Progetto di Intelligenza Artificiale Implementazione di DPOP per Multi Robot Patrolling

Alberto Lovato
VR356932
alberto.lovato@studenti.univr.it

30 gennaio 2013

Indice

1 Introduzione **1**

2 L'algoritmo **2**
2.1 Complessità 3

3 Struttura del codice **3**
3.1 Modellazione del problema 3
3.2 Esecuzione dell'algoritmo 4

4 Risultati sperimentali **6**
4.1 Istanze casuali lineari 9
4.2 Istanze casuali a griglia 11

1 Introduzione

Il progetto si propone di realizzare, in Java, un algoritmo completo per la risoluzione di problemi di ottimizzazione distribuiti Il programma accetta in ingresso una descrizione testuale del problema *DCOP* – un insieme di variabili e di vincoli *soft* tra di esse – e stampa in uscita un assegnamento ottimo di valori alle variabili.[1]

Per il *Multi Robot Patrolling* le variabili rappresentano i robot (e i valori che esse possono assumere le possibili strategie), mentre i *constraint* definiscono, per ogni stanza da controllare, l'utilità associata a una particolare scelta di strategie per i robot coinvolti.

[1] cioè un assegnamento (non necessariamente l'unico) che produce il valore massimo della funzione di *utility*.

Un esempio di descrizione è il seguente[2]

```
AGENT 1
VARIABLE 0 1 2
VARIABLE 1 1 2
CONSTRAINT 0 1 0
F 0 956
F 1 956
CONSTRAINT 1 1 0
F 0 957
F 1 957
CONSTRAINT 2 1 0 1
F 0 0 992
F 0 1 992
F 1 0 992
F 1 1 992
CONSTRAINT 3 1 1
F 0 1000
F 1 1000
CONSTRAINT 4 1 1
F 0 980
F 1 965
```

In questo caso ci sono due variabili (robot) con due possibili strategie ciascuno. A ogni variabile sono associati due vincoli unari, mentre il vincolo 2 lega entrambe le variabili. Gli assegnamenti possibili sono quattro, di cui due ottimi, $(0, 0)$ e $(1, 0)$, che producono il valore totale di utility 4885 – gli altri due, $(0, 1)$ e $(1, 1)$, producono il valore 4870.

2 L'algoritmo

L'algoritmo implementato – *DPOP*, *Distributed Pseudotree Optimization Procedure* o *Dynamic Programming Optimization Protocol*, descritto in [1] e in [2] – organizza gli agenti (ognuno dei quali controlla una variabile) in uno *PseudoTree*, costruito mediante una visita *Depth First* del grafo del problema.

Per creare uno *PseudoTree* a bassa *induced width* (si veda più avanti) risulta utile applicare nella visita DFS l'euristica *MCN* [2], che consiste nello scegliere come radice e successivamente come prossimo nodo da considerare il nodo con il numero maggiore di vicini (il nodo più connesso). L'opzione `-mcn` passata prima del nome del file consente di attivare l'uso di questa euristica.

Questa organizzazione serve a isolare il calcolo del valore delle variabili non collegate tra loro in rami distinti. In questo modo si può sfruttare l'indipendenza delle computazioni per un'esecuzione parallela (e/o distribuita) dell'algoritmo.

[2] Il programma assume che l'ingresso non abbia errori di sintassi.

2.1 Complessità

Sia i messaggi di tipo *UTIL* che quelli *VALUE* sono tanti quanti i *tree edge* dello *pseudotree*. La dimensione massima dei messaggi di tipo *UTIL* è pari alla *induced width* dello *pseudotree*, cioè al numero massimo di archi uscenti da ognuno dei sottoalberi (ossia la massima dimensione del separatore).

3 Struttura del codice

Il progetto è organizzato in *package* secondo la struttura qui sotto:

- `dpop/`
  - `algorithm/`
    - `Agent`
    - `AgentExecutor`
    - `AgentTask`
    - `PseudoTree`
  - `dcop/`
    - `Assignment`
    - `Constraint`
    - `DCOP`
    - `Graph`
  - `Main`
  - `Solver`

Due sono i package principali: `dcop` e `algorithm`.

Le classi presenti in `dcop` servono alla modellazione di un problema DCOP, mentre in `algorithm` il codice si occupa più in particolare dell'implementazione delle operazioni dell'algoritmo *DPOP*.

Nelle due sottosezioni seguenti vengono illustrate in dettaglio le classi appartenenti a questi due gruppi.

3.1 Modellazione del problema

Nella classe `DCOP` è contenuto il codice che esegue il *parsing* del file di input e contemporaneamente crea il grafo del problema.

