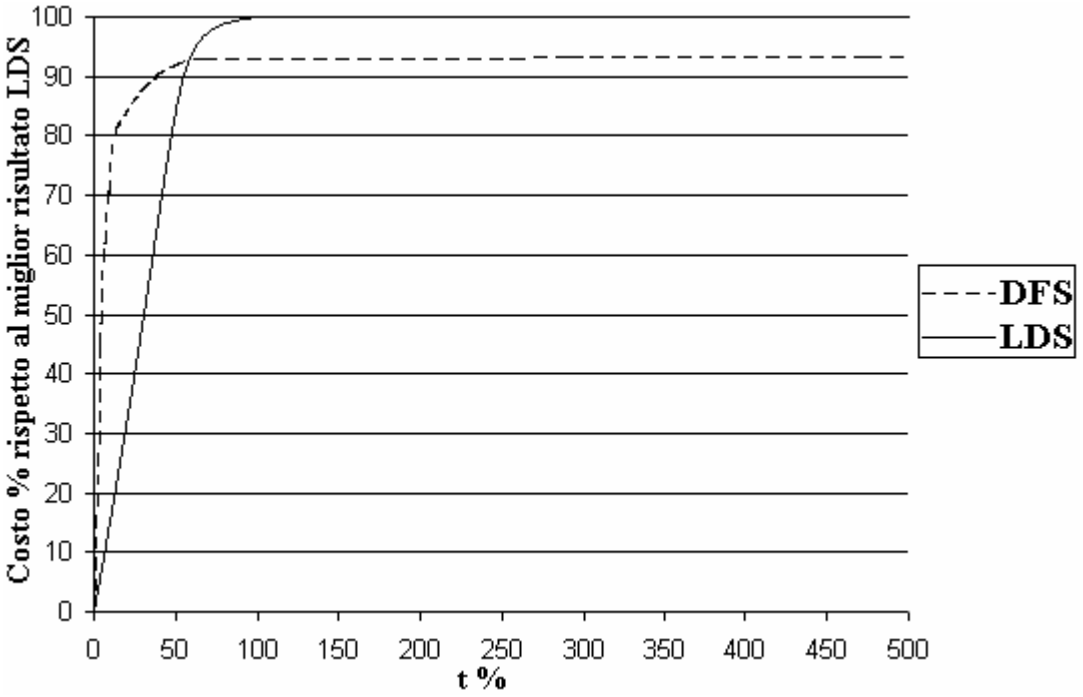

Fig. 7.5: *Comportamento anytime per problemi con 20 attività e 400 offerte.*

Diversamente da quanto descritto nel grafico precedente, in questo caso si evince che LDS trova la prima soluzione in ritardo rispetto a DFS, salvo poi recuperare questo svantaggio proseguendo la ricerca.

In conclusione, si può affermare che il sistema presentato nella tesi fornisce sempre dei risultati migliori rispetto a MAGNET, soprattutto per problemi di difficile soluzione, sia in termini temporali sia di ottimalità. Paragonando invece LDS e DFS si nota che la soluzione di LDS è sempre migliore, ma per problemi particolarmente difficili il tempo impiegato per fornire la prima soluzione è sensibilmente maggiore; essa è comunque migliore della prima fornita da DFS. Questo è dovuto al fatto che, essendo lo spazio di ricerca molto ampio, ed essendo quindi molti i vincoli in gioco, è probabile che una soluzione, nella fattispecie la prima, contenga alcune discrepanze. LDS impiega molto tempo ad analizzare le strade con poche discrepanze senza riuscire ad ottenere una soluzione. Tuttavia, si nota sperimentalmente che, aumentando il valore dello step di discrepanze dell'algoritmo LDS, si

avvicina la ricerca verso una depth-first, ottenendo quindi risultati finali analoghi a quelli ottenuti da DFS. I risultati descritti sono quindi i migliori che LDS riesce ad ottenere con la funzione euristica utilizzata. Come già detto, un possibile miglioramento al sistema potrebbe modificare tale funzione euristica per renderla maggiormente sensibile agli specifici problemi in esame.

7.3 Risultati ottenuti risolvendo problemi generati da CATS

I problemi generati con CATS non possono essere risolti con MAGNET. I risultati si riferiscono quindi ai soli algoritmi DFS e LDS.

CATS si propone come un modello di testing universale per tutte le tipologie di aste combinatorie. Modificando dei parametri, presenti nel motore di generazione dei problemi, si riescono ad ottenere differenti offerte che riproducono i problemi reali. La risoluzione di problemi generati da CATS permette quindi di misurare la robustezza del metodo proposto nei confronti di tipologie di aste differenziate.

