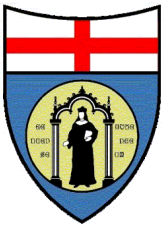

UNIVERSITÀ DEGLI STUDI DI GENOVA

SCUOLA DI SCIENZE UMANISTICHE

DIPARTIMENTO DI ITALIANISTICA, ROMANISTICA, ANTICHISTICA, ARTI E SPETTACOLO

Corso di Laurea in Lettere Moderne

Prova finale

Logica deduttiva nell'Intelligenza Artificiale per la risoluzione

di problemi computazionali complessi

Referente: prof. Carmine Dodaro

Coreferente: prof. Marco Maratea

Candidato: Chantal Lengua

Anno Accademico 2017/2018

Indice

Introduzione

Alla base di questo studio vi è l'obiettivo di dimostrare la stretta correlazione che può sussistere tra due discipline all'apparenza molto lontane tra loro: le materie umanistiche e le scienze informatiche, in particolare nell'ambito dell'Intelligenza Artificiale.

L'obiettivo di questa tesi di laurea è dunque quello di fornire una visione d'insieme di uno dei numerosi punti di collegamento tra queste due discipline: la logica. Nella prima parte si analizza il metodo sillogistico come descritto da Aristotele, partendo dalle origini con i filosofi della Grecia presocratica, per poi proseguire fino ai grandi padri della logica moderna, Gottlob Frege, Ludwig Wittgenstein ed Emil Post. Si procede dunque con l'illustrazione della logica enunciativa, seguita dalla logica del primo ordine e, infine, dalla logica non monotona, ossia quel tipo di linguaggio non classico che prevede eccezioni e si rivela dunque essere fondamentale all'interno dei calcolatori elettronici e, dunque, dell'Intelligenza Artificiale.

Il secondo capitolo si focalizza su Answer Set Programming (ASP), la forma di programmazione dichiarativa e *logic-based* per la risoluzione di problemi complessi di cui si tratterà per tutto il resto della tesi. Dopo una descrizione della sua sintassi e della sua semantica, si passa a una dimostrazione della sua efficacia nel terzo capitolo, in cui, mediante l'impiego di un'ampia bibliografia scientifica, vengono riportati numerosi esempi di applicazioni nel mondo reale e nel settore industriale.

Nel quarto e ultimo capitolo si è scelto di presentare un caso d'uso pratico, con una programmazione basata su Answer Set Programming che fosse in grado, in un ordine di pochi secondi, di risolvere complessi quesiti di problem-solving e di ottimizzazione come quello, preso in esame, dell'assegnazione di un numero di progetti (da due a otto) a un numero crescente di dipendenti di un'azienda (da dieci a sessanta), secondo una scalabilità progressiva. Si comincia quindi con il riportare l'*encoding* della programmazione, riprendendo la stessa presentata nel capitolo due e spiegandone il significato, per poi presentare i due solver che si è scelto di utilizzare, CLINGO e DLV.

I test che sono stati eseguiti su un personal computer hanno confermato come attraverso la logica deduttiva, nella fattispecie l'Answer Set Programming, è possibile risolvere problemi complessi, e nel modo più rapido ed efficiente.

Capitolo 1

Rappresentazione della conoscenza

1.1 Logica classica

1.1.1 Origini ed evoluzione

Sebbene la rappresentazione della conoscenza sia oggi un settore specifico dell'intelligenza artificiale (IA), le sue origini sono tutt'altro che moderne [1]: furono i filosofi presocratici della Grecia classica, infatti, i primi a formalizzare il ragionamento umano, secoli prima dell'invenzione dei calcolatori elettronici. Esempi di una logica di predicati si trovavano già nei pitagorici (IV secolo a.C.), nei parmenidei (in particolare, la logica di non-contraddizione e la tesi dell'immutabilità dell'Essere) e nella scuola di Elea (le dimostrazioni per assurdo di Zenone di Elea), ma fu con Aristotele (384 a.C. o 383 a.C. – 322 a.C) che assunse un'impostazione sistematica e cominciò a coincidere con il metodo deduttivo, ossia il procedimento razionale che permette di derivare una conclusione (tesi) partendo da alcune premesse più generiche (ipotesi), mediante un ragionamento concatenato detto "sillogismo".

Con lo stoicismo (III sec. a.C.) la logica cominciò anche a comprendere la retorica e a occuparsi dei *lògoi* (dal greco λόγος, *logos*, "parola", "discorso"), i ragionamenti espressi in forma di proposizioni: da una logica di soli predicati, quindi, si sviluppò una logica proposizionale che studiava la coerenza tra proposizioni.

