UNIVERSITÀ DEGLI STUDI DI PARMA

Facoltà di Scienze
Matematiche Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea Triennale

Progettazione e realizzazione di uno strumento per la risoluzione distribuita e decentralizzata di problemi di soddisfacimento di vincoli.

Candidato:
Andrea Zambon

Relatore:
Prof. Federico Bergenti

Anno Accademico 2008/2009

Indice

Introduzione

I CSP (Constraint Satisfaction Problem, Problemi di Soddisfacimento Vincoli) sono un tipo particolare di problemi che prevede di assegnare dei valori ad una serie di variabili in maniera consistente con una serie di vincoli. Un esempio banale di tale tipo di problema è il seguente: si immagini di dovere assegnare dei valori a tre variabili (chiamate x_1, x_2 ed x_3), ciascuna delle quali può assumere i valori 1, 2 o 3. Si stabilisce poi che tutte e tre le variabili debbano avere valori diversi tra loro. Una soluzione immediata del problema potrebbe essere $x_1 = 1$; $x_2 = 2$; $x_3 = 3$. Supponiamo ora di voler aggiungere il vincolo che il valore di x1 deve essere maggiore del valore di x_3. In questo caso la soluzione precedente non è più accettabile, ma è comunque possibile trovare la soluzione $x_1 = 3$; $x_2 = 2$; $x_3 = 1$. Notare che in entrambi i casi è stata trovata solo una soluzione tra tutte quelle possibili. Certi CSP hanno molte soluzioni possibili, certi hanno solo una soluzione, mentre altri non hanno nessuna soluzione.

Più formalmente, un CSP è costituito da n variabili, indicate con x_0, x_1, ..., x_{n-1}. Per ciascuna di queste variabili esiste un dominio finito di valori numerici: i possibili valori della variabile x_k sono tutti gli elementi dell'insieme D_k. Dunque si hanno n insiemi finiti D_0, D_1, ..., D_{n-1}. Inoltre sono presenti dei vincoli sui valori delle variabili. Ciascun vincolo $p_k(x_{k_1}, x_{k_2}, ..., x_{k_j})$ interessa le variabili x_{k_1}, x_{k_2}, ..., x_{k_j}, ed è una relazione tra D_{k_1}, D_{k_2}, ..., D_{k_j}. Un vincolo viene detto soddisfatto se e solo se i valori assegnati a tutte le variabili che esso interessa soddisfano la relazione.

L'importanza di questi problemi viene dalla loro generalità: molti problemi possono essere ricondotti a CSP, dunque avere un metodo per risolvere efficientemente e velocemente i CSP significa poter risolvere efficientemente tutta una serie di altri problemi [RN02]. In particolare, la definizione di CSP è una generalizzazione del problema di soddisfacibilità booleano (SAT). Tale problema appartiene alla classe dei problemi NP-completi, cioè quei problemi a cui ogni altro problema nelle classi P ed NP può essere ridotto in tempo polinomiale. Dato che i CSP sono una generalizzazione del problema

SAT, si ha immediatamente che il problema SAT può essere formalizzato come CSP. Questo significa che anche CSP è un problema NP-completo nel caso generale.

Il fatto che CSP sia un problema NP-completo nel caso generale lo rende contemporaneamente importante e difficile da risolvere in maniera efficiente. Infatti, tutti gli algoritmi oggi noti per risolvere problemi NP-completi hanno un tempo di esecuzione che è superpolinomiale nel caso peggiore. Questo significa che ogni algoritmo oggi noto che risolve esattamente un problema NP-completo ha almeno un caso in cui il tempo impiegato per risolvere il problema è superpolinomiale, il che significa che il tempo necessario per risolvere il problema cresce in maniera estremamente rapida con l'aumentare della dimensione del problema. Inoltre, se è vero che $P \neq NP$, si ha che non può esistere un algoritmo che risolva esattamente un problema NP-completo con complessità polinomiale nel caso pessimo.

Tutto questo significa che è importante cercare di migliorare per quanto possibile gli algoritmi usati per risolvere tale problema, cercando di ottenere ogni possibile piccola riduzione di complessità. Questo anche nel caso si dimostrasse impossibile trovare un algoritmo di complessità polinomiale per risolvere il problema.

D'altra parte, lo sviluppo della tecnologia elettronica ed informatica negli ultimi anni si sta dirigendo sempre più verso sistemi di calcolo distribuiti. Questo permette di continuare ad aumentare la potenza di calcolo disponibile, ma rende necessario adattare gli algoritmi usati per poter sfruttare l'intrinseco parallelismo dei sistemi distribuiti. Considerata la difficoltà di risolvere in maniera efficiente tale problema (CSP), è evidente che la potenza di calcolo aggiuntiva fornita da queste nuove tecnologie si dimostra molto utile per cercare almeno di arginare gli effetti derivanti dall'uso di algoritmi con complessità superpolinomiale.

Si aggiunga a questo anche la progressiva spinta verso l'adozione di nuovi linguaggi di programmazione ad alto livello (per esempio Java), iniziata dal settore della creazione di applicativi commerciali, ma ora emergente anche nel settore del calcolo ad alte prestazioni.

Si rende quindi necessario sviluppare dei sistemi software per la risoluzione dei problemi descritti sopra tenendo in considerazione queste nuove tendenze ed opportunità del panorama informatico globale.

I vantaggi di un sistema software come quello descritto possono essere molteplici: l'uso di un linguaggio di programmazione a più alto livello permette di esprimere concetti più complessi già nel codice sorgente, rendendo più comprensibile il programma e facilitan-

done la manutenzione e la modifica. Inoltre, la disponibilità di costrutti a più alto livello può permettere di ridurre le dimensioni stesse del programma, facilitandone ulteriormente la manutenzione e la modifica. In aggiunta, un sistema sviluppato per sfruttare i nuovi sistemi di calcolo paralleli e distribuiti promette di essere significativamente più veloce di un sistema basato su un singolo percorso di esecuzione, pur considerando gli inevitabili overhead in cui un sistema distribuito incorre per permettere le comunicazioni tra i nodi di calcolo. A questo si aggiunga anche la potenziale scalabilità del sistema, che può trarre vantaggio dall'aggiunta di ulteriori nodi di calcolo.

Da tutte queste considerazioni si comprende come un sistema distribuito per la risoluzione di CSP e scritto in un linguaggio ad alto livello possa essere utile.

Nel resto di questo lavoro di tesi si assumerà che tutti i vincoli del problema da trattare siano vincoli unari o binari, cioè vincoli che coinvolgono una o due variabili del problema. Questo non è restrittivo: se è presente un vincolo che interessa tre o più variabili è possibile scegliere tutte le variabili che compaiono nel vincolo eccetto una e sostituirle con una singola variabile che ha come valori possibili il prodotto cartesiano dei domini di tutte le variabili che erano state scelte. Una volta effettuata questa operazione, è possibile modificare tutti i vincoli che interessavano una o più variabili tra quelle che sono state unite, in modo che facciano riferimento ai valori della nuova variabile composita [BvB98].

Questa tecnica permette di concentrarsi solo su vincoli unari e binari, ma ha lo svantaggio di portare alla creazione di variabili con domini di dimensioni enormi, difficili (o quantomeno scomodi ed ingombranti) da rappresentare nella memoria di un calcolatore. Nonostante questo, il numero di possibili istanziazioni di tutte le variabili del problema rimane inalterato anche applicando questa tecnica.

Passiamo ora ad illustrare alcuni algoritmi utilizzabili per risolvere questi problemi.

Capitolo 1

Algoritmi per la Risoluzione di CSP

Come detto nel capitolo precedente, trovare un algoritmo per risolvere nella maniera più efficiente possibile i CSP è molto importante. Esiste tutta una serie di algoritmi per questo scopo, la maggior parte dei quali, però, non si adatta all'esecuzione distribuita descritta nel capitolo precedente. Per completezza, però, verranno presentati i principali algoritmi utilizzabili per la risoluzione di CSP.

Gli algoritmi utilizzabili per la risoluzione di CSP sono principalmente di due tipi: algoritmi di ricerca ed algoritmi di consistenza. Gli algoritmi di ricerca provano varie soluzioni del problema, verificando di volta in volta se quella generata è la soluzione corretta. Dato che questo metodo prevede di ricercare la soluzione corretta (se essa esiste) tra tutte le possibili istanziazioni delle variabili del problema, questi sono detti algoritmi di ricerca. Gli algoritmi di consistenza, invece, usano i vincoli presenti tra le variabili del problema per trovare ed eliminare i valori del dominio di ciascuna variabile che, se scelti, non permettono di trovare nessuna soluzione al problema.

1.1 Algoritmi di ricerca

1.1.1 Generate and test

Il più semplice algoritmo di ricerca è quello detto generate and test (GT). Questo algoritmo prevede di provare, una alla volta, tutte le possibili istanziazioni di tutte le variabili del problema, controllando ogni volta se si è trovata la soluzione al problema. Questo algoritmo, pur essendo estremamente semplice da implementare, ha il limite evidente di non essere in grado di risolvere rapidamente problemi di grandi dimensioni. Infatti questo algoritmo deve provare, una per volta, tutte le combinazioni di valori di tutte le variabili del problema. Il numero massimo di tentativi che l'algoritmo deve potenzialmente

fare è dato dalla produttoria del numero di elementi nel dominio di ciascuna variabile. Per fare un esempio, per risolvere il problema delle 4 regine (questo problema prevede di avere una scacchiera di dimensioni 4x4 e di dovere posizionare su di essa 4 regine in modo che nessuna regina minacci nessun'altra), questo algoritmo dovrebbe provare $4^4 = 256$ combinazioni possibili, ma già per risolvere il problema delle 8 regine (come sopra, ma la scacchiera ha dimensioni 8x8 e si devono posizionare 8 regine) dovrebbe provare $8^8 = 2^{24} = 16777216$ combinazioni, e il problema delle 10 regine richiederebbe $10^{10} = 10000000000$ tentativi. È facile capire perché questo algoritmo non sia adatto per risolvere problemi di grandi dimensioni.

```
procedure GT(Variables, Constraints)
  for each Assignment of Variables do
    if consistent(Assignment, Constraints) then
      return Assignment
    end if
  end for
  return fail
end GT
```

```
procedure consistent(Assignment, Constraints)
  for each C in Constraints do
    if C is not satisfied by Assignment then
      return fail
    end if
  end for
  return true
end consistent
```

Gli unici vantaggi di questo algoritmo sono l'estrema semplicità e la completezza, cioè il fatto di riuscire a stabilire sia che un problema ha soluzione, sia che un problema non ha soluzione. Tale algoritmo, inoltre, è facilmente parallelizzabile, anche se i miglioramenti di prestazioni ottenibili con un'esecuzione parallela dell'algoritmo non sono tali da compensare l'intrinseca inefficienza dell'algoritmo stesso. Di fatto, con lo stesso sforzo

richiesto per parallelizzare l'algoritmo GT, è possibile implementare delle alternative che sono notevolmente più efficienti.

Il problema principale è dato dal fatto che il controllo dei vincoli avviene alla fine, dopo che l'assegnamento è stato completamente eseguito, e che il generatore dei valori delle variabili non tiene conto di quali vincoli sono stati violati. Questo significa che il generatore continuerà a produrre valori che potrebbero essere facilmente scartati a priori come non validi e che invece devono essere testati uno per uno.

1.1.2 Standard backtracking

Un'alternativa molto migliore è l'algoritmo di backtracking (BT). Questo algoritmo di ricerca prova ad assegnare un valore ad una variabile e quindi controlla che tutti i vincoli verificabili (cioè quelli tra variabili assegnate) siano soddisfatti. Se tutti i vincoli verificabili sono soddisfatti, l'algoritmo procede con la variabile successiva, altrimenti prova ad assegnare un nuovo valore all'ultima variabile a cui aveva assegnato un valore, verificando nuovamente se tutti i vincoli verificabili sono soddisfatti. Se si esauriscono tutti i valori del dominio di una certa variabile, si ritorna indietro alla variabile precedente e si cerca di modificare il valore di tale variabile.

Il vantaggio di questo algoritmo rispetto all'algoritmo GT è che il controllo della violazione dei vincoli avviene molto prima, ogni volta che viene assegnato un valore ad una variabile. Questo significa che se un particolare valore assegnato ad una certa variabile genera una violazione di un vincolo, l'algoritmo BT cambia immediatamente il valore di tale variabile, senza nemmeno guardare le variabili non ancora assegnate. Questo permette di scartare in una volta sola una grande quantità di possibili assegnamenti che sicuramente non contengono una soluzione per il problema.

```
procedure BT(Variables, Constraints)
  BT-1(Variables, ∅, Constraints)
end BT
```

```
procedure BT-1(Unlabelled, Labelled, Constraints)
  if Unlabelled = ∅ then
    return Labelled
  end if
  pick first X from Unlabelled
  for each value V from D_x do
    if consistent({(X, V)} ∪ Labelled, Constraints) then
      R ← BT-1(Unlabelled/{X}, {(X, V)} ∪ Labelled, Constraints)
      if R ≠ fail then
        return R
      end if
    end if
  end for
  return fail
end BT-1
```

```
procedure consistent(Labelled, Constraints)
  for each C in Constraints do
    if all variables from C are Labelled then
      if C is not satisfied by Labelled then
        return fail
      end if
    end if
  end for
  return true
end consistent
```

1.1.3 Backjumping

L'algoritmo BT, pur essendo un notevole miglioramento rispetto all'algoritmo GT, presenta comunque delle inefficienze. Queste sono dovute al fatto che l'algoritmo, ogni volta che rileva un conflitto, non risale alla vera causa del conflitto, ma si limita a provare un valore diverso per l'ultima variabile istanziata. Questo significa che i conflitti non vengo-

no risolti immediatamente, ma continuano ad essere presenti. Dato che non è possibile trovare una soluzione finché tutti i conflitti non vengono risolti, ne risulta che il tempo trascorso tra il momento in cui si rileva un conflitto e quello in cui lo si risolve dovrebbe essere minimizzato. Inoltre, ogni tentativo di ricerca della soluzione intrapreso in una tale situazione non può avere successo, dato che il conflitto non ancora risolto impedisce di trovare una soluzione.

```
procedure BJ(Variables, Constraints)
  BJ-1(Variables,∅,Constraints,0)
end BJ
```

```
procedure BJ-1(Unlabelled, Labelled, Constraints, PreviousLevel)
  if Unlabelled = ∅ then
    return Labelled
  end if
  pick first X from Unlabelled
  Level ← PreviousLevel + 1
  Jump ← 0
  for each value V from D_X do
    C ← consistent({(X, V, Level)} ∪ Labelled, Constraints, Level)
    if C = fail(J) then
      Jump ← max(Jump, J)
    else
      Jump ← PreviousLevel
      R ← BJ-1(Unlabelled/{X},{(X, V, Level)} ∪ Labelled, Constraints, Level)
      if R ≠ fail(Level) then
        return R
      end if
    end if
  end for
  return fail(Jump)
end BJ-1
```

```
procedure consistent(Labelled, Constraints, Level)
  J ← Level
  NoConflict ← true
  for each C in Constraints do
    if all variables from C are Labelled then
      if C is not satisfied by Labelled then
        NoConflict ← false
        J ← min (J, max (L|X ∈ C ∧ (X, V, L) ∈ Labelled ∧ L < Level))
      end if
    end if
  end for
  if NoConflict then
    return true
  else
    return fail(J)
  end if
end consistent
```

Per questi motivi è stato introdotto l'algoritmo di backjumping (BJ). Questo algoritmo si basa sulla stessa idea generale del backtracking, ma quando rileva un conflitto BJ determina qual è la prima variabile che può risolvere il conflitto modificando il proprio valore ed esegue un backtracking fino a tale variabile.

Questo minimizza il tempo tra il rilevamento e la correzione di un conflitto, eliminando nel contempo tutti gli assegnamenti di valori alle variabili che includono il conflitto rilevato, riducendo ulteriormente lo spazio di ricerca della soluzione.

Anche questo algoritmo ha delle limitazioni, come il fatto che ogni volta che un vincolo è violato esso deve trovare quale variabile causa la violazione. Questo deriva dal fatto che i conflitti rilevati non vengono salvati, ma devono essere rilevati ad ogni passo della computazione.

Esistono altri algoritmi che cercano di risolvere questo problema, come gli algoritmi backchecking (BC) e backmarking (BM).

1.2 Algoritmi di consistenza

1.2.1 Node consistency

Passiamo ora alla famiglia degli algoritmi di consistenza. Come detto, questi algoritmi usano i vincoli del problema per eliminare dei valori dal dominio delle variabili. Dato che, come detto in precedenza, il numero di possibili assegnamenti di valori alle variabili del problema è dato dalla produttoria del numero di elementi nel dominio di ogni variabile del problema, si ha che eliminando anche pochi valori dal dominio di alcune variabili, il numero totale di possibili assegnamenti si riduce in maniera notevole.

Gli algoritmi di consistenza vengono classificati in base alla lunghezza della catena di vincoli che essi considerano. Le variabili e i vincoli tra di esse vengono visti come i componenti di un grafo in cui le variabili sono i nodi e i vincoli (unari e binari) sono gli archi. Iniziamo, quindi, considerando gli algoritmi più semplici in assoluto, quelli detti di consistenza di nodo (node consistency, NC). Una variabile x_k (che, come detto, può essere considerata come il nodo di un grafo) si definisce nodo-consistente se e solo se tutti i valori del relativo dominio D_k soddisfano tutti i vincoli unari che interessano tale variabile. In altre parole questo significa che una variabile è nodo-consistente se il suo dominio non comprende alcun valore che viola un qualche vincolo unario di tale variabile.

Si ha immediatamente che se una variabile non è nodo-consistente, esiste almeno un valore nel dominio di tale variabile che non permette di trovare una soluzione al problema. Questo perché, essendo la variabile non nodo-consistente, è presente un valore che viola un vincolo unario. Se si sceglie tale valore per la variabile, non sarà possibile trovare una soluzione per il problema indipendentemente dal valore assegnato a tutte le altre variabili, dato che il vincolo unario rimarrà sempre violato. Quindi, rimuovendo dal dominio di ogni variabile tutti i valori che violano uno o più vincoli unari, si ha la certezza di non eliminare nessuna soluzione del problema.

Di conseguenza è possibile applicare un semplice algoritmo per eliminare tutti i valori che rendono le singole variabili non nodo-consistenti. Tale algoritmo prova tutti i valori di ogni singola variabile e per ciascuna verifica se viene violato qualche vincolo unario. In caso affermativo, tale valore viene rimosso dal dominio della variabile.

```
procedure NC(G)
  for each variable X in nodes(G) do
    for each value V in the domain D_X do
      if unary constraint on X is inconsistent with V then
        delete V from D_X
      end if
    end for
  end for
end NC
```

Da notare che questo codice si presta molto facilmente ad una parallelizzazione, dato che non occorre scambiare alcuna informazione durante l'esecuzione tra le single variabili. Di conseguenza è possibile usare degli esecutori indipendenti per elaborare il dominio di ciascuna variabile. Nel caso poi di domini molto grandi, è anche possibile suddividere ciascun dominio tra più esecutori.

1.2.2 Arc consistency

L'algoritmo NC permette di eliminare alcuni valori dal dominio delle variabili del problema senza dover effettuare calcoli molto pesanti. Tuttavia la limitazione di questo algoritmo è che esso usa solo i vincoli unari delle variabili. Per poter sfruttare anche i vincoli binari si introduce il concetto di arco-consistenza.

Un arco (X_i, X_j) (con $i \neq j$) viene detto arco-consistente se e solo se X_i e X_j sono nodo-consistenti e per ogni valore a di X_i esiste un valore b di X_j tale che $X_i = a, X_j = b$ è permesso dai vincoli binari presenti tra X_i ed X_j. Una volta ottenuta la nodo-consistenza per tutti i vari nodi, ottenere la arco-consistenza è abbastanza semplice: per ogni coppia di variabili si verificano tutti i valori della prima variabile. Per ogni valore della prima variabile si provano tutti i valori della seconda variabile. Se nessun valore della seconda variabile permette di soddisfare tutti i vincoli tra le due variabili, il valore che è stato testato per la prima variabile può essere rimosso dal dominio.

```
procedure AC-1(G)
  Q ← {(V_i, V_j) ∈ arcs(G), i ≠ j}
  repeat
    CHANGED ← false
    for each arc (V_i,V_j) in Q do
      CHANGED ← REVISE(V_i, V_j) ∨ CHANGED
    end for
  until not(CHANGED)
end AC-1
```

```
procedure REVISE(V_i,V_j)
  DELETED ← false
  for each X in D_i do
    if there is no such Y in D_j such that (X,Y) is consistent,
    i.e. (X,Y) satisfies all the constraints on V_i, V_j then
      delete X from D_i
      DELETED ← true
    end if
  end for
  return DELETED
end REVISE
```

Questo algoritmo provvede a verificare ogni coppia di variabili del problema, possibilmente eliminando dei valori dal dominio della prima variabile di ogni arco se questi non permettono di soddisfare i vincoli binari tra le due variabili. Notare che ogni arco deve essere considerato almeno due volte invertendo l'ordine delle variabili. Questo perché i valori vengono rimossi solo dal dominio della prima variabile di ogni arco in ogni singolo controllo, quindi occorre considerare ogni arco due volte, invertendo le variabili, in modo da ridurre anche il dominio della seconda variabile.

Inoltre, se viene rimosso anche un solo valore dal dominio di una variabile, occorre ricontrollare tutti gli archi, dato che la rimozione di tale valore potrebbe permettere di rimuovere dei valori da altre variabili. L'algoritmo dunque termina solo quando non è più possibile rimuovere alcun valore dal dominio di ogni variabile.

