UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE
MATEMATICHE FISICHE E NATURALI
Corso di Laurea in Informatica

Tesi di Laurea Triennale

Un sistema web di prenotazione basato sulla programmazione a vincoli

Candidato:
Menozzi Luca

Relatore:
Dott. Federico Bergenti

Anno accademico 2008/2009

Ringraziamenti

Per prima cosa desidero ringraziare la mia famiglia, Loretta, Pier Luigi e Stefano che mi sono sempre stati vicini e mi hanno supportato durante questi lunghi anni di studio.
Un particolare ringraziamento ai compagni di studio e amici di Parma, con i quali ho condiviso questo interessante percorso: Kietto, Marco, Vigno e Silvia.
Un grazie ai miei amici Frigno, Titta, Gian e a tutti gli altri i quali mi hanno sempre aiutato e dimostrato fiducia.
Un grandissimo ringraziamento a Raoni che mi ha consigliato e sostenuto durante quest'ultimo anno.

Infine un ringraziamento al professor Bergenti Federico per la disponibilitá e per i consigli che ha saputo dispensare durante questi ultimi mesi.

Grazie a tutti.

Indice

1 Introduzione **1**

2 Problemi di soddisfacimento vincoli **3**

2.1 Definizione 3

2.2 Soluzione 3

2.3 Esempi 4

2.3.1 Problema delle *n*-regine 4

2.3.2 Problema del map-coloring 5

2.4 Constraint satisfaction problem e applicazioni reali 6

2.5 Principali algoritmi 7

3 Struttura della libreria JCL **21**

3.1 Descrizione 21

3.2 Vincoli implementati dalla libreria JCL 21

3.3 Algoritmi della libreria JCL 23

3.4 Esempio d'uso della libreria JCL 25

3.5 Modifiche alla libreria JCL 25

3.5.1 Creare nuovi domini 30

4 Sistema **33**

4.1 Descrizione del sistema 33

4.2 Analisi del dominio 33

4.2.1 Attori 33

4.2.2 Entità 34

4.2.3 Use Cases 34

4.3 Progettazione base di dati 39

4.3.1 Requisiti: 39

4.3.2 Schema logico 40
4.4 Schema entità relazione 41
4.5 Applicazione e constraint satisfaction problem 43
4.6 Domini e variabili 43
4.7 Vincoli 44
4.7.1 Vincoli binari essenziali 44
4.7.2 Vincoli binari non essenziali 47
4.7.3 Vincoli unari 47
4.8 Test e validazione 50
4.9 Applicazione 53

Conclusioni **57**

Elenco delle figure

2.1 Una delle 92 soluzioni del problema delle 8-regine. 4
2.2 Le regioni dell'Australia. 5
2.3 Constraint-graph per l'esempio della colorazione. 6
2.4 Mappa per problema del map coloring. 13
2.5 Parte dell'albero di ricerca relativo al Forward Checking. 14

4.1 Schema entità-relazione 42
4.2 Grafo rappresentante i vincoli 49
4.3 Grafico che rappresenta i test con accesso al database. 51
4.4 Grafico che rappresenta i test senza accesso al database. 51
4.5 Grafico che confronta l'algoritmo di arc consistency con e senza accesso al database. 52
4.6 Grafico che confronta l'algoritmo di forward checking with dynamic variable ordering con e senza accesso al database. 52
4.7 Form di registrazione. 54
4.8 Form di registrazione con notifica errori nell'inserimento dei dati. . 54
4.9 Form di ricerca. 55
4.10 Risultato ricerca, conferma prenotazione. 55
4.11 Prenotazione effettuata con successo. 56
4.12 Pagina di riepilogo prenotazioni eseguite. 56
4.13 Pagina di cancellazione prenotazione. 56

Elenco delle tabelle

4.1 Use case Registrazione. 35
4.2 Use case Log-in. 36
4.3 Use case ricerca prestazione. 37
4.4 Use case prenotazione prestazione. 38
4.5 Use case cancellazione prestazione. 39
4.6 Nome, tipo e descrizione delle variabili che rappresentano il dominio. 45
4.7 Vincoli binari Goodlist essenziali con breve descrizione. 46
4.9 Vincoli unari. 47
4.8 Vincoli binari GoodList non essenziali. 47

Elenco dei Listati

2.1 Algoritmo Generate and Test 7
2.2 Algoritmo di Backtracking 8
2.3 Algoritmo di Backjumping 10
2.4 Algoritmo di Backmarking 11
2.5 Algoritmo di Forward Checking con Arc Consistency 12
2.6 Algoritmo di Node Consistency 15
2.7 Funzione revise per algoritmi di Arc Consistency 16
2.8 Algoritmo di Arc Consistency (AC-1) 17
2.9 Algoritmo di Arc Consistency (AC-3) 17
2.10 Algoritmo di Path Consistency 19
3.1 Listato che risolve il problema del map coloring 26
3.2 Risultato dell'esecuzione del programma del map coloring 27
3.3 Funzione notifySolution() appartenente alla classe MySolutionManagerExtend 29

Capitolo 1

Introduzione

Lo scopo del seguente lavoro è quello di costruire un applicazione web che permetta ad un cittadino la prenotazione di una prestazione santiaria. Attualmente per eseguire la prenotazione di un prestazione è possibile seguire una delle seguenti tre vie:

- Prenotazione tramite CUP,
- Prenotazione telefonica tramite il numero verde 848 800 640,
- Prenotazione tramite farmacia convenzionata.

Per avere un idea più dettagliata del problema affrontato, possiamo pensare ad un utente che si reca ad uno sportello cup di una qualsiasi AUSL, questo presenterà un impegnative del medico dove è specificato quale prestazione l'utente dovrà effettuare. L'operatore allo sportello legge l'impegnativa e propone una data, un ora e un presidio ospedaliero dove effetturare la prestazione. Se l'utente è d'accordo accetta ed effettua una prenotazione altrimenti l'operatore varierà le proposte di data/ora/presidio ospedaliero in base alle sue esigenze.
Possiamo pensare che la procedura per la prenotazione di una prestazione tramite farmacia o mediante telefonata al numero verde, si molto simile alla procedura presentata sopra.
Sebbene alcuni di questi metodi di prenotazione, come la prenotazione telefonica, siano molto comodi per il cittadino, la regione Emilia-Romagna ha ritenuto di affiancare a questi anche la prenotazione web. Da questa esigenza nasce quasto lavoro.
Le realizzazione di questo prototipo per la prenotazione delle prestazioni sanitarie, è passata innanzi tutto dalla realizzazione di una prima bozza di applicazione web. Questa permetteva all'utente la registrazione, la visualizzazione

delle proprie prenotazione e la cancellazione delle stesse. Fatto ciò si è passati alla realizzazione del sistema di prenotazione delle prestazioni, abbiamo cercato di modellare questo problema utilizzando i problemi di soddifacimento vincoli. Per la loro struttura i problemi di soddisfacimento vincoli si prestano molto bene alla risoluzione di queste tipologie di applicazioni, tuttavia sono poche le effettive realizzazioni applicate al mondo web. Come linuguaggio di programmazione si è deciso di utilizzare Java, data la sua versatilità e la presenza di librerie adattate per la risoluzione dei problemi di soddsfacimento vincoli; in particolare la teconologia java server pages, che con il suo paradigma mvc si presta molto bene alla realizzazione di interfaccie web-oriented. Il tutto è stato pensato cercando si soddisfare i migliori requisiti di usabilitá.

Capitolo 2

Problemi di soddisfacimento vincoli

2.1 Definizione

Introduciamo per primo il concetto di vincoli; Consideriamo una sequenza finita di variabili $Y := y_1, ..., y_k$ con $k > 0$, associate ai loro rispettivi domini $D_1, ..., D_k$. In questo modo la generica varaibile y_i può assumere ogni valore del dominio D_i. Un vincolo C su Y è un sottoinsieme del prodotto cartesiano $D_1 \times D_2 \times ... \times D_k$. Se $k = 1$ chiameremo il vincolo unario, se $k = 2$ vincolo binario. Per constraint satisfaction problem (CSP), intendiamo una sequenza finita di variabili $X := x_1, ..., x_n$ con i rispettivi domini $D_1, ..., D_n$, con un insieme finito $\mathcal{C}$ di vincoli. Possiamo scrivere un CSP come $\langle \mathcal{C}; D\varepsilon \rangle$, dove $D\varepsilon := x_1 \in D_1, ..., x_n \in D_n$.

2.2 Soluzione

Intuitivamente possiamo intendere come soluzione di un CSP un assegamento alle variabili che soddisfi tutti i vincoli. Più formalmente una n-upla $(d_1, ...d_n) \in D_1 \times ... \times D_n$ soddisfa un vincolo $C \in \mathcal{C}$ sulle variabili $x_{i_1}, ..., x_{i_m}$ se $(d_{i_1}, ..., d_{i_m}) \in C$. Una n-upla $(d_1, ..., d_n) \in D_1 \times ... \times D_n$ è soluzione di $\langle \mathcal{C}; D\varepsilon \rangle$ se essa soddifa ogni vincolo $C \in \mathcal{C}$. Se un CSP ha soluzione diremo che è consistente, inconsistente altrimenti.

2.3 Esempi

Di seguito alcuni famosi esempi di CSP.

2.3.1 Problema delle *n*-regine

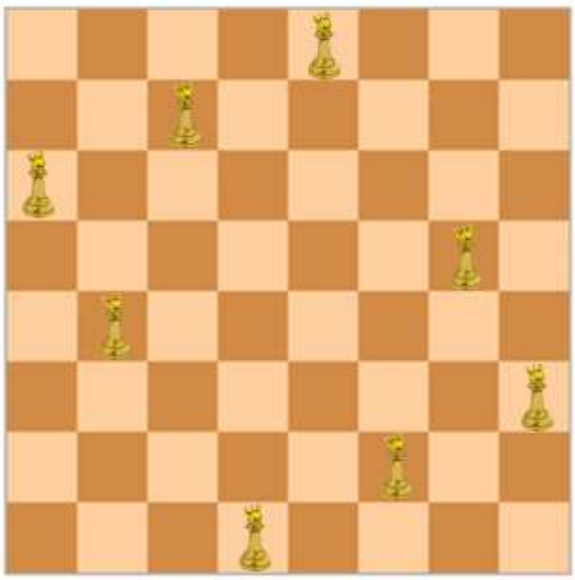

Figura 2.1: Una delle 92 soluzioni del problema delle 8-regine.

Il problema richiede di piazzare n-regine su una scacchiera di dimensione nxn, dove $n \geq 4$, senza che esse si minaccino a vicenda. Le regine hanno le stesse proprietà del gioco degli scacchi, cioè in una mossa possono spostarsi in un numero arbitrario di caselle in verticale, orizzontale e diagonale. Il problema dunque è quello di posizionare le regine in caselle sicure cioè non minacciate da altre regine. Intendiamo sicura una casella che non può essere raggiunta da una regina in una sola mossa.
Una possibile rappresentazione per questo problema è quella di usare n varaibili, $x_1, ...x_n$, ognuna con dominio$[1, ...n]$. L'idea è quella di denotare con x_1 la posizione della regina piazzata nella i-esima posizione della scacchiera. La soluzione presentata in figura 2.1 corrisponde alla sequenza di valori (6,4,7,1,8,2,5,3), la prima regina sulla sinistra è piazzata nella sesta riga contando dal basso, la seconda nella quarta riga e così via. I vincoli possono essere formulati come segue, per $i \in [1, ..., n-1]$ e $j \in [i+1, .., n]$:

- $x_i \neq x_j$,

- $x_i - x_j \neq i - j$,

- $x_i - x_j \neq j - i$.

2.3.2 Problema del map-coloring

Il problema consiste nel colorare le regioni di una mappa con numero limitato di colori, in modo che non vi siano due regioni adiacenti (per adiacenti intendiamo due regioni aventi un segmento in comune) che hanno lo stesso colore. È dimostrato che sono sufficenti quattro colori per colorare qualunque mappa in modo che nessuna coppia di regioni adiacenti abbia lo stesso colore. Come esempio possiamo considerare la mappa dell'Australia in Figura 2.2.

Figura 2.2: Le regioni dell'Australia.

Una rappresentazione del map-coloring può essere illustrata mediante un grafico, i nodi rappresentano le variabili mentre ogni arco rappresenta un vincolo binario. Come variabili usiamo (WA,NT,SA,Q,NSW,V,T) con dominio uguale per ogni varaibile (rosso,verde,blu). Un possibile assegnamento dei vincoli è dato da: $WA \neq NT \wedge WA \neq SA \wedge NT \neq SA \wedge NT \neq Q \wedge SA \neq Q \wedge SA \neq NSW \wedge SA \neq V \wedge Q \neq NSW \wedge NSW \neq V$; come in Figura 2.3. Una possibile soluzione per il problema è data da: $WA = rosso, NT = verde, Q = rosso, NSW = verde, V = rosso, SA = blu, T = verde$.