Per le aste su insiemi coordinati di attività, i valori da attribuire ai parametri possono essere reperiti in [14]. Tramite questi parametri è possibile specificare in maniera vincolante, oltre al numero di offerte e attività, anche il numero di attività da includere in ogni offerta. Per decidere tale numero, dapprima il generatore sceglie un numero *n* compreso tra 1 e un valore massimo, funzione del numero di attività totali. La probabilità con cui un numero può essere scelto è inversamente proporzionale al numero; con i parametri utilizzati, tale probabilità decresce linearmente da 1 a 0,2. Dopo aver generato le *n* attività, il sistema tenta di includerne altre: la probabilità con cui ogni ulteriore attività è aggiunta all'offerta, è ancora definita dai parametri sopra citati. Per il caso in esame tale valore si attesta attorno al 65%.

Per analizzare la variazione del comportamento dell'algoritmo a fronte della variazione dei parametri fondamentali, cioè numero di attività e di

offerte, sono stati generati dei problemi su un numero crescente di attività, per ognuno dei quali il numero di offerte varia da un minimo che rende il problema molto semplice, ad un massimo che lo rende relativamente complicato.

Il minor numero di attività prese in considerazione è 10; in questo caso le offerte variano da 40 a 500. Entrambi gli algoritmi riescono ovviamente a trovare la soluzione ottima. Nella figura 7.5 è rappresentato il tempo impiegato per fornire tale soluzione in funzione del numero di offerte.

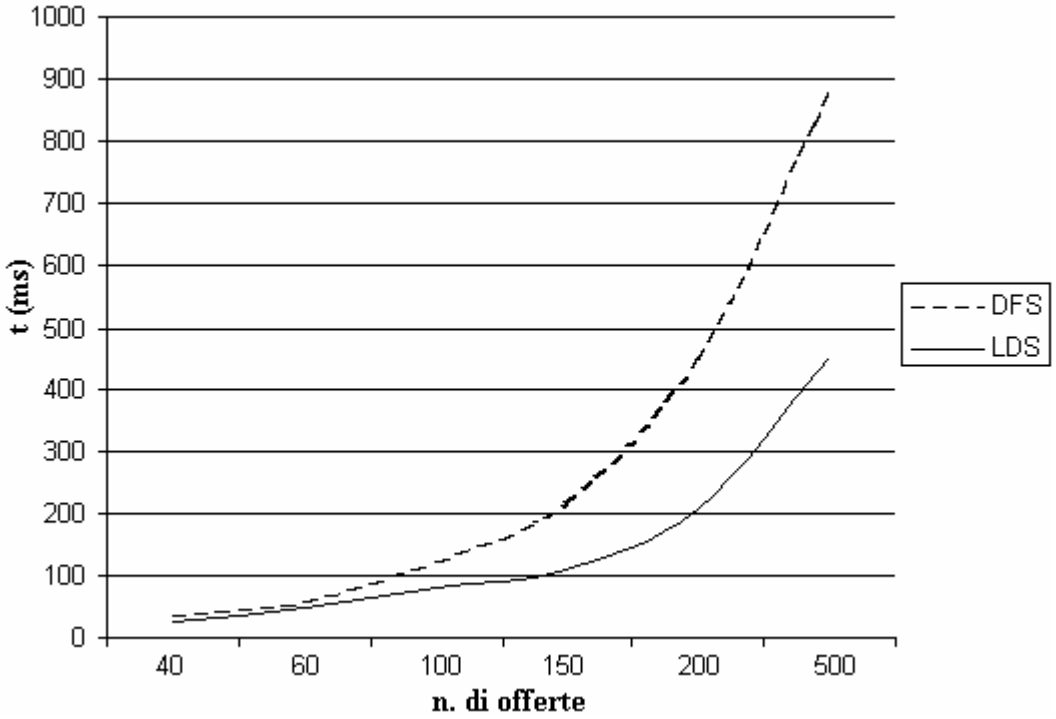

Fig. 7.6: *Tempo di ricerca per un problema con 10 attività in funzione del numero di offerte.*

Paragonando i risultati riportati in questa figura con quelli delle figure 7.1 e 7.2, si nota che LDS fornisce dei risultati, in termini temporali, migliori rispetto a DFS già per i problemi più semplici. Per i problemi CATS, infatti, i problemi con soltanto 5 attività sono molto più semplici di quanto non lo siano gli analoghi MAGNET, questo perché il maggior numero di attività contenute in ogni offerta fa sì che, a fronte di un assegnamento di un'attività ad un offerta, il vincolo *IloDistribute* riesca

ad eliminare molte più offerte ormai divenute incompatibili con la soluzione parziale trovata. Paragonando i tempi di risposta degli algoritmi, si deduce che la complessità di un problema MAGNET di 5 attività e 15 offerte è paragonabile con quella di un problema CATS di 10 attività e circa 40 offerte. Dalla figura 7.6 si vede che LDS impiega circa il 15% di tempo in meno rispetto a DFS per risolvere questi semplici problemi.