È tuttavia evidente che la rimozione di un valore dal dominio di una variabile non può influenzare tutti gli archi del problema, ma solo alcuni di essi. Di conseguenza si è migliorato l'algoritmo precedente per tenere conto di tale ridotta influenza.

```
procedure AC-3(G)
  Q ← {(V_i, V_j) ∈ arcs(G), i ≠ j}
  while not empty Q do
    select and delete any arc (V_k,V_m) from Q
    if REVISE(V_k,V_m) then
      Q ← Q ∪ {(V_i, V_k)|(V_i, V_k) ∈ arcs(G) ∧ i ≠ k ∧ i ≠ m}
    end if
  end while
end AC-3
```

```
procedure REVISE(V_i,V_j)
  DELETED ← false
  for each X in D_i do
    if there is no such Y in D_j such that (X,Y) is consistent,
    i.e. (X,Y) satisfies all the constraints on V_i, V_j then
      delete X from D_i
      DELETED ← true
    end if
  end for
  return DELETED
end REVISE
```

Questo algoritmo ricontrolla solo gli archi che possono essere influenzati dalla riduzione del dominio di una variabile, permettendo un notevole miglioramento di prestazioni rispetto all'algoritmo AC-1. Sono noti altri algoritmi ancora più efficienti di AC-3, ma questi tendono ad essere poco utilizzati a causa della loro complessità di implementazione.

1.2.3 Livelli di consistenza superiori

Si può cercare di ottenere un livello di consistenza ancora superiore con degli algoritmi di consistenza di cammino, che considerano i nodi e gli archi che compongono un cammino nel grafo. È anche possibile ottenere livelli di consistenza ancora superiori alla consistenza di cammino, cosa che permette in teoria di arrivare ad una soluzione del CSP senza dover effettuare alcun tipo di ricerca. Tuttavia l'ottenimento di livelli di consistenza così elevati è molto costoso, al punto che viene raramente utilizzato in pratica.

1.3 Algoritmi ibridi

Nel "mondo reale" il massimo livello di consistenza che può essere raggiunto in un tempo ragionevole è l'arco-consistenza. Questo permette comunque di eliminare un certo numero di valori dai domini delle varie variabili, cosa che riduce lo spazio di ricerca in cui si può trovare la soluzione del problema. Inoltre questi algoritmi di consistenza possono rilevare in anticipo l'assenza di una soluzione per il problema: se dopo l'esecuzione di questi algoritmi di consistenza il dominio di una variabile diventa vuoto, il problema non ha soluzione.

Questo suggerisce che gli algoritmi di consistenza possono essere usati in congiunzione con degli algoritmi di ricerca per utilizzare le migliori caratteristiche di ciascuno. È dunque possibile utilizzare degli algoritmi per ottenere la nodo-consistenza e poi l'arco-consistenza, in modo da eliminare alcuni valori dal dominio delle variabili del problema. Quindi è possibile eseguire un algoritmo di ricerca sul problema modificato, in modo da trarre vantaggio dalla riduzione delle dimensioni dei domini delle variabili.

1.3.1 Forward checking

Tuttavia è possibile combinare in maniera ancora maggiore i due tipi di algoritmi. Una prima tecnica è quella del forward checking (FC). Questa prevede, dopo ogni assegnamento di valore ad una variabile, di propagare l'assegnamento di tale valore alle variabili non ancora istanziate, in modo da escludere i valori non possibili per tali variabili.

Questo permette di eliminare a priori una serie di valori per le variabili non assegnate che non possono portare ad una soluzione del problema. Notare che, quando si deve scegliere un valore per una variabile, non è necessario controllare se il valore scelto nel dominio della variabile è consistente con i valori delle altre variabili già assegnate. Questo

perché i valori non consistenti nel dominio di ogni variabile vengono scartati man mano che le variabili vengono assegnate. Quando giunge il momento di assegnare il valore ad una variabile, tutti i vincoli tra tale variabile e tutte le altre variabili già assegnate saranno stati controllati. Quindi, tutti i valori nel dominio della variabile da assegnare che non sono consistenti con il valore di una qualche altra variabile già assegnata sono stati già rimossi. Questo significa che, quando si deve assegnare una variabile, tutti i valori rimanenti nel relativo dominio soddisfano tutti i vincoli con le variabili già assegnate.

```
procedure AC-FC(cv)
  Q ← {(V_i, V_cv) ∈ arcs(G) ∧ i > cv}
  consistent ← true
  while not Q empty ∧ consistent do
    select and delete any arc (V_k,V_m) from Q
    if REVISE(V_k,V_m) then
      consistent ← not empty D_k
    end if
  end while
  return consistent
end AC-FC
```

```
procedure REVISE(V_i,V_j)
  DELETED ← false
  for each X in D_i do
    if there is no such Y in D_j such that (X,Y) is consistent,
    i.e. (X,Y) satisfies all the constraints on V_i, V_j then
      delete X from D_i
      DELETED ← true
    end if
  end for
  return DELETED
end REVISE
```

1.3.2 Maintaining arc consistency

È ovviamente possibile eseguire il controllo di consistenza completo su tutti gli archi. Questo algoritmo prende il nome di full look ahead oppure maintaining arc consistency (MAC). L'idea di questo approccio, infatti, è quella di sfruttare gli assegnamenti effettuati da un algoritmo di backtracking durante la sua esecuzione per eseguire ulteriori controlli di consistenza ed eliminare un numero maggiore di valori non consistenti dal dominio delle variabili del problema.

```
procedure AC3-LA(cv)
  Q ← {(V_i, V_cv) ∈ arcs(G) ∧ i > cv}
  consistent ← true
  while not Q empty ∧ consistent do
    select and delete any arc (V_k,V_m) from Q
    if REVISE(V_k,V_m) then
      Q ← Q ∪ {(V_i, V_k)|(V_i, V_k) ∈ arcs(G) ∧ i ≠ k ∧ i ≠ m ∧ i > cv}
      consistent ← not empty D_k
    end if
  end while
  return consistent
end AC3-LA
```

```
procedure REVISE(V_i,V_j)
  DELETED ← false
  for each X in D_i do
    if there is no such Y in D_j such that (X,Y) is consistent,
    i.e. (X,Y) satisfies all the constraints on V_i, V_j then
      delete X from D_i
      DELETED ← true
    end if
  end for
  return DELETED
end REVISE
```

In pratica i valori scelti per ogni variabile dall'algoritmo di backtracking vengono considerati come vincoli aggiuntivi del problema, e ad ogni passo viene eseguito un algoritmo di arco-consistenza che usa tali vincoli aggiuntivi per ridurre il dominio dei valori delle variabili del problema.

1.4 Migliorare le prestazioni

Tutti questi algoritmi si basano sull'idea di ridurre le dimensioni dell'albero di ricerca che deve essere esplorato dall'algoritmo di backtracking. Per fare questo, essi introducono delle operazioni aggiuntive che, ogni volta che si assegna un valore ad una variabile, controllano i domini delle altre variabili ed eliminano da questi alcuni valori inconsistenti. Anche se queste operazioni riescono effettivamente a ridurre l'ampiezza dell'albero di ricerca, il costo aggiuntivo di dover effettuare tali operazioni ad ogni assegnamento porta l'algoritmo ad essere più lento di un altro algoritmo che, pur operando su un albero di ricerca più grande, deve eseguire meno operazioni ad ogni passo. Si aggiunga a questo la maggiore complessità (sia di comprensione, sia di implementazione) degli ultimi algoritmi presentati, e diventa facile comprendere perché gli algoritmi più semplici tra quelli presentati (backtracking e forward checking) restano molto utilizzati.

Un aspetto che può influenzare notevolmente l'efficienza di tutti gli algoritmi presentati finora è l'ordinamento delle variabili e l'ordinamento dei valori da provare per ogni variabile. In certi casi, infatti, è possibile trovare un ordine delle variabili che, se usato con un algoritmo di backtracking, permette di trovare, per ogni variabile, un valore che è consistente con tutti gli assegnamenti delle variabili già istanziate. Di conseguenza l'algoritmo non effettua mai un'operazione di backtracking, impiegando quindi un numero di passi proporzionale alla somma delle dimensioni dei domini delle variabili per trovare una soluzione al problema.

Tale comportamento permette ovviamente di risolvere un CSP molto velocemente, ma anche la scelta di un ordine appropriato per i valori di ogni singola variabile può portare molto velocemente a trovare una soluzione al problema. Basti pensare a cosa succede con un algoritmo di backtracking se i valori di ogni variabile sono ordinati in maniera tale che l'ennupla dei primi valori provati per ogni variabile costituisce una soluzione per il problema. In questo caso diventa possibile trovare la soluzione in un tempo proporzionale al numero di variabili del problema.

Con entrambi questi metodi il problema è riuscire a trovare tale ordinamento. Per quanto riguarda l'ordinamento delle variabili, in certi casi particolari (quando il grafo che rappresenta il problema è aciclico) è possibile trovare un ordinamento che, nel caso il problema sia arco-consistente, permette di risolverlo senza eseguire backtracking [Bar05]. Tuttavia, in tutti gli altri casi occorre usare altri metodi. Non si conoscono metodi esatti per determinare l'ordine migliore per le variabili o i valori, quindi, si cerca di creare delle euristiche che sembrano funzionare bene nella maggior parte dei casi, ma che presentano comunque la possibilità di scegliere un ordinamento non ottimale.

Una possibilità interessante è quella di cercare di eliminare eventuali cicli nel grafo dei vincoli. Se infatti il grafo dei vincoli è aciclico, diventa possibile trovare un ordinamento delle variabili che permette di effettuare una ricerca delle soluzioni senza backtracking. Il metodo per trasformare un grafo ciclico in uno aciclico è quello di trovare un insieme di variabili che, se rimosse, eliminano tutti i cicli del grafo. Quindi si scelgono dei valori per tutte queste variabili e si propagano tali valori attraverso i vincoli del problema; si rimuovono quindi le variabili istanziate e quello che rimane è un grafo aciclico. Per questo è possibile trovare un ordinamento delle variabili che permette di trovare una soluzione (se esiste) in maniera molto veloce.

Questo metodo presenta alcuni svantaggi: richiede comunque di effettuare alcuni passi di backtracking, dato che l'assegnamento alle variabili iniziali potrebbe non portare ad alcuna soluzione. Inoltre non si conoscono algoritmi in tempo polinomiale che riescano a trovare l'insieme minimo di nodi che occorre rimuovere per rendere aciclico un grafo ciclico. Di conseguenza occorre usare una procedura che fornisca un risultato non minimo, mentre un insieme minimo di nodi da rimuovere sarebbe preferibile, in quanto ridurrebbe lo spazio dei possibili assegnamenti da effettuare prima di poter verificare la soluzione.

Capitolo 2

CSP Distribuiti e Decentralizzati

2.1 Asynchronous backtracking search

Gli algoritmi per la risoluzione di CSP descritti nel capitolo precedente hanno un difetto evidente: essi richiedono che tutti i dati del problema siano disponibili ad un singolo esecutore. Se questo fatto rende semplice la realizzazione di algoritmi per la risoluzione di CSP, esso pone anche un limite drastico alla scalabilità dell'intero sistema di risoluzione. Questo perché la velocità di risoluzione del CSP con un certo algoritmo può essere migliorata solo utilizzando un esecutore più veloce, cosa che non sempre potrebbe essere possibile.

L'alternativa a questo approccio è quella di suddividere il problema tra più esecutori, in modo che questi possano combinare i propri sforzi per velocizzare la risoluzione del CSP. Tuttavia, per fare questo, occorre utilizzare un algoritmo di risoluzione che permetta la suddivisione del problema.

2.1.1 Assunzioni iniziali

Il primo algoritmo che rispetta questo requisito è una variante dell'algoritmo di backtracking presentato nel capitolo precedente. Tale algoritmo, definito Asynchronous Backtracking Search, prevede innanzitutto di suddividere il problema tra un certo numero di esecutori, detti agenti. Questa suddivisione avviene assegnando una variabile del problema ad ogni agente e stabilendo che il valore di una variabile può essere modificato solo dall'agente a cui è stata assegnata. Questo significa che riferirsi ad un agente od alla variabile gestita da tale agente è indifferente, dato che la relazione tra questi elementi è uno a uno. Inoltre, nel resto di questa tesi, si chiamerà "variabile locale" di un agente la variabile gestita da quell'agente.

In seguito occorre fornire ai singoli agenti un qualche tipo di informazione in modo che essi possano decidere autonomamente quando e come cambiare il valore della propria variabile. A tal proposito occorre fare alcune ipotesi sul modello di comunicazione utilizzato:

- Gli agenti comunicano tra loro usando dei messaggi. Si assume che tutti i tipi di messaggi permettano al destinatario di sapere chi è il mittente;
- I messaggi inviati da un agente A ad un agente B vengono ricevuti da B nello stesso ordine in cui sono stati inviati da A;
- Il tempo tra l'invio di un messaggio e la sua ricezione è casuale ma finito. Questo, tra l'altro, significa che la perdita di messaggi non è accettabile.

2.1.2 Agent view

Avendo presenti questi punti, occorre fare in modo che ogni agente abbia una conoscenza (parziale) dello stato del sistema, altrimenti un agente non potrebbe sapere che valore conviene assegnare alla propria variabile. A questo scopo si usa una struttura dati nota come agent view. Questa è sostanzialmente un insieme di associazioni chiave - valore, in cui le chiavi sono gli identificatori delle variabili e i valori sono i valori assegnati a ciascuna variabile. Formalmente:

$$AgentView : \mathbb{Z} \rightarrow \mathbb{Z} \cup \{\varepsilon\}$$

Quindi la agent view può essere vista come una funzione che associa ad ogni identificatore (intero) il valore della variabile corrispondente (intero), oppure risponde che il valore della variabile non è noto (ε).

Se si vuole creare un sistema totalmente distribuito e decentralizzato, tuttavia, non è possibile avere una struttura dati centralizzata di questo tipo a cui tutti gli agenti facciano riferimento. Per fare in modo che il sistema sia veramente distribuito e decentralizzato, occorre fare in modo che gli agenti non debbano fare riferimento ad alcun archivio di informazioni condiviso. Quindi ogni agente provvede a creare la propria agent view e a mantenerla aggiornata.

Questo crea dei problemi: come fa un agente a sapere il valore di una variabile che è gestita da un altro agente? Come fa a mantenerne aggiornato il valore nella agent view?

Inoltre, potrebbe non essere necessario avere nella agent view il valore di tutte le altre variabili, ma potrebbero servire solo i valori di alcune di esse.

Per risolvere questi problemi, per prima cosa si stabilisce che i valori contenuti nella agent view possano essere non aggiornati. Questo implica che l'algoritmo deve permettere il fatto che un agente effettui scelte sbagliate a causa di informazioni incomplete od obsolete. Poi, per permettere la costruzione e l'aggiornamento della agent view, si introduce un tipo di messaggio scambiabile tra gli agenti: il messaggio Ok. Questo messaggio contiene come dati l'identificatore di una variabile ed un valore per tale variabile. L'idea è che quando il valore di una variabile cambia, l'agente che ha assegnato il nuovo valore a tale variabile invia dei messaggi Ok agli altri agenti in modo da informali del cambiamento. Quando un agente riceve un messaggio Ok, esso aggiorna la propria agent view aggiungendo o modificando il valore della variabile indicata nel messaggio. Già da questa spiegazione risulta evidente come una agent view possa non essere aggiornata: nel tempo tra la modifica del valore di una variabile e la ricezione del messaggio Ok riguardante tale modifica, la agent view del destinatario conterrà ancora il valore vecchio per la variabile.

Infine occorre limitare le dimensioni della agent view, se possibile. A tale scopo si introduce il concetto di agente vicino: inizialmente gli agenti vicini ad un determinato agente A sono quegli agenti che compaiono in un vincolo binario insieme con la variabile locale di A. Si stabilisce poi che i messaggi Ok vengano inviati solo ai vicini, questo perché se un agente B non ha un vincolo che coinvolge anche l'agente A, a B non serve sapere il valore della variabile gestita da A, quindi A può evitare di inviare a B dei messaggi Ok.

2.1.3 Scelta dei valori per le variabili

A questo punto ogni agente ha a disposizione una agent view costruita ed aggiornata da lui stesso che contiene tutti i valori noti delle variabili dei vicini. Quello che ancora manca è un sistema per scegliere un valore per la variabile gestita da ogni agente in funzione della agent view.

Ora ci si ricollega all'algoritmo di backtracking: in tale algoritmo si sceglie un valore per una variabile, se è corretto (cioè se rispetta tutti i vincoli che possono essere verificati) si passa alla variabile successiva, mentre se non lo è si prova un nuovo valore. Se non sono disponibili altri valori per questa variabile che non siano già stati provati, si torna alla variabile precedente e si cerca un nuovo valore per quest'altra.

La prima cosa da fare per poter applicare tale schema ad un sistema ad agenti è

definire quale variabile / agente viene prima o dopo a quale altra. Occorre quindi stabilire un ordinamento tra le variabili / agenti. Tale ordinamento viene creato semplicemente usando gli identificatori delle variabili: si stabilisce che variabili con identificatore più basso abbiano priorità maggiore e viceversa. A questo punto, tornando alla descrizione dell'algoritmo di backtracking, si deduce che ogni agente assegna un valore alla propria variabile in base al dominio di tale variabile e ai vincoli che può verificare. Questi vincoli sono tutti i vincoli unari e i vincoli binari che coinvolgono, oltre alla variabile locale, una variabile di priorità superiore. Questo significa che un agente non ha bisogno di sapere il valore delle variabili di priorità inferiore. Di conseguenza è possibile modificare la regola di invio dei messaggi Ok descritta prima in modo che essi vengano inviati solo ai vicini con priorità inferiore a quella del mittente, invece che a tutti i vicini. In questo modo si può ridurre ulteriormente il numero di messaggi da scambiare tra gli agenti.

A questo punto basta stabilire che un agente, ogni volta che riceve un messaggio Ok, prova a vedere se un valore del dominio della sua variabile rispetta tutti i vincoli verificabili. Se il valore precedente va ancora bene, l'agente non fa nulla, altrimenti esso comunica ai propri vicini con priorità inferiore il nuovo valore assegnato alla propria variabile. Se un vincolo binario fa riferimento, oltre che alla variabile locale, ad una variabile non presente nella agent view, tale vincolo viene considerato rispettato.

2.1.4 Backtracking tramite messaggi Nogood

Rimane da descrivere cosa succede nel caso in cui un agente non riesca a trovare alcun valore per la propria variabile che soddisfi tutti i vincoli verificabili. In questo caso nell'algoritmo di backtracking centralizzato si torna alla variabile precedente e se ne cambia il valore. Tuttavia, nel caso distribuito, per eseguire questa operazione occorre trovare il modo di comunicare ad un agente con priorità superiore che è necessario che esso cambi il valore della propria variabile. Inoltre non bisogna dimenticare che tutte le operazioni di ogni agente si basano sui dati contenuti nella agent view di tale agente, che potrebbe non essere aggiornata. Questo significa che non necessariamente un agente di priorità superiore deve cambiare il valore della propria variabile quando un agente con priorità inferiore non riesce a trovare un valore per la propria variabile: potrebbe essere sufficiente aspettare per fare in modo che la agent view dell'agente con priorità inferiore venga aggiornata per trovare un valore alla sua variabile.

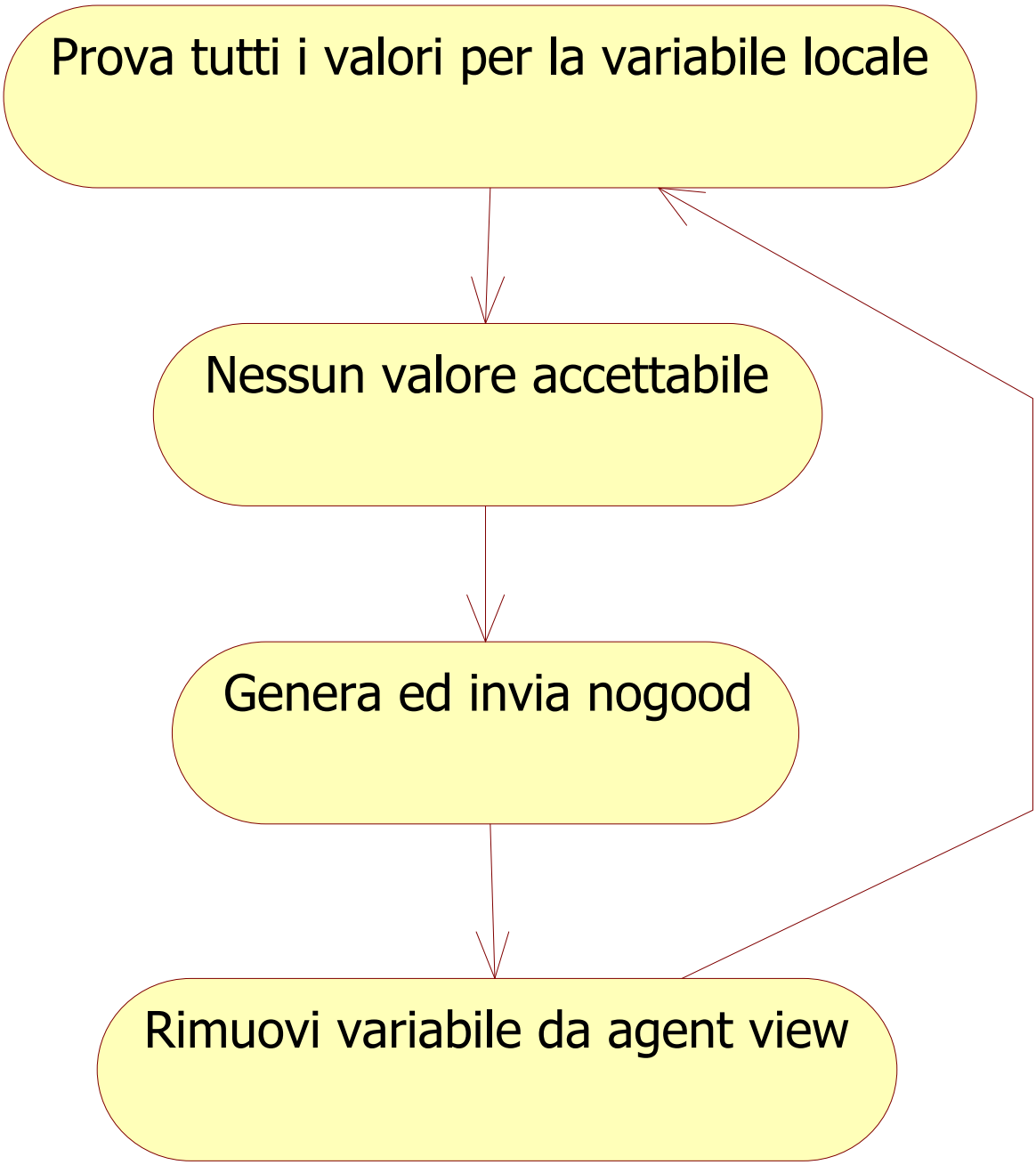
In ogni caso, se un agente non riesce a trovare un valore per la propria variabile,

significa che l'insieme dei valori presente nella sua agent view in quel momento non permette di trovare alcun valore per la variabile locale di tale agente. Occorre quindi comunicare agli altri agenti che tale combinazione di valori non permette di trovare una soluzione, indipendentemente dal fatto che i valori delle variabili degli altri agenti siano effettivamente quelli contenuti nella agent view.