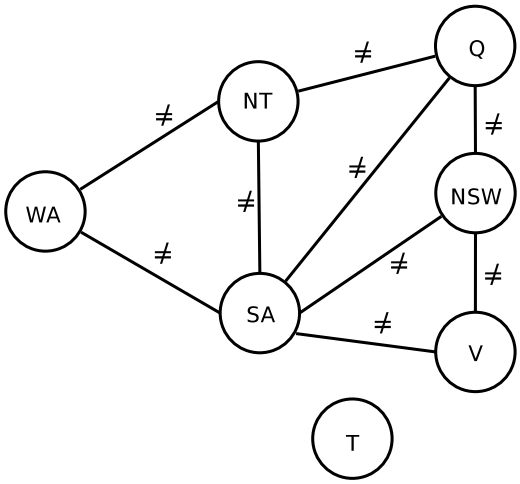

Figura 2.3: Constraint-graph per l'esempio della colorazione.

2.4 Constraint satisfaction problem e applicazioni reali

Nella realtà i problemi di soddisfacimento vincoli sono impiegati in innumerevoli campi, di seguito un elenco che non pretende di essere esaustivo:

- Problemi di allocazione risorse:
  - Distribuzione delle merci nei magazzini,
  - Pianificazione delle attività del personale, gestione dei turni e dei carichi di lavoro.
- Problemi di scheduling:
  - Pianificazione della produzione.
- Gestione e configurazione di reti di telecomunicazioni:
  - Pianificazione della stesura dei cavi,
  - Posizionamento di access point in reti wi-fi.
- Riconoscimneto di immagini e focalizzazione di attenzione in sistemi di visione.
- Progettazione di circuiti digitali.
- Realizzazione di assistenti intelligenti per applicazioni web.

2.5 Principali algoritmi

Breve panoramica sui principali algoritmi utilizzati per risolvere un CSP. Ogni algoritmo fà uso di un albero decisionale, ad ogni livello assegnamo una variabile e ad ogni nodo un valore corrispondente al dominio della variabile. I primi algoritmi descritti vengono chiamati di backtracking, questi algoritmi generalmente valutano le variabili in un determinato ordine, ad esempio $(x_1, ..., x_n)$.

Generate and Test: Questo algoritmo dapprima genera tutti valori per le variabili, dopo di che visita l'albero cercando i valori compatibili con i vincoli. Nel caso non trovi una soluzione continua la ricerca in backtracking risalendo l'albero fino a che non trova un nuovo nodo con strade inesplorate. Come si può facilmente notare è un algoritmo molto dispendioso in quanto generiamo tutte le combinazioni di variabili possibili per poi cercare una soluzione. Un esempio di questo algoritmo si trova nel listato 2.1

Listato 2.1: Algoritmo Generate and Test

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

Standard Backtracking: Lo standard backtracking, il cui algoritmo è presente nel listato 2.2, si basa sulla verifica dei vincoli a posteriori. La consistenza dei vincoli viene controllata a ogni assegnamento delle variabili, il controllo di consistenza viene effettuato su soluzioni parziali. Illustriamo un esempio con due variabili X_1, X_2. Subito viene istanziato il valore della variabile X_1 poi quello di X_2 se abbiamo una violazione dei vincoli l'algoritmo ritratta la

Listato 2.2: Algoritmo di Backtracking

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

procedure BT-1(Unlabelled, Labelled, Constraints)
  if Unlabelled = {} then return Labelled
  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 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 for
  return TRUE
end consistent
```

scelta fatta e assegna un nuovo valore alla variabile X_2, se l'assegnamento è compatibile con i vincoli viene mantenuto, altrimenti si procede con un nuovo assegamento. Se un assegnamento valido non viene trovato, risaliamo l'albero fino a che non troviamo un nuovo nodo con strade inesplorate. Lo standard backtracking ha prestazioni migliori rispetto al generate and test.

Backjumping: A differenza dello standard backtracking che risale sempre di un livello nell'albero di ricerca quando tutti i valori per una variabile sono testati, l'algoritmo di backjumping può risalire anche più di un livello, il metodo quindi cerca di minimizzare il numero di nodi visitati nell'albero di ricerca. In pratica l'algoritmo di backjumping torna indietro fino a una variabile che ha causato un conflitto; l'insieme di queste varibili è chiamato insieme dei

conflitti. Possiamo definire l'insieme dei confilitti per una variabile X come l'insieme delle variabili precedentemente assegnate che hanno vincoli con X. L'algoritmo 'salta indietro' fino alla variabile più recente nell'insieme dei confilitti. Troviamo l'algoritmo nel listato 2.3

Backmarking: Questo algoritmo minimizza l'esecuzione di controlli ridondanti sui vincoli congiuntamente ad un backtracking. Un controllo ridondante sui vincoli testa, nuovamente, se l'insieme di assegnamenti per le variabili soddisfa un vincolo quando siamo già a conoscenza del risultato del controllo.
Per prima cosa l'algoritmo evita di esguire controlli sui vincoli che sicuramente falliscono: se un assegnamento è precedentemente fallito nei conforonti di una varibile la cui istanziazione non è cambiata, lo stesso assegnamento fallirà ancora. Secondariamente un assegnamento che è riuscito su un insieme di variabili la cui istanziazione non è cambiata deve essere ancora consistente sugli stessi assegnamenti. Da notare che confronto al backtracking, l'algoritmo di backmarking non riduce lo spazio di ricerca, riduce solamente il numero di controlli di consistenza sui vincoli. L'algoritmo è presentato nel listato 2.4

Passiamo ora alla descrizione degli algoritmi di propagazione. L'idea alla base di questi algoritmi consiste nell'uso attivo dei vincoli che consente la riduzione a priori dell'albero di ricerca restringendo i domini delle varaibili nel corso della computazione.

Forward Checking: Il forward checking dopo ogni assegnamento ad una variabile, elimina dai domini delle variabili non ancora istanziate i valori non compatibili con i vincoli. Abbiamo così un maggiore carico computazionale ai primi livelli dell'albero per poi semplificare il carico computazionale ai livelli successivi. Il forward checking risulta molto efficace quando le ultime variabili ancora libere sono associate a un insieme di valori ammissibili ridotto, risultano perciò molto vincolate e facilmente assegnabili. Un ulteriore vantaggio consiste nel fatto che se ad un certo punto ci accorgiamo che un domino associato ad una variabile è vuoto si ha un fallimento senza proseguire in inutili tentativi e quindi backtracking come si comportano gli algoritmi precendenti.
Il forward checking utilizza i vincoli non solo sulle variabili già istanziate, ma anche su quelle ancora libere limitiando i valori che queste ultime possono assumere sulla base delle istanziazioni che, di volta in volta, vengono

Listato 2.3: Algoritmo di Backjumping

```
procedure BJ(Variables, Constraints)
  BJ-1(Variables,{},Constraints,0)
end BJ
procedure BJ-1(Unlabelled, Labelled, Constraints, PreviousLevel)
  if Unlabelled = then return Labelled
  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 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 in C & X/V/L in Labelled &
             L < Level}}
       end if
  end for
  if NoConflict then return TRUE
  else return FAIL(J)
end consistent
```

Listato 2.4: Algoritmo di Backmarking

```
procedure BM(Variables, Constraints)
  INITIALIZE(Variables)
  BM-1(Variables,{},Constraints,1)
end BM
procedure INITIALIZE(Variables)
  for each X in Variables do
    BackTo(X) ← 1
    for each V from D_x do
       Mark(X,V) ← 1
    end for
  end for
end INITIALIZE
procedure BM-1(Unlabelled, Labelled, Constraints, Level)
  if Unlabelled ={} then return Labelled
  pick first X from Unlabelled
  for each value V from D_x do
    if Mark(V,X) ≥ BackTo(X) then
       if consistent(X/V, Labelled, Constraints, Level) then
         R ← BM-1(Unlabelled−{X},
           Labelled+{X/V/Level}, Constraints, Level+1)
         if R ≠ FAIL then return R
       end if
    end if
  end for
  BackTo(X) ← Level−1
  for each Y in Unlabelled do
    BackTo(Y) ← min {Level−1, BackTo(Y)}
  end for
  return FAIL
end BM-1
procedure consistent(X/V, Labelled, Constraints, Level)
 for each Y/VY/LY in Labelled such that LY ≥BackTo(X) do
    if X/V is not compatible with Y/VY using Constraints
       then
       Mark(X,V)≤ LY
       return FAIL
    end if
 end for
 Mark(X,V)≤ Level−1
 return TRUE
end consistent
```

attuate.

Questo algoritmo deriva il suo nome dal fatto che ogni volta che assegnamo un valore ad una variabile i domini delle altre variabili vengono ridotti, questo fà si che i vincoli agiscano in avanti (forward), limitando lo spazio delle soluzioni prima che vengano effettuati tentativi su di esse. L'algoritmo è presentato nel listato 2.5 congiuntamente all'algoritmo di arc consistency.

Listato 2.5: Algoritmo di Forward Checking con Arc Consistency

```
procedure AC-FC(cv)
  Q ← {(V_i, V_cv) in arcs(G), i > cv};
  consistent ← TRUE;
  while not Q empty & consistent do
    select and delete any arc (V_k, V_m) from Q;
    //Funzione revise vedi il listato 2.7
    if REVISE(V_k, V_m) then
      consistent ← not empty D_k
    end if
  end while
  return consistent
end AC-FC
```

Partial Look Ahead: Il partial look ahead a differenza del forward checking permette la propagazione dei vincoli anche tra le variabili non ancora istanziate. In un primo momento l'algoritmo si comporta come il forward checking, ad ogni inizializzazione viene controllata la compatibilità dei vincoli con le variabili non ancora inizializzate e in più viene effettuato il look ahead, che controlla nei domini associati alle variabili ancora libere, l'esistenza di valori compatibili con i vincoli. In questo modo i domini delle variabili vengono ridotti durante la computazione, propagando anche relazioni contenenti coppie non ancora istanziate.

I controlli effettuati dal partial look ahead risultano essere più completi rispetto al forward checking in quanto verifichiamo che i futuri assegnamenti risultino compatibili con i vincoli. La verifica dei valori ammessi viene effettuata tra una variabile e quella successiva, supponendo quindi un ordinamento (lessicografico per stringhe, crescente per interi).

Applicheremo questo algoritmo all'esempio del map coloring in figura 2.4,

per renderne chiara la differenza rispetto all'algoritmo di forward checking. In figura 2.5 vi è una parte dell'albero di ricerca relativo al forward checking a cui ci riferiremo nel prossimo esempio. Dopo l'istanziazione delle variabili $X1$ al valore r e $X2$ al valore g, il partial look ahead esegue un forward checking lasciando nel dominio di $X3$ il solo valore b e nel dominio di $X4$ i valori b e r. A questo punto della computazione il forward checking si ferma a differenza del partial look ahead che controlla per ogni possibile valore della variabile $X3$, se esiste almeno un valore nella variabile $X4$ (r) compatibile con i vincoli. Se questo valore esiste il valore nel dominio di $X3$ non viene rimosso, altrimenti, il valore per la variabile $X3$ viene cancellato dal dominio. In questo esempio l'applicazione di questa tecnica non porta ad ulteriori semplificazioni dei domini, in quanto per ogni valore del dominio della variabile $X3$ esiste almeno un valore nel dominio della variabile $X4$ compatibile. Solitamente, comunque, alcuni valori vengono cancellati, riducendo così lo spazio di ricerca. In genere la riduzione dei domini non giustifica il maggiore carico computazionale dovuto alla propagazione dei vincoli.

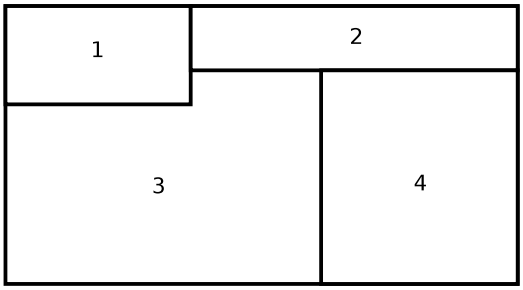

Figura 2.4: Mappa per problema del map coloring.

Full Look Ahead: Questo algoritmo risulta essere molto simili al partial look ahead, si differenza per il fatto che verifica la compatibilità delle successive variabili non ancora inizializzate in entrambe le direzioni.

Negli esempi precedenti, nella scelta dei prossimi valori da assegnare alle variabili abbiamo sempre presupposto di seguire un ordine (lessicografico, crescente) senza preoccuparci di possibili altri miglioramenti nella scelta fatta. Per ovviare a ció ci vengno incontro tecniche euristiche.

Euristiche: Una scelta migliore della variabile successiva da istanziare può essere eseguita applicando l'algoritmo first fail, questo metodo suggerisce di in-

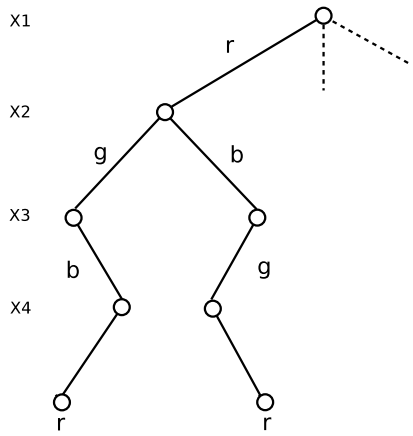

Figura 2.5: Parte dell'albero di ricerca relativo al Forward Checking.

izializzare come successiva variabile quella con il dominio minore, in modo da ridurre il numero di rami dell'albero nei livelli più alti della ricerca. In alternativa questa tecnica propone come altra scelta la variabile più vincolata (most constrining variable), questo porta ad una maggiore riduzione dei domini. Un altro metodo euristico è scegliere il valore che lascia aperto il maggior numero di scelte possibili rispetto alle variabili adiacenti nel constrain graph, questa tecnica viene chiamata least constraining value. Il least constraining value permette di arrivare più in fretta a una soluzione, però diventa inutile nel caso non vi siano soluzioni o nel caso ci servano tutte le soluzioni.

Oltre a queste esistono anche euristiche più complesse, in generale però, il carico computazionale è equivalente se non maggiore rispetto al risparmio di tempo nell'esplorazione dello spazio di ricerca.

Esistono tecniche che permetto di scegliere, oltre alla prossima variabile da istanziare, anche il valore, all'interno del dominio, che ci porta più velocemente ad una soluzione. Una di queste tecniche è least constraint principle che sceglie come primo valore quello che si ritiene abbia più probabilità di successo. Le euristiche si dividono in statiche e dinamiche. Quelle statiche determinano l'ordine delle varibili prima di iniziare la computazione a differenza di quelle dinamiche che svolgono questo compito durante la computazione, ogni volta che è richiesta una nuova selezione. Teoricamente le euristiche dinamiche sono da preferire a quelle statiche, è però da mettere in conto il costo computazionale elevato di queste tecniche, generalmente superiore ad euristiche statiche. Nella realtà si cerca un buon compromesso

Listato 2.6: Algoritmo di Node Consistency