Il linguaggio formale fin qui utilizzato, tuttavia, utilizzava forme grammaticali del linguaggio umano ordinario che potevano originare errori logici e insidie per la loro imprecisione ed equivocità. In tal senso, Gottfried Leibniz (1646-1716) fu il primo a cercare di creare un linguaggio scientifico universale, la "logica simbolica e combinatoria", ma si dovettero aspettare due secoli prima che alcuni dei padri della logica moderna, come Gottlob Frege (1848-1925), Ludwig Wittgenstein (1889-1951) ed Emil Post (1897-1954) si cimentassero nell'elaborazione di un *linguaggio formale* artificiale con cui esprimere la forma logica dei ragionamenti allo scopo di indagarne le condizioni di validità. Vennero quindi ideati e perfezionati i *linguaggi enunciativi* o *proposizionali*, linguaggi formali che evidenziano soltanto il tipo di connessione logica tra gli enunciati, espressa mediante connettivi o operatori logici enunciativi. Ed è proprio su questo

linguaggio rigoroso e privo di ambiguità che si basa la *logica enunciativa* o *proposizionale*, primo fondamentale tassello per comprendere l'automatizzazione del ragionamento nei sistemi artificiali sviluppata negli ultimi decenni e, nel complesso, la rappresentazione della conoscenza nell'intelligenza artificiale.

1.1.2 Logica proposizionale

Come anticipato in 1.1.1, la logica proposizionale è un linguaggio *formale* poiché astrae e semplifica il contenuto degli enunciati stessi (per esempio: "Piove" e "Prendo l'ombrello"), di cui non vuole verificare la verità (se è vero che sta piovendo), ma la *correttezza* delle condizioni logiche che intercorrono tra essi ("Se piove, allora prendo l'ombrello") [2]. Per questo motivo, questo tipo di logica è anche *simbolico*, poiché utilizza notazioni simboliche artificiali che compongono tre tipi di *alfabeto*:

- *descrittivo*, con simboli per indicare le proposizioni: A, B, C,…;
- *logico*, con simboli per indicare i connettivi: ¬, ∧, ∨, →, ↔;
- *ausiliario*, con simboli accessori: la virgola e le due parentesi (,).

1.1.2.1 Alfabeto descrittivo

Enunciati semplici (*atomici*) come "Piove" sono sostituiti da singole lettere – tipicamente P, Q, R – dette *variabili enunciative* o *proposizionali* che, combinate tra loro, originano proposizioni *molecolari* o *composte* (ossia, contenenti al proprio interno almeno un'altra proposizione), come la già citata "Se piove, allora prendo l'ombrello". La logica proposizionale si occupa di studiare i valori di verità di queste ultime, basandosi su due principi che riguardano gli enunciati semplici:

- *principio di determinatezza:* ogni enunciato ha uno ed un solo valore di verità;
- *principio di bivalenza*: i valori di verità sono vero (V) e falso (F).

A questi si aggiunge il *principio di vero-funzionalità*, che prende invece in considerazione le proposizioni molecolari, per cui lo stato di valore di verità degli enunciati composti dipende interamente da quello degli enunciati semplici che li compongono.

1.1.2.2 Alfabeto logico: i connettivi logici.

I connettivi logici – o *operatori logici enunciativi* – sono le espressioni che consentono di formare enunciati composti *connettendo* tra loro enunciati dati. Sono anche detti connettivi *enunciativi* o *vero-funzionali*, e sono di cinque tipi:

- *negazione*, espresso con «¬», l'unico simbolo *unario*, e non binario, tra tutti i connettivi logici, e avente il significato di *invertire il valore di verità* dell'enunciato negato;
- *congiunzione*, espresso con «∧» e corrispondente al linguaggio naturale «…e…»;
- *disgiunzione*, espresso con «∨». Esso corrisponde al significato *inclusivo* della disgiunzione «o» e non a quello *esclusivo*: pertanto l'enunciato composto P ∨ Q è vero se almeno uno dei suoi disgiunti è vero, senza che sia necessario che lo siano entrambi;
- *condizionale*, espresso con «→» e corrispondente all'espressione «se…, allora…». Un enunciato condizionale è falso nel caso in cui il suo antecedente (la variabile enunciativa a sinistra di "→") sia vero e il conseguente (la variabile enunciativa a destra di "→") falso. Inoltre, è vero anche se il suo antecedente è falso, indipendentemente dal valore di verità del conseguente. Questo tipo di condizionale (a differenza di «↔») è quindi un *condizionale sufficiente*, poiché la verità del suo antecedente è condizione sufficiente per la verità del suo conseguente, ma non necessaria;
- *bicondizionale*, espresso con «↔» e corrispondente a «… se e solo se…». Un enunciato bicondizionale è vero solo se entrambi gli enunciati semplici sono veri o entrambi sono falsi. Esso corrisponde alla congiunzione di due condizionali: (P → Q) ∧ (Q → P). La relazione che lega P e Q è dunque di *condizione necessaria e sufficiente*.

La Tabella 1 esprime la *matrice* di tutti e cinque i connettivi logici poiché mostra il modo in cui il loro valore di verità è interamente determinato in funzione dei valori dei due enunciati semplici che lo compongono (P, Q), e che si trovano nelle prime due colonne, dette per questo *colonne guida*. Questo tipo di tabella, detta *tavola* o *tabella di verità*, venne ideata da Gottlob Frege negli anni '80 del XIX secolo, per poi essere perfezionata nella forma odierna da Ludwig Wittgenstein, nel 1922.