A questo scopo si introduce un nuovo tipo di messaggio: il messaggio Nogood. Esso contiene un insieme di coppie identificatore – valore che rappresenta la combinazione di assegnamenti che non permette di trovare una soluzione. Esso è un sottoinsieme della agent view dell'agente che lo crea. Di fatto un Nogood è un nuovo vincolo che viene creato dinamicamente e che deve essere rispettato da tutte le variabili che coinvolge. Ogni volta che un agente riceve un Nogood, questo viene salvato e, ogni volta che si cerca un valore per la variabile locale di tale agente, occorre verificare che il valore scelto rispetti non solo i vincoli ordinari, ma anche tutti i Nogood ricevuti. Come nel caso dei vincoli, anche per i Nogood si stabilisce che se una variabile non è presente nella agent view, tutti i Nogood in cui compare tale variabile sono considerati non violati.

Quando un agente non trova alcun valore da assegnare alla propria variabile, esso genera dunque un nuovo Nogood. Questo Nogood deve essere inviato a qualcuno. Occorre inviarlo ad un agente che possa, modificando il valore della propria variabile, evitare che tutte le variabili presenti nel Nogood abbiano il valore indicato. Questo significa che il Nogood deve essere inviato ad un agente che gestisce una variabile tra quelle presenti nel Nogood stesso. Inoltre l'agente che riceve il Nogood deve poter verificare nella propria agent view il valore di tutte le altre variabili menzionate nel Nogood. Quindi, per i discorsi fatti in precedenza sulla costruzione della agent view e sulla priorità degli agenti, occorre inviare il Nogood all'agente con priorità più bassa tra quelli che gestiscono le variabili presenti nel Nogood stesso.

Una volta inviato il Nogood, bisogna ancora trovare un valore per la variabile locale. Dato che l'agente a cui si invia il Nogood probabilmente cambierà il valore della propria variabile, si decide di rimuovere dalla agent view il valore della variabile gestita da tale agente e di riprovare a trovare un valore per la variabile. Dato che un agente non controlla eventuali altri messaggi in arrivo prima di aver trovato un valore per la propria variabile, si ha che potenzialmente un agente, ripetendo più volte il ciclo in figura, arrivi a rimuovere vari elementi dalla propria agent view senza che nel frattempo ne venga aggiunto alcuno.


Prova tutti i valori per la variabile locale
Nessun valore accettabile
Genera ed invia nogood
Rimuovi variabile da agent view

Nel caso estremo, se anche con una agent view vuota non è possibile trovare nessun valore per la variabile, si ha che il problema non ha soluzione e quindi l'algoritmo può terminare. Più in generale, non è necessario che la agent view diventi vuota: è sufficiente che venga generato un Nogood vuoto per stabilire che il problema non ha soluzione. Un Nogood vuoto, infatti, significa che il problema non ha soluzione indipendentemente dal valore di tutte le variabili. Questo perché un Nogood rappresenta un sottoinsieme dei valori assegnati alle variabili che, se presente nell'assegnamento corrente, non permette di trovare una soluzione al problema. Dato che un insieme vuoto è sottoinsieme di ogni altro insieme, si ha che un Nogood vuoto implica che non è possibile trovare una soluzione al problema indipendentemente dal valore di tutte le variabili.

Se un agente riceve un Nogood, invece, deve salvarlo e verificare se tutte le condizioni del Nogood sono verificate. Per fare questo, l'agente che ha ricevuto il Nogood controlla nella propria agent view il valore di tutte le variabili indicate nel Nogood più, ovviamente, il valore della propria variabile, e verifica se tutti i vari valori corrispondono a quelli indicati nel Nogood. Se tutti i valori corrispondono, l'agente cerca di trovare un nuovo valore per la propria variabile nella maniera solita, inviando anche i messaggi Ok a tutti i vicini con priorità inferiore, altrimenti si limita a ribadire il valore della propria variabile all'agente che ha inviato il Nogood con un messaggio Ok. Ovviamente, se serve trovare un nuovo valore e nessun valore del dominio rispetta tutti i vincoli e i Nogood verificabili, verrà generato un nuovo Nogood e si ripeterà la stessa procedura eseguita dall'agente che ha inviato il primo Nogood.

2.1.5 Aggiungere dinamicamente nuovi vicini

Un problema di questo metodo è che se un agente riceve un Nogood, questo Nogood potrebbe contenere una o più variabili non presenti nella agent view del destinatario e che non sono nemmeno vicine dell'agente destinatario. In questo caso si avrebbe che il Nogood non potrebbe mai essere controllato, dato che una o più variabili del Nogood non sarebbero mai presenti nella agent view.

Per risolvere il problema, basta aggiungere una fase di controllo in cui si verifica se tutte le variabili del Nogood appena ricevuto sono dei vicini della variabile locale dell'agente che ha ricevuto il Nogood. Se una o più variabili non sono dei vicini del destinatario, questo invia agli agenti che gestiscono tali variabili un messaggio che richiede

di creare una nuova relazione di vicinanza. Inoltre l'agente che riceve il Nogood aggiunge tali nuovi agenti all'elenco dei suoi vicini.

Quando un agente riceve una richiesta di creazione di una nuova relazione di vicinanza, esso aggiunge il mittente della richiesta all'elenco dei suoi vicini, quindi risponde al mittente con un messaggio Ok per informarlo del valore corrente della propria variabile.

Tornando all'agente che riceve un messaggio Nogood, si ha dunque che se nel Nogood compaiono una o più variabili che non sono vicine dell'agente, esso invia delle richieste di creazione di nuove relazioni di vicinanza agli agenti che gestiscono tali variabili (oltre ad aggiungere tali agenti all'elenco dei propri vicini). Quindi passa a controllare se il Nogood è violato: se erano state trovate delle variabili che non erano vicine del destinatario, il Nogood non potrà essere controllato subito e quindi il destinatario del Nogood risponderà al mittente comunicandogli il valore corrente della propria variabile. Se il mittente del Nogood genera di nuovo lo stesso Nogood in un momento successivo, tutte le variabili presenti nel Nogood saranno già dei vicini del destinatario del Nogood e prima o poi il destinatario del Nogood riceverà tutti i messaggi Ok dagli agenti a cui ha chiesto di creare una nuova relazione di vicinanza. A quel punto il destinatario del Nogood potrà finalmente controllare se il Nogood è violato e, in tal caso, potrà cercare un nuovo valore per la propria variabile.

2.1.6 Descrizione complessiva

Ricapitolando, ogni agente si crea una propria agent view usando i messaggi Ok ricevuti. Ogni volta che arriva un messaggio Ok si aggiorna la agent view e si verifica se è possibile trovare un valore per la variabile che non violi alcun vincolo e Nogood verificabile. Se il valore assegnato in precedenza va ancora bene, non si fa nulla; se si trova un nuovo valore, si inviano messaggi Ok a tutti i vicini con priorità inferiore. Se non è possibile trovare alcun valore, si genera un Nogood, lo si invia all'agente che gestisce la variabile con priorità più bassa tra quelle presenti nel Nogood generato e si rimuove dalla agent view la variabile a cui è stato mandato il Nogood. Quindi si riprova a trovare un valore per la variabile locale ripetendo lo stesso procedimento. Se viene generato un Nogood vuoto, il problema non ha soluzione.

Se invece arriva un Nogood, lo si aggiunge all'elenco dei Nogood ricevuti e si verifica se qualche variabile contenuta nel Nogood non è un vicino dell'agente che ha ricevuto il Nogood. In questo caso si invia una richiesta di creazione di relazione di vicinanza tra

tale variabile e l'agente che ha ricevuto il Nogood e si aggiunge la variabile all'elenco dei vicini. Quindi si verifica se il valore assegnato alla variabile locale rispetta tutti i vincoli e i Nogood ricevuti verificabili (incluso quello appena ricevuto). Se il valore assegnato in precedenza è ancora accettabile, si invia un messaggio di Ok con tale valore al mittente del Nogood; se si trova un nuovo valore, si inviano messaggi Ok a tutti i vicini con priorità inferiore. Se non è possibile trovare alcun valore, si genera un Nogood come descritto sopra nel caso del messaggio Ok e si procede alla stessa maniera.

Infine, se si riceve una richiesta di creazione di relazione di vicinato, si aggiunge il mittente di tale richiesta all'elenco dei vicini e si invia al mittente della richiesta un messaggio Ok contenente il valore della variabile gestita dall'agente. L'invio del messaggio Ok di risposta, pur non essendo descritto nel lavoro di Yokoo [YH00], si dimostra necessario per fare in modo che l'agente che invia la richiesta di relazione di vicinato possa inserire nella sua agent view il valore della variabile gestita dall'agente a cui ha inviato la richiesta entro un tempo finito. Se, infatti, non si inviasse il messaggio Ok di riposta e l'agente che riceve la richiesta di relazione di vicinato non cambiasse più il valore della propria variabile, si avrebbe che al mittente della richiesta non arriverebbe mai un messaggio Ok con il valore della variabile, e dunque non potrebbe mai verificare il Nogood che ha portato all'invio della richiesta di vicinato.

2.1.7 Ridurre i Nogood

Rimane da descrivere un aspetto dell'algoritmo: la generazione dei Nogood. Si è detto che un Nogood è un sottoinsieme della agent view dell'agente che crea il Nogood, ma non si è detto come si può trovare questo sottoinsieme.

Il caso più semplice prevede di usare l'intera agent view come Nogood. Questa scelta permette di evitare ogni tipo di elaborazione per la generazione dei Nogood, ma porta alla generazione di Nogood molto grandi. Questo significa che chi riceve questi Nogood dovrà conoscere il valore di molte variabili per poterli verificare. Dato che per conoscere il valore di una variabile che non è sua vicina un agente deve creare una nuova relazione di vicinato, si ha che usare come Nogood l'intera agent view di ogni agente porta al rapido incremento del numero di vicini di ogni agente, al punto che, come verificato anche da alcuni test di applicazione dell'algoritmo ABT alla risoluzione del problema del Sudoku (usando come Nogood l'intera agent view di ogni agente), si arriva rapidamente alla situazione in cui ogni agente ha come vicini tutti gli altri. Questo aumenta la dimensione

della agent view di ogni agente, aumentando la possibilità che questa non sia aggiornata. Inoltre, il maggior numero di vicini per ogni agente necessita dell'invio di una maggiore quantità di messaggi tra agenti e, dato che la dimensione delle agent view aumenta, anche la dimensione dei messaggi Nogood aumenta di conseguenza. Quindi si viene a creare un ulteriore problema di invio di una maggiore quantità di dati tra gli agenti, che peggiora ulteriormente le prestazioni.

A tutto questo occorre aggiungere un'ulteriore considerazione. Un Nogood rappresenta un insieme di valori per alcune variabili che non permette di trovare una soluzione al problema. Se anche una sola delle variabili del Nogood ha un valore diverso da quello indicato, il Nogood non ci dice più nulla. Quindi Nogood più piccoli (cioè contenenti un minor numero di coppie variabile - valore) sono più generali e permettono di eliminare in una volta sola un maggior numero di assegnamenti errati.

Da tutte queste considerazioni si capisce come sia improponibile usare l'intera agent view come Nogood, specialmente quando la dimensione del problema da risolvere inizia a diventare importante. Si rende dunque necessario trovare un metodo per ridurre la dimensione dei Nogood generati, intendendo come dimensione di un Nogood il numero di coppie variabile - valore che esso contiene. Quello che si vuole fare è procedere alla generazione di un Nogood di dimensioni minori del Nogood contenente l'intera agent view e tale che, usando tale Nogood come agent view dell'agente che genera il Nogood, non sia possibile trovare alcun valore consistente per la variabile locale.

In teoria si potrebbe procedere ad una minimizzazione del Nogood generato (cioè procedere alla creazione del più piccolo Nogood che rispetti quanto detto sopra), tuttavia, anche se l'utilizzo del Nogood così prodotto dovrebbe portare dei benefici, tale operazione di minimizzazione rischia di essere lenta al punto da vanificare ogni vantaggio derivante dall'utilizzo di Nogood minimi. Per questo motivo si è scelto di seguire un approccio intermedio per ridurre le dimensioni dei Nogood generati senza procedere ad una minimizzazione completa. Tale approccio prevede semplicemente di partire dall'intera agent view e di provare a togliere da questa una variabile per volta. Se, togliendo una variabile, diventa possibile trovare un assegnamento per la variabile locale, tale variabile viene lasciata nel Nogood, altrimenti la variabile viene rimossa.

Questo procedimento non produce Nogood minimi, ma riduce comunque le dimensioni dei Nogood generati un maniera sensibile, oltre a non essere particolarmente dispendioso in termini di potenza di calcolo richiesta per applicarlo. In effetti l'uso di Nogood di

dimensioni ridotte favorisce il mantenimento di dimensioni contenute delle agent view. Dato che, come detto, i Nogood vengono generati a partire dalla agent view, se questa ha dimensioni ridotte, sarà necessario eseguire una minore quantità di elaborazioni per creare un Nogood a partire da questa.

Un altro aspetto su cui si può intervenire nel procedimento sopra descritto è l'ordine in cui le variabili della agent view vengono analizzate ed eventualmente rimosse per creare un Nogood. Per scegliere tale ordine si è considerato che un Nogood viene inviato all'agente che gestisce la variabile con priorità più bassa che è presente nel Nogood stesso. Pertanto risulta conveniente cercare di inviare un Nogood ad un agente con una priorità più alta possibile, in modo da cercare di velocizzare la revisione di una eventuale scelta sbagliata da parte di un agente con priorità elevata. Per cercare di ottenere questo risultato, si è deciso di eseguire l'eliminazione delle variabili partendo da quelle con priorità più bassa presenti nella agent view. In questo modo si cerca di rimuovere le variabili con priorità più bassa presenti nella agent view, in modo da ottenere un Nogood che massimizzi la priorità minima delle variabili presenti nel Nogood stesso.

when received (**Ok?**, (x_j, d_j)) **do**
  update agent view
  execute check_agent_view
end do

when received (**Nogood**, x_j, *nogood*) **do**
  record *nogood* as a new constraint
  if *nogood* contains an agent x_k that is not its neighbor **then**
    send (**add_neighbor**, x_i) to x_k
    add x_k to the neighbors
  end if
  $oldValue \leftarrow currentValue$
  execute check_agent_view
  if $oldValue = currentValue$ **then**
    send (**Ok?**, (x_i, $currentValue$)) to x_j
  end if
end do

```
when received (add_neighbor, x_j) do
  add x_j to the neighbors
  send (Ok?, (x_i, currentValue)) to x_j
end do
```

```
procedure backtrack
  create a copy V of the agent view
  sort the variables in V in ascending priority order
  for each variable x_k in V do
    remove x_k from V
    if a value in D_i is consistent with V then
      reinsert x_k in V
    end if
  end for
  if V is an empty Nogood then
    broadcast to other agents that there is no solution
    terminate this algorithm
  end if
  select (x_j, d_j) where x_j has the lowest priority in the Nogood
  send (Nogood, x_i, V) to x_j
  remove (x_j, d_j) from agent view
  execute check_agent_view
end backtrack
```

```
procedure check_agent_view
  if agent view and current value are not consistent then
    if no value in D_i is consistent with agent view then
      execute backtrack
    else
      select d ∈ D_i where agent view and d are consistent
      currentValue ← d
      send (Ok?, (x_i, d)) to neighbors
    end if
  end if
end check_agent_view
```

Da quanto detto, si nota come i messaggi Ok vengano inviati solo da agenti con priorità superiore ad agenti con priorità inferiore. Da ciò si ha che ogni agente riceverà messaggi Ok provenienti solo da agenti con priorità maggiore. Questo fatto, dato che questi messaggi vengono usati per creare e aggiornare le agent view di ogni agente, implica che le agent view di ogni agente contengono solo variabili con priorità superiore. Dato che ogni Nogood è un sottoinsieme della agent view dell'agente che lo crea, e dato che un Nogood viene inviato all'agente che gestisce una variabile tra quelle presenti nel Nogood stesso, si ha che i Nogood vengono sempre inviati da agenti con priorità inferiore ad agenti con priorità superiore.

Si ha dunque questa netta distinzione dei tipi di messaggi, con i messaggi Ok che "scendono" verso priorità più basse e i Nogood che "salgono" verso priorità più alte. Inoltre si ha che i Nogood ricevuti da ogni agente fanno riferimento, oltre che alla variabile gestita da tale agente, a variabili con priorità superiori.

2.1.8 Completezza dell'algoritmo

Si può ora procedere a dimostrare che l'algoritmo è completo, cioè che se il problema ha soluzione, dopo un tempo finito gli agenti si trovano tutti in uno stato di attesa di nuovi messaggi e tutti i valori assegnati alle variabili rispettano tutti i vincoli del problema. Se invece il problema non ha soluzione viene generato un Nogood vuoto. Questa dimostrazione è tratta da [YI98].

Proposizione 2.1. *Non è possibile che gli agenti che eseguono l'algoritmo ABT cadano in un ciclo di elaborazione infinito.*

Dimostrazione. Procediamo per induzione partendo dall'agente con priorità massima. Supponiamo per assurdo che l'agente con priorità massima si trovi in un ciclo di elaborazione infinito. Per quanto detto sopra, tale agente riceve solo messaggi Nogood. Ogni volta che tale agente sceglie un valore per la propria variabile può accadere una di queste cose: o riceve un Nogood riferito al valore che ha scelto, o non riceve nulla (in questo secondo caso ricade anche il caso in cui venga ricevuto un messaggio Nogood che fa riferimento ad un valore precedente della variabile, dato che se l'agente ha cambiato il valore della propria variabile, significa che un Nogood per tale valore precedente era già stato ricevuto). Se non riceve nulla, l'agente non cambierà il valore che ha scelto. Se invece riceve un Nogood riferito al valore attuale della variabile, esso salverà il Nogood e sceglierà un valore diverso, ripetendo la stessa situazione. Dato che i valori del dominio di ogni variabile sono finiti, si ha che l'agente può o non ricevere nulla per un qualche assegnamento di valore alla variabile (situazione descritta sopra), oppure ricevere un Nogood per tutti i possibili valori della propria variabile. In questo caso il successivo tentativo di trovare un valore per la variabile provocherebbe la generazione di un Nogood vuoto, e l'algoritmo terminerebbe. In tutti i casi, l'agente con priorità massima non può essere in un ciclo di elaborazione infinito, cosa che contraddice l'assunzione che l'agente sia in un ciclo di elaborazione infinito.

Supponiamo ora che tutti gli agenti x_0, x_1, ..., x_{k-1} siano in uno stato stabile e che l'agente x_k sia in un ciclo di elaborazione infinito. In questa situazione l'agente x_k riceve solo dei messaggi Nogood dagli agenti con priorità inferiore. Tali messaggi Nogood contengono solo variabili con identificatore k o inferiore, e tutte conterranno la variabile con identificatore k. Dato che gli agenti x_0, x_1, ..., x_{k-1} sono in uno stato stabile, tutti i Nogood ricevuti saranno compatibili con la agent view dell'agente k, e dunque obbligheranno l'agente k a cambiare il valore della propria variabile (anche qui è possibile che vengano ricevuti Nogood che fanno riferimento a valori precedenti della variabile dell'agente x_k, ma, come sopra, tali Nogood possono essere ignorati dato che sono sicuramente dei duplicati di Nogood che erano già stati ricevuti in precedenza da x_k). Dato che il dominio della variabile x_k è finito, si può verificare o che venga scelto un valore che non causa la successiva ricezione di un Nogood (cosa che contraddice l'assunzione del fatto che l'agente k sia in un ciclo di elaborazione infinito), oppure che per ogni valore scelto per la variabile x_k venga ricevuto un Nogood. In questo caso l'agente k genererebbe un Nogood. Un Nogood vuoto causerebbe la terminazione dell'algoritmo, contraddicendo l'assunzione che l'agente k sia in un ciclo di elaborazione infinito, mentre un Nogood non vuoto dovrebbe essere inviato ad un agente che era stato supposto essere in uno stato stabile, cosa che contraddirebbe tale ipotesi. Di conseguenza x_k non può entrare in un ciclo di elaborazione infinito.

