Relazione al progetto per il corso di Ragionamento Automatico

Andrea Fortunato

28 aprile 2010

Sommario

Questo documento rappresenta una relazione dettagliata per il progetto del corso Ragionamento Automatico. Lo scopo del progetto è costruire un risolutore automatico di problemi di ottimizzazione nel linguaggio Java, utilizzando l'algoritmo per *DPOP* (*Dynamic Programming Optimization Problem*). Il risolutore dovrà essere in grado di risolvere problemi di *massimizzazione del numero di vincoli soddisfatti* (*Max CSP*), accettando in input grafi i cui vincoli sono descritti tramite *NOGOOD* (coppie di valori *non consentite*) e risolvendoli massimizzando il numero di vincoli soddisfatti, indicando *quanti* e *quali* vincoli sono stati eventualmente violati.

Indice

1 Introduzione

I problemi che il risolutore deve essere in grado di manipolare sono problemi di ottimizzazione, in cui l'input sarà caratterizzato da variabili ciascuna con il suo dominio di valori, e dai vincoli sulle variabili. L'algoritmo per DPOP in particolare risolve problemi di massimizzazione con soli vincoli binari, dato che è possibile ridurre qualsiasi problema di ottimizzazione in questa versione. Questo progetto, in particolare, si focalizza sulla risoluzione di un sottoinsieme dei problemi di massimizzazione, vale a dire su problemi di *massimizzazione del numero di vincoli soddisfatti* (*Max CSP*). Il concetto di agente, di cui si fa menzione nel paper del DPOP, qui collassa sul concetto di variabile: in altre parole ogni agente detiene una e una sola variabile. In questo documento pertanto si parlerà di agente e variabile in maniera del tutto intercambiabile.

Riassumendo brevemente il "flusso di esecuzione" delle operazioni: viene letto in input un file di descrizione contenente l'elenco delle variabili, il loro dominio, e i vincoli binari a cui tali variabili sono sottoposte. I vincoli saranno descritti con *NOGOOD*, vale a dire saranno specificate solamente le coppie di valori *non ammesse* per le variabili del vincolo; questa descrizione può essere facilmente trasformata per risolvere *Max CSP,* impostando i vincoli in maniera tale che le coppie di valori non consentite abbiano peso -1 e quelle consentite 0. Il parsing dell'input genera un grafo non diretto, e successivamente tramite un'opportuna visita DFS viene generato lo pseudo-tree, vale a dire la struttura dati su cui si basa l'algoritmo DPOP. Una volta ottenuto lo pseudo-tree, la computazione si svolge visitando dapprima dal basso verso l'alto l'albero nella sua interezza (*util propagation*) e poi dall'alto verso il basso (*value propagation*). A questo punto la soluzione è disponibile: se il risultato è 0 nessun vincolo è stato violato, altrimenti un qualsiasi valore negativo rappresenta la violazione di uno o più vincoli. Per rendere la soluzione più completa, sono stati aggiunti frammenti di codice che memorizzano durante la computazione i vincoli violati, in modo da poter visualizzare al termine quali vincoli sono stati violati.

2 Requisiti dell'implementazione

Le caratteristiche dell'algoritmo per DPOP individuano naturalmente i requisiti che l'implementazione dell'algoritmo stesso deve soddisfare. Si possono individuare due obiettivi principali: il parallelismo delle computazioni degli agenti e l'efficienza in termini di dimensione dei messaggi scambiati tra gli agenti.

2.1 Parallelismo

L'algoritmo per DPOP si presta molto bene alle computazioni parallele in quanto, grazie alla programmazione dinamica, ciascun agente può concentrarsi solo su un sottoproblema in maniera (quasi) del tutto svincolata dagli altri agenti. In particolare, la conversione del grafo originale nello pseudo-tree aiuta ad individuare le relazioni tra gli agenti, in modo che agenti che si trovano su rami differenti dell'albero avranno computazioni completamente slegate tra di loro. Naturalmente non è possibile ottenere una parallelizzazione assoluta della computazione, infatti ogni nodo dell'albero è legato al nodo padre ed eventualmente agli pseudo-padri. In particolare il nodo padre dovrà attendere che le computazioni dei nodi figli siano terminate prima di poter procedere con la sua computazione. Nella sezione 3.6.2 a pagina 10 sarà illustrato come è stata gestita la parallelizzazione e la sincronizzazione tra i thread per poter simulare questo comportamento degli agenti.

2.2 Efficienza computazionale

Vista la "natura parallela" del problema, è verosimile aspettarsi un overhead in tempo o spazio, poiché si tratta pur sempre di un problema esponenziale in un qualche parametro. Nella fattispecie, come è dimostrato dettagliatamente nel paper sul DPOP, l'algoritmo utilizza un numero lineare di messaggi sia nella fase di propagazione dei messaggi *util* sia per i messaggi *value*, tuttavia il problema è esponenziale nella *dimensione* dei messaggi *util* scambiati tra gli agenti. Detto in maniera più formale, l'algoritmo risulta esponenziale nella *induced width* dello pseudo-tree indotto dalla visita DFS sul grafo originario. Nella sezione 3.4.3 a pagina 8 sarà illustrata l'euristica utilizzata per cercare di ridurre l'*induced width* nella fase di costruzione dello pseudo-tree, e sarà illustrato come l'applicazione di questa euristica aumenti drasticamente le prestazioni dell'algoritmo nella sezione 4.1.1 a pagina 17 (per i dettagli sulle motivazioni per cui questa euristica migliora le prestazioni guardare la sezione 3.4.3 a pagina 8).

2.3 Riusabilità del codice

L'implementazione dell'algoritmo per DPOP è stata data tenendo sempre presente la possibilità di volerne espandere in futuro le potenzialità. Per citare un esempio, i valori del dominio delle variabili non sono stati modellati con semplici tipi *integer*, ma è stata creata una superclasse classe *Value* per rendere più agevole una eventuale modifica di tali valori. Per esempio si potrebbe implementare una classe che rappresenta il dominio dei colori (rosso, verde, blu, ecc. . .) come possibili valori per le variabili del problema. Queste accortenze adottate provocano in alcuni casi un leggero overhead computazionale (banalmente un integer occupa meno spazio di un oggetto memorizzato nello heap), ma rendono il progetto meno "fine a se stesso" e più utile per utilizzi futuri.