P	Q	¬ Q	P ∧ Q	P ∨ Q	P⟶Q	P↔Q
V	V	F	V	V	V	V
V	F	V	F	V	F	F
F	V	F	F	V	V	F
F	F	V	F	F	V	V

Tabella 1. Tabella di verità dei connettivi logici.

Come già anticipato, la validità del ragionamento non dipende dalla veridicità (o non veridicità) del contenuto delle singole frasi ma dalla sua struttura. Questo si esplicita nel *criterio generale di correttezza logica*: un ragionamento è *corretto* se e solo se dalle sue premesse tutte vere si ottiene una conclusione vera. Reciprocamente, da premesse tutte false si deve ottenere una conclusione falsa. Dunque, più generalmente, un ragionamento è *corretto* se e solo se la sua conclusione è *conseguenza logica* delle sue premesse.

1.1.2.3 Ragionamenti scorretti: un esempio

Il criterio generale di correttezza logica è fondamentale per verificare la correttezza o non correttezza di un ragionamento ad applicazione pratica, come il seguente: “Se il gatto miagola, ha fame. Il gatto non miagola. Quindi non ha fame”.

L’enunciato può essere trasposto nella forma P → Q, ¬P / ¬Q, dove “/” ha il significato di «quindi».

P	Q	P → Q	¬P	/	¬Q
V	V	V	F		F
V	F	F	F		V
F	V	**V**	**V**		**F**
F	F	V	V		V

Tabella 2. Tabella di verità dell’enunciato P → Q, ¬P / ¬Q.

Questo ragionamento è *scorretto* poiché prevede una situazione in cui si ha il passaggio da premesse tutte vere a una conclusione falsa (terza riga della tabella): è quindi un caso

di *fallacia logica*, ossia di scorrettezza che a prima vista appare come correttezza, ma il cui valore di verità viene mostrato dal procedimento meccanico delle matrici.

1.1.2.4 Definizioni

Chiamando fp una qualsiasi forma proposizionale (per esempio, "P ∧ Q") e I una sua interpretazione (ossia l'assegnazione di V o F alle sue proposizioni atomiche P, Q), si dirà che il valore di verità fp^I è *assegnato* a fp da I.

Con questa premessa, possono essere introdotte quattro definizioni importanti [4]:

- *Soddisfacibilità:* è il problema di determinare se una forma proposizionale fp è soddisfacibile, ossia se le variabili in esse contenute possono essere assegnate in modo che la formula assuma il valore VERO. In generale, quindi, se fp^I = VERO, allora si dice che l'interpretazione *soddisfa* fp, o è un *modello* di fp. In tal modo, un insieme di forme proposizionali Γ è soddisfacibile se esiste un'interpretazione che soddisfi tutte le forme proposizionali in Γ. Dunque, Γ *comporta* una forma proposizionale fp (Γ |= fp) se ogni interpretazione soddisfacente Γ soddisfa fp;
- *Tautologia*: fp è una tautologia se ogni interpretazione soddisfa fp;
- *Contraddizione*: fp_1 è una contraddizione se assume il valore FALSO per qualunque valore di verità assegnato alle sue variabili enunciative. È la negazione (¬) di una tautologia;
- *Equivalenza*: due forme proposizionali o set di forme proposizionali (simbolicamente espresse con Γ) sono *equivalenti* tra loro se sono soddisfatte dalle stesse interpretazioni, cioè se e solo se hanno lo stesso valore di verità in corrispondenza di ogni assegnazione di verità alle lettere che compaiono in esse. Dalla precedente definizione di tautologia deriva che due forme proposizionali fp_1 e fp_2 sono equivalenti se e solo se $fp_1 \leftrightarrow fp_2$ è una tautologia.

1.1.3 Logica del primo ordine

1.1.3.1 Sintassi

Nella logica proposizionale analizzata nella sezione precedente era possibile formalizzare argomenti che coinvolgevano quantificatori solo nei casi in cui l'insieme in cui si quantificava era finito. Nel caso di insiemi imprecisati di elementi o di insiemi infiniti, come quelli che necessitano dei quantificatori ∀ ("per ogni ...") e ∃ ("esiste..."),

è la logica del primo ordine a fornire la sintassi e la semantica idonea alla rappresentazione. Essa infatti, a differenza della logica proposizionale, permette di esprimere variabili e quantificazioni, oltre a simboli che rappresentino predicati (cioè possibili proprietà o relazioni), mediante un alfabeto formato da:

- i nuovi quantificatori esistenziale (∃) e universale (∀), che vanno ad aggiungersi ai simboli di punteggiatura (le parentesi "(" e ")" e la virgola ",") e ai connettivi logici già propri della logica proposizionale (¬, ∧, ∨, → e ↔). Anche in questo alfabeto vi sono regole di precedenza, in particolare prima ¬, ∃, ∀, poi ∧, ∨, e infine →, ↔;
- simboli per *costanti individuali*, espressi con lettere minuscole corsive. Le costanti individuali, in particolare, sono singole entità del dominio del discorso (per esempio: "marco", "genova", …);
- simboli per *variabili* (infiniti), ossia entità non note del dominio;
- simboli per *costanti predicative*, o *lettere di predicazione*. Espressi con lettere maiuscole corsive, questi predicati n-ari rappresentano una generica relazione fra "n" oggetti del dominio del discorso. Per esempio: padre(giovanni, luca), da leggersi come "giovanni è il padre di luca".
- simboli di funzione, in cui una funzione n-aria individua univocamente un oggetto del dominio del discorso mediante una relazione tra altri "n" oggetti del dominio. Per esempio, "l'individuo *m* ha proprietà *F*" si renderà come *F*(*m*).

Date queste definizioni principali, si possono dunque definire:

- *termine*: una variabile è un termine, una costante è un termine, e se f è un simbolo di funzione e $t_1,\ldots t_n$ sono termini, allora anche $f(t_1,\ldots,t_n)$ è un termine (esempi: "f(X)", "giovanni");
- *atomo*: l'applicazione di un simbolo di predicato p a n termini $t_1,\ldots t_n$. La sua forma è $p(t_1,\ldots t_n)$ (per esempio, "padre(giovanni, luca)")
- *espressione* o *formula*: sequenza di simboli appartenenti all'alfabeto. È opportuno segnalare che esistono formule dette ground, ossia formule che non contengono variabili, di cui si tratterà più ampiamente nel Capitolo 3;
- *formule ben formate* (fbf): frasi sintatticamente corrette del linguaggio, ottenute con la combinazione di formule atomiche, connettivi e quantificatori;
- *letterale*; fbf atomica o la sua negazione.

1.1.3.2 Semantica

La differenza fondamentale che si ha tra linguaggi ordinari come l'italiano e linguaggi formalizzati è che i primi sono linguaggio *già* interpretati, ossia le cui espressioni si presentano come già dotate di significato. Le formule dei linguaggi verbali vengono invece intese come pure sequenze di simboli, costruite in base alle regole sintattiche di formazione. Pertanto, per attribuire un significato alle formule del linguaggio, occorre indicare un *dominio* o *universo del discorso*, ossia un insieme di riferimento *U* a cui appartengono gli "oggetti" presi in considerazione e denotati dalle costanti individuali, e in cui operano i quantificatori. Dunque, un *modello M* per un linguaggio formale L è una coppia ordinata M = <U, *i*> dove *U* è l'insieme sopra citato e *i* è una *funzione di interpretazione*, ossia una funzione che assegna significati a espressioni del linguaggio formale L. In altre parole, un'*interpretazione* è la costruzione di un rapporto fra i simboli del sistema formale e il dominio del discorso. Questa definisce un dominio non vuoto D e assegnerà significati ai simboli come segue:

- Per ogni simbolo di costante individuale *k*, il significato a essa associato nel modello *M* (scritto *M*(*k*)) sarà un certo individuo appartenente all'insieme *U*;
- A ogni simbolo di funzione n-ario *f*, sarà associata una funzione: $D^n \rightarrow D$;
- A ogni costante predicativa n-aria *p*, una relazione in D^n, cioè un sottoinsieme di D^n.

In questo modo, si fissa univocamente il significato delle costanti descrittive del linguaggio predicativo L: la semantica che si costruisce è dunque *referenzialista*, ossia assume che il significato delle espressioni subenunciative di L consista nel riferimento a oggetti o insiemi di oggetti del dominio *U*.

Per esempio, prendendo un linguaggio del primo ordine L, nel quale si ha una costante "0", un simbolo di funzione unaria "s" e un simbolo di predicato binario "p", si possono avere infinite interpretazioni, come le due seguenti, simili a quelle discusse in [3]:

Interpretazione I1.

Il dominio D è quello dei numeri naturali.

"0" rappresenta il numero zero.

"s" rappresenta il successore di un numero naturale.

"p" rappresenta la relazione binaria "≤".

Interpretazione I2.

Il dominio D è quello dei numeri interi negativi.

"0" rappresenta il numero zero.

"s" rappresenta il predecessore di un numero naturale.

"p" rappresenta la relazione binaria "≤"

1.1.3.3 Valori di verità

Data un'interpretazione il valore di verità di una fbf si diversifica a seconda che essa sia una formula atomica "ground", una formula composta, una formula quantificata esistenzialmente o una formula quantificata universalmente.

Nel caso di una *formula atomica "ground"*, essa ha valore vero sotto un'interpretazione quando il corrispondente predicato è soddisfatto (cioè quando la corrispondente relazione è vera nel dominio). Viceversa, ha valore falso. Esempi:

Interpretazione I1

- p(0, s(0)). Questa fbf ha valore vero, poiché equivale a dire che esiste sempre il successore di numero naturale (s(0)) che sia minore o uguale a 0.
- p(s(0), 0). Questa fbf ha valore falso, poiché significa che non può esistere un successore di numero naturale (s(0)) che sia minore o uguale a 0.