Questo significa che l'intero gruppo degli agenti non può entrare in un ciclo di elaborazione infinito. □

Questa dimostrazione implica o la terminazione dell'algoritmo se si genera un Nogood vuoto, oppure la stabilizzazione degli agenti in uno stato di attesa di nuovi messaggi. Come detto in precedenza, la generazione di un Nogood vuoto implica che il problema

non ha soluzione, mentre il raggiungimento di uno stato stabile significa che i valori delle variabili di tutti gli agenti sono consistenti tra loro, e quindi è stata trovata una soluzione per il problema.

2.2 Asynchronous weak-commitment search

2.2.1 Descrizione

Questo algoritmo è derivato dall'asynchronous backtracking, ma presenta delle modifiche. La più importante di queste è la possibilità dell'algoritmo di riordinare dinamicamente le variabili durante l'esecuzione. Questo gli permette di cambiare rapidamente delle scelte sbagliate effettuate da variabili con priorità iniziale più elevata, senza essere costretto prima ad eseguire una ricerca completa di tutti i valori delle variabili con priorità inferiore.

L'organizzazione di base dell'algoritmo è quasi uguale a quella dell'asynchronous backtracking: ogni agente gestisce una variabile ed ha a disposizione una agent view che esso stesso aggiorna quando riceve dei messaggi Ok da parte di altri agenti. Anche qui sono presenti dei vicini, che inizialmente sono gli stessi dell'algoritmo asynchronous backtracking, cioè i vicini di un certo agente A sono tutti quegli agenti le cui variabili compaiono in un vincolo binario insieme alla variabile dell'agente A.

Dato però che questo algoritmo, come detto, può cambiare dinamicamente l'ordinamento delle variabili, si rende necessario comunicare e memorizzare le priorità di ogni agente. Quindi ogni agente tiene traccia della propria priorità (rappresentata da un valore intero che all'inizio dell'esecuzione vale 0 per tutti gli agenti) e i messaggi Ok contengono anche la priorità oltre che l'identificatore e il valore della variabile. Si stabilisce inoltre che se due agenti hanno lo stesso valore di priorità, quello con identificatore più basso viene considerato avere una priorità maggiore dell'altro. In questo modo viene garantito che due agenti non possano mai avere una priorità uguale.

L'esecuzione dell'algoritmo segue in linea generale l'esecuzione dell'algoritmo asynchronous backtracking, ma sono presenti alcune differenze importanti. Anche qui quando si riceve un messaggio Ok si aggiorna la agent view e quindi si verifica se il valore assegnato alla variabile è consistente. Analogamente, se si riceve un messaggio Nogood, si contattano tutti gli agenti che non sono vicini e quindi si salva il Nogood, verificando se il valore assegnato alla variabile locale è consistente con la agent view e i Nogood ricevuti, inviando un messaggio Ok di risposta se il valore vecchio per la variabile locale è ancora

consistente. Una differenza è che se si contatta un agente per chiedergli di creare una nuova relazione di vicinanza, il valore della variabile locale del mittente della richiesta diventa parte del messaggio della richiesta, ed il destinatario usa tale valore per aggiornare la propria agent view quando riceve la richiesta.

Le differenze sono nelle operazioni da eseguire per verificare e, se necessario, modificare il valore della variabile locale. Per prima cosa occorre verificare se il valore corrente della variabile locale è ancora consistente (cioè non viola alcun vincolo o Nogood con variabili di priorità superiore): se lo è non occorre fare niente, altrimenti si verifica se esiste un valore per la variabile locale che è consistente. Se esistono due o più valori consistenti, si sceglie quello che viola il minor numero di vincoli con le variabili di priorità inferiore, cercando in questo modo di minimizzare i conflitti tra le variabili (questa scelta è detta euristica MIN-CONFLICT). Dopo aver scelto il valore, si inviano messaggi Ok a tutti i vicini. Notare che, a differenza dell'algoritmo asynchronous backtracking, i messaggi Ok vengono inviati a tutti i vicini, non solo a quelli con priorità inferiore.

Se invece non è possibile trovare alcun valore per la variabile locale, si procede a generare un Nogood. Anche qui se si genera un Nogood vuoto si termina l'algoritmo. Se invece il Nogood non è vuoto, si verifica se per caso un Nogood uguale a quello appena generato non è già stato generato ed inviato in precedenza. Se un Nogood uguale è già stato inviato in precedenza non si fa niente, altrimenti si invia il Nogood a tutti gli agenti che compaiono nel Nogood stesso. Quindi si trova la priorità massima di tutti i vicini e si aggiorna la priorità di questo agente impostandola al massimo più uno delle priorità dei vicini. Quindi si cerca un valore per la variabile che sia consistente e che violi il minor numero di vincoli considerando le variabili con priorità inferiore. Notare che ora la priorità dell'agente che esegue queste operazioni è maggiore di quella di tutti i suoi vicini, quindi non ci sono variabili con una priorità più alta nella sua agent view. Infine si inviano messaggi Ok a tutti i vicini.

Analizzando l'algoritmo, si nota che la priorità di un agente cambia solo se esso genera un Nogood nuovo. Si nota inoltre che i messaggi Ok vengono sempre inviati a tutti i vicini, anche a quelli con priorità superiore. Questo perché le priorità non sono fissate e dunque un agente che in un certo momento ha una priorità superiore a quella del mittente dei messaggi Ok, in seguito potrebbe ritrovarsi ad avere una priorità inferiore a quella del mittente. Lo stesso discorso vale per l'invio dei Nogood: questi sono inviati a tutti gli agenti che compaiono nel Nogood perché l'unico agente che potrà verificare tale Nogood

sarà l'agente tra quelli nel Nogood con priorità minima, ma dato che le priorità cambiano dinamicamente, questo agente con priorità minima non è noto a priori e potrebbe anche cambiare più volte durante l'esecuzione.

Il fatto di inviare i messaggi Ok a tutti i vicini implica che la agent view di ogni agente possa contenere i valori delle variabili di tutti i vicini, anche di quelli che hanno priorità inferiore. Quando si deve verificare la consistenza di un determinato valore per la variabile locale, si considerano solo le variabili nella agent view con priorità superiore a quella della variabile locale. Gli altri valori nella agent view vengono comunque usati e aggiornati per controllare il numero di violazioni di vincoli con variabili di priorità inferiore.

È anche importante notare che, quando si verifica il valore della variabile locale, per prima cosa si controlla se il valore precedente è ancora consistente, indipendentemente dal numero di vincoli con variabili di priorità inferiore che vengono violati da tale valore. Se poi occorre scegliere un nuovo valore, allora si controlla anche il numero di vincoli con variabili di priorità inferiore violati.

2.2.2 Completezza dell'algoritmo

Anche in questo caso è possibile dimostrare che l'algoritmo è completo. La dimostrazione è tratta, da [YH00].

Proposizione 2.2. *Non è possibile che gli agenti che eseguono l'algoritmo AWC cadano in un ciclo di elaborazione infinito.*

Dimostrazione. I valori di priorità degli agenti possono cambiare solo quando si genera un nuovo Nogood. Dato che il numero di Nogood distinti che possono essere generati è finito, si ha che dopo un tempo finito le priorità degli agenti saranno stabili. Una volta fissate le priorità degli agenti, l'algoritmo asynchronous weak-commitment search si comporta in maniera identica all'algoritmo asynchronous backtracking search, con l'unica eccezione dell'uso dell'euristica di selezione del valore con minore numero di conflitti con variabili di priorità inferiore. Dato che quest'ultimo algoritmo è stato dimostrato essere completo indipendentemente dal metodo usato per la scelta del valore da assegnare ad ogni variabile, anche l'algoritmo asynchronous weak-commitment search è completo. □

È anche presente una differente dimostrazione della completezza di questo algoritmo in [Yok95]

2.2.3 Riduzione dei Nogood

Anche in questo caso occorre determinare un metodo per la generazione dei Nogood. Dato che un Nogood può essere generato solo quando non è possibile trovare nessun valore consistente per la variabile locale, questo significa che il fatto che un Nogood venga creato dipende solo dalle variabili con priorità maggiore di quella dell'agente locale presenti nella agent view. In altre parole, dato che le variabili con priorità inferiore non vengono considerate durante il controllo della consistenza di un valore per la variabile locale, tutte le variabili con priorità inferiore possono essere escluse dal Nogood che si deve produrre senza nessuna conseguenza.

Di conseguenza si considerano solo le variabili con priorità superiore presenti nella agent view. Anche qui si segue il metodo di rimuovere una variabile per volta dall'insieme iniziale e vedere se, rimuovendo una certa variabile, diventa possibile trovare un valore consistente per la variabile locale. Quello che cambia è l'ordine in cui conviene eseguire la verifica. In questo caso il Nogood generato dovrà essere inviato a tutte le variabili che compaiono nel Nogood stesso. Quindi non ha senso cercare di massimizzare la priorità minima delle variabili nel Nogood per influenzare variabili con priorità superiore. Tuttavia cercare di rimuovere le variabili con priorità inferiore può avere un altro effetto: quando un agente riceve un Nogood, è possibile che, a causa del Nogood appena ricevuto, esso non riesca a trovare un valore per la propria variabile e sia costretto a generare un nuovo Nogood e ad aumentare la propria priorità. Quando le priorità cambiano, alcuni Nogood inviati in precedenza fanno riferimento a variabili con priorità inferiore. Questo significa che alcuni Nogood ricevuti non vengono utilizzati dopo il cambiamento delle priorità. Dato che la generazione di tali Nogood ha richiesto un certo sforzo computazionale, il non utilizzare tali Nogood vanifica parzialmente lo sforzo che ha portato alla loro creazione. Per questo motivo si è deciso comunque di procedere alla semplificazione dei Nogood rimuovendo le variabili in ordine di priorità, partendo da quelle con priorità più bassa. In questo modo, se uno o più destinatari del Nogood incrementano la propria priorità, si cerca di fare in modo che tali agenti avessero già prima una priorità più elevata possibile, in modo da limitare gli effetti di tale incremento di priorità.

when received (**Ok?**, (x_j, d_j)) **do**
  update agent view
  execute check_agent_view
end do

when received (**Nogood**, x_j, *nogood*) **do**
  record *nogood* as a new constraint
  if *nogood* contains an agent x_k that is not its neighbor **then**
    send (**add_neighbor** (x_i, $currentValue$)) to x_k
    add x_k to the neighbors
  end if
  $oldValue \leftarrow currentValue$
  execute check_agent_view
  if $oldValue = currentValue$ **then**
    send (**Ok?**, (x_i, $currentValue$)) to x_j
  end if
end do

when received (**add_neighbor**, (x_j, d_j)) **do**
  add x_j to the neighbors
  add (x_j, d_j) to the agent view
  send (**Ok?**, (x_i, $currentValue$)) to x_j
end do

procedure backtrack
  create a copy V of the agent view
  sort the variables in V in ascending priority order
  for each variable x_k in V **do**
    remove x_k from V
    if a value in D_i is consistent with V **then**
      reinsert x_k in V
    end if
  end for
  sort V in descending variable identifier order
  if V is an empty Nogood **then**
    broadcast to other agents that there is no solution
    terminate this algorithm
  end if
  if V is a new Nogood **then**
    send V to the agents in the Nogood
    $currentPriority \leftarrow 1 + pmax$
    where $pmax$ is the maximal priority value of neighbors
    select $d \in D_i$ where agent view and d are consistent
    and d minimizes the number of constraint violations
    with lower priority agents
    $currentValue \leftarrow d$
    send (**Ok?**, (x_i, d, $currentPriority$)) to neighbors
  end if
end backtrack

```
procedure check_agent_view
  if agent view and currentValue are not consistent then
    if no value in D_i is consistent with agent view then
      execute backtrack
    else
      select d ∈ D_i where agent view and d are consistent
      and d minimizes the number of constraint violations
      with lower priority agents
      currentValue ← d
      send (Ok?, (x_i, d, currentPriority)) to neighbors
    end if
  end if
end check_agent_view
```

2.2.4 Tenere traccia dei Nogood già inviati

Un aspetto ulteriore da notare è la necessità di mantenere un elenco dei Nogood inviati. Questo è necessario per evitare che possano essere generati ripetutamente gli stessi Nogood, con il risultato che diventerebbe possibile un ciclo infinito di creazione di Nogood e modifiche delle priorità che comprometterebbe la completezza dell'algoritmo. In realtà è sufficiente scegliere un valore massimo abbastanza grande per la struttura dati contenente tali Nogood per evitare problemi in tutti i case reali. Nel suo lavoro [YI98] Yokoo afferma che anche solo tenendo traccia degli ultimi 10 Nogood generati, non ci sono stati problemi di terminazione dell'algoritmo in tutti i test effettuati.

Per cercare di sfruttare le migliori prestazioni derivanti dal ridotto numero di Nogood memorizzati mantenendo nel contempo la completezza dell'algoritmo, è possibile partire con una coda di dimensione iniziale fissata (per esempio una coda di 10 elementi) contenente gli ultimi Nogood generati. Poi, usando un contatore che tiene traccia del numero di cicli eseguiti dall'agente, è possibile stabilire un numero di cicli fissato, raggiunto il quale la dimensione massima della coda di Nogood aumenta ed il contatore viene azzerato. In tutti i casi, dato che dopo un tempo finito la lunghezza della coda aumenta, si ha che o l'algoritmo raggiunge una condizione di terminazione (tutti gli agenti in attesa

di messaggi e nessun vincolo violato, oppure terminazione dell'algoritmo a causa di un Nogood vuoto), oppure la dimensione massima della coda diventa abbastanza grande da contenere tutti i Nogood generabili (che sono un numero finito). Se la coda diventa abbastanza grande da contenere tutti i possibili Nogood, si ricade nel caso della dimostrazione precedente di completezza dell'algoritmo asynchronous weak-commitment search, in cui non si era fatta nessuna assunzione sul massimo numero di Nogood memorizzabili. Questo semplicemente perché, pur essendo presente un numero massimo di Nogood memorizzabili, tutti i Nogood memorizzati sono diversi tra loro e il numero massimo di Nogood diversi che è possibile creare è minore della dimensione massima della coda. Di conseguenza, pur essendo presente un limite massimo alla dimensione della coda, questo limite non verrà mai raggiunto e dunque non influenza il funzionamento dell'algoritmo.

Questa modifica permette sia di risparmiare memoria, sia di velocizzare la verifica del fatto che un Nogood è nuovo. Inoltre, riducendo la quantità di memoria utilizzata, permette di sfruttare meglio la gerarchia di memoria dell'elaboratore, ottenendo prestazioni migliori grazie all'utilizzo migliore delle cache.

Un altro punto critico in entrambi gli algoritmi per quanto riguarda le prestazioni è il controllo dei Nogood ricevuti. Questo elenco cresce rapidamente di dimensioni ed ogni volta che si verifica se un valore per la variabile locale è consistente, occorre scandire tutti i Nogood ricevuti. Tale elenco di Nogood può contenere Nogood duplicati e/o ridondanti anche se non duplicati. Se infatti nell'elenco sono presenti i Nogood $\{(x_2, 3)\}$ e $\{(x_2, 3), (x_7, 4)\}$, è evidente come il primo Nogood implichi anche il secondo. Questo permetterebbe di eliminare in maniera sicura il secondo Nogood, riducendo l'occupazione di memoria e velocizzando le successive operazioni di controllo dei Nogood.

Per questo motivo può essere utile introdurre una operazione di rimozione della ridondanza dell'elenco dei Nogood ricevuti che periodicamente si occupi di rimuovere i Nogood duplicati e/o ridondanti. Una semplice procedura per eseguire tale rimozione della ridondanza può essere la seguente:

```
procedure redundancy_removal
  for each Nogood N_j in the received Nogoods list do
    for each Nogood N_k in the received Nogoods list (with k > j) do
      if N_k contains less variables than N_j then
        if every variable of N_k also appears in N_j with the same value then
          remove N_j from the received Nogoods list
          goto label
        end if
      else
        if every variable of N_j also appears in N_k with the same value then
          remove N_k from the received Nogoods list
        end if
      end if
    end do
    label
  end do
end redundancy_removal
```

Questa procedura rimuove sia i Nogood duplicati, sia quelli ridondanti. La complessità in tempo di tale operazione è $O(\frac{n^2}{2} \cdot m)$ nel caso peggiore (assumendo che le variabili all'interno di ogni Nogood siano già ordinate in base al loro identificatore), con n numero di Nogood inizialmente presenti ed m numero medio di variabili in ogni Nogood. Questo algoritmo, se l'elenco di Nogood supporta rimozioni in tempo costante e accesso sequenziale (come nel caso delle liste collegate), opera in-place, non richiedendo memoria aggiuntiva in misura dipendente dalla dimensione del problema.

Capitolo 3

Implementazione Java

3.1 Implementazione degli algoritmi

Ora occorre provvedere ad implementare il sistema descritto. Si è deciso di usare il linguaggio Java per l'implementazione, dato che questo linguaggio, possedendo dei costrutti e delle funzionalità di livello più alto di quelle fornite dal linguaggo C solitamente utilizzato per questi scopi, permette di esprimere più facilmente e più concisamente molti concetti necessari per l'implementazione del risolutore.

Si è deciso di iniziare dall'algoritmo più semplice dei due algoritmi distribuiti per la risoluzione dei CSP, il backtracking asincrono. Per implementare l'algoritmo mantenendo una possibilità di ampliamento e generalizzazione (necessaria per permettere in seguito l'aggiunta di nuovi algoritmi di risoluzione), si è proceduto alla creazione di una gerarchia di classi astratte. Tale gerarchia prevede una classe base astratta denominata `Agent`.

La classe astratta `Agent` fornisce alcune funzionalità utilizzate da tutti gli algoritmi di risoluzione, come i metodi `send(Message, int)` e `receive()` che permettono rispettivamente di inviare e ricevere un messaggio. A tale scopo, oltre ad implementare i metodi, la classe contiene anche la coda dei messaggi in arrivo per ciascun agente. Essa inoltre tiene traccia del valore della variabile locale dell'agente, e contiene un elenco di tutti i vicini dell'agente. In sostanza la classe `Agent` fornisce tutte le funzionalità di base necessarie per implementare un algoritmo distribuito per la risoluzione dei CSP. La classe `Agent` contiene anche un metodo di inizializzazione (necessario per preparare l'oggetto prima dell'esecuzione) e la dichiarazione di un metodo astratto `run()` che serve ad implementare l'interfaccia `Runnable`. Questa interfaccia permette di usare un thread per mandare in esecuzione ogni oggetto `Agent`, permettendo di simulare la presenza di più agenti in locale usando i thread.

L'aspetto interessante è che il metodo `run()`, essendo astratto, viene implementato dalle sottoclassi di `Agent`. Questo significa che ogni sottoclasse, implementando un algoritmo differente, fornirà una diversa implementazione di `run()`. Di conseguenza diventa possibile creare gli agenti in un punto del programma e poi passarli, se necessario, da un metodo all'altro come semplici oggetti `Agent`, senza sapere che algoritmo essi implementino. Infine, anche per avviare l'esecuzione non occorre conoscere niente degli oggetti `Agent`: basta creare un thread che esegua il metodo astratto `run()`, e l'esecuzione inizia. Questo uso dell'astrazione semplifica molto la gestione degli agenti nel codice di inizializzazione del sistema.

3.1.1 Asynchronous backtracking search

Una volta creata una versione iniziale della classe `Agent`, si è provveduto ad implementare l'algoritmo di backtracking asincrono usando come base tale classe. Questo ha portato alla creazione della classe `ABTAgent`. Tale classe implementa il metodo `run()` della classe `Agent`, ma per fare questo ha bisogno di poter eseguire delle operazioni che sono specifiche del problema da risolvere. Queste operazioni sono la ricerca di un nuovo valore per la variabile locale e la ricerca dei valori della variabile locale che non sono ammissibili a causa di qualche vincolo. Si assume che tali operazioni vengano eseguite da una coppia di metodi astratti. Tali metodi dovranno quindi essere implementati da una sottoclasse e, per permettere alla sottoclasse di avere abbastanza informazioni per poter implementare tali metodi, si fornisce un metodo protetto per cercare tutti i valori per la variabile locale che non sono ammissibili a causa di un Nogood o di un vincolo.

Per quanto riguarda le strutture dati interne, la classe `ABTAgent` implementa la agent view. L'implementazione del metodo `run()` è, di fatto, una trascrizione dello pseudocodice dell'algoritmo di backtracking asincrono, con alcuni controlli in più per quanto riguarda eventuali errori. Si è inoltre reso necessario implementare le operazioni da eseguire per la ricezione di messaggi di richiesta di vicinato (cosa che non era presente nello pseudocodice di Yokoo in [YH00]), ma sostanzialmente l'implementazione del metodo `run()` in questa fase è una semplice trascrizione dello pseudocodice dell'algoritmo di backtracking asincrono.

Per poter eseguire il programma, occorre ancora implementare le operazioni specifiche del problema. Quindi si crea un'ulteriore sottoclasse di `ABTAgent` che implementa i metodi di ricerca di un nuovo valore e la ricerca dei valori non ammissibili per la variabile a

causa di qualche vincolo. Si è dunque provveduto a creare tale classe. Inizialmente si è implementato un semplice problema con tre variabili, i cui vincoli imponevano che i valori delle tre variabili fossero tutti diversi tra loro. In seguito si è passati ad implementare il problema del sudoku.