```
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 for
  end for
end NC
```

tra l'algoritmo risolutivo e la tecnica euristica.

Vi sono poi algoritmi chiamati di consistenza che agiscono preventivamente sulla fase di istanziazione. Questi algoritmi modificano la rete di vincoli in modo che questi soddisfino deteminate propietà di consistenza. L'idea è quella di trasformare il CSP in un altro CSP equivalente dove però vengono diminuti i domini delle variabili. Una relazione di equivalenza tra CSP implica che questi hanno uguali soluzioni, di conseguenza se il dominio del nuovo CSP diventa vuoto significa che né il CSP originale né quello modificato hanno soluzione, in egual modo se il CSP modificato ammette soluzioni, queste saranno uguali per entrami i CSP. Questi algoritmi agiscono mediante un grafo (constraint graph) che rappresenta il problema di soddifacimento vincoli.

Node Consistency Un nodo di un grafo di vincoli è node-consitente se ogni vincolo unario sulla variabile x_i è soddisfatto da tutti valori del domino della varaibile. Supponiamo di avere una variabile x_k (rappresentata da un nodo del constrain graph), con dominio $\{1, 2, 3, 4\}$ e un vincolo su x_k dato da $x_k \leq 3$, la propietà di node-consistenza sul grafo non è soddisfatta. Per rendere il grafo node-consistente dobbiamo restringere il dominio di x_k ai valori $\{1, 2, 3\}$. Diremo che un constraint graph è node-consistente se tutti i suoi nodi sono node-consistenti.

Arc Consistency Una variabile appartenente a un problema di soddisfacimento vincoli è arc-consistente con un altra variabile se ognuno dei suoi valori ammissibli è consistente con almeno un valore ammissibili della seconda variabile. Formalmente, una variabile x_i è arc-consistente con la variabile x_j se,

Listato 2.7: Funzione revise per algoritmi di Arc Consistency

```
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, (X,Y) satisfies all the on V_i, V_j
    then delete X from D_i
      DELETED ← TRUE
    end if
  end for
  return DELETED
end REVISE
```

per ogni valore a appartenente al dominio di x_i, esiste un valore b appartenente al domino x_j tale che (a, b) soddisfa il vincolo binario tra x_i e x_j.
Come esempio consideriamo il vincolo $x < y$, il domino delle variabili x e y è $\{1, 2, 3\}$. Siccome x non potrà mai assumere valore 3, questi implica che il valore 3 può essere rimosso dal dominio della variabile x. y, invece, non potrà mai assumere valore 1, quindi questo valore può essere rimosso dal suo dominio. Quindi per rendere arc-consistente il problema dobbiamo rimuovere il valore 3 dal dominio di x e il valore 1 dal dominio di y.
Diremo che un constraint graph è arc-consistente se ogni suo arco è arc-consistente. Bisgona notare inoltre che la rimozione dei valori nel dominio di una variabile potrebbe rendere necessari ulteriori controlli su altri vincoli che coinvolgono questa variabile, questo porta ad un processo iterativo. Questi processi devono essere ripetuti sino a che non raggiungiamo una configurazione stabile.
Vi sono due tipi di arc-consistenza, debole che non prevede che il grafo si node-cosistente e una forte che lo prevede. In seguito quando faremo riferimento all'arc-consistenza intenderemo arc-consistenza di tipo forte. Nei listati 2.7, 2.8, 2.9 sono presenti rispettivamente la funzione *REVISE* e due versioni degli algoritmi di arc-consistency, entrambi fanno uso della funzione *REVISE*. La versione *AC-3* è più performante rispetto alla versione *AC-1*.

Path Consistency Applicheremo questa proprietà a problemi che soddisfano l'arc-consistenza e la node-consistenza, la definizione che daremo si riferisce a cammini di lunghezza arbitraria m. Un cammino tra i nodi (x_i, x_h, x_k) è path

Listato 2.8: Algoritmo di Arc Consistency (AC-1)

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

Listato 2.9: Algoritmo di Arc Consistency (AC-3)

```
procedure AC-3(G)
  Q ← {(V_i, V_j) in 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 union {(V_i, V_k) such that (V_i, V_k) in arcs(G),
          i ≠ k, i ≠ m}
    end if
  end while
end AC-3
```

consistente se per ogni $x \in D_i$ e $y \in D_h$ esiste un valore $z \in D_k$ che soddisfa i vincoli $A(i,k)$ e $A(k,h)$.

Se il grafo è completamente connesso vale il seguente teorema: Se ogni cammino di lunghezza due di un grafo è path-consistente allora l'intera rete è path-consistente.

Rendendo quindi un constraint graph completamente connesso, aggiungendo vincoli completamente rilassati possiamo considerare solo i cammini di lunghezza due e non cammini di lunghezza arbitraria. Un vincolo completamente rilassato tra due variabili X, Y con domini interi può essere dato da una relazione di maggiore uguale e minore, cioè $X \geq Y \wedge X < Y$.

Un versione dell'algoritmo di path consistency è presentata nel listato 2.10.

K-Consistency Dati $k-1$ variabili con valori consistenti con i vincoli del problema, diremo che una k-tupla è k-consistente se esiste un valore per ogni

k-esima variabile che soddisfi i vincoli tra tutte le k variabili.
Per k-consistenza strong intendiamo una rete che è j-consistente per ogni $j \leq k$. Se un grafo con n variabili è n-consistente strong, possiamo trovare una soluzione senza ricerca, infatti i valori delle variabili nei domini fanno parte di una soluzione consistente. Tenendo presente una problema con n variabili, la complessità di questo metodo è esponenziale in n.

Listato 2.10: Algoritmo di Path Consistency

```
procedure PC-2(Vars, Constraints)
  n ← |Vars|
  Q ← {(i,k,j) | 1 ≤ i ≤ j ≤ n & i ≠ k & k ≠ j}
  while Q ≠ {} do
    select and delete any path (i,k,j) from Q
    if REVISE_PATH((i,k,j),Constraints) then
      Q ← Q ∪ RELATED_PATHS((i,k,j))
  end while
end PC-2

procedure REVISE_PATH((i,k,j), C)
  Temp ← C_i,j & (C_i,k * C_k,k * C_k,j)
  if (Temp = Ci,j) then return FALSE
  else
    C_i,j ← Temp
    return TRUE
  end if
end REVISE_PATH