Soluzione ottima	**Numero di offerte**	**Tempo di ricerca (ms)**		**Fallimenti**		**Nodi visitati**	
		DFS	**LDS**	**DFS**	**LDS**	**DFS**	**LDS**
214	40	30	20	3	3	42	42
879	40	30	10	1	1	38	38
675	40	40	31	21	21	63	63
325	40	20	20	6	6	51	51
1431	40	50	40	50	50	93	93
494	60	20	20	1	1	66	66
708	60	40	30	8	8	66	66
344	60	150	150	171	171	179	179
563	60	20	10	2	2	61	61
790	60	30	30	9	9	76	76
242	100	741	130	157	157	171	171
605	100	70	70	108	108	114	114
293	100	360	100	245	176	266	209
415	100	70	40	168	57	196	164
530	100	100	90	176	160	197	187
326	150	150	120	383	272	413	325
396	150	70	70	168	173	168	173
863	150	180	161	287	231	294	247
515	150	350	120	342	267	375	313
570	150	120	80	228	17	247	222
254	200	80	70	203	203	213	213
308	200	151	71	332	65	499	272
477	200	181	170	336	287	346	297
480	200	60	40	68	23	271	229
588	200	461	671	1160	1092	1167	1053
122	500	1623	1272	4068	1687	4108	1884
115	500	230	160	824	600	830	620
212	500	921	200	2101	814	2107	847
182	500	60	60	4	4	507	507
149	500	4166	430	9793	721	9808	833

Tab. 7.6: Tempo di ricerca, fallimenti e nodi visitati per un problemi con 10 attività.

Dalla tabella si deduce ulteriormente che, per problemi semplici, i due algoritmi forniscono risultati equivalenti, mentre all'aumentare del numero di offerte, la differenza tra le prestazioni si accentua sempre di più.

Sono stati generati e risolti problemi più complicati. La successiva figura 7.7 descrive i risultati ottenuti su problemi con 15 attività e un numero di offerte variabile da 60 a 500.

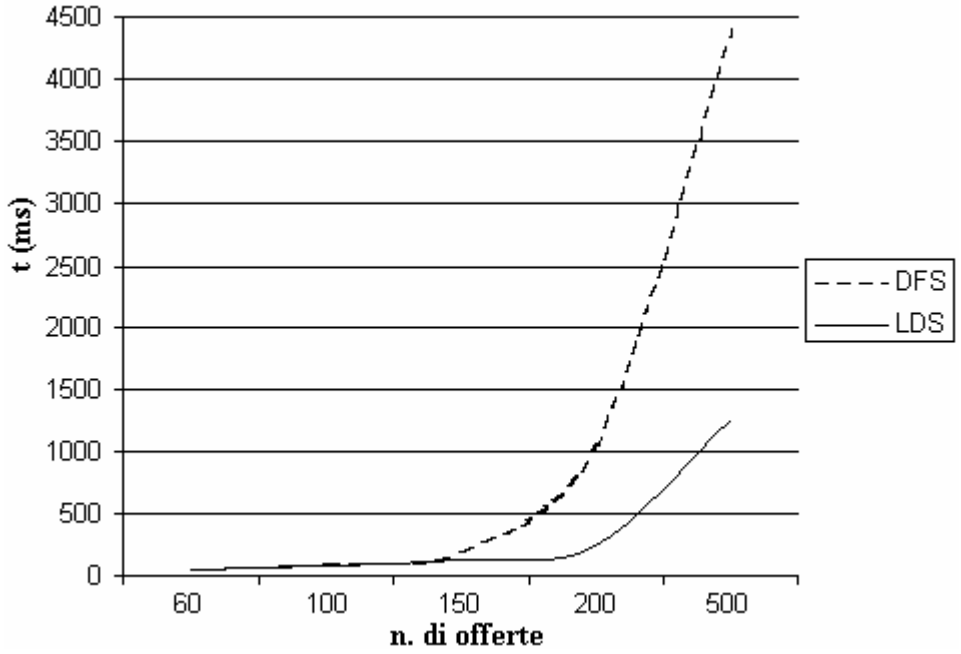

Fig. 7.7: *Tempo di ricerca per problemi con 15 attività in funzione del numero di offerte.*

Infine, sono stati analizzati problemi ancora più complessi, con 20 attività e offerte variabili tra 150 e 1000. La figura 7.8 nella pagina successiva descrive i risultati ottenuti risolvendo quest'ultimo tipo di problemi.