L'implementazione del sudoku ha evidenziato come l'approccio iniziale di usare l'intera agent view come Nogood fosse impraticabile. Si è quindi provveduto ad aggiungere un metodo privato alla classe `ABTAgent` in modo da creare dei Nogood più piccoli usando l'euristica descritta nel capitolo 2.1.7. L'introduzione di questa modifica ha portato ad un immediato e drastico miglioramento delle prestazioni del sistema, senza influenzare la correttezza dell'algoritmo.

Dopo questa modifica, si è proceduto a migliorare il metodo privato che cerca i valori della variabile locale che non sono ammissibili a causa di un Nogood. Si è sfruttato l'ordinamento delle variabili nei Nogood (ed il fatto che l'ultima variabile di un Nogood ricevuto è sempre la variabile locale) per migliorare le prestazioni di tale metodo, in modo da velocizzare un'operazione che è eseguita molte volte durante l'esecuzione del sistema. Da alcuni test eseguiti dopo la modifica, si è notato che, dopo la modifica, le prestazioni del sistema degradano più lentamente man mano che si accumulano dei Nogood ricevuti rispetto a quanto accadeva usando l'algoritmo precedentemente impiegato per la ricerca dei valori non ammessi.

Si è cercato di migliorare ulteriormente le prestazioni di tale metodo usando un algoritmo di ricerca speciale, un ibrido tra l'algoritmo di ricerca ad interpolazione e l'algoritmo di ricerca binaria. Tale ibrido richiede che il vettore in cui effettuare la ricerca sia ordinato in ordine crescente e che ogni elemento implementi una particolare interfaccia, tramite cui l'algoritmo di ricerca può ottenere una chiave intera per eseguire le proprie elaborazioni. Questo ibrido cerca di eseguire l'algoritmo di ricerca ad interpolazione, ma se si accorge di eseguire la scelta peggiore per quanto riguarda le dimensioni del sottovettore da considerare ad ogni passo, esegue delle ulteriori operazioni per assicurarsi che la lunghezza di ogni sequenza sia al più la metà della lunghezza della sequenza precedente. Questo significa che l'algoritmo ha complessità $O(1)$ nel caso migliore, $O(\log n)$ nel caso peggiore e $O(\log(\log n))$ nel caso medio. Tale algoritmo viene usato per ricercare valori nella agent view di ogni agente, che viene mantenuta ordinata secondo l'ordine degli identificatori delle variabili.

Inoltre, si è provato ad aggiungere del codice per ridurre le dimensioni dell'elenco dei

Nogood ricevuti. Invece di aggiungere semplicemente ogni nuovo Nogood alla lista dei Nogood ricevuti, si è provato ad eseguire un controllo per vedere se il nuovo Nogood è già implicato da qualche altro Nogood già presente o se esso può sostituire uno o più Nogood della lista. Tale serie di operazioni tende a diminuire le dimensioni medie dei Nogood nella lista dei Nogood ricevuti e le dimensioni della lista stessa, ma il costo computazionale del controllo da eseguire per ogni Nogood non è compensato dai vantaggi derivanti da un minor numero di Nogood da controllare. Questo significa che cercare di mantenere la lista dei Nogood ricevuti più compatta possibile richiede troppe risorse per essere efficiente. Per questo potrebbe essere più utile lo schema di rimozione della ridondanza dall'elenco dei Nogood descritto nel capitolo 2.2.4, in cui l'operazione di rimozione della ridondanza viene eseguita solo di tanto in tanto.

3.1.2 Asynchronous weak-commitment search

Dopo l'implementazione dell'algoritmo di backtracking asincrono, si è iniziato ad implementare l'algoritmo asynchronous weak-commitment search. Si è partiti ancora dalla classe `Agent` e si è creata una nuova sottoclasse chiamata `AWCAgent`. Come `ABTAgent`, questa classe implementa il metodo `run()` della classe `Agent` in modo da eseguire l'algoritmo per la risoluzione dei CSP. Anche in questo caso occorre affidarsi a delle operazioni specifiche del problema da risolvere, che sono rappresentate da dei metodi astratti che devono essere implementati dalle sottoclassi di `AWCAgent`.

Anche qui si è provveduto ad implementare una sottoclasse che risolvesse il problema del sudoku. In questo caso occorre fornire a tale sottoclasse una quantità di informazioni leggermente maggiore. Per implementare l'euristica MIN-CONFLICT descritta nello pseudocodice, infatti, occorre sapere anche i valori delle variabili nella agent view che hanno priorità inferiore alla variabile locale. Di conseguenza, si è reso protetto un metodo di ricerca nella agent view presente nella classe `AWCAgent`, in modo che le sottoclassi possano usarlo per controllare i valori di variabili con priorità inferiore a quella locale.

Nel caso della classe `AWCAgent`, si è provveduto subito ad implementare le ottimizzazioni che sono state descritte per la classe `ABTAgent`. Si è implementata la riduzione dei Nogood, la ricerca rapida nell'elenco dei Nogood ricevuti e l'uso dell'algoritmo di ricerca ibrido per la agent view. L'algoritmo asynchronous weak commitment, tuttavia, presenta delle difficoltà aggiuntive nell'applicare tali ottimizzazioni rispetto all'algoritmo asynchronous backtracking. In ABT, infatti, la priorità delle variabili è determinata dal

loro identificatore, mentre in AWC è potenzialmente diversa. Mantenere la agent view ordinata in base alla priorità effettiva degli agenti non è proponibile a causa del costo dei continui riordinamenti. D'altra parte, l'euristica per la minimizzazione dei Nogood dell'algoritmo AWC richiede di avere le variabili del Nogood ordinate in base alla loro priorità. Infine, per verificare efficientemente quali valori della variabile locale sono proibiti dai Nogood, occorre che le variabili nei Nogood ricevuti siano ordinate in base al loro identificatore.

Per risolvere queste esigenze contrastanti, si è deciso di mantenere la agent view ordinata in base all'identificatore delle variabili, fornendo operazioni che considerano solo le variabili con priorità maggiore o minore di quella locale per poter effettuare ricerche selettive in tali categorie. Nella generazione dei Nogood si è quindi proceduto ad implementare un riordinamento di una copia della agent view in base alle priorità, procedendo alla rimozione di alcune variabili secondo l'euristica e quindi riordinando il Nogood secondo l'identificatore delle variabili.

Si è quindi cercato di migliorare l'efficienza delle operazioni specifiche del problema nelle sottoclassi di `ABTAgent` e `AWCAgent` che implementano il sudoku. Tali operazioni, essendo eseguite molte volte, possono portare a notevoli miglioramenti di efficienza complessiva del sistema se si riesce a renderle anche solo leggermente più efficienti.

3.2 Comunicazione tramite socket

Dopo aver implementato tali migliorie ed aver corretto alcuni bug, si è iniziato a lavorare sul metodo di comunicazione remoto per scambiare messaggi tra agenti posizionati su macchine fisicamente distinte e collegate tra loro da una rete.

A questo punto occorreva scegliere come procedere per la ricezione dei messaggi provenienti da agenti remoti. Normalmente, infatti, è presente un thread od un processo che resta continuamente in attesa di dati in arrivo dalla rete, e provvede poi ad inoltrare tali dati al destinatario corretto. Tale approccio è ampiamente utilizzato in quanto semplifica molto la logica di ricezione dei messaggi, ma presenta il problema di richiedere un thread o un processo dedicato che utilizza risorse controllando continuamente se sono presenti dei dati in arrivo. Inoltre, se un ipotetico utente malintenzionato volesse interrompere il funzionamento del sistema, tale thread o processo diventerebbe un bersaglio naturale, dato che interrompendone l'esecuzione, l'intero sistema smetterebbe di funzionare corret-

tamente. Per questi motivi si è deciso di seguire un approccio diverso: invece di avere un thread dedicato per la ricezione dei messaggi, si è fatto in modo che, quando un agente vuole ricevere un messaggio e la sua coda di messaggi è vuota, esso prova a scaricare i messaggi provenienti da agenti remoti e continua a tentare fino a che non riesce a scaricare almeno un messaggio, finché qualcuno non inserisce un messaggio nella sua coda o finché un altro agente non si ritrova nella stessa situazione. È necessario un sistema di controlli per fare in modo che il sistema non vada in deadlock se tutti gli agenti locali hanno una coda di messaggi vuota: bisogna garantire che quando questo succede, un agente continui a cercare di scaricare i messaggi provenienti dall'esterno, in modo che il gruppo di agenti possa reagire a messaggi provenienti da agenti non locali.

Quindi occorre creare il sistema di comunicazione remoto. Si è creato un oggetto astratto denominato `RemoteCommunicator`. Tale oggetto fornisce metodi (astratti) quali `send(Message)` e `receive(int)` di messaggi da e verso agenti posizionati su macchine remote. L'idea è quella di implementare una serie di sottoclassi di `RemoteCommunicator`, ciascuna delle quali usi un metodo di comunicazione diverso per implementare le varie funzionalità, in modo da verificare quale metodo di comunicazione sia migliore. Notare che l'oggetto `RemoteCommunicator` di ogni agente è condiviso tra tutti gli altri agenti che sono in esecuzione nella stessa istanza del programma. Questo significa che il numero di oggetti `RemoteCommunicator` presenti in totale è pari al numero di istanze del programma. Tale numero può essere diverso dal numero di macchine fisicamente presenti: se si lanciano due istanze del programma sulla stessa macchina, ciascuna istanza creerà un oggetto `RemoteCommunicator` per comunicare con l'altra.

Si è iniziato da un'implementazione basata direttamente sui socket, racchiusa in una classe chiamata `SocketRemoteCommunicator`. Tale oggetto deve svolgere numerose funzioni, come stabilire dei canali di comunicazione con le altre macchine che contengono almeno un agente, stabilire su quale macchina si trovi un determinato agente e provvedere ad inviare e ricevere i messaggi attraverso i socket.

3.2.1 Apertura delle connessioni

Per aprire le connessioni, si è creato un metodo dell'oggetto che prende un elenco di indirizzi e cerca di aprire un socket verso ciascuno di tali indirizzi. Il problema di questo metodo è che se ogni oggetto apre un canale verso tutti gli altri oggetti, si verranno a creare due canali aperti tra ogni coppia di oggetti `SocketRemoteCommunicator`. Per

risolvere il problema occorre quindi che ogni coppia di oggetti stabilisca un solo canale di comunicazione. Questo significa che, per ogni coppia di oggetti, solo uno dei due deve aprire un canale di comunicazione verso l'altro. L'oggetto che non deve aprire il canale di comunicazione deve invece mettersi in attesa di nuove connessioni in arrivo. Per poter fare questo, si è deciso di usare l'ordinamento lessicografico degli indirizzi IP e delle porte delle macchine coinvolte. Questo permette di stabilire che, per ogni coppia di oggetti, quello che ha l'indirizzo IP lessicograficamente maggiore può aprire un socket verso quello che ha l'indirizzo lessicograficamente minore, che invece deve mettersi in attesa delle connessioni in arrivo. Questo schema elimina il problema delle doppie connessioni, ma presenta dei problemi legati al sistema di indirizzi IP. Se infatti sono presenti indirizzi di loopback o se sono presenti degli apparati di NAT nella rete, possono verificarsi problemi con il meccanismo dell'ordinamento degli indirizzi, che possono portare all'apertura di doppie connessioni o alla mancata apertura di alcune connessioni. Questo problema non è stato ancora risolto e necessita di ulteriore lavoro.

3.2.2 Gestione della rubrica e protocollo HLARP

Una volta ottenuti tutti i vari socket, occorre sapere dove si trovano i vari agenti per poter inviare i messaggi al destinatario corretto. A tale scopo si crea una struttura dati (rubrica) che associa ad ogni identificatore di variabile l'indice del socket che comunica con l'istanza del programma dove si trova l'agente che gestisce tale variabile. Il problema è che inizialmente tale rubrica è vuota ed occorre popolarla. Per fare questo, si è creato un semplice protocollo simile al protocollo ARP (Address Resolution Protocol), denominato High Level Address Resolution Protocol (HLARP). Tale protocollo prevede, quando occorre sapere dove si trova un determinato agente, di inviare una richiesta contenente l'identificatore dell'agente cercato attraverso tutti i socket aperti. Quando una tale richiesta viene ricevuta, si verifica se l'agente richiesto è presente nell'istanza locale. Se non lo è, la richiesta viene scartata, altrimenti si invia una risposta al mittente della richiesta. Quando il mittente della richiesta si vede arrivare una risposta, esso salva l'identificatore del socket attraverso cui tale risposta è arrivata e lo usa per aggiornare la rubrica. Inoltre, ogni volta che un istanza riceve un messaggio, esso controlla l'identificatore dell'agente mittente e l'indice del socket attraverso cui tale messaggio è arrivato, se necessario usando tali informazioni per aggiornare la rubrica.

Questo meccanismo prevede di inviare una richiesta HLARP quando si rende necessa-

rio sapere dove si trova un determinato agente. Tale necessità si verifica principalmente quando occorre inviare un messaggio ad un agente non nella rubrica. Quando si verifica tale situazione, viene immediatamente inviata la richiesta HLARP, ma finché la risposta non viene ricevuta, il messaggio non può essere inviato. Quindi il messaggio viene salvato all'interno dell'oggetto in una coda. Ogni volta che si riceve o si cerca di inviare un messaggio, si verifica se è possibile inviare uno o più messaggi tra quelli nella coda. Questo permette di evitare di doversi affidare a delle chiamate esterne per l'invio ritardato dei messaggi se questi non possono essere inviati immediatamente: se i messaggi non possono essere inviati subito, essi sono salvati all'interno dell'oggetto `SocketRemoteCommunicator` e vengono inviati appena possibile. È importante notare che l'ordine dei messaggi è fondamentale: tutti gli algoritmi distribuiti considerati si basano sull'assunzione che i messaggi inviati tra ogni coppia di agenti vengano ricevuti dal destinatario nell'ordine in cui sono stati inviati dal mittente. Se i messaggi fossero scambiati di ordine in una fase qualsiasi del processo di invio o ricezione, il sistema non funzionerebbe correttamente.

3.2.3 Invio e ricezione dei messaggi

Rimane da trovare un metodo per inviare e ricevere messaggi attraverso i socket. Il numero di socket da gestire rende necessario trovare un modo efficiente per controllare le operazioni di invio, ricezione e apertura di eventuali nuove connessioni. A tale scopo si usa un oggetto `Selector`. Per poter usare l'oggetto `Selector` con i socket in Java, è necessario che i socket siano configurati in modalità non bloccante.

Una volta che si deve provvedere all'invio di un messaggio (ed anche alla sua ricezione), occorre convertire il messaggio dalla sua rappresentazione in memoria ad una serie di byte che possano essere inviati attraverso il socket. Queste operazioni possono essere eseguite dagli oggetti `ObjectOutputStream` ed `ObjectInputStream`, che possono anche lavorare direttamente sui socket senza aver bisogno di altre strutture dati. Tuttavia tali oggetti richiedono che i socket siano configurati in modalità bloccante.

Per risolvere tale contraddizione tra le necessità dell'oggetto `Selector` e quelle degli oggetti `ObjectInputStream` ed `ObjectOutputStream`, si è provveduto ad implementare dei metodi che permettessero di serializzare un oggetto all'interno di un buffer e di inviarlo attraverso un socket non bloccante, oltre a permettere di eseguire l'operazione inversa. Questi metodi usano un buffer locale per permettere di usare gli oggetti `ObjectInputStream` ed `ObjectOutputStream`, provvedendo manualmente ad inviare e

ricevere il contenuto di tale buffer. Un problema da risolvere è la necessità di sapere, quando si riceve un messaggio, la lunghezza del messaggio stesso per allocare un buffer grande abbastanza e per sapere quando tutti i dati sono stati ricevuti. Per questo motivo, nella fase di invio si serializza il messaggio nel buffer e si aggiunge nel buffer, prima del messaggio stesso, un intero serializzato che rappresenta la lunghezza della parte rimanente del messaggio. In questo modo il ricevente può scaricare prima quattro byte di dati dal socket, ricomporre l'intero ed usare il suo valore per sapere quanto grande deve essere il buffer per il messaggio. Quindi, una volta creato il buffer, si continua ad attendere finché non sono stati ricevuti tutti i dati, si usa l'oggetto `ObjectInputStream` sul buffer e si restituisce l'oggetto ottenuto.

Un problema di tale approccio scoperto durante i test è la possibilità di deadlock di rete: due o più oggetti `SocketRemoteCommunicator` cercano di inviare dei messaggi l'uno verso l'altro, ma durante l'invio i buffer di ricezione del destinatario e di invio del mittente si riempiono. In questa situazione nessuno riesce ad inviare altri dati, ma tutti continuano a provare ad inviare. Quindi nessuno cerca mai di ricevere i messaggi in arrivo, operazione che potrebbe svuotare i buffer e permettere lo sblocco del sistema.

Per risolvere il problema, si è provveduto ad aggiungere un contatore in modo che, se l'operazione di invio richiede più di un certo numero di tentativi, si interrompe l'invio dei dati per cercare di scaricare tutti i messaggi in arrivo e svuotare i buffer. Una volta che l'operazione di ricezione è completa, si ritorna all'invio dei dati, cercando di inviare quanto non si era riusciti ad inviare prima. I messaggi ricevuti per svuotare i buffer vengono salvati in un elenco all'interno dell'oggetto `SocketRemoteCommunicator` e verranno recuperati alla successiva chiamata del metodo receive. Anche qui occorre fare attenzione a non scambiare l'ordine di ricezione dei messaggi.

A questo punto il sistema è in grado di risolvere problemi usando i due algoritmi implementati, terminando correttamente se il problema non ha soluzione e raggiungendo uno stato stabile se il problema ha soluzione. Il passo successivo è stato la creazione di un sistema per il rilevamento della terminazione con successo.

3.3 Rilevamento della terminazione con successo

Si è deciso di procedere con un approccio a due fasi: per prima cosa occorre stabilire se tutti i gruppi di agenti hanno un agente che controlla continuamente i messaggi in arrivo,

segno che tutti gli altri agenti del gruppo sono in attesa di messaggi. Se tutti i gruppi sono in attesa, occorre poi verificare se tutti i singoli agenti hanno trovato un valore per la propria variabile che soddisfa tutti i vincoli.

Per procedere alla prima fase, si è fatto in modo che quando un agente cerca di ricevere messaggi da mittenti remoti per un certo tempo senza ricevere niente, esso prova ad inviare a tutti gli altri gruppi di agenti una richiesta speciale per verificare se tali gruppi di agenti sono anch'essi in attesa di messaggi. I gruppi di agenti che ricevono tali messaggi rispondono comunicando il loro stato. Quando tutte le risposte sono tornate al mettente, si verifica se tutti sono in attesa di messaggi. Se qualche gruppo di agenti non è in attesa, il mittente delle richieste non fa nulla e si rimette in attesa; in caso contrario il mittente delle richieste provvede ad inviare a tutti gli agenti (incluso se stesso) un messaggio di richiesta che chiede di verificare la consistenza del valore assegnato alla propria variabile locale. Quando gli agenti ricevono tale messaggio, essi verificano se il valore assegnato alla loro variabile è consistente e rispondono al mittente comunicando o che l'assegnamento è consistente o che non è consistente (notare che il mittente della richiesta invia un messaggio anche a se stesso, quindi tale agente invierà la risposta ancora una volta a se stesso). L'agente che ha inviato le richieste raccoglie tutte le risposte e quindi verifica se tutti gli agenti hanno un assegnamento consistente. Se almeno un agente ha un assegnamento non consistente non si fa nulla, altrimenti si invia a tutti gli agenti una notifica che avvisa che la soluzione è stata trovata. Tutti gli agenti che ricevono tale notifica terminano.

Tale sistema si è dimostrato efficace nelle prove eseguite, ma occorre verificare ulteriormente la sua correttezza. Potrebbe essere possibile, infatti, che tutti gli agenti abbiano un assegnamento consistente per la propria variabile locale, ma che il valore di una variabile nell'agent view di un agente non corrisponda al valore vero di tale variabile.