procedure RELATED_PATHS((i,k,j))
  if (i < j) then
    return {(i,j,p) | i ≤ p ≤ n & p ≠ j} ∪
           {(p,i,j) | 1 ≤ p ≤ j & ≠ i} ∪
           {(j,i,p) | j < p ≤ n} ∪
           {(p,j,i) | 1 ≤ p < i}
  else
    return {(p,i,r) | 1 ≤ p ≤ r ≤ n} − {(i,i,i),(k,i,k)}
  end if
end RELATED_PATHS
```

Capitolo 3

Struttura della libreria JCL

3.1 Descrizione

La libreria utilizzata in questo lavoro è JCL (Java Constraint Library) è una libreria Java per la risoluzione di problemi a vincoli. Nella libreria sono contenuti algoritmi che lavorano su problemi di soddifacimento vincoli su domini finiti e continui, tuttavia in questo lavoro si fà riferimento solo a problemi su domini finiti. La parte della libreria utilizzato in questo lavoro è contenuta nel package JCL, questo package implementa buona parte degli algoritmi che riguardano problemi di soddifacimento vincoli su domini finiti.
I domini supportati sono interi, stringhe e tuple. Una tupla o n-upla è un collezione o un elenco ordinato di oggetti; si differenzia da un insieme di n elementi in quanto fra gli elementi di un insieme non è dato alcun ordine, inoltre gli elementi di una tupla possono anche essere ripetuti. Sebbene altri domini non siano previsti, questi si possono creare, come illustrato in sezione 3.5.1. La libreria offre inoltre la posssibilità di lavorare con vincoli di tipo soft.

3.2 Vincoli implementati dalla libreria JCL

Per una definizione del concetto di vincolo rimandiamo alla sezione 2.1.
Tutti i vincoli unari che descriveremo rappresentano delle classi java che estendono l'interfaccia *UnaryConstraint*.

Vincoli unari su interi e stringhe: Di seguiti i vincoli unari supportati dalla libreria su interi e stringhe, in base al tipo di dominio utilizzato, sostituiamo ad *X* i caratteri *ID* o *SD*, rispettivamete per domini interi e domini stringa.

- *UC_X_Equals*,

- *UC_X_NotEquals*,
- *UC_X_GreaterThan*,
- *UC_X_GreatherThanEquals*,
- *UC_X_LessThan*,
- *UC_X_LessThanEquals*.

Vincoli unari per tuple: Di seguiti i vincoli unari supportati dalla libreria per gli elementi di tipo tupla.

- *UC_TD_Equals*,
- *UC_TD_NotEquals*,
- *UC_TD_UnaryConstraint*: Vincolo che ci serve per impostare un vincolo unario su un elemento di una tupla, ad esempio l'elemento in posizione *i*-esima deve essere uguale a 3,
- *UC_TD_BinaryConstraint*: Questo vincolo agisce su due elementi della stessa tupla.

Come per quelli unari, i vincoli binari che elencheremo di seguito rappresentano delle classi java, quasi tutte fanno parte dell'interfaccia *BinaryConstraint*. Altre invece sono un estensione dell'interfaccia *BC_ExplicitConstraint* che a sua volta ha come interfaccia *BinaryConstraint*.

Vincoli binari su interi e stringhe: Di seguito i vincoli binari supportati dalla libreria su interi e stringhe, in base al tipo di dominio utilizzato, sostituiamo ad *X* i caratteri *ID* o *SD*, rispettivamete per domini interi e domini stringa.

- *BC_X_Equals*,
- *BC_X_NotEquals*,
- *BC_X_GreaterThan*,
- *BC_X_GreatherThanEquals*,
- *BC_X_LessThan*,
- *BC_X_LessThanEquals*,
- *BC_X_GoodList*,
- *BC_X_NoGoodList*.

Vincoli binari su tuple: Elenco dei vincoli binari su tuple.

- *BC_TD_Equals*,

- *BC_TD_NotEquals*.

Altri vincoli binari: Elenco di vincoli binari che agiscono anche su tipi diversi, rendendo vera o falsa la loro associazione.

- *BC_GoodList*,
- *BC_NoGoodList*.

3.3 Algoritmi della libreria JCL

Per una descrizione sul funzionamento dei principali algoritmi che compaione nella libreria rimandiamo alla sezione 2.5. La classe *Solver* è un interfaccia, i metodi più importanti della classe, che permettono la risoluzione un problema di soddisfacimento vincoli precedentemente impostato, sono:

- *run():* Risolve il problema.
- *startSolving():* Risolve il problema mediante un nuovo thread, per completezza và segnalato che vi sono anche dei metodi che sospendono, riprendono o fermano l'esecuzione del thread, questi sono *suspendSoving()*, *resumeSolving()*, *stopSolving()*; nel loro uso però vi é da prestare la massima attenzione in quanto usano metodo della classe Thread di Java che sono deprecati in quanto possono causare deadlock o lasciare l'esecuzione in stati inconsistenti.

Entrambi i metodi chiamano il metodo *solve()*, specifico per ogni algoritmo.
Altri metodi importanti della classe sono:

- *setSolutionManager(SolutionManagerInterface manager)*: Permette di impostare un solution manager in modo che ci ritorni le soluzioni trovate, di default la libreria permette di stampare le soluzioni solo sullo standard output.
- *setNumberOfSolutionToFind(int n):* Permette di impostare il numero di soluzioni da trovare, per $n = -1$ il risolutore trova tutte le soluzioni.

Di seguito un elenco dei principali algoritmi della libreria JCL, adibiti alla risoluzione di CSP:

- *BTSolver:* Implementa l'algoritmo simple Backtracking,
- *BJSolver:* implementa l'algoritmo Backjumping,

- *BMSolver:* Implementa l'algoritmo Banckmarking,
- *CBJSolver:* Implementa l'algoritmo Constraint-Directed Backjumping,
- *GBJSolver:* Implementa l'algoritmo Graph-Based Backjumping,
- *FCSolver:* Implementa l'algoritmo Forword Checking,
- *ARCSolver:* Implementa l'algoritmo Simple Arc Consistency.

Spesso questi algoritmi vengono combinati tra loro per sfruttare le diverse potenzialità degli stessi, di seguito un elenco di algoritmi 'ibridi':

- *BM_BJSolver:* Implementa l'algoritmo Banckmarking con Backjumping,
- *BM_CBJSolver:* Implementa l'algoritmo Backmarking con Conflict-Directed Backjumping,
- *BM_GBJSolver:* Implementa l'alogritmo Backmarking con Graph-Based Backjumping,
- *FC_BJSolver:* Implementa l'algoritmo Forward Checking con Backjumping,
- *FC_CBJSolver:* Implementa l'algoritmo Forword Checking con Constraint-Directed Backjumping,
- *FC_GBJSolver:* Implementa l'algoritmo Forword Checking con Graph-Based Backjumping,
- *FC_DVOSolver:* Implementa l'algoritmo Forword Checking con Dynamic Varaible Ordering,
- *FC_ARC_DVOSolver:* Implementa l'algoritmo Forword Checking con Full Arc Consistency e Dynamic Variable Ordering.

Gli ultimi due algoritmi presentati fanno uso anche di una tecnica euristica, in paritcolare usano la tecnica first fail, spiegata in Sezione 2.5.
Un'importante interfaccia della libreria è *SolutionManagerInterface*, dalla quale deriva la classe *MySolutionManager* la cui funzione principale è quella di notificare i risultati dell'elaborazione, da notare che tutte le notifiche vengono effettuata sullo standard output.
I metodi più importanti della classe sono:

- *notifyStart():* Notifica l'inizio della risoluzione.

- *notifySolution():* Ogni volta che viene trovata una soluzione coerente con i vincoli, questa viene notificata.

- *notifyEnd():* Notifica la fine del processo di risoluzione, visulizza anche il numero di soluzioni trovate.

3.4 Esempio d'uso della libreria JCL

Nel listato 3.1 un esempio dell'utilizzo della libreria JCL al problema del map-coloring applicato alla nazione dell'Australia, come in sezione 2.3.
Il listato 3.2 presenta invece il risultato dell'esecuzione del programma.

3.5 Modifiche alla libreria JCL

Nel corso della realizzazione dell'applicazione si sono rese necessarie delle modifiche alla libreria per adattarla alle esigenze dell'applicazione stessa. Una prima limitazione è data dal fatto che una volta impostato il problema, ed eseguito il solver mediante il metodi *run()* o *startSolving()*, questo restituisce tutte le soluzioni trovate; questo può rivelarsi un grosso problema dal lato delle performance. Una seconda limitazione della libreria JCL è quella della stampa dei risultati su standard output, questa caratteristica è pressochè inutile per un applicazione web.
Entrambe le limitazioni vengono superate creando una nuova classa, nominata *MySolutionManagerExtend*, per il primo problema si è optato per terminare la risoluzione al primo risultato utile, lasciando tuttavia all'utente la possibilità di ottenere nuovi risultati, mediante un bottone presente nella pagina di notifica dei risultati; mentre per il secondo problema salveremo il risultato in un oggetto di tipo *booking* e presenteremo questo risultato nella pagina di notifica dei risultati.
Per ospitare tutte le nuove classi della libreria è stato creato un nuovo package, nominato *jclExtends*;

Di seguito un elenco di queste classi con una descrizione delle nuove funzionalità; molte delle classi che presenteremo servono da interfeccia e le modifiche apportate a queste classi sono minime.

- *SolutionManagerInterfeceExtend:* L'interfaccia che ci permette di creare la classe *MysolutionManagerExtend*. Differisce dalla vecchia classe del package JCL per il fatto che il metodo *notifySolution()* accetta, oltre ad un oggetto *Solution* anche un oggetto *SolverExtend*.

Listato 3.1: Listato che risolve il problema del map coloring

```
import JCL.*;
   public class MapColoring {
        public static void main(String[] args) {
           //creo il domino.
           StringDomain colori_d = new StringDomain();
           colori_d.setName("Dominio colori");
           colori_d.addElement("rosso");
           colori_d.addElement("verde");
           colori_d.addElement("blu");
           //Creo le variabili che rappresentano il problema,
           //impostando il medesimo dominio a tutte le varibili.
           CSPVariable WA_v = new CSPVariable(colori_d, "WA");
           CSPVariable NT_v = new CSPVariable(colori_d, "NT");
           ..... //Dichiariamo le stesse variabili anche per
           .....// SA, Q, NSW, V, T.
           //creo un nuovo problema csp.
           CSP csp_problem = new CSP();
           //aggiungo le variabili al problema csp.
           csp_problem.addVariable(WA_v);
           csp_problem.addVariable(NT_v);
           .....//Aggiungiamo al problema anche
           .....//SA_v, Q_v, NSW_v, V_v, T_v.
           //Imposto i vincoli, tutti i vincoli sono binari di
           //NotEquals, vengono impostati tra regioni confinanti.
           //Dapprima creiamo il nuovo vincolo binario
           BC_SD_NotEquals bc_notequals = new BC_SD_NotEquals();
           //setto i vincoli tra le regioni confinanti
           csp_problem.setConstraint(WA_v, NT_v, bc_notequals);
           csp_problem.setConstraint(WA_v, SA_v, bc_notequals);
           csp_problem.setConstraint(NT_v, SA_v, bc_notequals);
           csp_problem.setConstraint(NT_v, Q_v, bc_notequals);
           csp_problem.setConstraint(SA_v, Q_v, bc_notequals);
           csp_problem.setConstraint(SA_v, NSW_v, bc_notequals);
           csp_problem.setConstraint(SA_v, V_v, bc_notequals);
           csp_problem.setConstraint(Q_v, NSW_v, bc_notequals);
           csp_problem.setConstraint(NSW_v, V_v, bc_notequals);
           //creo un sover, risolvera' il problema mediante
           //l'algoritmo di Backtracking.
           Solver bt_solver = new BTSolver(csp_problem);
           //creo un nuovo solution manager,
           // essenziale per la stampa su std output dei risultati.
           MySolutionManager my_sol = new MySolutionManager();
           //attacco il solution manager al solver.
           bt_solver.setSolutionManager(my_sol);
           //setto il numero di soluzioni da trovare,
           // -1 per tutte le soluzioni.
           bt_solver.setNumberOfSolutionsToFind(-1);
           bt_solver.run(); //eseguo il solver
        }
    }
```

Listato 3.2: Risultato dell'esecuzione del programma del map coloring

```
Starting the Solving Process
Solving CSP using Simple Backtracking algorithm
 Solution 1
  WA : rosso
  NT : verde
  SA : blu
  Q : rosso
  NSW : verde
  V : rosso
  T : rosso

 Solution 2
  WA : rosso
  NT : verde
  SA : blu
  Q : rosso
  NSW : verde
  V : rosso
  T : verde

  .
  .
  .
  .
  .

 Solution 18
  WA : blu
  NT : verde
  SA : rosso
  Q : blu
  NSW : verde
  V : blu
  T : blu

Ending the Solving Process
Solutions found: 18
```

- *MySolutionManagerExtend:* Questa classe estende la classe *SolutionManagerInterfaceExtend*, quindi ne eredità tutti i metodi. In particolare a noi è utile il metodo *public void notifySolution(Solution solution, SolverExtend solExt)*, che come detto in precedenza ci permette di salvare la soluzione in un oggetto di tipo *Booking* che contiene i dati relativi ad un prenotazione ed è un attributo privato della classe.
  Siccome l'applicazione è strutturata in modo tale da presentare un solo risultato dopo la ricerca, si è pensato di impostare un contatore. Questo contatore inizialmente (prima ricerca) avrà valore 0 e viene incrementato ogni volta che l'utente chiederà un nuovo risultato, fermando la risoluzione al risultato successivo.
  Per recuperare le soluzioni trovate, abbiamo definito il metodo *public Booking getSolution()*.

- *SoverExtend*: La ridefinizione di questa classe ci è utile per utilizzare il metodo *setSolutionManager(SolutionManagerInterfaceExtend manager)* con un oggetto *SolutionManagerInterfaceExtend* descritto in precedenza.

Abbiamo anche definito nuove classi che implementano gli algoritmi di risoluzione, utilizzati nella fase di test il cui nome termina con *Extend*. Questa ridefinizione è resa necessaria per mantenere la compatibilità con le modifiche precedenti. In particolare le classi originali estendono la classe *Solver*, siccome questa è stata ridefinita in *SolverExtend*, è stato necessario ridefinire anche queste classi.
Le classi differiscono da quelle della libreria JCL solamente per il nome, in quanto non sono state apportate modifiche agli algoritmi.
Tutte le classi estendono la classe *SolverExtend*, di seguito un elenco di queste:

- *BTSolverExtend*.
- *ACSolverExtend*.
- *FCSolverExtend*.
- *FC_ARCSolverExtend*.
- *FC_DVOSolverExtend*.
- *FC_ARC_DVOSolverExtend*.

Un altra limitazione, per ciò che riguarda i nostri scopi, avviene al momento della creazione di una tupla; per chiarire il problema ci serviremo di un esempio. Supponiamo di avere due domini di interi $\{1,2,3\}$ e $\{4,5\}$, con il costruttore della

Listato 3.3: Funzione notifySolution() appartenente alla classe MySolutionManagerExtend

```
// Funzione che ci permette di controllare se una soluzione
// corrisponde ad uno slot libero di prenotazione.
public void notifySolution(Solution solution, SolverExtend solExt) {
   try {
      solutions++;
      DBManager dbManager = new DBManager();
      // Estraiamo dalla soluzione le variabili che ci interessano.
      String id_esameString = problem.getVariable(3).getLabel().↩
          getElement(solution.getValue(3)).toString();
      String id_presdioString = problem.getVariable(4).getLabel().↩
          getElement(solution.getValue(4)).toString();
      String dataString = problem.getVariable(0).getLabel().↩
          getElement(solution.getValue(0)).toString();
      String oraString = problem.getVariable(1).getLabel().↩
          getElement(solution.getValue(1)).
      // Cerchiamo la disponibilita' di una prestazione all'interno
      // del database.  Salviamo i risultati in un vettore.
      Vector esameVect = dbManager.checkReserved_extractPren_id(↩
          id_esameString,
      id_presdioString, dataString, oraString);
      // Se il vettore e' vuoto, lo slot di prenotazione e' libero
      if (esameVect.isEmpty()) {
      // La variabile count ci serve nel caso di ulteriori risultati,
      // per fermare la risoluzione al risultato voluto.
           if (count == 0) {
              // Estriamo le informazioni che ci servono.
              String sede_esame_id = dbManager.extractSede_esame_ID(↩
                  id_esameString, id_presdioString);
              Esame esame = new Esame(Integer.parseInt(↩
                  id_esameString.toString()),
                                      Integer.parseInt(↩
                                          id_presdioString.toString()↩
                                          ),
                                      Integer.parseInt(sede_esame_id↩
                                          ));

              booking.setEsame(esame);
              booking.setData(dataString);
              booking.setOra(oraString);
              // Fermiamo la risoluzione.
              solExt.setNumberOfSolutionsToFind(0);
           } else count--;
       }
             // Blocco di istruzioni che ci serve nel caso abbiamo
             // trovato una prestazione in precedenza cancellata.
          else  {
             if (esameVect.get(1).equals("0")) {
                .  //Vengono estratte le informazioni che
                .  //ci servono e fermata la risoluzione
  .... //blocco catch omesso
```

classe *TupleDomain* otterremo le seguenti tuple: (1,4), (2,5), (3,4); Ciò che è necessario al nostro scopo è il prodotto cartesiano dei due domini interi, di conseguenza si è deciso di creare una nuova classe, nominata

- *TupleDomainExtend*: In questa classe abbiamo apportato una modifica al metodo *setDomain()*, così da rendere possibile il prodotto cartesiano tra due domini. Le tuple che vengono costruite mediante questa classe hanno arità due.

Durante l'implementazione della parte che riguarda la ricerca delle prestazioni sanitarie si è verificato un errore a runtime della libreria, dovuto ad un errato casting di un oggetto. In particolare nel metodo *satisfies(int index, CSPLabel l)* della classe *UC_TD_Equals* l'oggetto *l* viene convertito in un *IntervalDomaniLabel* ciò causa un errore a runtime poichè non è possibile la conversione.
La soluzione proposta consiste nella definizione di una nuova classe:

- *UC_TD_EqualsExtend* che risolve questo problema sostituendo

  IntervalDomainLabel idl = (IntervalDomainLabel)l;

  con

  TupleDomainLabel idl = (TupleDomainLabel)l;

  nel metodo *satisfies(int index, CSPLabel l)*.

3.5.1 Creare nuovi domini

Durante lo sviluppo dell'applicazione non abbiamo avuto la necessità di creare nuovi domini, in quanto ci sono serviti unicamente quelli già implementati dalla libreria.
Tuttavia è possibile creare nuovi domini estendendo la classe *CSPDomain*.
Supponendo che ci serva lavorare con i numeri reali è possibile creare la classe *RealsDomain* nel seguente modo:

```
package JCL;
import java.util.*;

public class RealsDomain extends CSPDomain {
.....
.....
```

implementando poi le operazioni che sono presenti anche per gli altri domini (addElement, getSize, getElement, ecc..). Dopo di che creiamo la classe *RealDomainLabel* nel seguente modo:

```
package JCL;
import java.util.*;

public class RealsDomainLabel extends CSPLabel {
.....
.....
```

Capitolo 4

Sistema

4.1 Descrizione del sistema

La funzione principale del portale è quella di permette la prenotazione di prestazioni sanitarie da parte di qualsiasi cittadino. Possiamo identificare due attori principali, ospite e utente registrato, che descriveremo dettagliatamente in seguito.
Le funzionalità quindi offerte dall'applicazione sono:

- Registrazione,
- Visualizzazione delle prestazioni prenotatate,
- Cancellazione di una prenotazione,
- Ricerca di un prestazione,
- Conferma prenotazione.

4.2 Analisi del dominio

Descriveremo ora le principali entità e le azioni che possono compiere.

4.2.1 Attori

- ospite: l'utente generico che accede all'applicazione, può accedere alla prima pagina e alla pagine di registrazione, una volta registrato può eseguire ogni azione che compie un utente.
- utente registrato: L'utente registrato ha la possibilità, dopo aver eseguito il login, di accedere ad ogni parte dell'applicazione, in particolare può eseguire le seguenti azioni:

- ricercare un prestazione;
- dopo aver cercato una prestazione ha la possibilità di effettura una prenotazione;
- visualizzare tutte le sue prenotazioni,
- cancellare una prestazione in precedenza prenotata.

Un utente ha le seguenti proprietà: nome, cognome, codice fiscale, e-mail, età, sesso, data di nascita, luogo di nascita, comune di residenza.

4.2.2 Entità

- Presidio: È una struttura sanitaria in cui sono presenti un insieme di ambulatori, questi possono cambiare da presidio a presidio, le proprietá di un presidio sono il nome, l'indirizzo e il comune in cui si trova.

- Ambulatorio: È un reparto di un presidio, al suo interno vengono effettuate le prestazioni sanitarie, le proprietà di un ambulatorio sono il nome e l'appartenenza ad un presidio.

- prestazione sanitaria: Viene anche chiamata prestazione o esame sanitario. Ha come proprietà il nome e l'appartenenza ad un ambulatorio che a sua volta afferisce ad un presidio.

- Prenotazione: L'utente può scegliere di riservare un posto per una prestazione sanitaria. Le sue proprietà sono date da: ora, data e la terna ambulatorio, presidio, esame.

- Richiesta: Vengono chiamate richeste tutte le informazioni relative alla ricerca di un esame. Una richiesta ha le stesse propretà della prenotazione.

4.2.3 Use Cases

Nelle tabelle 4.1, 4.2, 4.3, 4.4, 4.5 gli use cases dell'applicazione.

Tabella 4.1: Use case Registrazione.

Nome	REGISTRAZIONE.
Requisiti	Nessuno.
Attori	Ospite, Sistema.
Descrizione	Permette da un ospite la registrazione al servizo. È obbligatoria per accedere ai successivi use cases.

Numero	**Attore**	**Azione**
1	Ospite	Accede all'applicazione.
2	Sistema	Presenta la pagina principale.
3	Ospite	Accede all'area della registrazione
4	Sistema	Presenta la pagina della registrazione.
5	Ospite	Inserisce i dati e preme il bottone RESGISTRATI.
6	Sistema	Se i dati non sono corretti, avvisa l'utente e non gli permette di continuare.
7	Ospite	Modifica i dati non corretti e preme il bottone REGISTRATI.
8	Sistema	Se i dati non sono corretti torna al punto 6, altrimenti procede e visualizza la pagina degli esami.

Tabella 4.2: Use case Log-in.

Nome	LOG-IN.
Requisiti	L'utente deve essere registrato.
Attori	Utente, Sistema.
Descrizione	Permette da un utente di accedere ai servizi offerti dall'applicazione . È obbligatoria per accedere ai successivi use cases.

Numero	**Attore**	**Descrizione**
1	Utente	Accede all'applicazione.
2	Sistema	Presenta la pagina principale.
3	Utente	Inserisce i dati, l'username (codice fiscale) e la password. Clicca sul bottone LOGIN
4	Sistema	Se i dati inseriti sono errati, avvisa l'utente con un messaggio.
5	Ospite	Modifica i dati e preme il bottone LOGIN.
6	Sistema	Se i dati inseriti sono errati torna al punto 4, altrimenti visualizza la pagina degli esami .

Tabella 4.3: Use case ricerca prestazione.

Nome	RICERCA PRESTAZIONE.
Requisiti	Log-in.
Attori	Utente, Sistema.
Descrizione	Permette da un utente la ricerca di una prestazione, le opzioni combinabili per la ricerca di una prestazione sono: • nome prestazione sanitaria; • presidio (selezionabile da un elenco); • ambulatorio (selezionabile da un elenco). Almeno due delle precedenti opzioni sono obbligatorie. • periodo in cui si desidera effettuare la prestazione; • periodo della giornata in cui si desidera effettuare la prestazione (mattino, pomeriggio, pranzo); • giorni della settimana in cui non si ha la possibilità di effettuare una prestazione.

Numero	Attore	Descrizione
1	Utente	Accede alla pagine ricerca perstazioni.
2	Sistema	Presenta la pagina della ricerca.
3	Utente	Inserise i dati richesti (vedere la descrizione dello use case) e preme il bottone RICERCA
4	Sistema	Se non vengono inseriti due dei campi obbligatori il sistema avvisa l'utene e non lo fà continuare.
5	Utente	Modifica i campi errati.
6	Sistema	Se ancora non sono stati inseriti due dei campi obbligatori si torna al punto 4, altrimenti esegue la ricerca. Se non vengono trovati risultati, questo viene notificato all'utente tramite un messaggio.
7	Utente	Modifica i dati e preme il bottone ricerca. In caso di errori si torna al punto 6.
8	Sistema	Presenta il risultato trovato.

Tabella 4.4: Use case prenotazione prestazione.

Nome	PRENOTAZIONE PRESTAZIONE.
Requisiti	Ricerca prestazione.
Attori	Utente, Sistema.
Descrizione	Permette da un utente la prenotazione di una prestazione ricercata in precedenza.

Numero	**Attore**	**Descrizione**
1	Utente	L'utente che in precedenza ha effettuato la ricerca ha come risultato una prestazione.
2	Sistema	Presenta la pagina con la prestazione trovata e i relativi dettagli.
3	Utente	L'utente può scegliere di visualizzare un nuovo risultato, cliccando il bottone NUOVO RISULTATO.
4	Sistema	Il sistema presenta il nuovo risultato.
5	Utente	Se il risultato presentato è adeguato per l'utente questo clicca il pulsante CONFERMA PRENOTAZIONE, altrimenti torna al punto 3.
6	Sistema	Presenta una pagina di conferma prenotazione con i dettagli della prestazione.
7	Utente	Può scegliere se effettuare una nuova ricerca oppure andare alla pagine degli esami, cliccando sui rispettivi link.

Tabella 4.5: Use case cancellazione prestazione.

Nome	CANCELLAZIONE PRESTAZIONE.
Requisiti	Log-in. Almeno una prestazione prenotata.
Attori	Utente, Sistema.
Descrizione	Permette da un utente la cancellazione di una prestazione in precedenza prenotata.

Numero	**Attore**	**Descrizione**
1	Utente	Dopo aver effettuato il log-in si trova nella pagina esami.
2	Sistema	Visualizza la pagina, con tutte le prestazioni prenotate in precedenza.
3	Utente	Sceglie quale prestazione cancellare, cliccando sul bottone CANCELLA PRENOTAZIONE che è presente in ogni box relativo alla prestazione.