3 Implementazione

L'implementazione del progetto si compone di un certo numero di classi. Alcune hanno il ruolo di "classi di servizio", vale a dire servono per modellare particolari strutture dati necessarie, oppure compiere determinati aggiornamenti dell'interfaccia grafica, o altre operazioni simili; altre classi invece rappresentano il cuore centrale della computazione, e sono quelle su cui sarà speso maggior dettaglio in questa relazione.

3.1 Struttura generale del progetto

Per descrivere la struttura generale del progetto, può essere utile dare un elenco completo delle classi Java, raggruppandole per la loro funzione all'interno del programma.

3.1.1 Gestione interfaccia grafica

- *DPOPApp*: classe di avvio del programma; intercetta gli eventi dell'interfaccia grafica.
- *Interface*: interfaccia grafica del programma.
- *ViewGraph*: pannello per visualizzare, tramite apposita libreria esterna, il grafo letto in input e lo pseudo-tree generato.

- *RunTimer*: timer su un thread separato che ogni secondo aggiorna le informazioni di stato sull'interfaccia grafica, per esempio la progress bar e il tempo trascorso dall'inizio della computazione.

3.1.2 Classi di servizio

- *TaggedDictionary*: rappresenta una struttura dati dizionario, vale a dire un insieme di coppie (chiave, valore), identificato da un oggetto aggiuntivo chiamato *tag*. Il senso del *tag* è quello di identificare a chi appartiene l'insieme di coppie rappresentato dall'oggetto.
- *Handler*: racchiude operazioni relative sia alla gestione della UI, sia alla gestione delle informazioni ricevute dalla computazione. In particolare, l'handler gestisce la lettura dell'input, l'avvio della computazione, la collezione delle soluzioni parziali, l'avanzamento di stato del processo, la terminazione e l'aggiornamento delle informazioni sull'interfaccia grafica.
- *Couple*: rappresenta una generica coppia (variabile, valore), necessaria in alcuni punti del programma.
- *ValuesIterator*: è una classe che implementa l'interfaccia *Iterator*, e consente di generare tutte le possibili combinazioni di valori delle variabili (con i relativi domini) specificate. Per esempio, supponendo di dare al ValuesIterator le variabili x_0, x_1, x_2 con i valori del dominio 0 e 1, ogni chiamata del metodo *next()* restituisce la successiva configurazione di valori, partendo da 000 fino a 111, reimpostando eventualmente a 000 la configurazione quando viene raggiunta l'ultima configurazione disponibile.

3.1.3 Classi centrali

- *Value*: superclasse per modellare il concetto generico di valore, sia per i domini che per i pesi dei vincoli.
- *IntegerValue*: estende *Value*, rappresenta un valore intero.
- *Variable*: rappresenta il concetto di variabile, con il suo dominio di valori, e legata ad altre variabili attraverso vincoli.
- *GraphNode*: estende *Variable*, è un nodo di un grafo non diretto, contiene al suo interno sia le funzioni classiche di un nodo, sia quelle per la visita DFS che trasforma il sottografo che parte dal nodo stesso nello pseudo-tree corrispondente.
- *PseudoTreeNode*: estende *Variable*, rappresenta un nodo dello pseudo-tree, con i relativi concetti di padre, pseudo-padri, pseudo-figli. Contiene inoltre tutte le funzioni per la computazione vera e propria, in particolare la funzione *start()* che fa partire la computazione sul sottoalbero radicato in questo nodo.
- *Constraint*: superclasse per modellare il concetto generico di vincolo.
- *BinaryConstraint*: estende *Constraint*, e rappresenta un vincolo binario, utile per modellare i vincoli tra le variabili.

- *NAryConstraint*: estende *Constraint*, rappresenta un vincolo n-ario e viene utilizzato solo per modellare i messaggi *util* multidimensionali ricevuti (per esempio un messaggio *util* che coinvolge x_0, x_1, x_2, x_3 sarà un ipercubo e volendo trovare il peso di una certa configurazione di queste variabili si può vedere il messaggio come un vincolo a 4 variabili in cui, passando i valori per ciascuna di esse, viene restituito il peso corrispondente).

3.2 Classe Handler

Come già accennato precedentemente, la classe Handler svolge due funzioni principali: aggiornamento dell'interfaccia grafica e controllo delle operazioni, a partire dalla lettura dell'input fino alla ricezione della soluzione al problema. Di seguito una breve descrizione dei metodi più significativi di questa classe.

3.2.1 Parsing dell'input: *_parse_graph_coloring()*

In questo metodo il file di input viene letto riga per riga fino al termine; ogni volta che si incontra la direttiva *variable* viene creato un nodo di tipo *GraphNode*; ogni volta che si incontra la direttiva *constraint* viene generato un *BinaryConstraint* tra i nodi specificati, che devono necessariamente essere già stati computati precedentemente onde evitare un'eccezione di tipo null pointer. Infine la direttiva *nogood* specifica che per il vincolo letto precedentemente vi è una combinazione di valori non ammessa; se tale direttiva viene letta prima di una direttiva *constraint* un'eccezione sarà generata. Da notare che i nodi generati vengono mano a mano salvati in una variabile di classe *_nodes*, in modo da poterli recuperare in fasi successive. Inoltre durante il parsing viene mantenuta aggiornata la variabile *_most_connected_node* che è un riferimento al nodo attualmente più connesso tra tutti quelli letti finora; tale riferimento servirà quando il grafo dovrà essere trasformato nello pseudo-tree utilizzando l'euristica *Most Connected Node.* Infine al termine del parsing dell'input la variabile *_first_node* sarà settata con il riferimento del primo nodo letto, e servirà successivamente quando il grafo dovrà essere trasformato nello pseudo-tree senza usare alcuna euristica.