3.4 Altri metodi di comunicazione: RMI e MPI

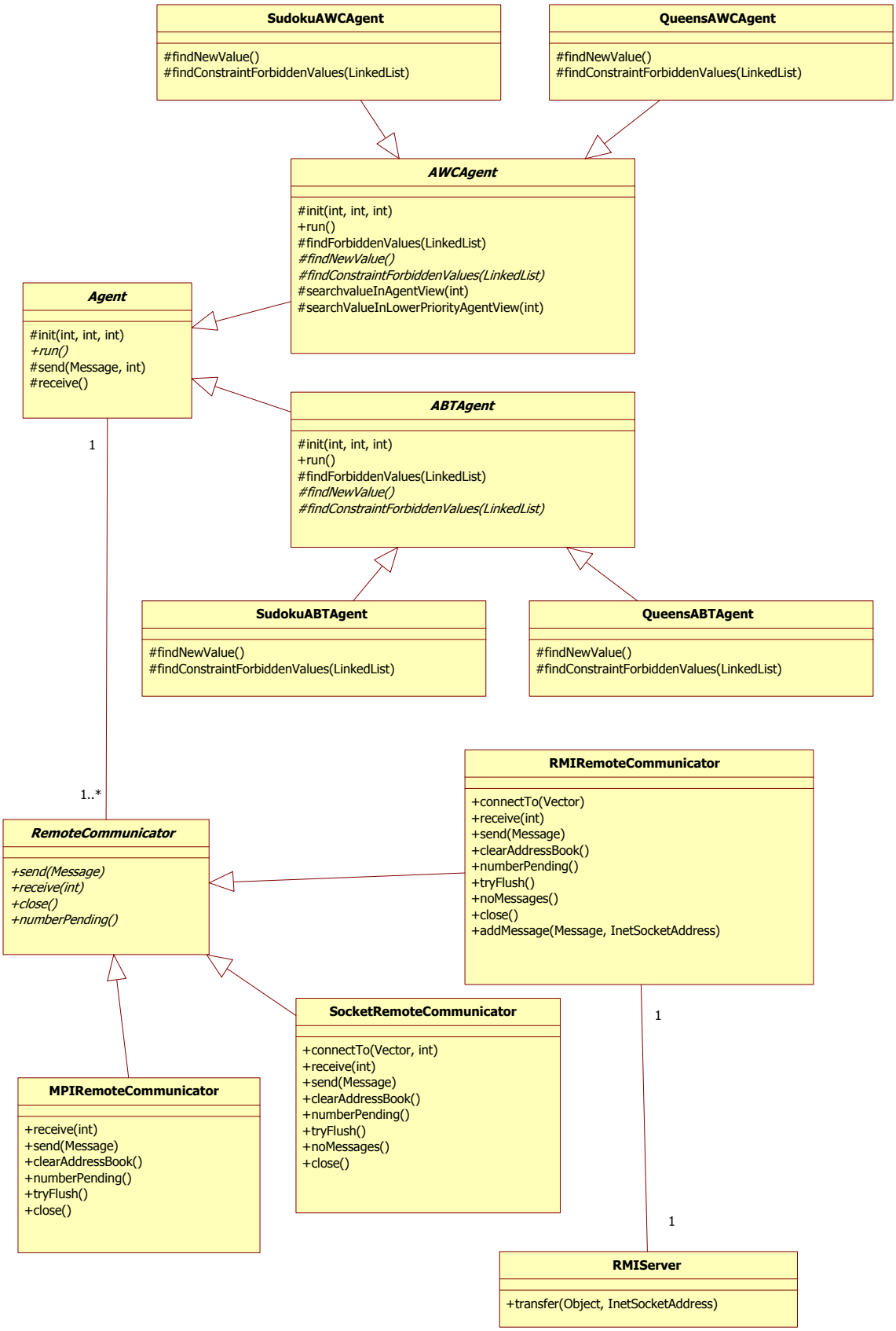
Si è poi proceduto ad implementare altri metodi di comunicazione per poter confrontare le prestazioni dei vari metodi utilizzabili. Si è provveduto ad implementare lo scambio di messaggi attraverso RMI ed usando un'implementazione di MPI per Java (MPJ Express). Tali implementazioni sono state racchiuse in ulteriori sottoclassi della classe `RemoteCommunicator`.

L'implementazione MPI è stata racchiusa nell'oggetto `MPIRemoteCommunicator`. Tale oggetto implementa le funzionalità della classe astratta `RemoteCommunicator` usando le funzionalità di MPI. In questo caso, però, non è necessaria la fase di connessione, dato che il sistema MPI si occupa di questo in automatico all'inizio dell'esecuzione. Occorre anche in questo caso la rubrica per sapere dove si trova ogni agente, cosa che implica la presenza del protocollo HLARP per inviare le richieste necessarie a costruire la rubrica. Questo significa anche che serve la coda di messaggi da inviare per cui si sta attendendo la risposta ad una richiesta HLARP. A parte questo, le parti di invio e ricezione de messaggi sono molto semplificate: per inviare un messaggio singolo si usano le primitive di invio di MPI, come pure per ricevere. Non occorre provvedere alla serializzazione e alla deserializzazione manuale dei messaggi, come non occorre tenere traccia di tutte le connessioni aperte e ricordarsi di chiuderle quando si finisce e quando si verifica un errore.

Si è poi provveduto ad implementare le stesse funzionalità usando RMI (Remote Method Invocation). La fase di connessione prevede di cercare di creare degli oggetti proxy per ogni destinatario, in modo che i successivi invii si riducano all'invocazione di un metodo su questi oggetti proxy. Anche qui serve la rubrica, il protocollo HLARP e la coda di messaggi da inviare. Tuttavia, per usare le funzionalità RMI, occorre anche creare un oggetto server che implementi un'interfaccia. Quindi ogni oggetto `RMIRemoteCommunicator` contiene un oggetto server che viene reso visibile e contattabile da tutti i processi che usano RMI. Tale server implementa un'interfaccia che definisce un metodo invocabile da remoto. Questo metodo prende come parametri un messaggio e l'indirizzo IP del mittente, e si preoccupa di copiare il messaggio nella coda di messaggi da ricevere del destinatario, aggiornandone la rubrica con l'indirizzo del mittente.

Tutti questi metodi di comunicazione (socket, MPI ed RMI) implementano la stessa interfaccia `RemoteCommunicator`, cosa che significa che gli agenti non hanno bisogno di sapere il metodo usato per scambiare i messaggi. Questo permette di eseguire dei test comparativi per verificare quale metodo di comunicazione sia il migliore tra i tre disponibili. Inoltre con questo approccio risulta molto facile implementare ulteriori metodi di comunicazione, per esempio per trarre vantaggio da particolari architetture di rete.

A questo punto si è implementato un nuovo problema per entrambi gli algoritmi: il problema delle n regine. Questo problema ha il vantaggio, rispetto al sudoku, di poter essere dimensionato a piacere, senza essere limitato ad una dimensione fissa.

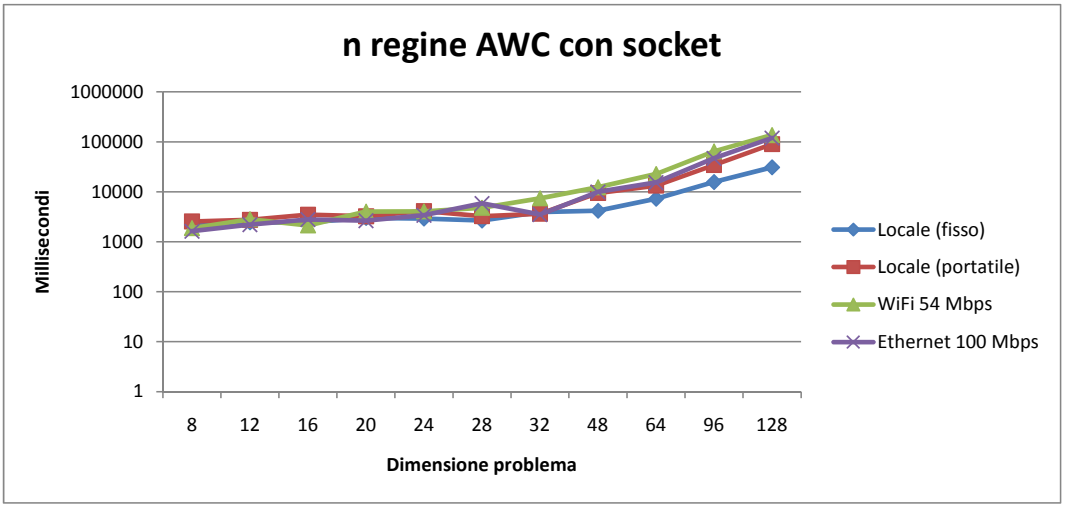

SudokuAWCAgent
#findNewValue()
#findConstraintForbiddenValues(LinkedList)
QueensAWCAgent
#findNewValue()
#findConstraintForbiddenValues(LinkedList)
AWCAgent
#init(int, int, int)
+run()
#findForbiddenValues(LinkedList)
#findNewValue()
#findConstraintForbiddenValues(LinkedList)
#searchvalueInAgentView(int)
#searchValueInLowerPriorityAgentView(int)
Agent
#init(int, int, int)
+run()
#send(Message, int)
#receive()
ABTAgent
#init(int, int, int)
+run()
#findForbiddenValues(LinkedList)
#findNewValue()
#findConstraintForbiddenValues(LinkedList)
1
SudokuABTAgent
#findNewValue()
#findConstraintForbiddenValues(LinkedList)
QueensABTAgent
#findNewValue()
#findConstraintForbiddenValues(LinkedList)
1..*
RMIRemoteCommunicator
+connectTo(Vector)
+receive(int)
+send(Message)
+clearAddressBook()
+numberPending()
+tryFlush()
+noMessages()
+close()
+addMessage(Message, InetSocketAddress)
RemoteCommunicator
+send(Message)
+receive(int)
+close()
+numberPending()
SocketRemoteCommunicator
+connectTo(Vector, int)
+receive(int)
+send(Message)
+clearAddressBook()
+numberPending()
+tryFlush()
+noMessages()
+close()
1
MPIRemoteCommunicator
+receive(int)
+send(Message)
+clearAddressBook()
+numberPending()
+tryFlush()
+close()
1
RMIServer
+transfer(Object, InetSocketAddress)

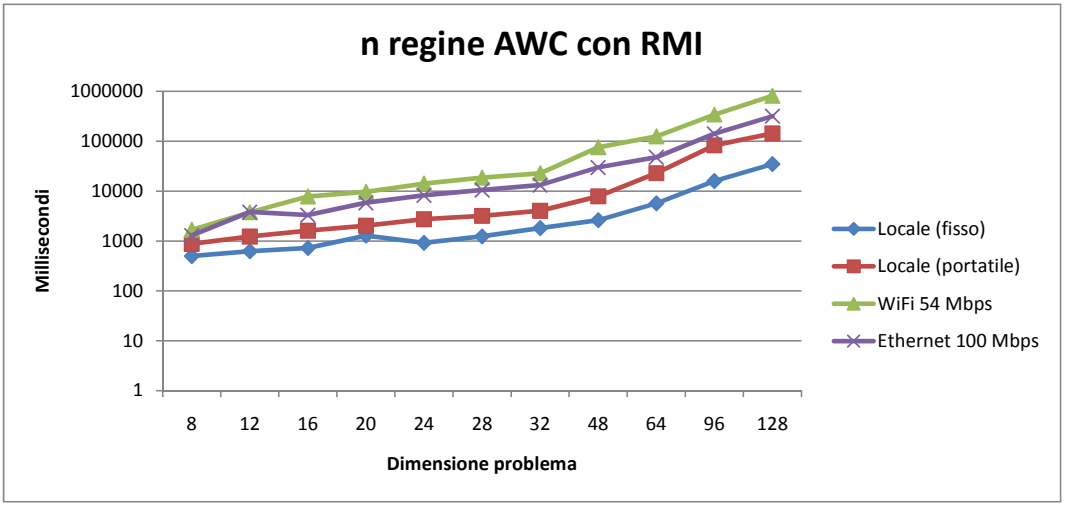
Capitolo 4

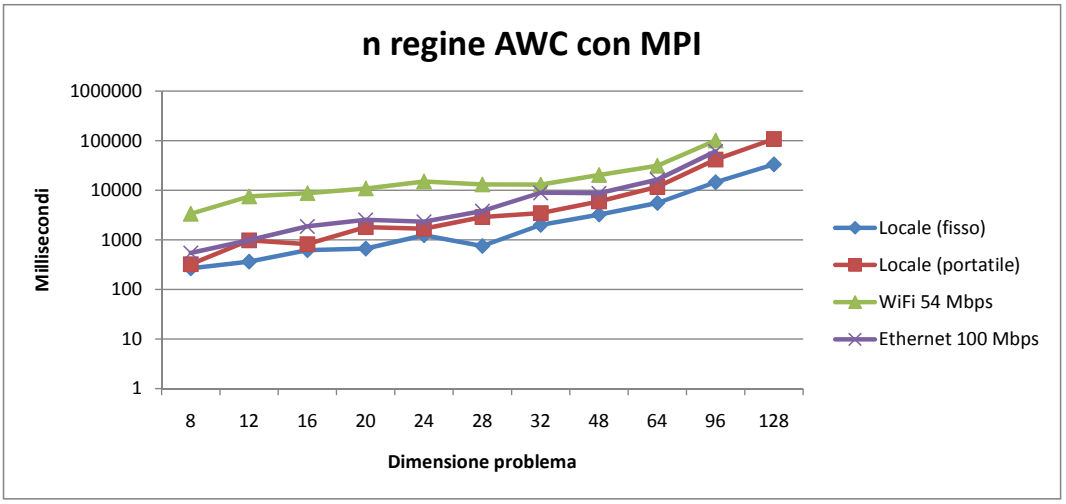
Analisi delle Prestazioni

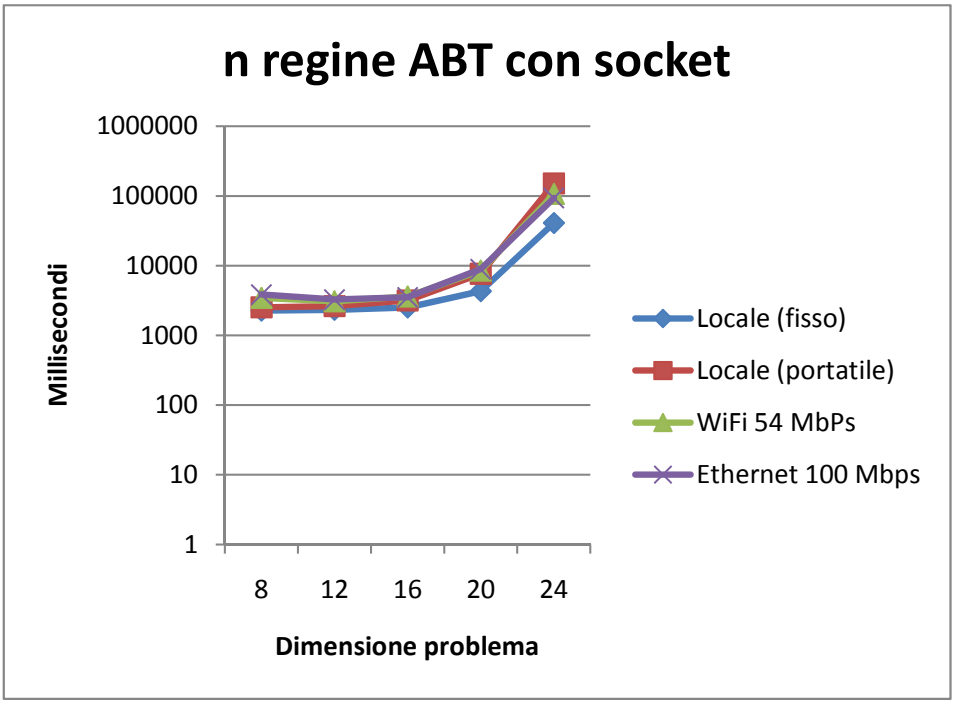
A questo punto, con due algoritmi di risoluzione (asynchronous backtracking search ed asynchronous weak-commitment search), due problemi di test (sudoku e n regine) e tre meccanismi di comunicazione di rete (socket, MPI ed RMI), si è proceduto ad effettuare una serie di test per verificare le prestazioni del sistema.

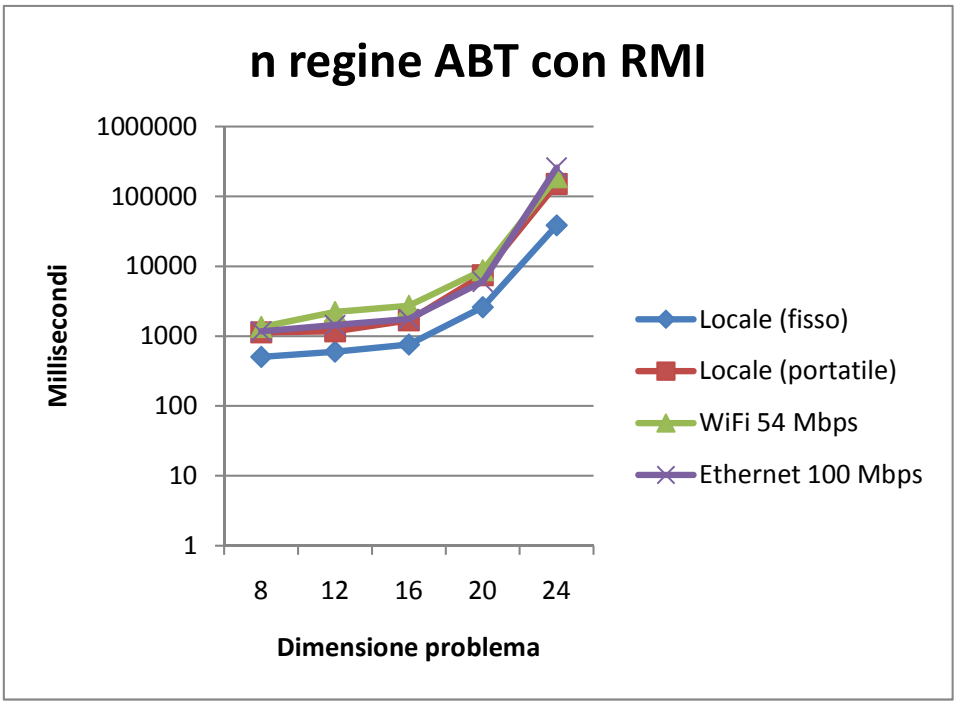
Per effettuare i test sono stati usati due computer. un computer portatile Toshiba M60-162 dotato di processore Intel Pentium M 2 GHz, 2 Gb di RAM, disco rigido Western Digital WD2500BEVE da 250 Gb a 5400 rpm, schede di rete Intel PRO/Wireless 2200BG e Realtek RTL8139/810x fast ethernet (questo computer viene identificato nei risultati dei test come "portatile") ed un PC assemblato dotato di processore Intel Core 2 Duo E6850 a 3 GHz, 4 Gb di memoria RAM, 2 dischi rigidi Seagate Barracuda ST3500320AS da 500 Gb e 7200 rpm in RAID 0 ed un disco rigido Maxtor 6V320F0 da 320 Gb e 7200 rpm non in RAID, scheda di rete Marvell-Yukon 88E8056 PCI-E Gigabit Ethernet (questo computer viene identificato nei risultati dei test come "fisso"). Entrambi i computer hanno installato un sistema operativo Windows XP professional con SP3. I due computer comunicano tra di loro attraverso un router wireless Belkin N1 Vision. Nei test che utilizzano la connessione wireless, il computer portatile è stato posto a circa due metri dal router, mentre per i test con il collegamento ethernet cablato si è provveduto a disattivare la scheda di rete wireless del portatile ed a collegarlo con un cavo di rete ad

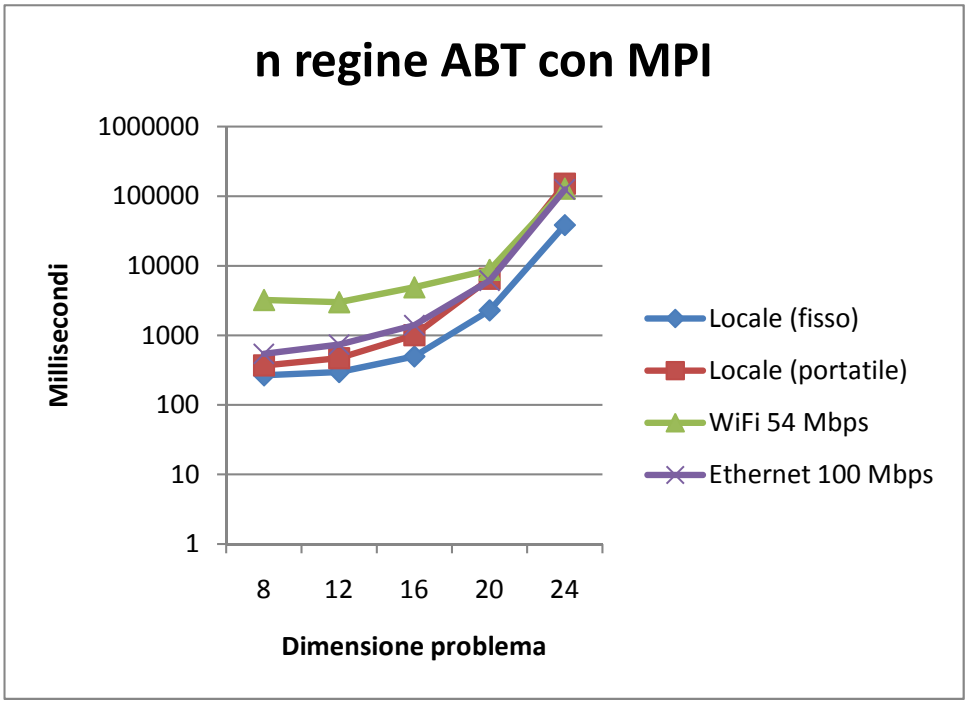

n regine AWC con socket
Millisecondi
1000000
100000
10000
1000
100
10
1
8
12
16
20
24
28
32
48
64
96
128
Dimensione problema
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps

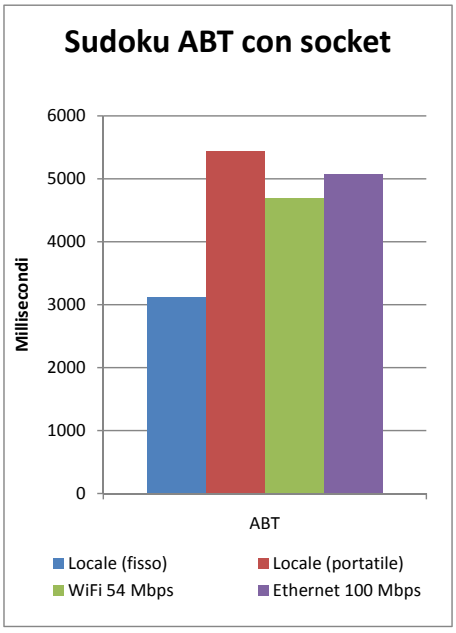

n regine AWC con RMI
Millisecondi
1000000
100000
10000
1000
100
10
1
8
12
16
20
24
28
32
48
64
96
128
Dimensione problema
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps

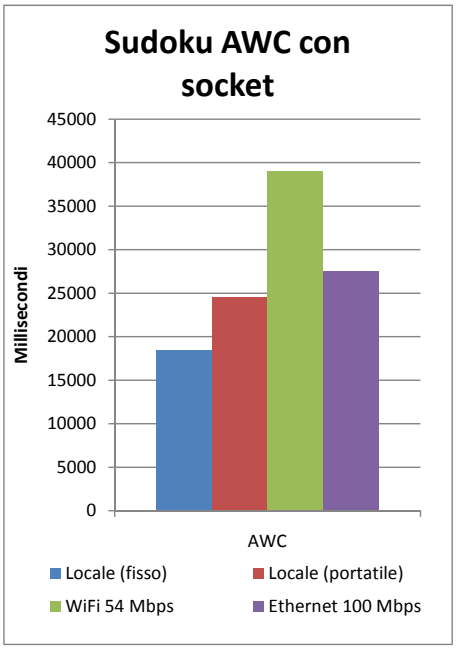

n regine AWC con MPI
Millisecondi
1000000
100000
10000
1000
100
10
1
8
12
16
20
24
28
32
48
64
96
128
Dimensione problema
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps

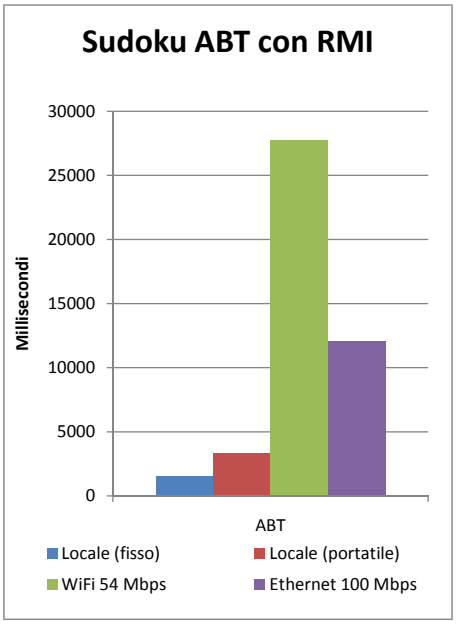

n regine ABT con socket
Millisecondi
1000000
100000
10000
1000
100
10
1
8
12
16
20
24
Dimensione problema
Locale (fisso)
Locale (portatile)
WiFi 54 MbPs
Ethernet 100 Mbps

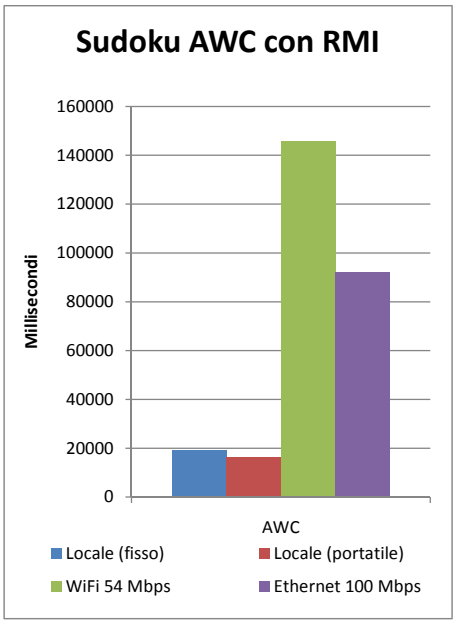

n regine ABT con RMI
Millisecondi
1000000
100000
10000
1000
100
10
1
8
12
16
20
24
Dimensione problema
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps

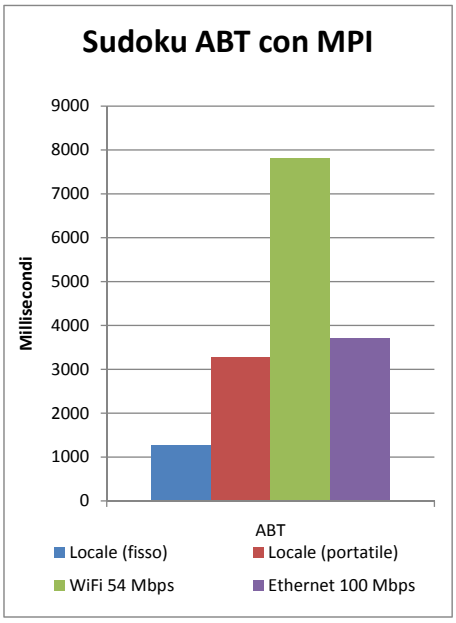

n regine ABT con MPI
Millisecondi
1000000
100000
10000
1000
100
10
1
8
12
16
20
24
Dimensione problema
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps

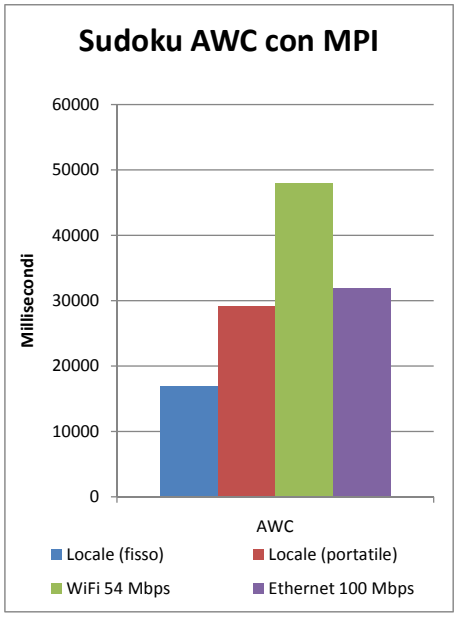

Sudoku ABT con socket
6000
5000
4000
3000
2000
1000
0
Millisecondi
ABT
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps


Sudoku AWC con socket
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
Millisecondi
AWC
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps


Sudoku ABT con RMI
30000
25000
20000
15000
10000
5000
0
Millisecondi
ABT
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps


Sudoku AWC con RMI
160000
140000
120000
100000
80000
60000
40000
20000
0
Millisecondi
AWC
Locale (fisso)
Locale (portatile)
WiFi 54 Mbps
Ethernet 100 Mbps





una delle porte ethernet del router.

I test mostrano chiaramente come l'utilizzo di RMI come metodo di comunicazione non sia assolutamente adatto ad un sistema ad alte prestazioni, in quanto produce un notevole rallentamento del sistema rispetto alle comunicazioni tramite socket o MPI. Altra cosa da notare è che i due algoritmi utilizzati si adattano in maniera molto diversa ai due problemi di test: nel problema delle n regine, l'algoritmo ABT riesce a risolvere rapidamente problemi fino ad un valore di n pari a 20, dopodiché il tempo di esecuzione cresce rapidamente. L'algoritmo AWC, invece, riesce ad arrivare senza problemi fino ad n pari a 128. Tuttavia, nel caso del sudoku, i ruoli si invertono: ABT riesce a risolvere il problema molto più velocemente di AWC. Questo illustra chiaramente come nessuno dei due algoritmi sia sempre più veloce dell'altro, ma come occorra considerare anche il problema a cui l'algoritmo viene applicato per stabilire quale sia più vantaggioso.

Conclusioni

Questo lavoro di tesi si propone di progettare e realizzare uno strumento per la risoluzione distribuita e decentralizzata di problemi di soddisfacimento di vincoli.

Visti gli innumerevoli vantaggi offerti dai linguaggi di programmazione ad alto livello, si è scelto di realizzare il sistema in Java. Questo ha permesso di implementare gli algoritmi per la risoluzione distribuita di CSP illustrati da Yokoo utilizzando un paradigma ad oggetti. Questo permette una migliore organizzazione ed una maggiore facilità di comprensione e modifica del codice. Inoltre la realizzazione di un sistema distribuito permette di avvantaggiarsi delle ultime tendenze nel campo dell'elettronica, che hanno visto l'introduzione di architetture di calcolo parallele nei computer.

Sono stati implementati due algoritmi, l'asynchronous backtracking search e l'asynchronous weak-commitment search. L'asynchronous backtracking search è una versione distribuita dell'algoritmo di backtracking standard, in cui ogni variabile del problema viene gestita da un esecutore (chiamato agente) che ne modifica il valore e che comunica con gli altri agenti tramite messaggi. L'asynchronous weak-commitment search è un algoritmo in cui, a differenza dell'asynchronous backtracking, l'ordine delle variabili può cambiare dinamicamente, cosa che permette dei miglioramenti di prestazioni rispetto all'algoritmo di backtracking.

Sono stati poi implementati dei sistemi per lo scambio di messaggi che permettessero di far comunicare i vari agenti che costituiscono il sistema sia in locale sia attraverso la rete, permettendo di scegliere tra alcuni metodi di comunicazione di rete (comunicazione tramite socket, tramite RMI o tramite MPI).

Infine il sistema è stato testato su due tipi di problemi di test: il problema del sudoku e il problema delle n regine. I test hanno permesso di verificare le differenze nel comportamento dei due algoritmi e dei vari sistemi di comunicazione in tutte le combinazioni possibili. Questi test hanno mostrato come, nonostante un calo delle prestazioni verificatosi passando da un esecuzione in locale ad una distribuita su due macchine, il sistema

prometta di scalare rapidamente con l'aggiunta di ulteriori nodi di calcolo. Questa scalabilità viene verificata a livello teorico nell'appendice di analisi dei risultati dei test di questa tesi.

Sviluppi possibili

Durante la progettazione e la realizzazione del sistema descritto in questo lavoro di tesi, sono emersi dei possibili sviluppi e dei problemi che non è stato possibile trattare per mancanza di tempo.

Rappresentazione simbolica dei vincoli

Il primo di questi sviluppi del sistema è un metodo per permettere di esprimere in maniera esplicita i vincoli di un problema. Il sistema attuale, infatti, richiede che per ogni problema che deve risolvere si crei un nuovo oggetto Java che si occupa di controllare i vincoli specifici del problema. Questo significa che i vincoli del problema sono inglobati all'interno del codice di tali oggetti.

Il possibile miglioramento di questo aspetto prevederebbe di creare un oggetto generico in grado di controllare i vincoli di un qualsiasi problema che gli siano forniti in un formato opportuno. Una volta creato questo oggetto, quindi, per adattare il sistema a risolvere un nuovo problema, basterebbe scrivere una opportuna formalizzazione dei vincoli di tale problema e fornire tale informazione all'oggetto generico, senza dover scrivere del codice ad hoc.

Addirittura, sarebbe possibile creare un piccolo compilatore che, prendendo in ingresso le espressioni dei vincoli del problema, produca in uscita la descrizione dei vincoli necessaria per il funzionamento dell'oggetto descritto prima. In questo modo, per utilizzare il sistema, basterebbe indicare il numero delle variabili, il dominio di ciascuna e scrivere le espressioni dei vincoli per poter iniziare l'esecuzione.

Rimozione della ridondanza dall'elenco dei Nogood ricevuti

Questo è già stato descritto in precedenza: in entrambi gli algoritmi distribuiti descritti in questo lavoro, occorre mantenere un elenco dei Nogood ricevuti da ogni agente. Quando si riceve un nuovo Nogood, è possibile che questo sia una generalizzazione di uno o più Nogood già ricevuti in precedenza, oppure che il nuovo Nogood sia una specializzazione

di un Nogood già ricevuto in precedenza. Questo significa che è possibile rimuovere vari Nogood ricevuti che sono ridondanti.

Durante i test, eseguendo questa operazione di rimozione della ridondanza ogni volta che veniva ricevuto un nuovo Nogood, si è notato come l'introduzione di ulteriori controlli da eseguire ogni volta che si riceve un nuovo Nogood annulli ogni beneficio derivante dall'avere una lista di Nogood di dimensioni più ridotte. Tuttavia potrebbe essere possibile eseguire questa operazione solo saltuariamente, per esempio quando l'agente è in attesa di messaggi.

Lo pseudocodice per questa operazione di rimozione della ridondanza è presente nel capitolo 2.2.4.

Rilevamento della terminazione

Durante la fase di implementazione, si è reso necessario creare un sistema di rilevamento della terminazione con successo. Il sistema descritto nel testo sembra funzionare correttamente dai test effettuati, ma occorrerebbe o verificarne formalmente la correttezza, oppure sostituirlo con un algoritmo distribuito per il rilevamento della terminazione. La descrizione di un tale algoritmo è presente in [CL85].

Massimo numero di Nogood memorizzati

Nel caso dell'algoritmo Asynchronous weak-commitment search si è detto che spesso non è necessario poter memorizzare un numero illimitato di Nogood già inviati. Questo è necessario per la completezza teorica dell'algoritmo, ma, nella pratica, anche solo con una decina di Nogood salvati l'algoritmo non presenta problemi di terminazione.

Per cercare di sfruttare la migliore gestione della memoria derivante dalla memorizzazione di un minor numero di Nogood senza dover rinunciare alla completezza teorica dell'algoritmo, si può adottare la tecnica descritta nel capitolo 2.2.4: inizialmente il numero massimo di Nogood generati memorizzabili è impostato ad una valore fisso e limitato (per esempio 10). Se dopo un certo intervallo di tempo non si è ancora raggiunta la terminazione, il numero di Nogood memorizzabili aumenta e si azzera il contatore che tiene traccia del tempo trascorso. In questo modo il numero massimo di Nogood memorizzabili aumenta progressivamente.

Algoritmo di serializzazione ad hoc

Come spiegato nell'appendice, ridurre la dimensione dei dati da spedire in rete ha un effetto benefico sulle prestazioni. L'algoritmo di serializzazione standard di Java produce un output che ha dimensioni più grandi di quanto strettamente necessario. Scrivendo un algoritmo di serializzazione ad hoc, sarebbe possibile ridurre le dimensioni dei messaggi serializzati, migliorando così le prestazioni del sistema.

Multicast di messaggi

Anche questo descritto nell'appendice: quando si devono inviare dei messaggi tutti uguali a molto destinatari, si può ridurre il traffico di rete usando un protocollo multicast. Si inviano in rete dei messaggi contenenti un elenco di destinatari, invece che un solo destinatario. In questo modo, al costo di un solo invio di messaggi in rete, si possono raggiungere più destinatari.

Metodo di connessione tramite socket

Nel capitolo 3 si è descritto come si debba provvedere ad aprire un solo canale di comunicazione bidirezionale tra ogni coppia di istanze del sistema. Il metodo attualmente utilizzato è quello di usare l'ordinamento lessicografico degli indirizzi e delle porte di ogni istanza per evitare di aprire doppie connessioni, ma questo presenta dei problemi legati alla gestione degli indirizzi IP.

Un possibile metodo per risolvere il problema sarebbe quello di procedere inizialmente ad una fase di apertura di tutte le connessioni verso gli altri indirizzi. Questo porterebbe, come detto, alla creazione di doppie connessioni, che verrebbero gestite nella fase seguente: ogni istanza del programma invierebbe un messaggio di esplorazione su ogni singolo socket aperto, ciascun messaggio marcato con un identificatore unico per l'istanza mittente. Ogni volta che un'istanza riceve uno di questi messaggi, essa risponde con un messaggio di risposta su tutti i suoi socket aperti tranne quello da cui è arrivata la richiesta.

Quando un'istanza riceve una risposta ad un messaggio inviato da un'altra instanza, la risposta ricevuta viene scartata. Viceversa, se si tratta della risposta ad un proprio messaggio, tale risposta arriverà tramite uno dei due socket aperti tra la coppia di istanze (mentre la richiesta iniziale era stata inviata sull'altro socket). In questo modo l'agente mittente della richiesta sa che una particolare richiesta era stata inviata su un certo socket

e che la relativa risposta è arrivata da un altro socket. In questo modo ogni istanza del programma può raggruppare tutti i propri socket aperti in una serie di coppie di socket che condicono alla stessa destinazione.

Quindi occorre stabilire, per ogni coppia di istanze, quale sia tra le due quella che ha il diritto di chiudere uno dei due canali aperti. Per effettuare tale scelta, è possibile utilizzare il minimo degli identificatori degli agenti contenuti in ciascuna istanza, stabilendo, per esempio, che l'istanza con l'identificatore minimo minore ha il diritto di scegliere quale canale di comunicazione chiudere.

Infine, l'istanza scelta decide quale canale chiudere (in maniera casuale o in base a qualche criterio) e lo comunica all'altra istanza. Quindi si chiude il canale scelto. Quando tutte le istanze hanno fatto questo, tra ogni coppia di istanze rimarrà un solo canale di comunicazione bidirezionale attivo.

Appendice A

Stima del Costo della Comunicazione

Osservando i risultati dei test si nota che, usando due macchine distinte, le prestazioni del sistema spesso calano rispetto ad usare una macchina singola. Questo fatto è causato dal grande numero di messaggi da scambiare sulla rete e dall'overhead che tale scambio comporta, senza che la maggiore suddivisione del sistema sia compensata dalla maggiore potenza di calcolo disponibile.

A.1 Stime dei costi di scambio dei messaggi

Supponiamo, infatti, di suddividere n agenti in m gruppi, in modo tale che il numero medio di agenti in ogni gruppo sia p. Supponiamo ora che il numero di messaggi scambiati tra ogni coppia di agenti sia circa lo stesso e, per semplicità, supponiamo anche che gli agenti inviino messaggi anche a se stessi.

Ogni gruppo di agenti tratta solo due categorie di messaggi: quelli locali (mittente e destinatario fanno entrambi parte dello stesso gruppo di agenti) e quelli non locali (mittente e destinatario fanno parte di gruppi di agenti diversi). Se il problema viene suddiviso tra soli due gruppi di agenti (indicati con A e B), ci sono tre categorie di messaggi scambiate in questo sistema: i messaggi locali del gruppo A, i messaggi locali del gruppo B e i messaggi non locali. Ogni gruppo, in questa situazione, deve considerare i suoi messaggi locali e tutti quelli non locali. Indichiamo con M_A ed M_B gli insiemi dei messaggi locali di ogni gruppo, ed indichiamo con M_{AB} l'insieme dei messaggi non locali. È evidente come ogni gruppo debba considerare tutti i suoi messaggi locali e tutti quelli non locali. Il gruppo di agenti A, per esempio, dovrà considerare tutti i messaggi M_A e tutti i messaggi M_{AB}.

Avendo ipotizzato che il numero di messaggi scambiati tra ogni coppia di agenti sia

circa uguale, si ha che il numero di messaggi scambiati localmente all'interno di un gruppo è circa p^2 volte il numero di messaggi scambiati tra una coppia di agenti, mentre il numero di messaggi scambiati tra una coppia di gruppi di agenti è circa $2 \cdot p^2$ volte il numero di messaggi scambiati tra una coppia di agenti. Infatti, se all'interno di un gruppo ogni agente invia un messaggio ad ogni agente del gruppo stesso (incluso se stesso), il numero totale di messaggi locali è dato dal numero di agenti che inviano messaggi moltiplicato per il numero di messaggi inviato da ciascuno. Dato che il numero di messaggi inviati da ciascuno è pari al numero di agenti presenti nel gruppo, il numero totale di messaggi locali inviati è dato da p^2 volte il numero di messaggi scambiato tra ogni coppia di agenti. Invece, considerando due gruppi di p agenti ciascuno, supponiamo che ogni agente di ciascun gruppo invii un messaggio ad ogni agente dell'altro gruppo. Anche qui il numero totale di messaggi inviati è dato dal numero di agenti che inviano messaggi ($2 \cdot p$, dato che sono coinvolti due gruppi di agenti), moltiplicato per il numero di messaggi inviati da ciascuno (p, il numero di agenti nell'altro gruppo). Il numero di messaggi è dunque dato da $2 \cdot p^2$ volte il numero di messaggi scambiati tra ogni coppia di agenti, ma in questo caso ogni messaggio remoto deve essere o solo inviato o solo ricevuto da ogni gruppo di agenti, mentre i messaggi locali di un gruppo devono essere sia inviati sia ricevuti da degli agenti di quel gruppo. Per esprimere questo fatto nel costo di scambio dei messaggi, è possibile considerare che il numero di messaggi remoti sia p^2 invece che $2 \cdot p^2$, in modo da tenere in considerazione che tali messaggi devono essere elaborati solo per metà da ogni gruppo di agenti.

La frazione di messaggi che devono essere considerati dal gruppo A è dunque data da:

$$\frac{\text{messaggi da considerare}}{\text{numero totale di messaggi}} = \frac{p^2 + p^2}{2 \cdot p^2 + 2 \cdot p^2} = \frac{2 \cdot p^2}{4 \cdot p^2} = \frac{1}{2}$$

Infatti, i messaggi locali del gruppo B non vengono minimamente considerati, mentre i messaggi remoti vengono considerati solo per metà.

Supponiamo ora di suddividere il problema tra quattro gruppi i agenti (denominati A, B, C e D). Anche qui si distingue tra messaggi locali ad ogni gruppo e messaggi non locali tra ogni coppia di gruppi. Ogni gruppo deve trattare i suoi messaggi locali e tutti i messaggi non locali che lo interessano (dato che in questo caso i messaggi non locali non interessano tutti i gruppi).

Indichiamo gli insiemi dei messaggi locali di ogni gruppo con M_A, M_B, M_C, ed M_D;

indichiamo gli insiemi dei messaggi non locali tra ogni coppia i gruppi con M_{AB}, M_{AC}, M_{AD}, M_{BC}, M_{BD} ed M_{CD}.