Interpretazione I2

- p(0, s(0)). Viceversa, in questa interpretazione, la stessa fbf già analizzata risulta con valore falso (non esiste un numero intero negativo che sia maggiore o uguale a 0).
- p(s(0), 0). Qui, invece, il valore è vero: qualunque numero intero negativo è minore o uguale a 0.

Nel caso di una *formula composta*, il suo valore di verità rispetto a un'interpretazione si ottiene da quello delle sue componenti utilizzando le tavole di verità dei connettivi logici (si veda Tabella 1, sezione 1.1.2.2). Per esempio, la formula F p(s(0),0) → p(0,s(0)) ha valore vero nell'*interpretazione I1*, poiché l'antecedente ha valore falso, mentre ha valore falso in *I2*, poiché a un antecedente vero corrisponde un conseguente falso.

Si prenda dunque una *formula quantificata esistenzialmente*: una formula del tipo ∃X F è vera in un'interpretazione se esiste almeno un elemento *d* del dominio D tale che la formula F', ottenuta assegnando *d* alla variabile X, è vera in I. In caso contrario F ha valore falso.

Per esempio, la formula ∃X p(X,s(0)) ha valore vero nell'interpretazione *I1* in quanto esiste un numero naturale, zero, minore di uno, tale che la formula F'=p(0,s(0)) abbia valore vero in *I1*.

Infine, una *formula quantificata universalmente* (del tipo ∀X) è vera in un'interpretazione I se per ogni elemento *d* del dominio D, la formula F', ottenuta da F sostituendo *d* alla variabile X, è vera in I. Altrimenti F ha valore falso.

Per esempio, ∀Y p(0,Y) ha valore vero rispetto alle interpretazioni *I1* (dove viene interpretata come "0 è minore o uguale a ogni intero positivo Y"), mentre ha valore falso rispetto a *I2* poiché esiste almeno un elemento del dominio che la falsifica (per esempio, non è vero che "0 è minore o uguale a –1").

1.1.3.4 Modelli e semantica

Visti gli esempi precedenti, si può dare una formulazione più chiara del concetto di *modello* proposto in 1.1.3.2.

Data una interpretazione I e una fbf chiusa F, I è un *modello* per F se e solo se F è vera in I. Per esempio, per la fbf ∀Y p(0,Y) l'interpretazione *I1* è un modello, mentre *I2* non lo è.

Inoltre, una fbf è *soddisfacibile* se e solo se è vera almeno in una interpretazione, ovvero se esiste almeno un modello per essa. Allo stesso modo, un insieme di formule chiuse del primo ordine S è soddisfacibile se esiste una interpretazione I che soddisfa tutte le formule di S (cioè che è un modello per ciascuna formula di S). Tale interpretazione è detta modello di S. Al contrario, un insieme di formule S che non può essere soddisfatto da alcuna interpretazione, è detto *insoddisfacibile* (o inconsistente); ad esempio l'insieme di formule {F, ¬F} è insoddisfacibile.

Infine, una fbf che ha valore vero per tutte le possibili interpretazioni, cioè per cui ogni possibile interpretazione è un modello, è detta *logicamente valida*.

Per esempio, la fbf: ∀X p(X) ∨ ¬(∀Y p(Y)) è logicamente valida. Infatti, le formule ∀X p(X) e ∀Y p(Y) sono semplici varianti della stessa formula F e quindi hanno i medesimi valori di verità per qualunque interpretazione. In generale, F ∨ ¬F ha sempre valore vero, in modo indipendente dall'interpretazione.

Una formula F segue logicamente (o è conseguenza logica) da un insieme di formule S (scritto: S |= F), se e solo se ogni interpretazione I che è un modello per S, è un modello per F.

1.1.3.5 Monotonicità

Un'altra proprietà fondamentale delle teorie del primo ordine è la monotonicità.

In particolare, una teoria T è monotona se l'aggiunta di nuovi assiomi non invalida i teoremi trovati precedentemente. Della monotonicità si dirà più lungamente nella sezione successiva.

1.2 Linguaggi non classici: la logica non monotona

L'invenzione dei calcolatori elettronici ha aperto la strada all'eventualità di usare la logica come linguaggio per l'intelligenza artificiale: un'eventualità che è poi divenuta realtà. Pur avendo creato rigorosi linguaggi formali come la logica proposizionale e la successiva logica del primo ordine, tuttavia, è stato subito chiaro ai ricercatori nel campo dell'IA che esistono le *eccezioni*.

Come specificato nel paragrafo 1.1.3.5, la logica classica è infatti *monotona*, nel senso che ogni affermazione inizialmente vera sarà vera anche dopo l'aggiunta di un nuovo assioma qualsiasi. In altre parole, aggiungere informazioni non invalida la conclusione.