3.2.2 Inizio della computazione: *_start_computation()*

Avvia la computazione sul nodo root dello pseudo-tree generato successivamente al parsing del grafo in input. Per avviare la computazione viene invocato il metodo *start()* sul nodo root, inoltre viene invocato lo stesso metodo su tutti i nodi isolati del grafo, vale a dire gli eventuali nodi che sono dichiarati nell'input ma non sono legati agli altri tramite nessun vincolo. Questa operazione è necessaria per ritornare la soluzione completa.

3.2.3 Aggiornamento dello stato: *util_sent()* e *value_sent()*

Questi due metodi vengono chiamati dai nodi dello pseudo-tree durante la computazione e servono per avvisare, rispettivamente, l'invio di un messaggio *util* e di un messaggio *value*. Le informazioni sull'interfaccia grafica possono così essere aggiornate.

3.2.4 Avviso di vincolo violato: *broken_constraint()*

Ogni volta che uno *PseudoTreeNode*, durante la fase di *value propagation*, individua che il valore assegnato alla variabile viola un vincolo *nogood*, chiama questo metodo per memorizzare l'informazione. Se ciò non fosse fatto, infatti, sarebbe necessario rileggere la descrizione del grafo a

computazione terminata per individuare se i valori assegnati violano qualche vincolo; in questo modo invece le informazioni sui vincoli violati vengono aggiornate in tempo reale e non sono necessarie altre ispezioni al termine.

3.2.5 Ricezione della soluzione: *receive_solution()* e *receive_solution_weight()*

Questi metodi vengono invocati per ricevere le informazioni sulla soluzione computata.

Il metodo *receive_solution()* viene chiamato da *ogni nodo* durante la computazione dopo aver deciso quale valore assegnare alla variabile che esso stesso rappresenta. Pertanto, se il grafo contiene n nodi, tale metodo sarà chiamato esattamente n volte.

Il metodo *receive_solution_weight()* viene chiamato una sola volta dal nodo root all'inizio della fase *value propagation*, per comunicare il peso totale della soluzione trovata.

Le informazioni raccolte da questi due metodi verranno poi comunicate sull'interfaccia grafica al termine della computazione.

3.3 Classe TaggedDictionary

Questa classe viene ampiamente utilizzata nel progetto. Come già spiegato precedentemente, consiste di un dizionario con un oggetto aggiuntivo chiamato *tag*, che in alcuni casi serve ad identificare il "proprietario" dell'oggetto. Un esempio di utilizzo dell'oggetto *tag* è il seguente: supponiamo di voler modellare il messaggio util di x_1 contenente solamente informazioni sul padre x_0.

$util_1$	$x_0 = 0$	$x_0 = 1$
	20	30

Tabella 1: Messaggio util di x_1 da inviare a x_0

Per modellare questo messaggio util viene creato un *TaggedDictionary* con tag x_1, proprietario del messaggio, in cui il tipo chiave sarà l'oggetto *PseudoTreeNode* rappresentato da x_0, e come valore un altro *TaggedDictionary* che associa *valori* del dominio (0 e 1) con i pesi della configurazione (20 e 30 rispettivamente). Nella sezione 3.6.4 a pagina 14 è spiegato in maniera dettagliata come è stato modellato il messaggio *util* che ciascun nodo invia al proprio nodo padre.

3.3.1 Brevi cenni sull'implementazione

Il dizionario è stato implementato con un *entry set*, ovvero un insieme in cui sia gli elementi chiave che i relativi valori sono rappresentati da vettori di tipo *ArrayList*. La ricerca degli elementi nell'*entry set* è stata semplicemente implementata con una ricerca lineare, visto che quasi sempre i dizionari creati non conterranno più di 20-30 coppie (chiave, valore). Comunque, questa classe lascia spazio ad eventuali miglioramenti nell'implementazione, a cominciare appunto dalla ricerca degli elementi nell'*entry set*.

3.4 Classe GraphNode

Questa classe rappresenta un nodo di un grafo, e contiene quindi tutti i metodi classici per aggiungere un nodo adiacente, recuperare la lista dei nodi adiacenti, e un metodo aggiuntivo per sapere se si tratta o meno di un nodo isolato (vedere la sezione 3.2.2 nella pagina precedente per maggiori

dettagli sullo scopo di questa informazione del nodo). Oltre a questi metodi, troviamo due metodi di visita DFS che permettono di generare, a partire dal nodo su cui viene invocato il metodo, lo pseudo-tree che sarà necessario per la computazione successiva. Vediamo nel dettaglio questi due metodi di visita.

3.4.1 Visita DFS senza euristica: *dfs_visit()*

Questo metodo esegue una banalissima visita DFS, avendo cura di creare ad ogni step il corrispondente nodo *PseudoTreeNode*. Tale metodo utilizza l'omonimo metodo privato *_dfs_visit()* per effettuare la visita, in quanto tale metodo richiede come parametro la lista dei nodi che fanno parte del cammino dalla root fino al nodo corrente (per il nodo root pertanto questa lista sarà vuota). Non vi sono concetti significativi da analizzare, dato che si tratta di una normale DFS (notare il flag *_visited* sul nodo per sapere se è già stato visitato).

3.4.2 Visita DFS con euristica Most Connected Node: *dfs_visit_mcn()*

Questo metodo esegue una visita DFS, utilizzando la stessa tecnica della funzione *dfs_visit()* ma avendo cura di ordinare i nodi adiacenti in base al grado: i nodi più connessi saranno così ordinati prima dei nodi meno connessi, in questo modo si assicura che la visita scelga sempre per primi i nodi più connessi. A parte questo accorgimento iniziale, il resto della visita si svolge esattamente come la visita DFS standard vista nella sezione 3.4.1.