`Assignment` rappresenta un'assegnamento a delle variabili (l'implementazione usa una mappa ordinata). Il messaggio *VALUE* trasmesso ai figli consiste in un riferimento a un'istanza di questa classe.

Un `Constraint` è un vincolo *soft*. Qui tra l'altro è implementato il codice di join e di massimizzazione.

Infine `Graph` è un semplice grafo avente nodi etichettati con interi.

3.2 Esecuzione dell'algoritmo

Il nucleo principale del programma è la classe `Solver`. Nel suo metodo `start()` viene caricato il problema dal *file* fornito, viene costruito lo *pseudotree* corrispondente e viene avviato l'algoritmo *DPOP*. Lo *pseudotree* (si veda la classe `PseudoTree`) è composto di oggetti di tipo `Agent`; ogni agente ha un identificatore (un numero intero) univoco, la lista dei vincoli che hanno nello *scope* la variabile dell'agente, le informazioni relative allo *pseudotree* (genitore, lista dei figli, insiemi di *pseudofigli* e *pseudogenitori*) e altri campi che servono per la successiva esecuzione dell'algoritmo.

Data la natura distribuita della computazione dei vari agenti la scelta più logica è stata quella di far eseguire ognuno dei *task* indipendentemente dagli altri. Infatti ad ogni oggetto di classe `Agent` è associato un oggetto di classe `AgentTask`, che implementa l'interfaccia `Callable`. Il codice della classe `AgentExecutor` sottopone tutti gli `AgentTask` a un esecutore[3] e ne attende la terminazione.

Ogni oggetto `Agent` contiene i campi `utility` di tipo `Constraint` e `parentValue` di tipo `Assignment`. Questi campi verranno modificati durante l'esecuzione di *DPOP*, e alla fine conterranno l'*utility* relativa all'agente e l'assegnamento ottimo a tutte le variabili. L'oggetto `parentValue` è unico, viene creato dall'agente alla radice e aggiornato da tutti gli altri con il valore della rispettiva variabile.[4] Questo valore può essere ricavato chiamando il metodo `getAssignment()` di `AgentExecutor`.

Nel metodo `call()` di `AgentTask` sono esplicitati i vari passi di *DPOP* relativi all'agente corrispondente. Viene quindi aggiornata la lista dei vincoli (i vincoli già considerati a livelli più bassi dello *pseudotree* vengono rimossi) e vengono avviate le fasi di propagazione di *UTIL* e *VALUE*.

Nella fase di *UTIL propagation* l'agente deve attendere che tutti i figli calcolino la propria *utility*, calcolare la propria e inviarla al genitore.

Nella fase di *VALUE propagation* l'agente deve aspettare che il genitore calcoli il valore della sua variabile che massimizza l'*utility* e aggiorni l'assegnamento `parentValue`, fare lo stesso per la sua variabile relativamente alla sua *utility* e inviare l'assegnamento ai figli.

Ovviamente le foglie non hanno bisogno di aspettare alcuna *utility*, e la radice non deve aspettare il valore dal genitore.

A livello implementativo i *task* sono sincronizzati usando delle **code bloccanti**: ogni agente ha una coda per le *utility* dei figli e una per il *proprio* valore di *VALUE*.
Quando un agente ha pronta la sua *utility* la mette (`put()`) nella coda del genitore, mentre quando ha calcolato il proprio *VALUE* mette l'assegnamento

[3]un oggetto di tipo `ExecutorService`, con implementazione restituita da `Executors.newFixedThreadPool(int)` – viene usato un *thread* per ogni *task*.

[4]Il metodo che aggiunge una coppia variabile/valore è dichiarato `synchronized`, poiché comporta una modifica strutturale della *TreeMap*.

aggiornato nella *propria* coda.
Per attendere le utility dei figli l'agente prende (`take()`) un `Constraint` dalla propria coda delle *utility* tante volte quanti sono i figli. Allo stesso modo per attendere *VALUE* prende dalla coda *del genitore* l'assegnamento aggiornato.

Il metodo `BlockingQueue.take()` è bloccante: il chiamante sospende la sua esecuzione fin quando nella coda non ci sarà un valore da prendere.

4 Risultati sperimentali

Nella tabella che segue sono riportati i grafi relativi ai problemi usati per testare il codice. Alcuni di questi condividono la stessa struttura e quindi sono stati raggruppati nella stessa riga.