Tuttavia, come si sperimenta continuamente nella vita quotidiana, esistono ragionamenti *non monotoni*, per cui informazioni aggiuntive possono invalidare le conclusioni. Un classico esempio è "gli uccelli volano". Questo è naturalmente vero, ma come propone [5], esistono eccezioni: i pinguini, gli emù, i tacchini e altre specie ancora. Formalizzare una teoria di uccelli come animali volanti scrivendo

$$\forall x\ (\text{Uccello}(x) \rightarrow \text{Vola}(x))$$

è errato, poiché non tiene conto delle varie eccezioni. Si può quindi perfezionare la formula in

$$\forall x\ (\text{Uccello}(x) \wedge \neg\ \text{Pinguino}(x) \wedge \neg\ \text{Emù}(x) \wedge \neg\ \text{Tacchino}(x) \rightarrow \text{Vola}\ (x)),$$

che significa che tutti gli uccelli volano, a meno che non siano pinguini, emù o tacchini. Dal punto di vista rappresentazionale, è meglio dividerlo in tre formule:

$$\forall x\ (\text{Uccello}(x) \wedge \neg\ \text{Anormale}(x) \rightarrow \text{Vola}\ (x)),$$

$$\forall x\ (\text{Pinguino}(x) \rightarrow \text{Anormale}(x)),$$

$\forall x$ (Emù(x) → Anormale(x)),

$\forall x$ (Tacchino(x) → Anormale(x))

che mostrano come tutti gli uccelli volino, a meno che non siano anomali, e che pinguino, emù e tacchino sono anomali.

La tipologia di queste ultime quattro formule è stata ideata negli anni Ottanta del secolo scorso da John McCarthy, informatico statunitense Premio Turing nel 1971, e prevede che il nuovo predicato Anormale(x) è considerato falso se non può essere provato che sia vero. È quello che, più o meno inconsciamente, svolge già la mente umana. Per esempio, si ponga Skipper un uccello: chiedendo a un umano se esso voli, è probabile che egli risponda di sì, poiché assumerà che se un particolare non gli è stato menzionato, significa che esso è falso. Questo stesso ragionamento, tuttavia, non vale per un sistema artificiale: se esso sa soltanto che Skipper è un uccello, non può usare la formula sopra descritta all'interno di un ragionamento deduttivo, poiché è necessario che sappia che Skipper non è né un pinguino né un emù né un tacchino. Questo ragionamento è non monotono, dunque, perché aggiungere informazioni (come, per esempio, che Skipper sia un pinguino) può significare che la conclusione (ossia che Skipper possa volare) venga ritrattata.

Il sistema artificiale appena menzionato si muove all'interno di quella che è definita "*ipotesi del mondo chiuso*" (nota anche in inglese come closed-world assumption) dall'informatico e logico canadese Raymond Reiter nel 1978, per cui ogni affermazione il cui valore di verità non è noto è considerata falsa.

Il "mondo", quindi, è "chiuso" nel senso che ogni elemento che esiste, è stato definito nel programma o può essere derivato da esso; di conseguenza, un elemento che non si trova nel programma, non è vero ed è vera la sua negazione.

Sulla base della conoscenza di "Il pinguino è un uccello", alla domanda "Il pinguino è un animale?", un sistema basato sull'assunzione del mondo chiuso non potrà far altro che rispondere, dunque, "No".

A quest'assunzione è strettamente legata una regola di inferenza nota come *negation as failure* o *negation by default* (in italiano, "negazione come fallimento", simbolo "not"), che equivale a credere in falso ogni predicato che non può essere dimostrato vero. Più precisamente, essa è definita nel seguente modo: "*dato il predicato not(p), se p ha successo, allora not(p) fallisce, altrimenti not(p) ha successo*", in cui p è un atomo

che non si può dedurre automaticamente dal programma (poiché non è conseguenza logica dei fatti e delle regole contenute nel programma).

La negation as failure può sembrare simile al "¬" della logica proposizionale, ma le due sono molto diverse. Il migliore esempio esplicativo viene dal già citato John McCarty [6], che propone uno scuolabus fermo di fronte alle rotaie di un treno: può attraversarle solo se non c'è un treno in avvicinamento. La rappresentazione di questa situazione per mezzo della negation as failure è

not Train → Cross

e non è adeguata, perché suggerisce che lo scuolabus possa attraversare in assenza di informazioni a proposito di un treno in avvicinamento. Ossia, se fosse un programma, lo scuolabus effettuerebbe una ricerca all'interno dei suoi fatti e delle sue regole, ottenendo tre possibili risultati: vero ("treno in avvicinamento"), falso ("treno non in avvicinamento") e un "non si sa" che verrebbe registrato automaticamente come falso. Lo scuolabus attraverserebbe le rotaie, dunque, in assenza dell'informazione di pericolo.

¬ Train → Cross

è invece una rappresentazione adeguata, poiché indica che lo scuolabus può attraversare se non c'è alcun treno in avvicinamento.