3.4.3 Dettagli sull'euristica MCN

L'idea di questa euristica è quella di cercare di diminuire la dimensione dei messaggi *util* che ciascun nodo deve inviare al proprio padre. Poiché la dimensione di tali messaggi dipende fortemente dal numero di pseudo-padri dei nodi, selezionando sempre i nodi più connessi durante la visita DFS si cerca di aumentare il numero di nodi figli dell'albero, in altre parole si cerca di creare un albero più profondo, ma che conterrà meno pseudo-collegamenti. L'euristica non garantisce ovviamente la visita perfetta, in quanto valuta la situazione "passo passo", si può vedere cioè come una tecnica greedy che cerca un ottimo localmente ma non assicura la visita migliore in assoluto. Comunque computazionalmente risulta molto efficace e per questo è stata implementata, visto che aumenta drasticamente le prestazioni del calcolo (guardare la sezione 4.1.1 a pagina 17 per i risultati sperimentali).

3.5 Classe ValuesIterator

Questa classe è indispensabile per la composizione del messaggio *util* da inviare al nodo padre, in quanto deve essere ispezionata ogni possibile configurazione di valori delle variabili da cui dipende lo stato attuale, e per ciascuna configurazione deve essere preso il valore massimo rispetto alla variabile corrente. Per fare questa operazione, il ValuesIterator viene creato passando come *ultima variabile* anche la variabile relativa al nodo corrente, in modo tale che sia quella di posizione *meno significativa*. Utilizzando questo stratagemma è possibile tenere "bloccate" tutte le configurazioni delle altre variabili e scorrere solo la variabile del nodo corrente per individuare il valore che massimizza la configurazione corrente. Un breve esempio dovrebbe chiarire questo discorso: supponiamo di lavorare con soli domini $\{0, 1\}$, di essere al nodo x_4, e supponiamo che la valutazione del suo messaggio *util* dipenda dai nodi precedenti x_0, x_1, x_2, x_3. Per creare il messaggio *util* deve essere

valutata ogni possibile configurazione di queste variabili e massimizzare rispetto a x_4, quindi viene creato un *ValuesIterator* che agirà nel seguente modo:

x_0	x_1	x_2	x_3	x_4	valore	comportamento
0	0	0	0	0	10	$util_{x_4}(x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0) = 20$
0	0	0	0	1	20	$argmax_{x_4}(x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0) = 1$
0	0	0	1	0	15	$util_{x_4}(x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 1) = 15$
0	0	0	1	1	5	$argmax_{x_4}(x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 1) = 0$
...	...	...	...	...	...	...

Tabella 2: Computazione del *ValuesIterator*

In altre parole ogni volta che il nodo corrente x_4 resetta la sua configurazione si potrà calcolare il valore finale per la configurazione corrente di *util* e *argmax* rispetto alle variabili x_0, x_1, x_2, x_3 da cui dipende il nodo x_4.

3.6 Classe PseudoTreeNode

Questa classe è da ritenersi il cuore vero e proprio della computazione. Essa infatti non solo rappresenta uno pseudo-nodo dell'albero generato dalla visita del grafo originale, ma contiene anche le funzioni necessarie per la computazione locale dei messaggi *util* e la scelta del valore ottimo per la successiva fase di propagazione dei valori.

I metodi per modellare il concetto di pseudo-nodo sono semplici ed intuitivi, troviamo *setParent()*, *addChild()*, *addPseudo_parent()*, *addPseudo_child()*, *get_childs()*, *get_pseudo_parents()* la cui semantica è autoesplicativa. La parte più interessante è rappresentata dai metodi per la computazione vera e propria.

3.6.1 Svolgimento della computazione, descrizione ad alto livello

Di seguito una breve descrizione ad alto livello della computazione: affinché le operazioni abbiano inizio, ciascun nodo deve ricevere il *segnale di start*; si può chiamare questa fase *propagazione dello start*. Questo segnale viene propagato dalla radice fino alle foglie, ciascun nodo infatti avrà il compito di propagare il segnale ai suoi figli fino ad arrivare alle foglie, e a quel punto la computazione vera e propria può avere inizio con la fase di *propagazione dei messaggi util*: ciascun nodo foglia computa il messaggio *util* e lo invia al proprio padre; ogni nodo dovrà attendere l'arrivo di tutti i messaggi *util* dei figli prima di poter creare il proprio messaggio. Quando la computazione arriva al nodo radice, questa fase ha termine e comincia una nuova fase di *propagazione dei valori*: la root decide il peso della soluzione ottima e lo comunica all'*handler*, dopodichè sceglie il valore appropriato per la variabile che rappresenta e lo invia a tutti i figli; questa operazione viene eseguita da tutti i nodi fino ad arrivare ai nodi foglia; ciascun nodo deve sempre comunicare all'*handler* il valore prescelto. Quando tutti i nodi figli hanno inviato il proprio valore, l'*handler* dichiara conclusa la computazione comunicando all'utente il risultato.

Da questa breve descrizione si possono identificare dunque tre fasi: una fase di *propagazione dello start*, dall'alto verso il basso, di cui non si fa menzione nel paper di riferimento sul DPOP (è stata implementata per facilitare l'avvio della computazione), la fase di *propagazione dei messaggi util,* dal basso verso l'alto, e infine la *propagazione dei valori* dall'alto verso il basso.