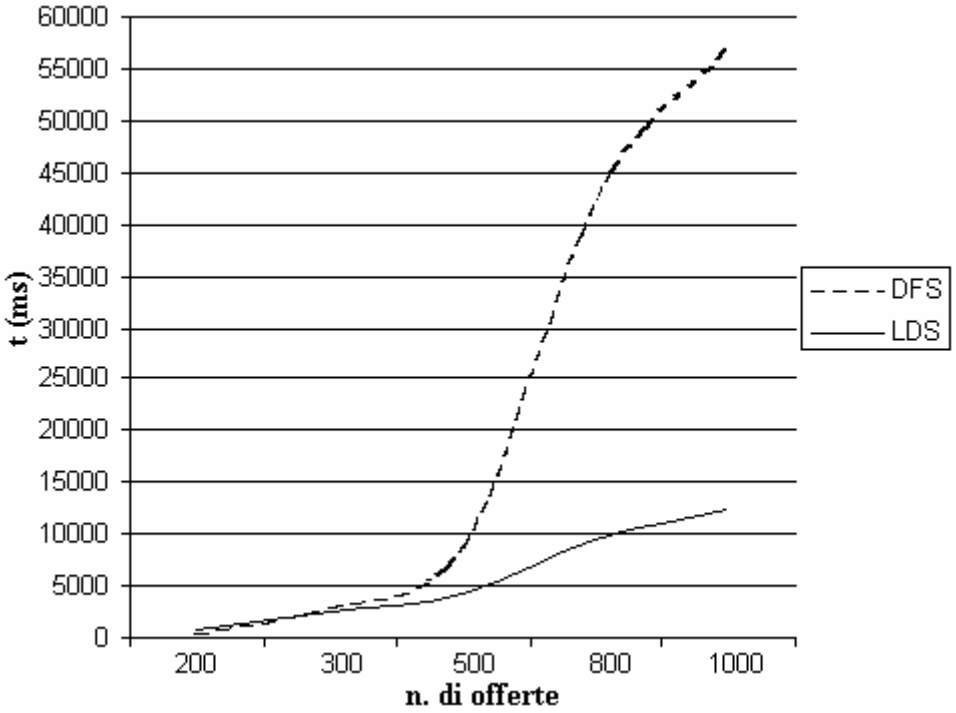

Fig. 7.8: *Tempo di ricerca per problemi con 20 attività in funzione del numero di offerte*

.

Paragonando questi due grafici, si nota che, al crescere della complessità del problema, sia per l'aumento delle offerte sia delle attività, la differenza tra le prestazioni dei due algoritmi aumenta sensibilmente. Si nota inoltre che, quando poche foglie dell'albero di ricerca rappresentano una soluzione, tipicamente quando sono in gioco molte attività e relativamente poche offerte, DFS si comporta leggermente meglio di LDS poiché è necessario esplorare tutto l'albero per trovare la soluzione ottima. La leggera flessione nella parte finale delle curve sul secondo grafico è dovuta al fatto che l'asse delle ascisse non è in scala. Va infine specificato che, per tutti questi problemi, entrambi gli algoritmi terminano fornendo la soluzione ottima.

Diversamente dai problemi MAGNET, in questo caso, all'aumentare della difficoltà del problema, LDS non perde di efficacia, anzi incrementa la differenza di prestazioni rispetto a DFS. La funzione euristica, per come è implementata, è più attendibile per problemi con offerte che contengano molte attività.

Nella tabella che segue vengono riportate le soluzioni alle istanze di problemi con 15 attività.

Soluzione ottima	**Numero di offerte**	**Tempo di ricerca (ms)**		**Fallimenti**		**Nodi visitati**	
		DFS	**LDS**	**DFS**	**LDS**	**DFS**	**LDS**
1133	60	120	110	194	194	230	230
872	60	50	40	25	25	84	84
1161	60	10	10	1	1	63	63
1536	60	50	50	83	83	149	149
792	60	10	10	1	1	72	72
1302	100	70	71	84	84	191	191
314	100	90	80	103	103	211	211
918	100	80	80	121	121	222	222
718	100	90	41	133	106	240	218
1257	100	120	120	240	240	293	293
956	150	190	170	375	312	527	481
1206	150	371	281	871	536	927	714
477	150	271	221	563	475	597	523
920	150	340	310	736	645	763	688
1496	150	200	230	454	350	511	504
323	200	90	90	226	226	294	294
682	200	271	90	573	338	595	566
1218	200	490	351	781	486	985	724
361	200	321	250	686	417	837	624
785	200	81	80	80	78	280	279
352	500	661	540	1530	659	1550	1237
260	500	6860	1583	10602	1572	10659	1977
625	500	4046	1272	4943	985	5450	1697
412	500	9373	1472	14391	1580	14428	1902
435	500	6019	1412	10781	1846	10861	2202

Tab. 7.7: Tempo di ricerca, fallimenti e nodi visitati per un problemi con 15 attività.

Così come il grafico, anche la tabella 7.7 è simile a quella del problema con 10 attività. Con poche offerte i risultati si assomigliano, con più offerte LDS è più efficace rispetto a DFS.

La tabella 7.8 nella pagina successiva riporta i risultati della risoluzione del problema con 20 attività.