4	Sistema	Visualizza la pagina di conferma cancellazione prestazione con tutti i dettagli relativi alla prestazione.
5	Utente	Può scegliere di tornare alla pagine esami, (punto 2). Oppure cliccare sul bottone annulla prestazione.
6	Sistema	Visualizza un messaggio relativo all'avvenuta prestazione.
7	Utente	Torna alla pagina esami.

4.3 Progettazione base di dati

4.3.1 Requisiti:

- Ad ogni utente è associato un nome, cognome, codice fiscale, username (non utilizzato), password, nome, e-mail, sesso, data di nascita, comune di nascita, comune di residenza, l'utente viene individuato attraverso un id univoco. Questi dati ad esclusione dell'id devono essere inseriti al momento della registrazione;

- Per effettuare la registrazione l'utente deve essere residente nella regione Emilia Romagna;
- Ogni utente ha la possibilità di prenotare uno o più esami;
- A prenotazione avvenuta un esame più essere cancellato dall'utente che ha effettuato la prenotazione;
- Ogni prestazione viene effettuata in orari e giorni specifici a seconda del presidio in cui la prestazione viene erogata, i dettagli che ci servono sono ora di inizio prestazione, ora di fine prestazione, tempo che un prestazione impiega per essere esguita (ad es. 15 minuti) e i giorni della settimana in cui viene eseguita.
- Ogni presidio ospedaliero appartiene ad un comune dell'Emilia Romagna;
- Ad ogni presidio sono associati degli ambulatori, questi variano a seconda del presidio;
- Ad ogni ambulatorio sono associate delle prestazione sanitarie, le prestazioni presenti in un ambulatorio variano a seconda del presidio;
- Viene salvata ogni richesta di prenotazione con tutti i dati relativi, in modo da tenere traccia delle richieste.

4.3.2 Schema logico

Dall'analisi dei requisiti si passa alla realizzazione di uno schema logico che preserva le caratterisitiche del database che andrà in produzione. Di seguito lo schema logico del database:

UTENTI (id, username, password, cf, cognome, nome, email, sesso, datanascita, comunenascita, residenza).

COMUNE (nome, provincia).

PRESIDI (id_presidio, nome, comune)

foreign key comune → COMUNE(nome).

AMBULATORI (id_ambulatorio, nome).

PRESTAZIONI (id, nome).

SEDE_AMBULATORIO (id, id_presidio, id_ambulatorio)

foreign key id_presidio → PRESIDI(id_presidio).

foreign key id_ambulatorio → AMBULATORI(id_ambulatorio).

SEDE_PRESTAZIONE (id, id_sedeambulatorio, id_prestazione, ora_inizio, ora_fine, durata, giorni)

unique key (id_sedeambulatorio, id_prestazione).

foreign key id_sedeambulatorio → SEDE_AMBULATORIO(id).

foreign key id_prestazione → PRESTAZIONI(id).

PRENOTAZIONI (id, id_utente, id_sede_prestazione, data, ora, prenotata, data_prenotazione).

unique key (id_sede_prestazione, data, ora).

foreign key id_sede_prestazione → SEDE_PRESTAZIONE(id).

foreign key id_utente → UTENTI(id).

RICHIESTE (id, id_utente, id_sede_prestazione, data_richiesta, dalgg, algg, periodo_giorno, giorni_non_disp).

foreign key id_utente → UTENTI(id).

foreign key id_sede_prestazione → SEDE_PRESTAZIONE(id).

GIORNI (id, giorni_it).

PERIODO_GIORNO (id, periodo_it).

4.4 Schema entità relazione

In figura 4.1 è rappresentato lo schema entità relazione, le linee tratteggiate tra le tabelle indicano che la relazione viene gestita tramite l'appicazione e non con vincoli di referenza. Il motivo di questa scelta è dato dal fatto che nelle tabelle RICHESTE e SEDE_PRESTAZIONE gli attributi che dovrebbero essere legati alle tabelle GIORNI o PERIODO_GIORNO sono di tipo SET, quindi un insieme. A causa di ciò mysql non permette la definizione di vincoli di chiave esterna. Per il resto, la simbologia utilizzata è quella standard UML.
Per quanto riguarda la relazione SEDE_PRESTAZIONE, si è ritenuto necessario realizzare una tabella, in quanto alcune delle relazioni che la compongono hanno cardinalità N.

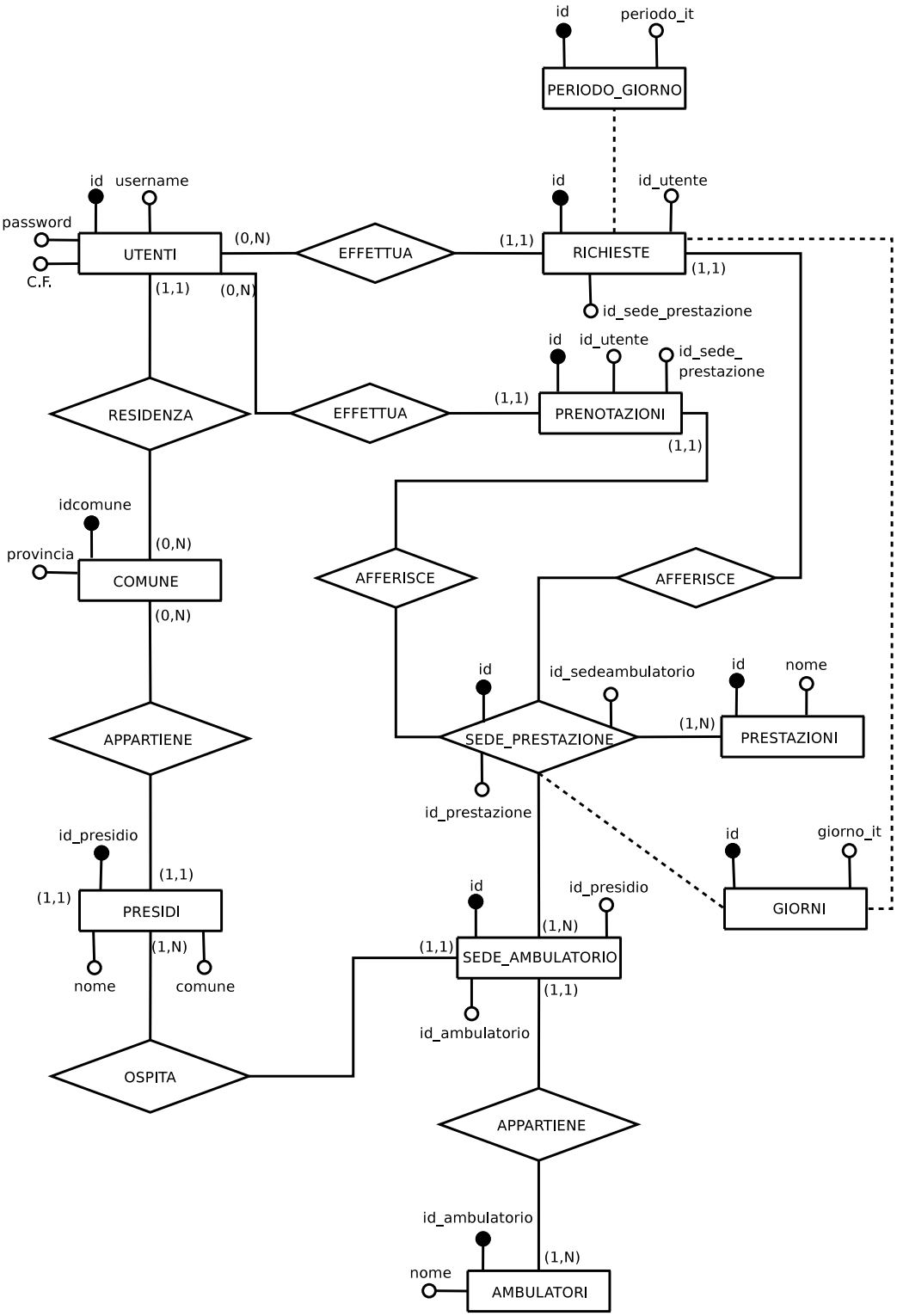

Figura 4.1: Schema entità-relazione

4.5 Applicazione e constraint satisfaction problem

L'integrazione delle normali tecniche per la progettazioni di applicazioni web e la programmazione logica a vincoli ha comportato il sorgere di alcune problematiche.
Per prima cosa come modellare il problema di soddisfacimento vincoli al problema della ricerca esami, in quanto le variabili in gioco erano molte e di conseguenza sono stati consistenti anche i vincoli da assegnare. In secondo luogo si ha una sostanziale parziale replicazione di dati, tra il database e la parte che si occupa di gestire il problema di soddisfacimento vincoli. Un spiegazione di questo fatto è che normalmente un applicazione web quando accede per una ricerca al database lo fà estraendo solamente i dati necessari al momento. Prendiamo come esempio la ricerca di uno slot libero per una prestazione in seguito a una ricerca da parte di un utente. I dati che imposta l'utente vengono usati per trovare uno slot libero, non abbiamo bisogno di estrapolare molti dati, in quanto effettuiamo una ricerca nel database.
Per come sono strutturati i problemi di soddisfacimento vincoli vi è la necessità di caricare tutti i dati nel risolutore, in seguito mediante vincoli, che impostiamo per mantenere le relazioni o che vengono impostati dall'utente per la ricerca riduciamo i domini. Inoltre una volta che abbiamo trovato un risultato, controlliamo mediante connessione al database se lo slot è libero. Ciò appesantisce notevolemente il programma. Per ridurre i domini e quindi per alleggerire la risoluzione abbiamo cercato di impostare alcuni vincoli che chiameremo non essenziali. Questi vincoli non servono per ottenere una giusta soluzione che si sarebbe ugualmente ottenuta con un numero minore di vincoli, ma solo per diminuire il carico computazionale, di fatti come abbiamo visto in sezione 2.5 vi sono molti algoritmi che riducono i domini delle variabili a seconda dei vincoli unari o binari impostati sulle stesse variabili.. Nell'impostare i vincoli non essenziali terremo altresì conto che un numero molto elevato di vincoli può aumentare notevolemente il carico computazione degli algoritmi, cercheremo quindi di ottenere un giusto equilibrio tra il numero di vincoli e la possibilità che questi rendano più complessa la risoluzione.

4.6 Domini e variabili

Tutte le variabili usate per impostare il problema finiscono con il suffisso _d, per sottolineare che sono variabili che contengo i domini del problema. In tabella 4.6

sono indicati il nome delle variabili, il tipo del dominio e una breve descrizione del contenuto.

L'utilizzo di queste variabili con una coerente applicazione dei vincoli ci permetterà di ottenere un settupla che identifica univocamente ogni prestazione.
L'uso della tupla esamipresidi_d a prima vista può sembrare ridondate, il suo scopo è quello di legare le variabili esamiID_d, ambulatoriID_d e presidiID_d, permettendo così di ottenere una soluzione univoca; approfondiremo meglio questo dettaglio nella sezioni successiva, trattando i vincoli.

4.7 Vincoli

Nella seguente descrizione differenzieremo i vincoli in:

- Vincoli binari essenziali;
- Vincoli binari non essenziali.