3.6.2 Parallelizzazione del calcolo e sincronizzazione dei thread

Un'anticipazione sull'implementazione, ampiamente discussa nella sezione 3.6.3, riguarda la parallelizzazione della computazione: nella fase di *propagazione dello start*, ogni volta che un nodo deve inviare il segnale di start ai suoi figli, crea un nuovo thread, in questo modo al termine di questa fase ci saranno tanti thread quanti sono i nodi foglia dell'albero. Con questo stratagemma, nella successiva fase di *propagazione dei messaggi util*, i nodi potranno eseguire parallelamente la computazione. Ogni volta che un nodo ha inviato il suo messaggio *util* al padre, invoca il controllo (sul nodo padre) per verificare se tutti i nodi figli hanno inviato il messaggio: se ciò non è verificato, il nodo figlio semplicemente termina e il thread associato viene killato, se invece il nodo risulta essere l'*ultimo* nodo figlio che ha inviato il suo messaggio *util* al padre, il thread associato sposta la sua computazione sul nodo padre, e quindi il meccanismo si ripete ricorsivamente verso l'alto. Arrivati al nodo radice, un solo thread sarà attivo, e avvierà la fase di *propagazione dei valori*, inviando il proprio valore ai figli. L'invio del valore a un figlio comporta la generazione di un nuovo thread sul nodo figlio per gestire la propagazione dei valori. Al termine di questa fase dunque ci saranno nuovamente tanti thread quanti sono i nodi foglia, ciascuno dei quali, dopo aver comunicato all'*handler* l'esito, terminerà killando il relativo thread associato. A questo punto nessun thread è più in esecuzione e la soluzione finale è disponibile.

Questa gestione per le computazioni parallele può sembrare un pò macchinosa ma in realtà è alquanto semplice: ogni nodo ha sempre un processo associato dedicato in ognuna delle fasi della computazione, e al termine dell'elaborazione sul nodo il thread viene killato, con l'accortenza che nella fase di *propagazione dei messaggi util* il thread associato ad un nodo che ha inviato per ultimo l'*util* al padre non viene killato ma si sposta sul nodo padre, in questo modo si evita di killare e ricreare subito dopo un nuovo thread.

E' stata scelta questa implementazione particolare della parallelizzazione per cercare di rendere minime le risorse occupate durante il programma nella computazione: se si fosse associato in maniera fissata un thread ad ogni nodo dell'albero sarebbero stati attivi dall'inizio alla fine molti thread, la maggiorparte dei quali inattivi per un lungo periodo; questa implementazione sarebbe stata una simulazione più vicina al concetto di calcolo distribuito (ogni thread su un nodo rappresenta un'agente completamente scorrelato dagli altri) però sarebbe stata probabilmente meno efficiente. Ai fini di questo progetto dunque è stata preferita una soluzione a metà via tra simulazione esatta ed efficienza, generando sì un thread su ogni nodo, ma terminandolo quando la computazione sul nodo è finita, ed eventualmente riattivandolo (da un thread vicino, padre o figlio) se necessario in un secondo momento.

3.6.3 Svolgimento della computazione, descrizione dettagliata con riferimenti all'implementazione

In questa sezione sarà descritto il flusso delle operazioni, con riferimenti alle chiamate dei metodi effettuate, senza però entrare troppo nel dettaglio del codice di tali metodi. La computazione ha inizio con la fase di *propagazione dello start*, che viene avviata grazie al metodo pubblico *start()* invocato sul nodo radice. Questo metodo crea il primo thread associato alla root e quindi esce,

poiché da questo punto in poi sarà compito del thread svolgere le operazioni. Il thread in particolare invocherà il metodo *_start_signal()*, che si occupa di propagare il segnale di *start* a tutti i nodi: questo metodo infatti avvia ricorsivamente nuovi thread associati ai nodi figli, oppure avvia la fase di *propagazione del messaggio util* sul nodo, a seconda che il nodo corrente sia rispettivamente un nodo intermedio o un nodo foglia:

```
private void _start_signal() {
    if (_childs == null) {  //il nodo è una foglia: parte la fase di
        propagazione dell'util
        _send_util();
    } else {    //il nodo non è una foglia, propaga il segnale di start
        Iterator it = _childs.keySet().iterator();
        while (it.hasNext()) {
            _nthreadcalls++;
            //la chiamata seguente avvia un nuovo thread, ovvero invoca
                su un nuovo processo il metod run() il quale invocherà
               nuovamente questo metodo _start_signal()
            new Thread((PseudoTreeNode)it.next()).start();
        }
    }
}
```

La fase di propagazione degli *util* ha inizio dunque alle foglie, ciascuna sul proprio thread indipendente, a partire dal metodo *_send_util()*. Questo metodo rappresenta il cuore della computazione, in quanto genera il messaggio *util* e lo invia al nodo padre. Per dettagli sulla generazione del messaggio *util*, vedere la sezione 3.6.4. Di rilevo nel metodo *_send_util()* è la generazione della variabile *_argmax*: questa variabile di classe sarà disponibile al termine della generazione del messaggio *util* e rappresenta i valori che massimizzano la soluzione; ha la stessa struttura multidimensionale del messaggio *util* e sarà indispensabile nella fase successiva (per un esempio, vedere la tabella 2 a pagina 9).

Ogni volta che un thread ha terminato di computare il messaggio, lo invia al nodo padre, invocando il suo metodo *_receive_util()*. Questo metodo, eseguito dal thread corrente ma *sul nodo padre*, colleziona il messaggio e controlla se tutti i figli hanno terminato la propria computazione (controllando numero di messaggi ricevuti rispetto al numero di nodi figli), e in caso affermativo può cominciare la propria computazione, utilizzando sempre il metodo *_send_util*():

```
//metodo eseguito sul nodo padre dal thread del nodo figlio
private void _receive_util(TaggedDictionary<Value, Object> u,
    PseudoTreeNode from) {
    //memorizza il messaggio util ricevuto
    _util_received.put(from, u);

    if (_util_received.size() == _childs.size()) {  //se tutti i figli
        hanno terminato
        if (_parent == null) {  //se è la radice
            //avvia la fase di propagazione dei valori
            _send_value();
        } else {    //se non è la radice
```

```
            //il nodo può iniziare la propria computazione del
                messaggio util, notare che il thread che ha attualmente
                il controllo è il thread dell'ultimo nodo figlio che ha
                inviato il messaggio al suo padre (questo nodo),
                utilizzando la tecnica descritta anche nella sezione
                sulla parallelizzazione del codice
            _send_util();
        }
    }
}
```