Soluzione ottima	Numero di offerte	Tempo di ricerca (ms)		Fallimenti		Nodi visitati	
		DFS	LDS	DFS	LDS	DFS	LDS
1241	200	90	80	89	89	297	297
820	200	691	671	1860	1782	2073	2013
923	200	151	181	226	273	436	488
1141	200	691	1992	2212	3912	2223	3984
1105	200	561	1121	2051	3183	2083	3344
1365	300	2703	1432	4145	2085	4204	2523
710	300	3505	3114	8299	5529	8309	5939
1553	300	2464	1102	4851	1946	5159	2407
1317	300	4015	3735	4835	6048	5144	6133
903	300	1622	2784	2256	3536	2387	3979
1384	500	13449	11106	16686	13451	17192	14808
706	500	21031	4888	28865	4766	29021	5886
603	500	1432	1131	3110	2352	3220	2525
1476	500	4536	2784	3960	2377	4461	2992
1035	500	3015	1402	4682	1945	4715	2126
526	800	14900	2484	17808	2241	17821	2781
558	800	76379	18807	98483	13722	98498	15973
375	800	42181	10946	65132	7607	65160	9013
387	800	99263	17886	146574	15658	146596	17914
409	800	59015	8312	70972	8176	70989	9800
160	1000	94296	27189	102710	21113	102768	24320
363	1000	103739	20770	106260	13703	106320	17369
389	1000	36202	5668	49075	5776	49090	6513
504	1000	19589	1882	26070	1146	26491	2301
311	1000	10536	821	15110	448	15382	1550

Tab. 7.8: Tempo di ricerca, fallimenti e nodi visitati per un problemi con 20 attività.

Si ritrova anche in tabella 7.8 il comportamento migliore di DFS per i problemi con poche offerte, quindi con poche soluzioni. Si nota inoltre che, per problemi difficili, LDS ha un comportamento decisamente migliore di DFS.

Pure per i problemi CATS è possibile mostrare il comportamento in funzione del tempo. Nei grafici che seguiranno l'ascissa è il tempo e l'ordinata il costo; entrambe le coordinate sono espresse in percentuale, e il valore di riferimento è la soluzione ottima fornita da LDS. Il vantaggio di utilizzare questo tipo di piano cartesiano è che, al variare del numero di offerte, le curve caratteristiche restano molto simili tra loro.

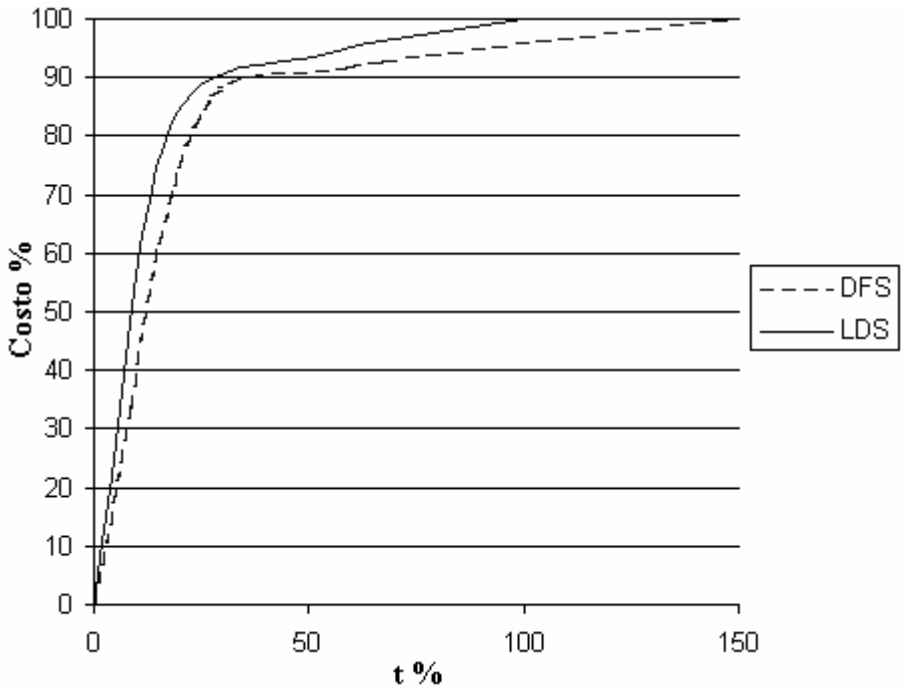

Fig. 7.9: *Comportamento anytime per problemi con 10 attività.*

Questo grafico è relativo ad un problema con 100 offerte. I grafici con un numero minore o maggiore di offerte presentano la linea DFS rispettivamente leggermente più vicina o più lontana da quella LDS. Si vede che entrambi gli algoritmi riescono a trovare la prima soluzione molto velocemente e in un tempo analogo, ma poi LDS riesce a raggiungere più velocemente l'ottimo. La percentuale di ottimalità della prima soluzione trovata è attorno all'80%.