Per vincoli binari essenziali intendiamo quei vincoli che ci permettono di ottenere i giusti abbinamenti tra le variabili, ad esempio a quale ambulatorio appartiene una prestazione sanitaria, quali ambulatorio appartengono ad un presidio e così via. I vincoli non essenziali invece servono esclusivamente all'algoritmo di risoluzione per ridurre i domini durante la computazione.

4.7.1 Vincoli binari essenziali

I vincoli binari essenziali sono tutti vincoli di GoodList, ricordiamo che questi vincoli agiscono su due variabili rendendo vera la loro associazione.

I vincoli n°1 e n°2 in tabella 4.7 servono per conoscere quali prestazioni si effettuano in un presidio o mutuamente quali prestazioni possono venire svolte in un presidio. Il vincolo n°3 ci indica in quale ambulatorio una prestazione viene effettuata.
Anche se non abbiamo impostato un vincolo diretto questi primi tre vincoli ci permettono di conoscere anche quali ambulatori sono presenti in un presidio. Siccome un esame appartiene univocamente ad un ambulatorio, se almeno un esame di un ambulatorio viene svolto in un presidio, allora anche l'ambulatorio sarà presente in quel presidio.
Il vincolo n°4 permette di conoscere in quali giorni una prestazione viene eseguita.
Il vincolo n°5 permette di conosce gli orari in cui una prestazione viene effettuata.
Il vincolo n°6 permette di conoscere a quale data corrisponde un giorno.

Tabella 4.6: Nome, tipo e descrizione delle variabili che rappresentano il dominio.

Nome variabile	Tipo	Descrizione
esamiID_d	Intero	Contiene gli id delle prestazioni così come si presentano nel database.
ambulatoriID_d	Intero	Contiene gli id degli ambulatorio così come si presentano nel database.
presidiID_d	Intero	Contiene gli id dei presidi così come si presentano nel database.
esamipresidi_d	Tupla	Contiene il prodotto cartesiano di esamiID_d $\times$ presidiID_d.
giorni_d	Intero	Contiene gli i valore da 0 a 6; 0 per domenica, 1 per lunedì e cosi via.
date_d	Stringa	Contiene tutte le date di un anno, partendo dal giorno in cui si effettua la ricerca.
orari_d	Stringa	Contiene le ore della giornata ad intervalli di 15 minuti; da 00:00 a 23:45.

Tabella 4.7: Vincoli binari Goodlist essenziali con breve descrizione.

Numero	Nome vincolo	Breve descrizione
1	esami_esamipresidiGood	Associa la variabile esamiID_d alla variabile esamipresidi_d.
2	presidi_esamipresidiGood	Associa la variabile presidiID_d alla variabile esami_presidi.
3	esami_ambulatoriGood	Associa la variabile esamiID_d alla variabile ambulatoriID_d.
4	esamipresidi_giorniGood	Associa la variabile esamipresidi alla variabile giorni_d
5	esamipresidi_orariGood	Associa la variabile esamipresidi alla variabile orari_d.
6	date_giorniGood	Associa la variabile date_d alla variabile giorni_d.

Una nota importate è data dal fatto che tutte le informazioni che riguardano i precedenti vincoli vengono estratte dal database, ogni volta che un utente effettua una ricerca.
Questa assegnazione dei vincoli ci permette di ottenere come risultato una set-tupla con tutte le soluzioni possibili. Dovremo poi ridurre queste soluzioni con l'assegnamento di vincoli unari, selezionati in base alla preferenze dell'utente che effettua la ricerca.
L'uso della variabile esamipresidi_d che contiene una tupla con gli id delle prestazioni e dei presidi, può apparire ridondante, eppure svolge un ruolo centrale in quanto permette di legare tra di loro le variabili, come si può notare in figura 4.2. Sarebbe stato anche possibile eliminara a costo di incrementare notevolemente i vincoli.

Tabella 4.9: Vincoli unari.

Numero	Tipo vincolo	Varaibile coinvolta
1	NotEquals	esamiID_d
2	Equals	AmbulatoriIDi_d
3	Equals	presidiID_d.
4	LessThenEquals, GreaterThenEquals	date_d
5	$\geq, >, \leq, <, =$	orari_d.
6	NotEquals	giorni_d.

4.7.2 Vincoli binari non essenziali

Ricordiamo che questi vincoli servono per ridurre i domini durante la computazione, l'esattezza delle soluzioni trovate è ugualmente valida anche senza di essi. In tabella 4.8 un elenco e una descrizione di questi vincoli.

Tabella 4.8: Vincoli binari GoodList non essenziali.

Numero	Nome vincolo	Breve descrizione
1	ambulatorio_presidiGood	Associa la variabile ambulatoriID con la variabile presidiID.

4.7.3 Vincoli unari

Sono i vincoli impostabili dall'utente al momento della ricerca tramite la maschera web. Sono tutti vincoli opzionali che servono per raffinare la ricerca, vengono settati in base alle preferenze e alla disponibilità dell'utente. In tabella 4.9 un elenco di questi vincoli.

Il vincolo unario n°1 viene impostato quando l'utente sceglie di effettuare una ricerca in base al nome della prestazione, il vincoli che settiamo sono vincoli di

NotEquals in quanto scartiamo tutte le prestazioni che non corrispondono alla ricerca.
Il vincolo unario n°2 viene impostato quando l'utente seleziona un ambulatorio dal menù a tendina che elenca tutti gli ambulatori. È un vincolo di Equals in quanto l'utente seleziona un determinato ambulatorio a cui è associato un id univoco.
Il vincolo unario n°3 viene impostato quando l'utente seleziona un presidio dal menù a tendina che elenca tutti i presidi. È un vincolo di Equals in quanto l'utente selezionando un presidio indica un id che rappresenta univocamente quel presidio.
Il vincolo unario n°4 viene impostato ogni volta che una ricerca viene eseguita, nel caso l'utente non imposti il valore la ricerca viene effettuata per solo un giorno, altrimenti in base al periodo di ricerca che l'utente imposta. Ipostiamo un vincolo di GreterThanEquals che contiene la data di inizio ricerca e un vincolo LessThenEquals con la data di termine ricerca. Entrambi i vincoli vengono impostati sulla variabile data_d.
Il vincolo unario n°5 viene impostato a seconda del periodo della giornata in cui l'utente vuole effettuare l'esame. Le possibili opzioni sono (il menù è a scelta multipla):

- Mattino;
- Pausa pranzo;
- Pomeriggio;
- Qualsiasi orario.

I vincoli che impostiamo variano molto a seconda della scelta dell'utente; ad esempio se un utente sceglie solo un opzione o due opzioni i cui orari sono congiunti, ad esempio mattino e pomeriggio, c'è la caviamo semplicemente impostando due vincoli, uno di LessThenEquals e uno di GreaterThenEquals mentre se la scelta cade sulle opzioni mattino e pomeriggio andiamo a settare un vincolo di NotEquals con tutti gli orari che compongono la pausa pranzo. Questi vincoli vengono impostati sulla variabile orari_d.
Il vincolo unario n°6 viene impostato dall'utente scegliendo, mediante un check box a scelta multipla, i giorni della settimana in cui non ha disponibilità per effettuare la prestazione. Settiamo uno o più vincoli di NotEquals sulla variabile giorni_d in base alle scelte effettuate dall'utente.

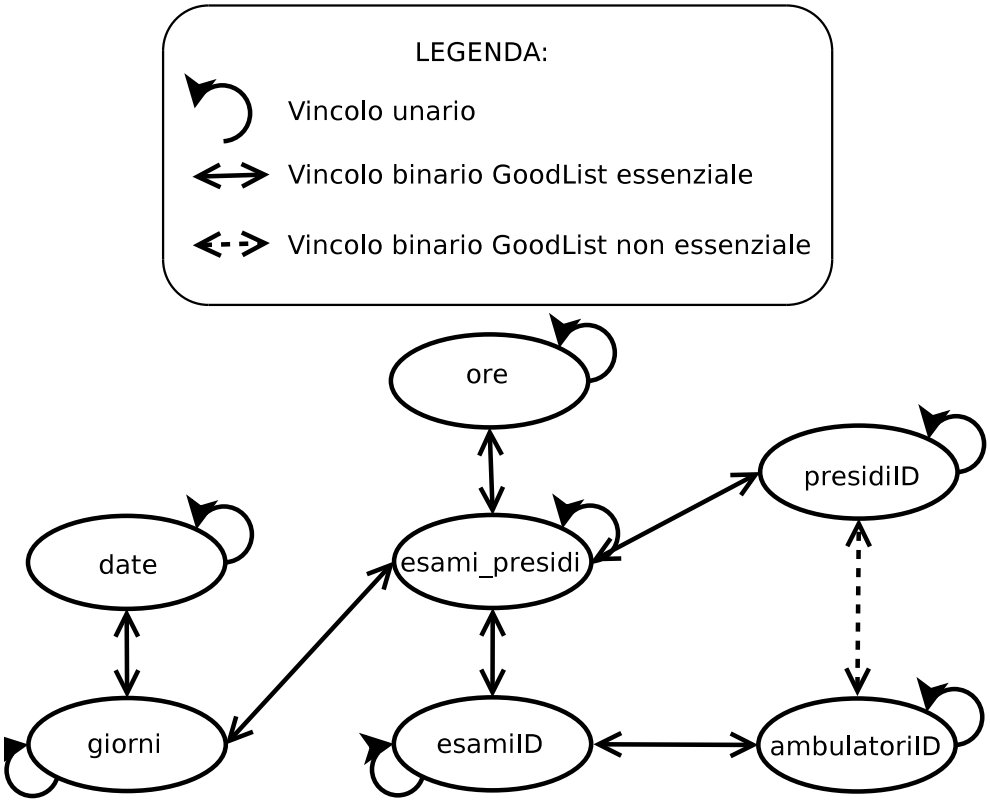

Figura 4.2: Grafo rappresentante i vincoli

4.8 Test e validazione

Per verificare le prestazioni del sistema e verificare quale tra i tanti algoritmi risolutivi si adatta meglio alla risoluzione di questo tipo di problema, abbiamo effettuato alcuni test.
La postazione di test è costituita da un pc portatile ed è dotata di un processore Intel Core 2 Duo T7200 a 2GHz, 2Gb di memoria RAM e disco fisso da 80 Gb 7200rpm. Sistema operativo Debian 5.0 Lenny.