Come si può notare nel codice, se il nodo corrente è la radice, non deve essere computato nessun messaggio *util* ma deve essere avviata l'ultima fase, ovvero la propagazione dei valori, e per fare ciò viene invocato il metodo *_send_value()*. Questo metodo ha una duplice funzione: sul nodo radice consente di scegliere il peso che ottimizza la soluzione finale, ispezionando banalmente i messaggi *util* ricevuti e quindi scegliendo il proprio valore che rende ottima la soluzione, mentre sui rimanenti nodi semplicemente viene letta la soluzione parziale ricevuta dai nodi padri e, in base a tali valori, viene ispezionata la variabile *_argmax* generata nella fase precedente in modo da ottenere immediatamente il valore del nodo corrente che rende ottima la soluzione. In entrambi i casi viene sempre avvisato l'*handler* del valore scelto, e nel caso del nodo radice viene anche inviato il peso della soluzione. Infine, la propagazione viene avviata ricorsivamente su tutti i figli (se ce ne sono) del nodo corrente:

```
private void _send_value() {
    ...
    //computa il valore che rende ottima la soluzione per il nodo
        corrente (codice omesso)
    ...
    //invia all'handler la soluzione trovata per il nodo corrente
    _handler.receive_solution(this, (Value)...);
    ...
    if (_childs != null) {  //se ci sono ancora figli
        Iterator it = _childs.keySet().iterator();
        while (it.hasNext()) {
            PseudoTreeNode next = (PseudoTreeNode)it.next();
            _handler.value_sent(this, next);     //avvisa l'handler che
                è stato inviato il valore al figlio
            next._receive_value(_received_solution);     //propaga il
                valore
        }
    }
    ...
}
```

Il metodo *_receive_value()* è quello che si occupa di memorizzare la soluzione parziale ricevuta dal nodo padre e di avviare il thread sul nodo corrente per la propagazione dei valori:

```
private void _receive_value(TaggedDictionary<PseudoTreeNode, Value> sol
    ) {
```

```
        _phase = Phase.ValuePropagation;
        _received_solution = sol;
        _nthreadcalls++;

        //esegue un nuovo thread per la fase di propagazione dei valori.
            Questa chiamata invoca ancora una volta il metodo run() che,
            differentemente dalla _start_signal(), causerà la chiamata al
            metodo _send_value(), grazie alla flag _phase che consente al
            metodo run() di avviare la fase corretta
        new Thread(this).start();
    }
```

Per completezza, il metodo *run()* è così composto:

```
    //metodo che implementa l'omonimo dell'interfaccia Runnable e viene
        eseguito all'avvio di un thread
    public void run() {
        ...
        if (_phase == Phase.StartPropagation) {
            ...
            start_signal();
            ...
        }
        else if (_phase == Phase.ValuePropagation) {
            _send_value();
            ...
            if (_nthreads == 0 && _nthreadcalls == 0) {
                //se tutti i thread hanno terminato, comunica all'handler
                    che la computazione è terminata
                _handler.end();
            }
        }
        ...
    }
```

Come si può vedere nel codice del metodo *run()* vi è il test di terminazione: quando ogni thread associato ai nodi foglia ha terminato, ovvero *_send_value()* è terminata, il controllo passa al metodo *run()* che controllerà se ci sono ancora thread in esecuzione: in caso negativo, significa che esso è l'*ultimo thread* ad aver terminato, pertanto comunica all'*handler* il termine della computazione, il quale comunicherà all'utente i risultati, e quindi il calcolo ha termine.

Per rendere ancora più chiaro tutto il flusso di esecuzione, si può schematizzare lo stack delle chiamate in questo modo:

```
start();
_start_signal();     //genera un nuovo thread per la Util propagation
_start_signal();     //genera un nuovo thread per la Util propagation
...
_start_signal();     //genera un nuovo thread per la Util propagation
```

```
    Thread 1:
    _send_util();        //eseguito sul nodo corrente
    _receive_util();     //eseguito sul nodo padre

...

    Thread n:
    _send_util();        //eseguito sul nodo corrente
    _receive_util();     //eseguito sul nodo padre

//quando rimane vivo solo il thread del nodo root:

    Thread del nodo root:
    _send_value();       //eseguito sul nodo root
    _receive_value();    //eseguito sul nodo figlio
    ...
    _receive_value();    //eseguito sul nodo figlio

...

    Thread n:
    _send_value();       //eseguito sul nodo corrente
    _receive_value();    //eseguito sul nodo figlio
    ...
    _receive_value();    //eseguito sul nodo figlio
```

3.6.4 Generazione del messaggio *util*

Il messaggio *util* viene generato seguendo esattamente lo schema dato nel paper sul DPOP: il messaggio dovrà comprendere tante dimensioni quante sono le variabili da cui dipende il nodo corrente. Il caso più semplice è quello monodimensionale, vale a dire in cui il nodo dipende solo dal padre (esempio nella sezione 3.3 a pagina 7); quando la computazione raggiunge il nodo radice non vi sono nodi padri, dunque nessun messaggio viene computato ma viene invece avviata la fase di propagazione dei valori. Il nodo corrente potrebbe però dipendere anche da uno o più pseudo-padri; queste dimensioni aggiuntive collasseranno a 0 quando il nodo non ha pseudo-padri. Infine il nodo potrebbe dipendere, oltre che dal padre e dagli eventuali pseudo-padri, anche dai nodi che vengono "ereditati" dai messaggi *util* ricevuti dai figli; queste dimensioni aggiuntive collasseranno mano a mano che la computazione sale verso l'alto, infatti quando uno dei nodi "ereditati" è proprio il nodo corrente, esso viene scartato dai nodi ereditati (e non sarà più passato ai livelli superiori).