Non dissimile è la caratteristica per problemi con 15 attività. Ovviamente, in accordo con i dati esposti nei grafici precedenti, le due linee LDS e DFS sono più distanziate. La curva LDS è molto simile al caso precedente, mentre quella DFS fa capire come l'algoritmo non riesca, diversamente da LDS, a fornire le soluzioni con le stesse prestazioni dei casi più semplici. La prima soluzione è in ogni caso veloce ad essere fornita, anche se la percentuale di ottimalità, che per LDS resta attorno all'80%, cala al 65% per DFS. La figura sottostante si riferisce al caso con 200 offerte.

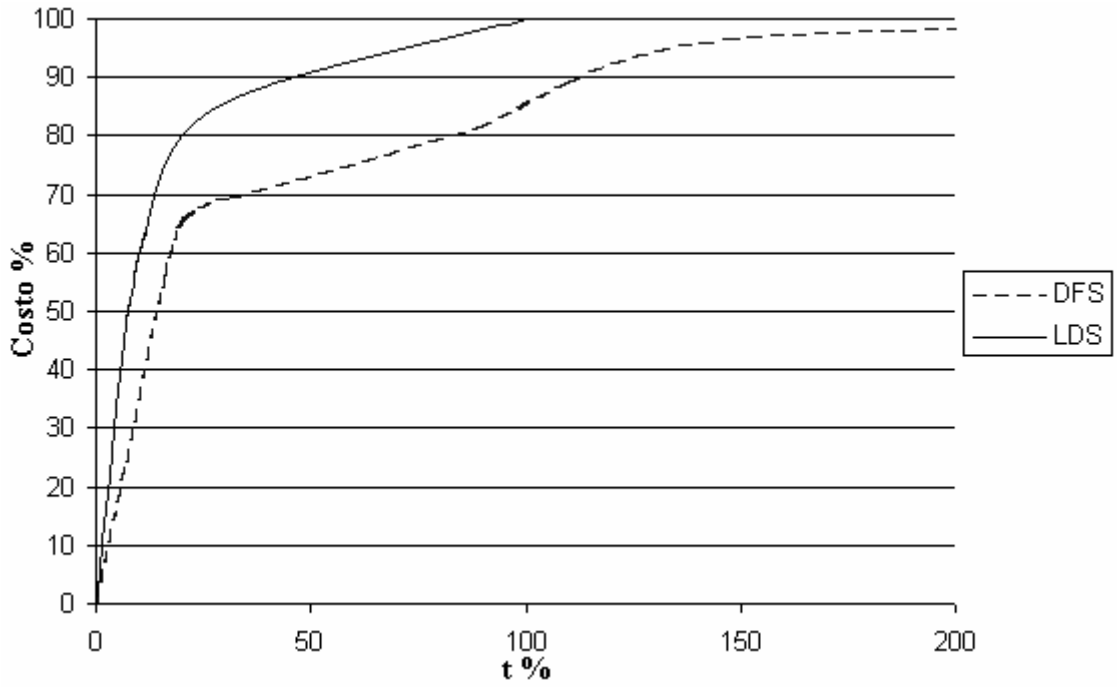

Fig. 7.10: *Comportamento anytime per problemi con 15 attività.*

La figura nella pagina successiva rappresenta infine le caratteristiche ottenute risolvendo problemi con 20 attività ed è riferita al caso di 800 offerte. Si nota che le prestazioni di LDS peggiorano leggermente, poiché la prima soluzione, fornita tuttavia dopo poco tempo, si attesta attorno al 55% di quella ottima. La motivazione è anche in questo caso da ricercarsi nell'euristica utilizzata, poiché, essendo i vincoli da soddisfare proporzionali alla complessità del problema, quindi al numero di offerte e di attività, diminuisce di conseguenza la probabilità che le offerte economiche, le prime scelte dall'euristica, facciano parte della soluzione migliore. Aumentano quindi le discrepanze delle buone soluzioni, che vengono prese in considerazione da LDS soltanto dopo aver percorso tutte le strade con poche discrepanze.

Il picco che si nota in prossimità del 100% del costo è stato riscontrato in tutti i grafici ed è dovuto al fatto che, una volta trovata una soluzione quasi ottima, cioè dal 95% in poi, l'albero è ormai molto ridotto e la ricerca arriva velocemente alla soluzione ottima. Dalla figura 7.8 si ricava che, per problemi complessi, il tempo per trovare la

soluzione è dell'ordine della decina di secondi. Il fatto di impiegare molto tempo a trovare soluzioni quasi ottime, e di riuscire poi ad arrivare subito alla soluzione ottima, spiega il picco presente nel grafico. Tale picco sarebbe presente anche negli altri grafici, ma è mascherato dalla scala adottata per rappresentare i risultati.

La curva DFS mostra che l'algoritmo peggiora ulteriormente nei confronti di LDS.