Gli algoritmi testati e tra parentesi la loro abbreviazione, utilizzata nei grafici:

- Backtracking (BT);
- Arc-Consistency (AC);
- Forward Checking con Arc-Consistency (FC_ARC);
- Forward Checking con Dynamic Variable Ordering (FC_DVO);
- Forward Cheking con Arc-Consistency e Dynamic Variable Ordering (FC_ARC_DVO).

Abbiamo eseguito i test in due modalità; la prima modalità prevede che ad ogni risultato trovato, effettuiamo una connessione al database per verificare che lo slot di prenotazione sia libero. Mentre nella seconda modalità abbiamo ugualmente chiamato la funzione per la connessione al database, ma questa è stata modificata e 'svuotata' del suo contenuto, facendo in modo che il suo valore di ritorno sia sempre false (slot di prenotazione sempre occupato).
Per entrambi i test la ricerca è stata svolta in modo identico, stessa prestazione sanitaria, stesso presidio e naturalmente stesso ambulatorio. Per ogni test è stato misurato il tempo di esecuzione della sola parte di risoluzione; incrementando ogni volta il numero di giorni occupati. Abbiamo in questo modo effettuato test in cui non vi è nessuna prenotazione, un intero giorno occupato, tre giorni occupati, sette, quindici, venticinque, e quaranta giorni occupati.

I grafici in figura 4.3 e 4.4 mostrano nettamente come il miglior algoritmo sia il Forward Checking with Dynamic Variable Ordering. Tuttavia durante i test si è notato che, mentre ogni algorimo passa per ogni soluzione che appartiene ai giorni già occupati, l'FC_DVO esplora molte meno soluzioni; questa anomalia si è verificata per le prove superiori a venticinque giorni. A causa di ciò, sebbene non si sia mai verificato nei test, si può non avere la soluzione più recente. Tuttavia in prove con ricerce casuali effettuate in seguito, l'algoritmo si discosta al massimo

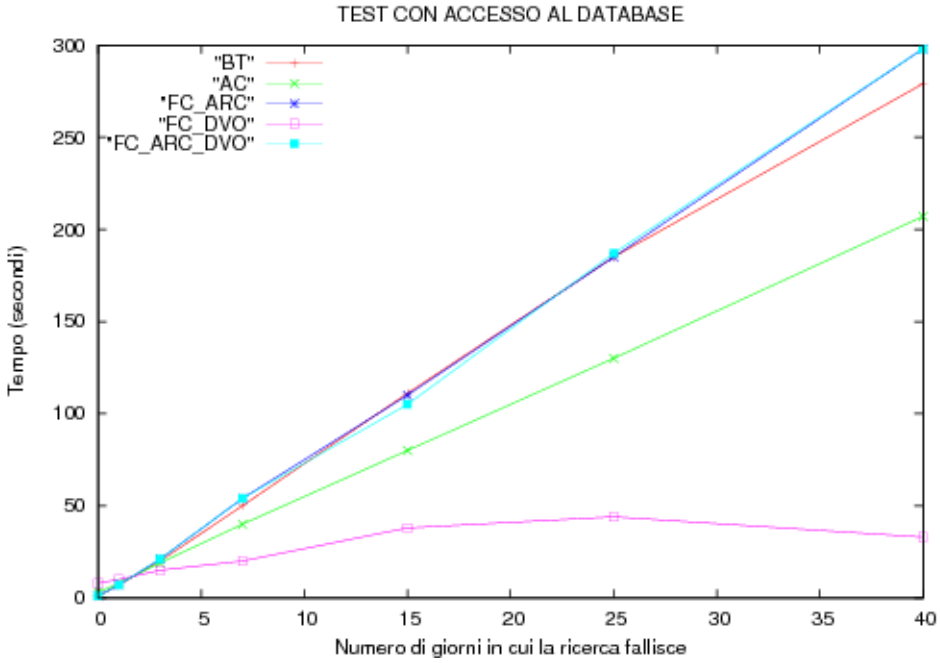

Figura 4.3: Grafico che rappresenta i test con accesso al database.

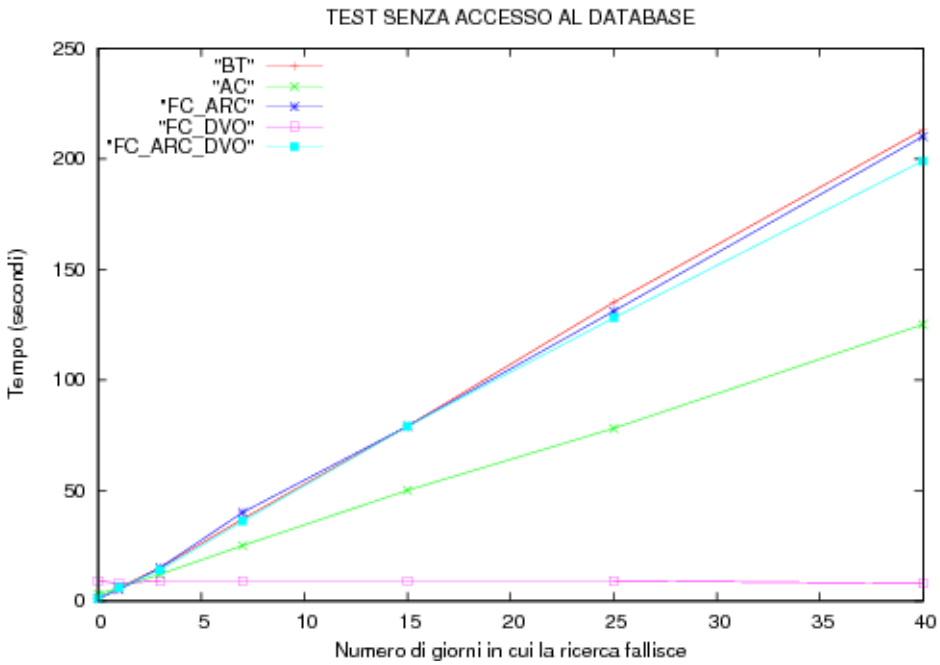

Figura 4.4: Grafico che rappresenta i test senza accesso al database.

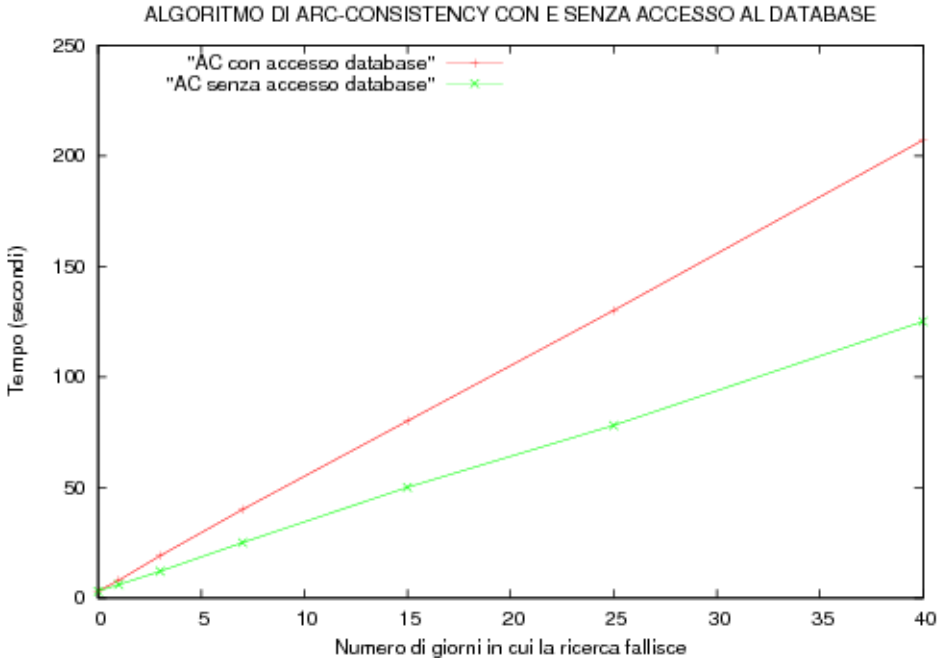

Figura 4.5: Grafico che confronta l'algoritmo di arc consistency con e senza accesso al database.

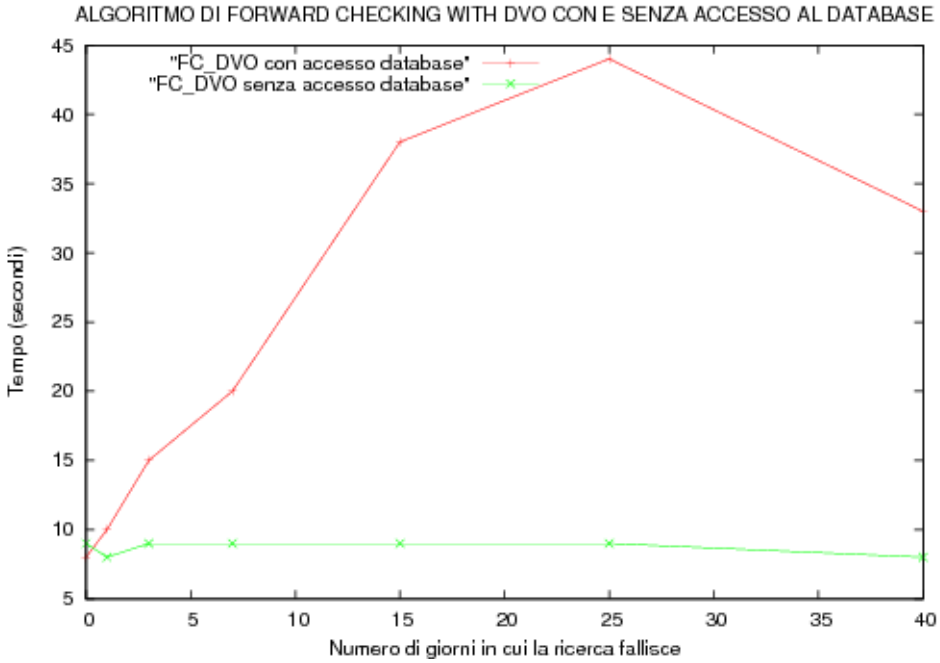

Figura 4.6: Grafico che confronta l'algoritmo di forward checking with dynamic variable ordering con e senza accesso al database.

di una settimana dal primo slot libero.
I grafici in figura 4.5 e 4.6 evidenziano, rispettivamente, il costo dell'algoritmo di Arc-Consistency e dell'algoritmo Forward Checking with DVO, con accesso al database e senza accesso al database. Come si nota dai grafici la differenza di tempo è notevole. Il grafico 4.5, relativo all'algorimo di arc-consistency mostra che questa differenza tende a crescere all'aumentare delle soluzioni. Questo comportamento è medesimo anche per gli altri algoritmi testati ad esclusione del Forward Checking with DVO. Infatti in figura 4.6 si nota che, sebbene il tempo richiesto per la connessione al database si in precentuale maggiore rispetto agli altri algoritmi, questo può anche non aumentare al crescere delle soluzioni trovate. Questo fatto dipende da quante soluzioni l'algoritmo è passato prima di arrivare ad uno slot libero.

4.9 Applicazione

Per concludere presenteremo le schermate dell'appicazione che riteniamo più importanti. In figura 4.7 e 4.8 rispettivamente la form di registrazione vuota e con notifica errori in caso di inserimento errato dei campi, in quest'ultimo caso sono stati lasciati vuoti tutti i campi, da ciò derivano le notifiche di errore.
In figura 4.9 la form di ricerca prestazione santiaria con alcuni campi inserti, il risultato della ricerca si trova in figura 4.10. Cliccando il tasto 'Conferma prenotazione' la prenotazione diventa effettiva e viene visualizzata una pagina di conferma prenotazione, come in figura 4.11.
In figura 4.12 è presente la pagina dove vengono visualizzate le prestazioni prenotate. Cliccando su tasto 'Cancella prenotazione', si passa alla pagina 4.13, confermando la cancellazione mediante click sul tasto 'Annulla prenotazione', viene notificato un messaggio di avvenuta cancellazione.

Figura 4.7: Form di registrazione.

Figura 4.8: Form di registrazione con notifica errori nell'inserimento dei dati.

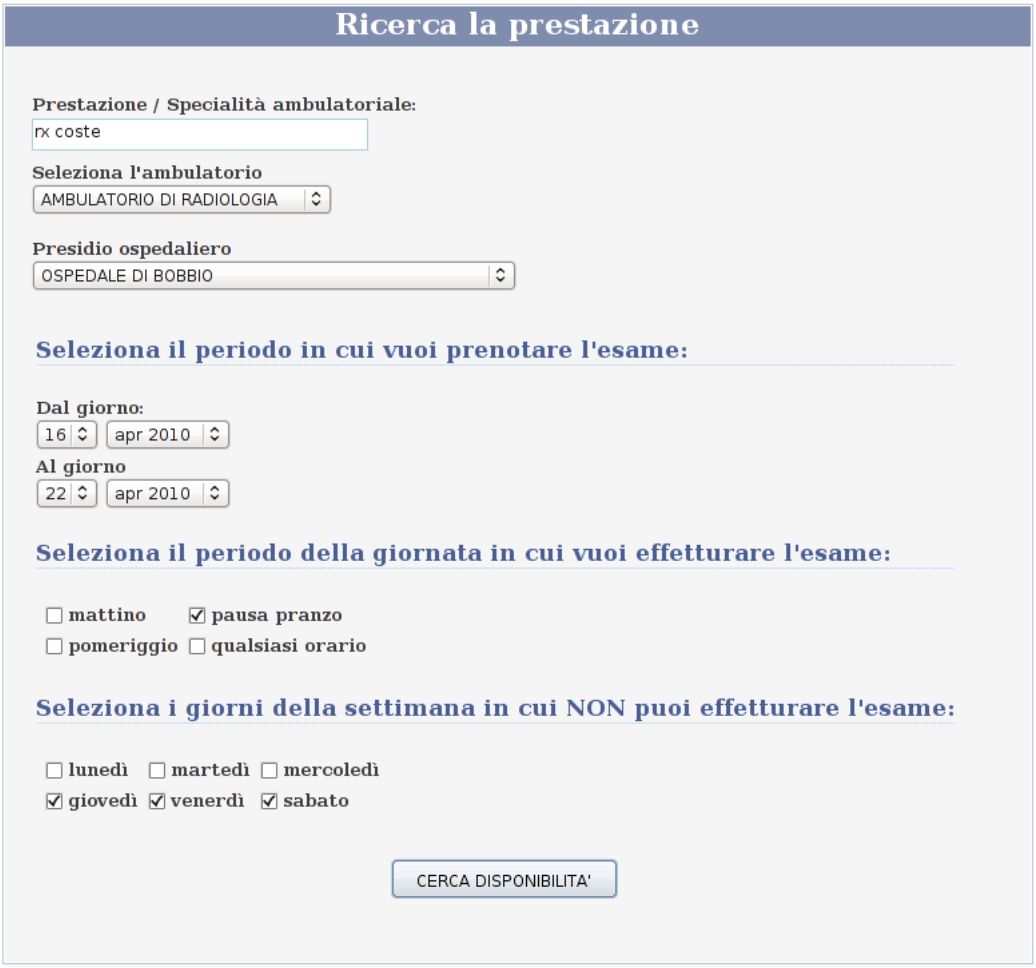

Figura 4.9: Form di ricerca.

Figura 4.10: Risultato ricerca, conferma prenotazione.

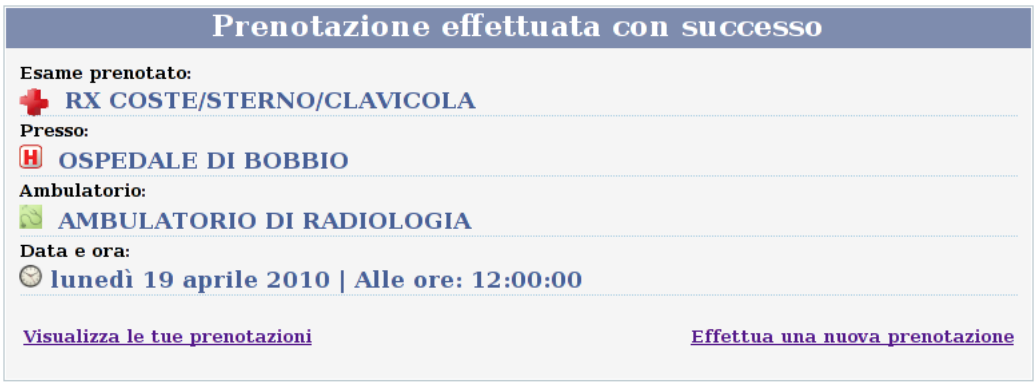

Figura 4.11: Prenotazione effettuata con successo.

Figura 4.12: Pagina di riepilogo prenotazioni eseguite.

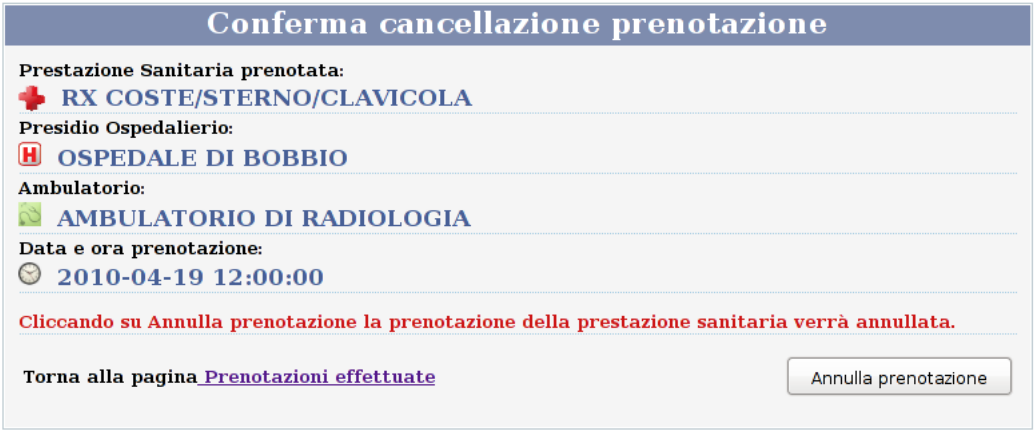

Figura 4.13: Pagina di cancellazione prenotazione.

Conclusioni

Questo lavoro di tesi è nato dall'esigenza della Regione Emilia-Romagna, in accordo con l'Università di Parma, di estendere le prenotazioni delle prestazioni sanitarie al mondo web.
Si è quindi realizzato un appicativo web, con funzionalità basilari che permette al cittadino la registrazione al servizio, il login, la ricerca e la prenotazione delle prestazioni e la cancellazione di prenotazioni effettuate in precedenza. Ci si è soffermati in particolare sulla parte di ricerca delle prestazioni sanitarie, realizzando la medesima mediante la programmazione logica a vincoli, con l'ausilio della libreria JCL. Durante il lavoro si sono rese necessarie modifiche alla libreria, per adattarla alle esigenze dell'applicazione.
Sebbene la parte di ricerca, prenotazione, cancellazione e registrazione di un utente siano complete, ulteriori funzionalità possono essere aggiunte per migliorare la fruibilità e la completezza generale dell'applicazione. Di seguito un elenco di funzionalità che non sono state implementate per mancanza di tempo:

- Implementazione di vincoli di tipo soft, dando la possibilità agli utenti di inserire preferenze non restrittive.
- Migliorare la ricerca nel caso l'utente richeda un nuovo risultato. Dopo una ricerca viene presentato un risultato all'utente, se questo risultato non soddisfa le sue esigenze, l'utente può richedere un nuovo risultato. In questo caso la ricerca riparte dall'inizio. Si potrebbe modificare la libreria JCL in modo da fermare la ricerca al risultato voluto e nel caso l'utente richieda un nuovo risultato riprendere l'esecuzione dal punto in cui ci si era fermati.
- Una pagina di amministrazione che permetta l'inserimento tramite l'appicazione di presidi, ambulatori, esami. E che più in generale permetta di gestire l'intera applicazione.
- Una fase si test per verificare che l'applicazione si comporti adeguatamente in ogni contesto.

Bibliografia

[1] Kim Marriott and Peter J. Stuckey. *Programming with Constraint: An Introduction*. The MIT Press, 1998.

[2] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[3] L. Console, E. Manna, P. Mello, M. Milano. *Programmazione Logica e Prolog, Nuova edizione*. UTET Libreria, 2005.

[4] Roman Bartàk. *Constraint Propagation and Backtracking-based Search*. Charles University, Faculty of Mathematics and Physics Malostranské nàm. 2/25, 118 00 Praha 1, Czech Republic.

[5] Sito Web Java Constraint Library. *http://liawww.epfl.ch/JCL/*