Per generare il vero e proprio messaggio *util*, è necessario ispezionare *tutte le possibili configurazioni* di valori del dominio di ciascuno dei nodi da cui dipende il nodo corrente. Se, per esempio, il nodo x_4 dipende dal padre x_3, dallo pseudo-padre x_2 e ha una dimensione su x_1 ereditata, dovranno essere ispezionate tutte le configurazioni di x_3, x_2, x_1e in più x_4 tenuta in posizione meno significativa, in modo da poter agevolmente massimizzare rispetto a questa variabile (vedere la sezione 3.5 a pagina 8 per maggiori dettagli). Le configurazioni vengono generate grazie alla classe *ValuesIte-*

rator (vedere sezione 3.5 a pagina 8), mentre di seguito c'è un riassunto del codice necessario per computare il messaggio *util*:

```
...
//crea un ValuesIterator tra le variabili da cui dipende il nodo
    corrente
ValuesIterator vi = new ValuesIterator(variables, ...);
..
while(vi.hasNext()) {    //finchè c'è una nuova configurazione di valori
     per le variabili
     ...
     current_configuration = vi.next();
     ...
     Value sum = null;    //inizializza il peso della configurazione
         corrente a "zero"
     ...
     //ora deve ispezionare se la configurazione corrente migliora la
         soluzione

     //la variabile relations contiene l'elenco dei vincoli Constraint
         che il nodo corrente ha con i nodi da cui dipende; questi
         vincoli saranno di tipo BinaryConstraint. Notare che per
         modellare l'"eredità" delle variabili verso l'alto, relations
         contiene anche i messaggi util ricevuti, sottoforma invece di
         NAryConstraint (vedere il breve cenno su NAryConstraint nella
         sezione Struttura Generale del progetto).
     for (int j = 0; j < relations.size(); j++) {
          Constraint r = relations.get(j);     //vincolo corrente
          ArrayList<Variable> nodes = r.getVariables();    //nodi del
              vincolo corrente (binario o n-ario che sia)
          Value v;

          if (r instanceof BinaryConstraint) {     //è un normale vincolo
              binario tra due nodi
               //crea le coppie nodi-valori con i valori della
                   configurazione corrente
               Couple c1 = new Couple((PseudoTreeNode)nodes.get(0),
                   current_configuration.get(nodes.get(0)));
               Couple c2 = new Couple((PseudoTreeNode)nodes.get(1),
                   current_configuration.get(nodes.get(1)));
               ...
               //recupera il valore del nodo corrente associato al vincolo
                    usando le coppie c1, c2
               v = r.getConstraint(...);
               ...
          }
          else {  //è un vincolo n-ario che modella un messaggio util
```

```
            ricevuto
            for (int i = 0; i < nodes.size(); i++) {     //per tutti i
                nodi ereditati
                PseudoTreeNode n = (PseudoTreeNode)nodes.get(i);
                Couple current_couple = new Couple(n,
                    current_configuration.get(n));     //coppia nodo
                    ereditato-valore della configurazione corrente
                ...
            }
            //recupera il valore del nodo corrente usando le coppie
                generate prima
            v = r.getConstraint(...);
        }

        ...
        sum = sum.sum(v);    //somma il peso
        ...
    }

    //se sum calcolato migliora il maxvalue corrente, viene aggiornato
    if (current_maxvalue == null || current_maxvalue.compareTo(sum) <
        0) {
        current_maxvalue = sum;
        //aggiorna anche il valore argmax con la configurazione attuale
        current_argmax = current_configuration.get(this);
    }

    //se la variabile meno significativa (nodo corrente) è arrivata all
        'ultimo valore nella successione di configurazioni allora
        significa che sono state investigate tutte le possibili
        configurazioni del nodo corrente tenendo fissati i nodi da cui
        dipende, dunque il maxvalue e argmax correnti devono essere
        salvati
    if (last_domain_value.equals(current_configuration.get(this))) {
        //memorizza dentro util il maxvalue e argmax corrente (la
            funzione _store_argmax_util() è omessa, semplicemente
            riempie i campi del messaggio util in maniera corretta)
        _store_argmax_util(util, current_maxvalue, current_argmax,
            current_configuration, variables);
        current_maxvalue = null;
        current_argmax = null;
    }

    //sposta la computazione del thread corrente sul nodo padre,
        inviandogli il messaggio util appena computato
    _parent._receive_util(util, this);
```

}

Unica nota di rilevo: dopo aver ispezionato ogni possibile configurazione di valori mantenendo fissate le variabili da cui dipende quella corrente, viene aggiornato sia il messaggio *util*, impostando nella configurazione corrente il *peso massimo* trovato (nel codice, *current_ maxvalue*), sia la variabile *_ argmax*, che sarà invece riempita per la configurazione corrente con il *valore del dominio* che rende *massimo il peso* inserito nel messaggio *util.* Poichè *util* e *argmax* vengono riempite mano a mano nello stesso modo, avranno la stessa struttura multidimensionale ma conterranno valori differenti (la prima i pesi massimi per ogni configurazione, la seconda i valori del dominio della variabile corrente che rendono tali pesi massimi). La funzione *_ store_ argmax_ util*() è quella che si occupa di riempire *util* e *argmax* dopo aver ispezionato ogni configurazione.

4 Risultati sperimentali

I risultati sperimentali sull'efficienza del programma sono abbastanza soddisfacenti, in generale la computazione si comporta bene per istanze di piccole-medie dimensioni, con tempi di esecuzione nella media e talvolta migliori del risolutore ADOPT, con cui sono stati effettuati diversi test comparativi nella sezione 4.1.2.

Sono stati tuttavia riscontrati problemi generalizzati sull'utilizzo della memoria da parte del programma realizzato, specialmente su istanze di grandi dimensioni. Questi problemi sono stati dettagliatamente approfonditi nella sezione 4.2 a pagina 19. A causa di queste problematiche quindi i test sperimentali che seguono sono stati sempre effettuati *utilizzando l'opzione -Xmx2g* che consente di aumentare la memoria massima per lo heap a 2 GB.

4.1 Efficienza in termini di tempo di esecuzione

I tempi di esecuzione sono molto buoni, e vanno da 0 a 1 secondi per istanze piccole, da 1 a 20 per istanze di piccole-medie dimensioni, e da 20 secondi in su per computazioni più complesse[1]. Ricordiamo che non conta molto il numero dei nodi del grafo, quanto più il grado di connessione e specialmente la visita utilizzata per creare lo pseudo-tree.