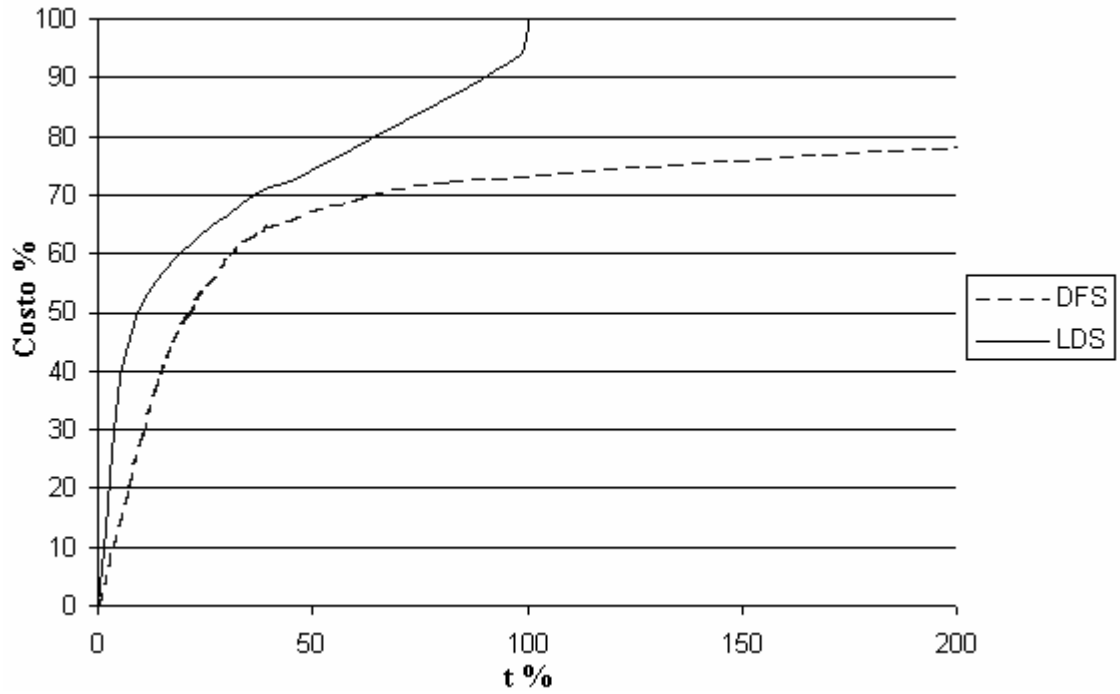

Fig. 7.11: *Comportamento anytime per problemi con 20 attività.*

7.4 Considerazioni conclusive sui risultati

Dall'analisi dei risultati è emerso che il sistema presentato riesce a risolvere problemi di aste combinatorie su insiemi coordinati di attività fornendo delle prestazioni migliori rispetto a MAGNET, un sistema esistente utile per risolvere lo stesso tipo di problemi. La differenza tra i 2 metodi diventa tanto più evidente quanto più complesso è il problema; soprattutto, il sistema in oggetto riesce a trovare soluzioni ottime laddove MAGNET fallisce.

La funzione euristica implementata e la capacità da parte di ILOG di eliminare rami di albero che porterebbero ad una soluzione comunque peggiore dell'ultima trovata, consentono anche ad un algoritmo poco efficiente come la DFS di raggiungere il risultato ottimo anche nel caso di problemi più complessi. Questo è risultato particolarmente evidente risolvendo le istanze di asta generate da CATS, dove il maggior numero di attività per offerta porta ad una maggiore riduzione dei domini delle variabili, tramite il metodo *propagate*, in seguito all'istanziazione di una di esse.

Il fatto che il sistema fornisca le prestazioni migliori quando applicato a problemi CATS non è un limite, poiché è proprio nella filosofia delle aste combinatorie poter fare offerte su più oggetti contemporaneamente. Problemi come quelli MAGNET, in cui ogni offerta si riferisce tipicamente ad una sola attività, focalizzano troppo la ricerca sullo scheduling e poco sulla vendita di tutte le attività.

Dai risultati si confermano i dati introdotti nel capitolo 2, secondo cui l'algoritmo di esplorazione LDS fornisce risultati migliori rispetto a DFS, soprattutto quando le dimensioni del problema si fanno considerevoli.

Conclusioni

Dal lavoro svolto e dai risultati ottenuti, si deduce come la programmazione a vincoli possa essere efficacemente utilizzata per trovare soluzione a problemi di aste combinatorie. L'algoritmo LDS utilizzato si è dimostrato efficiente, come descritto anche in letteratura, riuscendo a trovare la soluzione in un tempo minore rispetto ad un algoritmo completo, tipicamente la DFS. LDS riesce inoltre a trovare una soluzione ottima laddove altri metodi, nello specifico dell'analisi effettuata MAGNET, falliscono. In generale, rispetto a MAGNET, LDS trova soluzioni migliori, o la soluzione ottima, in un tempo decisamente minore. Le funzioni euristiche per la selezione di variabili e valori forniscono ottimi risultati a problemi di piccola e media complessità, mentre perdono progressivamente di efficacia se applicati a problemi molto complessi, poiché si focalizzano troppo sulla minimizzazione del costo, perdendo di vista i vincoli che intercorrono tra le offerte. I risultati ottenuti sono comunque indiscutibilmente migliori di quelli ottenuti con MAGNET e in generale buoni.