1.2.1 Il frame problem

Il frame problem (variamente tradotto con "problema del cambiamento" o "problema della cornice") venne definito da John McCarthy e Patrick J. Hayes nell'articolo *Some Philosophical Problems from the Standpoint of Artificial Intelligence* (1969) [7] e si occupa di esprimere i fatti che persistono (ossia, che non si modificano) all'interno di un ambiente di intelligenza artificiale.

Per esempio, il fatto che una finestra venga aperta, presuppone che, intanto, il colore delle pareti e la posizione delle mattonelle non cambino. Il frame problem si interroga dunque su come rappresentare la grande quantità di *non modifiche* che permangono quando occorre un'azione.

Si veda l'esempio della stessa finestra che si apre e di un telefono che suona: in questo caso le due proposizioni sono *aperta* e *suona*. Poiché questi due predicati possono cambiare, sono detti fluenti, ed espressi con *aperta(t)* e *suona(t)*, in quanto dipendono

dal tempo (t). Si ponga una situazione (0) in cui il telefono non suona e la finestra è chiusa, e una situazione (1) in cui la finestra si apre, ma il telefono rimane silenzioso:

¬ aperta(0)

¬ suona(0)

vero → aperta(1)

Queste tre formule non sono sufficienti per trarre conseguenze. Si può infatti ottenere la situazione effettivamente voluta:

¬ aperta(0) aperta(1)

¬ suona(0) ¬ suona(1)

ma anche un altro insieme di condizioni, coerente alle tre formule precedenti:

¬ aperta(0) aperta(1)

¬ suona(0) suona(1)

Il frame problem può essere risolto aggiungendo dei *frame axioms* (assiomi): essi specificano che, durante l'esecuzione di un'azione, tutte le condizioni non influenzate esplicitamente da tale azione, non vengono modificate. Nel caso proposto, un esempio può essere:

suona(0) ↔ suona(1)

che indica che lo stato della finestra non cambia dal tempo 0 al tempo 1.

Altra possibilità è usare una regola non monotona come una regola di persistenza come: ciò che si mantiene tipicamente in una situazione, si mantiene nella situazione dopo che un'azione è stata eseguita, a meno che non contraddica la descrizione degli effetti dell'azione.

Una delle soluzioni più complete, inoltre, prevede la formalizzazione del principio di inerzia (ossia la persistenza delle condizioni nel tempo) e l'inserimento della possibilità di "occlusione": viene pertanto chiamata Fluent occlusion solution, o Soluzione di occlusione del fluente.

1.2.1.1 Soluzione di occlusione del fluente

Fu il professore svedese Erik Sandewall a proporre questa soluzione [8,9]. Sandewall decise di rappresentare non solo il valore delle condizioni nel tempo (t), ma anche la loro possibilità di essere influenzate dall'ultima azione eseguita: possibilità, questa, definita "occlusione", e che può essere vista come una sorta di "permesso di cambiare".

Se in un dato tempo (t) l'effetto di un'azione appena eseguita rende una condizione vera o falsa, questa condizione si dice *occlusa*. Una condizione occlusa è dunque sollevata dall'obbedire al vincolo di inerzia, cui tutte le altre condizioni continueranno, invece, a sottostare.

Nell'esempio proposto, l'occlusione può essere indicata con i predicati *occlusaaperta(t)* e *occlusasuona(t)*. Il fondamento logico è che il predicato di occlusione è vero solo quando viene eseguita un'azione che influenza la condizione; a sua volta, una condizione può cambiare valore.

¬ aperta (0)

¬ suona(0)

vero → aperta(1) ∧ occlusaaperta(1)

∀t ¬occlusaaperta (t) → (aperta(t – 1) ↔ aperta(t))

∀t ¬occlusasuona (t) → (suona(t – 1) ↔ suona(t))

In questo caso, *occlusaaperta(1)* è vero, rendendo l'antecedente (si ricorda: ciò che sta a sinistra del condizionale "→") della quarta formula falso per t = 1. In tal modo, *aperta(t – 1)* ↔ *aperta(t)* non si mantiene per t = 1.

Ovviamente, l'occlusione non implica necessariamente che avvenga un cambiamento, come nel caso in cui si esegua l'azione di aprire la finestra quando era già aperta: il predicato *occlusaaperta* diviene vero e rende *suona* vero, ma quest'ultimo non ha cambiato valore, essendo già in precedenza vero.

Capitolo 2

Answer Set Programming

2.1 Introduzione

I concetti che sono stati introdotti nel Capitolo 1 e che rientrano nell'ambito della logica monotòna (logica per default, ipotesi del mondo chiuso, frame problem, negation as failure) costituiscono la base dell'Answer Set Programming (ASP), una forma di programmazione dichiarativa che utilizza la logica per la risoluzione di problemi complessi, come la pianificazione (planning), l'ottimizzazione e la rappresentazione della conoscenza.