Nome file	Rete	PseudoTree	PseudoTree, MCN
`big.txt`			
`in01.txt`, `in05.txt`, `in06.txt`, `dcopInput.cop`			

Nome file	Rete	PseudoTree	PseudoTree, MCN
`in02.txt`, `in03.txt`, `in04.txt`, `mrpGenerated.cop`	0 — 1	0 → 1	0 → 1
`dcopInput2.cop`, `d3values.txt`	0 — 1, 0 — 2, 2 — 3	0 → 1, 0 → 2, 2 → 3	0 → 2, 0 → 1, 2 → 3

Di seguito sono stampate le statistiche dell'esecuzione dell'algoritmo senza uso di euristiche nella generazione dello PseudoTree.

Nome file	T	U	V	Dim. U	A	Val
`big.txt`	3.753033	8	8	6	1 2 1 0 1 0 1 1 1	98648
`d3values.txt`	1.189761	3	3	2	1 2 1 0	54075
`dcopInput.cop`	1.130033	3	3	8	0 0 0 1	46639
`dcopInput2.cop`	1.372594	3	3	2	1 1 1 0	53315
`in01.txt`	1.387181	3	3	8	0 0 1 0	46755
`in02.txt`	0.297791	1	1	2	1 1	4878
`in03.txt`	0.651285	1	1	2	1 1	4895
`in04.txt`	0.365215	1	1	2	1 1	4742
`in05.txt`	1.710573	3	3	8	0 0 0 1	45091
`in06.txt`	1.768452	3	3	8	0 0 0 1	46754
`mrpGenerated.cop`	0.637057	1	1	2	0 0	4885

Qui sotto invece ci sono le statistiche registrate usando l'euristica MCN.

Nome file	T	U	V	Dim. U	A	Val
`big.txt`	2.941412	8	8	6	1 2 1 0 1 0 1 1 1	98648
`d3values.txt`	0.970124	3	3	2	1 2 1 0	54075
`dcopInput.cop`	1.320196	3	3	8	0 0 0 1	46639
`dcopInput2.cop`	0.92263	3	3	2	1 1 1 0	53315
`in01.txt`	1.695875	3	3	8	0 0 1 0	46755
`in02.txt`	0.284329	1	1	2	1 1	4878
`in03.txt`	0.441514	1	1	2	1 1	4895
`in04.txt`	0.362066	1	1	2	1 1	4742
`in05.txt`	1.733435	3	3	8	0 0 0 1	45091
`in06.txt`	1.743838	3	3	8	0 0 0 1	46754
`mrpGenerated.cop`	0.660005	1	1	2	0 0	4885

Legenda:

- **T** tempo medio di esecuzione in millisecondi su 1000 istanze dell'algoritmo
- **U** numero di messaggi *UTIL* scambiati nel corso dell'esecuzione dell'algoritmo
- **V** numero di messaggi *VALUE*
- **Dim. U** dimensione massima dei messaggi *UTIL* - in numero di tuple/combinazioni
- **A** assegnamento trovato dall'algoritmo
- **Val** valore della funzione di *utility* corrispondente all'assegnamento trovato

4.1 Istanze casuali lineari

In questa sezione sono presentati i risultati dell'esecuzione di DPOP su istanze lineari generate in modo casuale. Queste istanze sono generate usando la classe `dpop.RandomDCOPGeneratorLinear`[5], immettendo nell'ordine i parametri:

- numero di robot
- numero massimo di strategie per robot
- numero massimo di stanze per robot
- massima sovrapposizione (overlap) tra robot (in numero di stanze)
- valori minimo e massimo della funzione di ogni vincolo/stanza

I problemi considerati sono i seguenti:

Nome file	Parametri	Numero dei vincoli generati
`r10a.txt`	10 3 5 3 -10 100	16
`r10b.txt`	10 4 6 3 -100 1000	25
`r100a.txt`	100 4 10 4 -100 1000	391
`r100b.txt`	100 5 6 2 -10 100	221
`r200a.txt`	200 5 10 6 -10 100	494
`r200b.txt`	200 4 5 2 -1 10	422
`r200c.txt`	200 6 10 3 -1 10	901
`r500.txt`	500 5 8 3 -1 10	1562
`r600.txt`	600 4 20 5 -1 10	4928
`r700.txt`	700 4 20 5 -1 10	5389
`r800.txt`	800 4 30 10 -1 10	8629
`r900.txt`	900 4 30 10 -1 10	9583
`r1000.txt`	1000 4 6 2 -1 10	2531

[5] avviabile anche tramite lo script `randoml`

Rsultati senza uso di euristiche:

Nome file	T	U	V	Dim. U
`r10a.txt`	9.261533	9	9	27
`r10b.txt`	4.785273	9	9	16
`r100a.txt`	116.505037	99	99	144
`r100b.txt`	35.546686	98	98	100
`r200a.txt`	10260.920727	199	199	115200
`r200b.txt`	59.463475	199	199	36
`r200c.txt`	71.819337	199	199	75
`r500.txt`	521.834562	499	499	1125
`r600.txt`	568.173566	599	599	96
`r700.txt`	852.393929	699	699	1536
`r800.txt`	1099.062686	799	799	972
`r900.txt`	1922.880896	899	899	3072
`r1000.txt`	1156.268753	999	999	144

Risultati con l'uso dell'euristica MCN:

Nome file	T	U	V	Dim. U
`r10a.txt`	4.761093	9	9	54
`r10b.txt`	3.51396	9	9	12
`r100a.txt`	41.151972	99	99	108
`r100b.txt`	29.333032	98	98	100
`r200a.txt`	852.357433	199	199	14400
`r200b.txt`	46.210988	199	199	36
`r200c.txt`	58.897417	199	199	144
`r500.txt`	379.593334	499	499	1125
`r600.txt`	303.482897	599	599	72
`r700.txt`	703.822452	699	699	256
`r800.txt`	881.7316	799	799	108
`r900.txt`	1143.829944	899	899	1536
`r1000.txt`	974.482991	999	999	192

Legenda:

- **T** tempo medio di esecuzione in millisecondi
- **U** numero di messaggi *UTIL* scambiati nel corso dell'esecuzione dell'algoritmo
- **V** numero di messaggi *VALUE*
- **Dim. U** dimensione massima dei messaggi *UTIL* - in numero di tuple/combinazioni

4.2 Istanze casuali a griglia

Le istanze a griglia qui considerate sono state generate usando la classe `dpop.RandomDCOPGeneratorGrid`[6], immettendo nell'ordine i parametri:

- dimensione orizzontale totale
- dimensione verticale totale
- dimensione orizzontale della cella
- dimensione verticale della cella
- massima sovrapposizione (overlap) tra robot (in orizzontale)
- massimo numero di strategie per robot
- valori minimo e massimo della funzione di ogni vincolo/stanza

Nome file	Parametri	Numero dei robot generati
`rg5x3.txt`	5 3 2 2 1 2 -1 10	30
`rg5x5.txt`	5 5 2 2 1 2 -1 10	39
`rg10x5.txt`	10 5 4 2 1 2 -1 10	36
`rg10x10.txt`	10 10 5 3 1 2 -1 10	39
`rg10x15.txt`	10 15 3 2 1 2 -1 10	162
`rg10x20.txt`	10 20 3 2 1 2 -1 10	200
`rg15x15.txt`	15 15 4 2 1 2 -1 10	166
`rg20x10.txt`	20 10 4 2 1 2 -1 10	140

[6] avviabile anche tramite lo script `randomg`

Rsultati senza uso di euristiche:

Nome file	T	U	V	Dim. U
`rg5x3.txt`	189.56131	29	29	4096
`rg5x5.txt`	406.73496	38	38	4096
`rg10x5.txt`	54.219326	35	35	1024
`rg10x10.txt`	133.159921	38	38	1024
`rg10x15.txt`	2544.053908	161	161	32768
`rg10x20.txt`	6364.247373	199	199	32768
`rg15x15.txt`	57998.603654	165	165	524288
`rg20x10.txt`	2240.453274	139	139	16384

Risultati con l'uso dell'euristica MCN:

Nome file	T	U	V	Dim. U
`rg5x3.txt`	12.221975	29	29	64
`rg5x5.txt`	22.668501	38	38	128
`rg10x5.txt`	6.913245	35	35	16
`rg10x10.txt`	13.902776	38	38	64
`rg10x15.txt`	163.698769	161	161	2048
`rg10x20.txt`	1390.73104	199	199	16384
`rg15x15.txt`	964.111801	165	165	8192
`rg20x10.txt`	22146.840789	139	139	131072

Legenda:

- **T** tempo medio di esecuzione in millisecondi
- **U** numero di messaggi *UTIL* scambiati nel corso dell'esecuzione dell'algoritmo
- **V** numero di messaggi *VALUE*
- **Dim. U** dimensione massima dei messaggi *UTIL* - in numero di tuple/combinazioni

Riferimenti bibliografici

[1] Petcu, Adrian and Faltings, Boi (August 2005), DPOP: A Scalable Method for Multiagent Constraint Optimization, Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005, Edinburgh, Scotland, pp. 266-271, http://liawww.epfl.ch/Publications/Archive/Petcu2005.pdf

[2] Petcu, Adrian, A Class of Algorithms for Distributed Constraint Optimization, 2009, Ios Press, http://books.google.it/books?id=y9lZ4Y8zEtMC