Possibili sviluppi futuri possono orientarsi nel senso di progettare funzioni euristiche problem dependent, in grado cioè di adattarsi maggiormente al problema oggetto dell'analisi, riducendo l'importanza data al costo all'aumentare dei vincoli in gioco.

Altri possibili sviluppi possono viceversa utilizzare gli stessi algoritmi per la risoluzione di problemi rilassati derivati da quelli analizzati. Eliminando i vincoli temporali, il problema si riduce alla determinazione del vincitore di un'asta combinatoria su un insieme di beni in generale. In letteratura è presente una vasta analisi di tali problemi e sono proposti vari metodi e algoritmi per trovare la soluzione ottima. La programmazione a vincoli, applicata a tali problemi tramite l'utilizzo di algoritmi incompleti, potrebbe fornire risultati decisamente migliori rispetto a quelli riportati in letteratura.

BIBLIOGRAFIA

[1] W. D. Harvey e M. L. Ginsberg, "Limited Discrepancy Search", in "Proceedings of the 14th International Joint Conference on Artificial Intelligence" (IJCAI-95), Montreal 1995, pagg. 607-613.

[2] R. E. Korf, "Improved Limited Discrepancy Search", in "Proceedings of the 13th National Conference on Artificial Intelligence" (AAAI-96), Portland 1996, pagg. 288-291.

[3] M. R. Garey e D. S. Johnson, "Computers and Intractability: a Guide to the Theory of NP-Completeness", W. H. Freeman, S. Francisco 1979.

[4] N. Karmarkar e R. M. Karp, "The differencing method of set partitioning", Technical Report UCB/CSD 82/113, Computer Science Division of the University of California, Berkeley 1982.

[5] T. Walsh, " Depth-bounded Discrepancy Search", in "Proceedings of the 15th International Joint Conference on Artificial Intelligence" (IJCAI-97), Nagoya 1997, pagg. 1388-1395.

[6] R. E. Korf, "Depth-first Iterative Deepening: an optimal admissible tree search", in "Artificial Intelligence" Vol. 27, No 1, 1985, pagg. 97-109.

[7] P. Meseguer e T. Walsh, "Interleaved and Discrepancy based Search", in "Proceedings of the 13th European Conference on Artificial Intelligence" (ECAI-98), Brighton 1998, pagg. 239-243.

[8] J. Jaffar e J. L. Lassez, "Constraint Logic Programming", in "Proceedings of the 14th ACM Symposium on Principles of Programming Languages" (POPL-87), Munich 1987, pagg.111-119.

[9] N. Beldiceanu ed E. Contejean, "Introducing global constraint in CHIP", in "Mathematical and Computer Modelling" Vol. 12, 1994, pagg. 97-123.

[10] Y. Deville, P. Van Hentenryck e V. A. Saraswat, "Design, Implementation and Evaluation of the Constraint Language cc(FD)", in "Constraint Programming: Basics and Trends", A. Podelsky, Berlin 1995, pagg. 293-316.

[11] N. Nisan, "Bidding and Allocation in Combinatorial Auctions", in "Proceedings of the 2nd Conference on Electronic Commerce" (EC-00), Minneapolis 2000, pagg. 1-12.

[12] J. Collins e M. Gini, "Bid evaluation for coordinated tasks: an Integer Programming formulation", in "IJCAI-2001 Workshop on Economic Agents, Models, and Mechanisms", Seattle Washington 2001.

[13] J. Collins, M. Gini e W. Ketter, "A Multi-Agent Negotiation Testbed for Contracting Tasks with Temporal and Precedence Constraints", in "International Journal of Electronic Commerce" Vol. 6, 2002.

[14] K. Leyton-Brown, M. Pearson e Y. Shoham, "Towards a Universal Test Suite for Combinatorial Auction Algorithms", in "Proceedings of the 2nd Conference on Electronic Commerce" (EC-00), Minneapolis 2000, pagg. 66-76.

[15] A. Andersson, M. Tenhunen e F. Ygge, "Integer Programming for Combinatorial Auction Winner Determination", in "Proceedings of the 4th International Conference on MultiAgent Systems" (ICMAS-2000), Boston 2000, pagg.39-46.

[16] J. Collins, M. Gini,B. Mobasher e R. Sundareswara, "Bid Selection Strategies for Multi-Agent Contracting in the presence of Scheduling Constraints", in "Agent Mediated Electronic Commerce II" (IJCAI-99), Stockholm 1999, pagg.113-130.