4.1.1 Confronto computazioni con e senza euristica

L'euristica migliora notevolmente le prestazioni della computazione, come si può vedere da alcune esecuzioni dell'algoritmo su istanze prese casualmente :

[1] Risultati ottenuti con un Intel Core 2 Duo @ 2.66 Ghz, i tempi sono perlopiù indicativi

Istanza utilizzata	N° nodi	Tempo (senza euristica)	Tempo (con euristica)	% miglioramento con euristica
Problem-GraphColor-16_3_3_0.4_r19	16	0,985 secondi	0,171 secondi	+83%
Problem-GraphColor-20_3_3_0.4_r10	20	7,131 secondi	1,359 secondi	+77%
Problem-GraphColor-30_3_2_0.4_r0	30	13,602 secondi	14,633 secondi	-8%
Problem-GraphColor-20_3_3_0.4_r9	20	10,654 secondi	22,113 secondi	-108%
Problem-GraphColor-40_3_2_0.4_r6	40	1 minuto e 35,66 secondi	27,177 secondi	+72%

Tabella 3: Confronto tempi con e senza euristica

Trattandosi di un'euristica è lecito aspettarsi casuali peggioramenti, comunque nel caso medio l'aumento di prestazioni è notevole (è evidente che 5 casi non rappresentano un campione sufficientemente elevato per poter trarre conclusioni sull'incremento di prestazioni, ma durante tutta la realizzazione del progetto sono state testate svariate istanze e la sensazione generale è di un notevole boost prestazionale).

4.1.2 Confronto con il risolutore ADOPT

Il risolutre ADOPT risulta essere in molti casi più lento (e in alcuni casi molto più lento) del programma creato in questo progetto; tuttavia la gestione della memoria di ADOPT sembra funzionare decisamente meglio. Nella tabella sottostante alcune comparazioni con ADOPT:

Istanza utilizzata	Tempo totale (ADOPT)	Tempo totale (progetto corrente)	% miglioramento
Problem-GraphColor-16_3_3_0.4_r19	48,422 secondi	0,171 secondi	+99,6%
Problem-GraphColor-20_3_3_0.4_r10	5 minuti e 0,854 secondi	1,359 secondi	+99,5%
Problem-GraphColor-30_3_2_0.4_r0	3,699 secondi	14,633 secondi	-296%
Problem-GraphColor-20_3_3_0.4_r9	53,502 secondi	22,113 secondi	+59%
Problem-GraphColor-40_3_2_0.4_r6	1 minuto e 47,771 secondi	27,177 secondi	+75%

Tabella 4: Confronto con il risolutore ADOPT

In questo tipo di confronto si evidenzia come in generale il progetto sia efficiente in termini di tempo di esecuzione, raggiungendo quasi sempre risultati migliori rispetto ad ADOPT.

4.2 Problematiche della memoria riscontrate

I test hanno evidenziato un comportamento anormale dell'esecuzione che tende ad incrementare la memoria occupata dal programma senza mai deallocarla. Come è noto, Java non consente di deallocare manualmente la memoria occupata dagli oggetti, ma utilizza il Garbage Collector che grazie ad algoritmi euristici è in grado di liberare la memoria in maniera autonoma, nel momento in cui gli oggetti non sono più referenziabili. Sfortunatamente, nel progetto realizzato la Garbage Collection non sembra essere molto efficace: i problemi sono stati riscontrati nel ciclo che computa il messaggio *util* (vedere sezione 3.6.4 a pagina 14), che potenzialmente viene eseguito milioni di volte in caso di istanze di grafi particolarmente connessi. Questo ciclo non presenta apparentemente particolari problemi (per esempio oggetti non necessari che rimangono referenziati) e sono state provate diverse modifiche per migliorare la situazione, ma senza grandi risultati. Se Java consentisse di deallocare la memoria manualmente probabilmente questi problemi non sarebbero sorti, motivo per cui se il progetto fosse stato realizzato in un linguaggio come C++ probabilmente i risultati sarebbero stati migliori.

Per cercare di risolvere il problema è stata analizzata l'esecuzione della Garbage Collection avviando il programma con le opzioni -verbose:gc -XX:+PrintGCDetails, che consentono di mostrare in output le esecuzioni del GC: in questi test è stato evidenziato come nelle fasi iniziali della computazione, finché è disponibile ancora memoria nello heap, venga avviata più volte una Garbage Collection "slim", che tuttavia fallisce nel suo intento, non liberando molta memoria; quando poi lo heap raggiunge la sua dimensione massima consentita, il GC avvia una Garbage Collection più approfondita, che riesce nell'intento di liberare memoria per il programma ma che rallenta notevolmente la computazione. Il problema risiede dunque nella Garbage Collection "slim", che dovrebbe cercare di liberare la memoria dalle allocazioni recenti di oggetti: in effetti il ciclo incriminato alloca molti oggetti che hanno "vita breve", in quanto all'interno dello scope del ciclo stesso, e che

dovrebbero teoricamente essere non più disponibili nell'iterazione successiva. Non si spiega perciò come mai la Garbage Collection "slim" non riesca a liberare lo heap da questo tipo di oggetti.

Queste problematiche non sono state risolte e restano il punto debole del progetto realizzato, anche se un'analisi più approfondita del codice potrebbe riuscire a risolvere definitivamente il problema. Per poter eseguire comunque i test sperimentali, è stata quindi utilizzata l'opzione -Xmx2g che consente allo heap di potersi estendere fino a 2 GB, in questo modo è stato cercato di ritardare il più possibile l'avvio della Full GC che avrebbe rallentato molto la computazione e avrebbe reso non attendibili le comparazioni del programma con ADOPT. La memoria che Java di default concede per lo heap è di 150 MB, che sperimentalmente risulta essere molto limitata per questo programma: già istanze di medie dimensioni saturerebbero in fretta questa quantità e causerebbero dunque ripetute Full GC rallentando tutta la computazione.