Anche qui consideriamo il gruppo di agenti A: esso dovrà considerare gli insiemi di messaggi M_A, M_{AB}, M_{AC} ed M_{AD}. La frazione di messaggi che esso dovrà considerare è dunque:

$$\frac{\text{messaggi da considerare}}{\text{numero totale di messaggi}} = \frac{p^2 + 3 \cdot p^2}{4 \cdot p^2 + 6 \cdot 2 \cdot p^2} = \frac{4 \cdot p^2}{16 \cdot p^2} = \frac{1}{4}$$

Generalizzando questa considerazione, se il problema ha n agenti suddivisi in m gruppi (con $1 \leq m \leq n$, ovviamente) che hanno in media p agenti ciascuno (cioè $p = \frac{n}{m}$ circa, e $p > 0$), allora ci sono m insiemi di messaggi locali e $\binom{m}{2} = \frac{m!}{(m-2)! \cdot 2} = \frac{m \cdot (m-1)}{2}$ insiemi di messaggi non locali. Ogni gruppo di agenti considererà l'insieme dei suoi messaggi locali e $m-1$ insiemi di messaggi non locali. La frazione di messaggi considerati da ogni gruppo di agenti quando il sistema è suddiviso in m gruppi è dunque:

$$\frac{p^2 + (m-1) \cdot p^2}{m \cdot p^2 + \frac{m \cdot (m-1)}{2} \cdot 2 \cdot p^2} = \frac{(m - 1 + 1) \cdot p^2}{m \cdot p^2 + (m^2 - m) \cdot p^2} = \frac{m \cdot p^2}{(m^2 - m + m) \cdot p^2} = \frac{m}{m^2} =$$

$$= \frac{1}{m} \tag{A.1}$$

Di conseguenza, la frazione di messaggi considerati da ogni gruppo di agenti diminuisce sempre più man mano che si suddivide il problema tra più gruppi.

Ogni gruppo di agenti considera m insiemi di messaggi: un insieme per i suoi messaggi locali ed $m - 1$ insiemi per i messaggi scambiati con gli altri $m - 1$ gruppi di agenti. Volendo quantificare la parte del lavoro di scambio dei messaggi da parte di ciascun gruppo di agenti che viene indirizzata verso lo scambio dei messaggi locali e quella che viene indirizzata verso lo scambio dei messaggi remoti, la frazione del costo dei messaggi locali di ogni gruppo è dunque:

$$\frac{1}{m} \tag{A.2}$$

e la frazione del costo dei messaggi remoti è:

$$\frac{m-1}{m} \tag{A.3}$$

Questo deriva immediatamente se supponiamo che un singolo agente nel sistema invii un messaggio a tutti gli altri agenti (incluso se stesso): tale agente invierà n messaggi, di cui $\frac{n}{m}$ avranno dei destinatari locali ed $\frac{n\cdot(m-1)}{m}$ avranno destinatari non locali.

Si può dunque affermare che (indicando con C_l il costo medio di scambio di un messaggio locale, con C_r il costo medio di scambio di un messaggio remoto ed assumendo che $0 < C_l \leq C_r$):

$$\begin{gathered}\text{costo medio di scambio di un messaggio} = \\ = (\text{frazione di messaggi scambiati in locale} \cdot C_l) + \\ + (\text{frazione di messaggi scambiati in remoto} \cdot C_r)\end{gathered}$$

Indicando con "costo di scambio di un messaggio" il tempo che passa dal momento in cui un agente inizia ad eseguire le operazioni per inviare il messaggio a quando tale messaggio è inserito nella coda di messaggi del destinatario. In altre parole, il costo di scambio di un messaggio è la latenza dell'operazione di scambio del messaggio. Dall'equazione sopra riportata si ha che, distribuendo gli agenti tra più macchine fisiche collegate in rete, il costo di scambio dei messaggi da parte di ogni gruppo diventa:

$$\begin{gathered}\text{costo di scambio dei messaggi per un gruppo di agenti} = \\ = \text{numero totale di messaggi scambiati nel sistema} \cdot \\ \cdot \text{frazione di messaggi da considerare} \cdot \\ \cdot \text{costo medio di scambio di un messaggio} =\end{gathered}$$

(indicando con G il numero totale di messaggi scambiati nel sistema e sostituendo l'equazione (A.1))

$$\begin{gathered}= G \cdot \frac{1}{m} \cdot ((\text{frazione di messaggi scambiati in locale} \cdot C_l) + \\ + (\text{frazione di messaggi scambiati in remoto} \cdot C_r)) =\end{gathered}$$

(Sostituendo (A.2) e (A.3))

$$\begin{gathered}= G \cdot \frac{1}{m} \cdot \left(\left(\frac{1}{m} \cdot C_l \right) + \left(\frac{m-1}{m} \cdot C_r \right) \right) = G \cdot \frac{1}{m} \cdot \frac{C_l + (m-1) \cdot C_r}{m} = \\ = G \cdot \frac{C_l + m \cdot C_r - C_r}{m^2} = G \cdot \left(\frac{C_l}{m^2} + \frac{C_r}{m} - \frac{C_r}{m^2} \right) \qquad \text{(A.4)}\end{gathered}$$

Quindi:

$$\lim_{m \to +\infty} G \cdot \left(\frac{C_l}{m^2} + \frac{C_r}{m} - \frac{C_r}{m^2} \right) = 0$$

Questo significa che, aumentando il numero di nodi in cui si suddivide il sistema, se il numero totale di messaggi non aumenta, il costo di scambio dei messaggi per ogni gruppo diminuisce.

Verifichiamo ora quanto occorre distribuire un sistema di agenti per fare in modo che il costo di scambio dei messaggi per ogni gruppo di agenti sia minore del costo di scambio dei messaggi per l'intero sistema eseguito in locale:

costo di scambio dei messaggi per un gruppo di agenti <

< costo di scambio dei messaggi per l'intero sistema eseguito in locale

costo di scambio dei messaggi per un gruppo di agenti <

< numero totale di messaggi scambiati nel sistema·

·frazione dei messaggi scambiati in locale·

·costo medio di invio di un messaggio in locale

(Sostituendo (A.4) e considerando che, eseguendo il sistema in locale, la frazione di messaggi scambiati in locale è 1 si ha)

$$G \cdot \left(\frac{C_l}{m^2} + \frac{C_r}{m} - \frac{C_r}{m^2} \right) < G \cdot 1 \cdot C_l$$

$$\frac{C_l}{m^2} + \frac{C_r}{m} - \frac{C_r}{m^2} - C_l < 0$$

$$C_l + C_r \cdot m - C_r - C_l \cdot m^2 < 0$$

$$C_l \cdot m^2 - C_r \cdot m + C_r - C_l > 0$$

Si risolve l'equazione associata:

$$m = \frac{C_r \pm \sqrt{C_r^2 - 4 \cdot C_l \cdot (C_r - C_l)}}{2 \cdot C_l} = \frac{C_r \pm \sqrt{C_r^2 - 4 \cdot C_r \cdot C_l + 4 \cdot C_l^2}}{2 \cdot C_l} =$$

$$= \frac{C_r \pm \sqrt{(C_r - 2 \cdot C_l)^2}}{2 \cdot C_l} = \frac{C_r \pm |C_r - 2 \cdot C_l|}{2 \cdot C_l}$$

Supponiamo che $C_r - 2 \cdot C_l \geq 0$, cioè $C_r \geq 2 \cdot C_l$:

$$m = \frac{C_r \pm (C_r - 2 \cdot C_l)}{2 \cdot C_l}$$

Le due radici sono:

$$m_1 = \frac{C_r - C_r + 2 \cdot C_l}{2 \cdot C_l} = \frac{2 \cdot C_l}{2 \cdot C_l} = 1$$
$$m_2 = \frac{C_r + C_r - 2 \cdot C_l}{2 \cdot C_l} = \frac{2 \cdot (C_r - C_l)}{2 \cdot C_l} = \frac{C_r - C_l}{C_l}$$

Dato che abbiamo supposto $C_r \geq 2 \cdot C_l$, si ha che $\frac{C_r - C_l}{C_l} \geq 1$, quindi $m_1 \leq m_2$. La disequazione, dunque, è soddisfatta per $m < m_1 \vee m > m_2$, cioè per $m < 1 \vee m > \frac{C_r - C_l}{C_l}$. Dato però che m è il numero di gruppi di agenti del sistema, m non può essere minore di 1. Dunque le uniche soluzioni accettabili sono $m > \frac{C_r - C_l}{C_l}$.

Supponiamo ora $C_r - 2 \cdot C_l < 0$, cioè $C_r < 2 \cdot C_l$:

$$m = \frac{C_r \pm (2 \cdot C_l - C_r)}{2 \cdot C_l}$$

Le due radici sono:

$$m_1 = \frac{C_r - 2 \cdot C_l + C_r}{2 \cdot C_l} = \frac{2 \cdot C_r - 2 \cdot C_l}{2 \cdot C_l} = \frac{2 \cdot (C_r - C_l)}{2 \cdot C_l} = \frac{C_r - C_l}{C_l}$$
$$m_2 = \frac{C_r + 2 \cdot C_l - C_r}{2 \cdot C_l} = \frac{2 \cdot C_l}{2 \cdot C_l} = 1$$

Sapendo che $C_r < 2 \cdot C_l$, si ha che $m_1 < m_2$. Quindi le soluzioni sono date da $m < m_1 \vee m > m_2$, cioè $m < \frac{C_r - C_l}{C_l} \vee m > 1$. Per lo stesso motivo detto sopra, però, l'unica soluzione accettabile è $m > 1$.

Unendo i due casi, si ha che la disequazione è soddisfatta per:

$$m > \max\left(1, \frac{C_r - C_l}{C_l}\right) \tag{A.5}$$

Inoltre, ponendo:

$$C_r = k \cdot C_l$$

in cui k è un reale positivo e maggiore od uguale ad 1, dato che $C_r \geq C_l$, si ha che deve essere:

$$\frac{C_r - C_l}{C_l} = \frac{k \cdot C_l - C_l}{C_l} = \frac{C_l \cdot (k-1)}{C_l} = k - 1$$

Quindi, sostituendo:

$$m > \max(1, k-1) \tag{A.6}$$

Questa disequazione mostra come, più l'invio dei messaggi remoti diventa lento rispetto all'invio dei messaggi locali, più occorre distribuire gli agenti per compensare tale maggiore inefficienza. In certi casi si può addirittura verificare:

$$k - 1 > n$$

Questo significa che, dovendo essere $m \leq n$, non è possibile suddividere il sistema a tal punto da essere sicuri di ottenere un'esecuzione più veloce rispetto all'esecuzione in locale.

A.2 Altre considerazioni

Occorre però puntualizzare su due ulteriori aspetti: il costo medio di scambio dei messaggi remoti e l'effetto della suddivisione sulla potenza di calcolo disponibile. Nei calcoli precedenti, infatti, si è supposto che il costo medio di scambio di un messaggio remoto fosse costante. Questo non è del tutto vero: quando il traffico sulla rete cresce, il costo associato allo scambio di ogni messaggio cresce anch'esso a causa della congestione della rete. Già con due gruppi di agenti si ha che circa metà dei messaggi viaggia sulla rete: nel caso del sistema diviso in due gruppi di agenti A e B, infatti, gli insiemi dei messaggi scambiati in locale da ciascun gruppo contengono ciascuno p^2 volte il numero di messaggi scambiati tra ogni coppia di agenti. L'insieme dei messaggi remoti, invece, contiene $2 \cdot p^2$ volte il numero di messaggi scambiati tra ogni coppia di agenti (nei calcoli precedenti ci interessava il costo dello scambio dei messaggi remoti, che si era stimato in p^2 dato che i messaggi remoti devono essere o solo inviati o solo ricevuti da ogni gruppo di agenti; adesso, invece, ci interessa il numero totale di messaggi remoti scambiati tra ogni coppia di gruppi di agenti). Con quattro gruppi di agenti la frazione sale ai $\frac{3}{4}$. Nel caso generale, la frazione dei messaggi non locali è data da:

$$\frac{\text{numero messaggi scambiati in remoto}}{\text{numero messaggi totali}} =$$

$$= \frac{2 \cdot \text{numero di insiemi di messaggi remoti}}{2 \cdot \text{numero di insiemi di messaggi remoti} + \text{numero di insiemi di messaggi locali}} =$$

$$= \frac{2 \cdot \binom{m}{2} \cdot p^2}{2 \cdot \binom{m}{2} \cdot p^2 + m \cdot p^2} = \frac{\frac{m!}{2 \cdot (m-2)!} \cdot 2 \cdot p^2}{\frac{m!}{2 \cdot (m-2)!} \cdot 2 \cdot p^2 + m \cdot p^2} = \frac{\frac{m \cdot (m-1)}{2} \cdot 2 \cdot p^2}{\frac{m \cdot (m-1)}{2} \cdot 2 \cdot p^2 + m \cdot p^2} =$$

$$= \frac{m \cdot (m-1) \cdot p^2}{m \cdot (m-1) \cdot p^2 + m \cdot p^2} = \frac{m \cdot (m-1) \cdot p^2}{m \cdot ((m-1) \cdot p^2 + p^2)} = \frac{m \cdot (m-1) \cdot p^2}{m^2 \cdot p^2} =$$

$$= \frac{m \cdot (m-1)}{m^2} = \frac{m-1}{m}$$

Questo significa che, aumentando la suddivisione del sistema, il quantitativo di messaggi da scambiare in rete crescerà, ma sarà sempre minore del doppio del quantitativo di messaggi scambiati in rete suddividendo il sistema in due gruppi di agenti:

$$\frac{m-1}{m} < 2 \cdot \frac{2-1}{2} = 2 \cdot \frac{1}{2} = 1$$

Questo incremento della quantità dei messaggi scambiati in rete può far sì che il numero di gruppi di agenti in cui si deve suddividere il sistema per ottenere prestazioni migliori rispetto all'esecuzione in locale sia in realtà superiore a quanto indicato dai calcoli precedenti.

Tuttavia i calcoli non tengono in considerazione un altro aspetto: la maggiore potenza di calcolo disponibile suddividendo maggiormente il sistema. Suddividendo il problema tra più macchine, infatti, si ha a disposizione una maggiore potenza di calcolo per ogni agente. Questo, ovviamente, permette di eseguire più rapidamente le operazioni di elaborazione in risposta alla ricezione di un messaggio, cosa che può compensare la maggiore quantità di messaggi inviati in rete suddividendo il sistema tra più macchine. Questa considerazione, in realtà, non coinvolge solo la potenza di calcolo pura disponibile: ogni macchina, eseguendo un minor numero di agenti, utilizza una minore quantità di memoria. Questo permette di tenere una proporzione maggiore dei dati degli agenti in cache, cosa che può avere un ulteriore effetto benefico sulle prestazioni del sistema. È noto, infatti, che l'evoluzione tecnologica ha permesso di aumentare la velocità di elaborazione dei microprocessori molto più della velocità delle memorie. Quindi, fare in modo di migliorare l'uso della memoria può avere un impatto molto importante sulle prestazioni.

Inoltre, tutti i calcoli fatti in precedenza assumono che la quantità di messaggi scambiati tra ogni coppia di gruppi di agenti e all'interno di ogni gruppo di agenti sia circa la stessa. Se però è possibile suddividere gli agenti in modo che la proporzione di messaggi remoti sia molto inferiore a quella ipotizzata, diventa possibile ottenere benefici dalla

suddivisione del sistema anche con un numero di macchine molto inferiore. Questo è visibile nei test del problema delle n regine con l'algoritmo ABT: durante questi test, il portatile ha eseguito quasi tutto il lavoro di elaborazione, inviando solo saltuariamente qualche messaggio in rete. Il risultato è che l'esecuzione in rete è stata più veloce di quella in locale sul portatile. Questo grazie al fatto che, al costo di qualche messaggio inviato e ricevuto sulla rete, il gruppo di agenti sul portatile si è ridotto di dimensioni in memoria ed ha avuto a disposizione una quantità di potenza di elaborazione leggermente superiore, non dovendo suddividere tale potenza con gli agenti dell'altro gruppo.

Appendice B

Un semplice algoritmo per la risoluzione di CSP

Questa appendice presenta un semplice algoritmo distribuito e decentralizzato per risolvere i CSP. Tale algoritmo non vuole essere un metodo pratico per risolvere i CSP, ma vuole fornire un metodo semplice per dimostrare la correttezza di altri algoritmi più complessi per la risoluzione di CSP. Dimostrando, infatti, che un certo algoritmo rispetta la descrizione dell'algoritmo presentato qui di seguito, si avrà automaticamente che tale algoritmo è corretto.

B.1 Descrizione dell'algoritmo

Ogni agente gestisce una variabile. Quando il valore di una variabile cambia, l'agente che gestisce tale variabile invia dei messaggi Ok per comunicare il cambiamento. Ogni agente usa i messaggi Ok ricevuti per creare una propria agent view. Ogni volta che la agent view cambia, l'agente controlla i vincoli. Se tutti i vincoli che interessano la sua variabile sono rispettati, il valore della variabile non cambia. Altrimenti l'agente sceglie se mantenere lo stesso valore per la propria variabile o se assegnargliene uno nuovo (entrambe queste possibilità vengono scelte con probabilità maggiore di 0). Se si sceglie un nuovo valore per la variabile, la probabilità che si scelga un valore che fa parte di una soluzione del problema deve essere maggiore di 0.

B.2 Dimostrazione di correttezza

Proposizione B.1. *Se il CSP ha almeno una soluzione, l'algoritmo sopra indicato riesce a trovare una soluzione del problema dopo un tempo finito, anche se illimitato*

Dimostrazione. Dimostrazione: supponiamo che tutti gli agenti siano in uno stato tale che l'insieme dei valori assegnati alle rispettive variabili costituisce una soluzione al problema. Se tutti i messaggi Ok sono stati ricevuti ed elaborati, nessun agente cambierà più il proprio stato, e l'algoritmo termina. Viceversa, se rimangono ancora dei messaggi Ok da ricevere, potrebbe essere che uno o più agenti cerchino di cambiare il valore delle loro variabili, che in realtà erano corretti. Dato che esiste una probabilità maggiore di 0 che il valore di una variabile rimanga quello vecchio, esiste una probabilità maggiore di 0 che il valore di tutte le variabili che potrebbero cambiare il loro valore rimanga inalterato. Quindi c'è una possibilità maggiore di 0 che nessuna variabile cambi il proprio valore nel tempo necessario ad attendere la ricezione di tutti i messaggi Ok ancora in transito. Se nessuna variabile cambia il proprio valore finchè l'ultimo messaggio non viene ricevuto, si ricade nel caso precedente e l'algoritmo termina.

Supponiamo ora che l'insieme dei valori assegnati alle singole variabili non sia una soluzione del problema. Esiste dunque almeno una variabile che, prima o poi (potrebbe essere necessario attendere la ricezione di uno o più messaggi Ok), si accorgerà che uno dei suoi vincoli è violato. Quando questo avviene, tale variabile cambierà il proprio valore. Se la variabile non modifica il proprio valore, non cambia niente e la variabile tenterà ancora di cambiare il proprio valore. Altrimenti, se il nuovo valore è diverso da quello vecchio, l'insieme dei valori delle variabili del problema cambia. Quindi, se l'assegnamento di valori alle variabili del problema non è una soluzione, prima o poi almeno una variabile cambierà il proprio valore.

Dato che i nuovi valori hanno una probabilità maggiore di 0 di essere parte di una soluzione del problema, esiste una probabilità maggiore di 0 che tutti i valori scelti per le variabili modificate siano parte di una soluzione del problema. Se tale situazione non si verifica (cioè se una o più variabili assumono un valore che non è parte di una soluzione del problema), si ritorna al caso precedente. Altrimenti possono verificarsi due situazioni: l'insieme degli assegnamenti di valori a tutte le variabili del problema è una soluzione del problema (e quindi si ricade nel caso descritto all'inizio), oppure l'insieme degli assegnamenti di tutte le variabili non è una soluzione al problema.

In quest'ultimo caso, invece, si ha ancora che una o più variabili cercheranno di cambiare il proprio valore, anche qui con probabilità maggiore di 0 di scegliere dei valori che sono parte di una soluzione del problema. Si ripete lo stesso ragionamento fatto sopra e si arriva a concludere che esiste una probabilità maggiore di 0 che, ad ogni passo, tutte le variabili che cercano di cambiare valore scelgano dei valori che sono parte di una soluzione del problema. Di conseguenza esiste una probabilità maggiore di 0 che, ad un certo punto, tutti i valori assegnati alle variabili del problema costituiscano una soluzione al problema.

Dato che abbiamo dimostrato sopra che, se l'insieme dei valori assegnati a tutte le variabili del problema costituisce una soluzione, esiste una probabilità maggiore di 0 che l'algoritmo raggiunga lo stato di terminazione e dato che abbiamo dimostrato che esiste una probabilità maggiore di 0 che le variabili raggiungano tale stato con l'assegnamento dei rispettivi valori, si ha che esiste una probabilità maggiore di 0 che l'algoritmo raggiunga lo stato di terminazione un un certo periodo di tempo.

Tuttavia questo significa che, in un determinato lasso di tempo, o l'algoritmo termina, oppure esso continua ad elaborare. Se consideriamo un intervallo di tempo più lungo, la probabilità che l'algoritmo termini cresce. Di conseguenza si ha che la probabilità

che l'algoritmo non termini in un certo lasso di tempo decresce quanto più lungo è il lasso di tempo considerato. Di conseguenza, la probabilità che l'algoritmo non termini mai è 0, pur non essendo possibile trovare un limite massimo alla durata dell'esecuzione dell'algoritmo. □

Bibliografia

[Bar05] R. Barták. Constraint propagation and backtracking-based search. In *First International Summer School on Constraint Programming*, September 11-15 2005.

[BvB98] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *National Conference on Artificial Intelligence*, Madison, Wisconsin, 1998.

[CL85] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[RN02] S. Russel and P. Norvig. *Artificial Intelligence: a Modern Approach.* Prentice Hall, 2002.

[YH00] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2), 2000.

[YI98] M. Yokoo and T. Ishida. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE transactions on knowledge and data engineering*, 10(5), 1998.

[Yok95] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. *International Conference on Principles and Practice of Constraint Programming*, 1995.