L'ASP nasce nel 1999 quando viene identificata come specifico paradigma di programmazione nella pubblicazione "*Stable models and an alternative logic programming paradigm*" [10] degli studiosi polacchi Victor W. Marek (1943 –) e Jan Truszczyński (1949 –), e il linguaggio da essa impiegato è AnsProlog (Answer Set Programming in Logic). AnsProlog è un sottoinsieme del Prolog ideato nel 1972 dall'informatico francese Alain Colmerauer (1941 – 2017) [11]: la sua sintassi consente la rappresentazione dei valori di default e delle loro eccezioni (operazioni inattuabili dalla logica classica) e i suoi meccanismi di inferenza permettono a un programma di trovare un set di valide risposte, ossia conclusioni, in un lasso di tempo finito e ragionevole. Inoltre, AnsProlog può anche includere affermazioni sugli effetti causali delle azioni ("l'affermazione A diventa vera a seguito dell'esecuzione dell'azione B", come già mostrato in altri termini nell'occlusione del fluente, in 1.2.1.1.), affermazioni che esprimono una mancanza di informazioni (come: "non è noto se l'affermazione A è vera o falsa") e altre ancora.

Come semantica, ASP utilizza la semantica del modello stabile (answer set semantics).

2.2 Semantica del modello stabile

La semantica del modello stabile è una semantica per programmi come ASP che si compongono di insiemi di regole. Queste regole sono formate da atomi e letterali. Un letterale è un atomo *a* o la sua negazione not *a*. Per esempio, la regola *r*:

a ← b, not c.

è interpretata come "se *b* è conosciuto e *c* non è conosciuto, allora *a* è conosciuto", e può anche essere scritta come:

a :- b, not c.

L'atomo *a* è detto testa ($H(r)$, o head) della regola e {*b*, not *c*} ne è il corpo ($B(r)$, o body): esso può essere anche vuoto, e in questo caso le regole sono chiamate *fatti*. Quando il body non è vuoto, si divide in un body negativo $B^-(r)$ che comprende gli atomi negati, nell'esempio sopra proposto {*c*}, e un body positivo $B^+(r)$, in questo caso {*b*}. Di conseguenza, ci possono anche essere programmi (identificati dalla lettera greca *Π*) positivi, ossia che non contengono negazioni nelle loro regole. Questi sono i più semplici e incontrovertibili: per ricavare le loro soluzioni, o insiemi di risposte (answer set) si applica consequenzialmente un operatore detto T_P, che comprende soltanto l'head e il body positivo delle regole appartenenti al programma.

Dal punto di vista analitico, ciò corrisponde a iniziare senza alcuna conoscenza e quindi apprendere informazioni solo quando vengono soddisfatte le condizioni di una regola (e che sono contenute nel suo body).

Di seguito si riporta un esempio simile a quello discusso in [12]: dato il programma

a :- c.

b :- d.

c :- .

d :- c, a.

e :- b.

f :- g.

g :- f.

Il suo modello stabile è dato dall'applicazione consequenziale dell'operatore T_P:

$T_P(\emptyset) = \{c\}$

$T_P(\{c\}) = \{c, a\}$

$T_P(\{c, a\}) = \{c, a, d\}$

T_P ({c, a, d}) = {c, a, d, b}

T_P ({c, a, d, b}) = {b, a, c, d, e}

T_P ({c, a, d, b, e}) = {b, a, c, d, e}

Naturalmente, *f* ed *g* non sono inclusi nel modello perché non esiste una "ragione" indipendente per cui dovrebbero essere considerati noti.

Si hanno dunque le due definizioni formali:

Sia M un modello, M è minimale se e solo se non esiste un modello M_1 sottoinsieme di M.

Sia Π un programma positivo, allora un modello è stabile se e solo se è un modello minimale di Π.

Nel caso in cui, invece, un programma contenga anche regole con body negativi, bisognerà effettuare un passaggio intermedio, ossia la riduzione di Gelfond-Lifschitz, che genera un nuovo insieme di regole a partire dal programma di partenza, ma rimuovendo ogni regola che dipenda dalla negazione di un atomo del set di risposte e lasciando cadere tutte le altre dipendenze negative. Il concetto di riduzione si basa sul fatto che i body negativi contengano non un tipo di negazione classica ("si sa che x non è vero") ma una negation as failure ("non si sa se x è vero").

Per esempio, se Π è il seguente programma:

a :- not b.

b :- not a.

c :- not d.

e :- a, c, not b.

f :- not g, e.

g :- not f, e.

Allora il set {*b*, *c*} dà il programma ridotto $\Pi^{\{b,c\}}$

b :- .

c :- .

f :- e.

g :- e.

Intuitivamente è semplice: poiché la prima regola si interpreta come "se *b* non è conosciuto, allora *a* è conosciuto", e dato che *b* è conosciuto dal set di risposte, allora *a* non è conosciuto, e l'intera regola viene pertanto eliminata.

Per verificare che il modello stabile della riduzione $\Pi^{\{b,c\}}$ sia proprio $\{b, c\}$ basta applicare l'operatore T_P sulla riduzione $\Pi^{\{b,c\}}$, ottenendo rapidamente il set di risposte $\{b, c\}$.

Dal punto di vista formale, dunque, si definisce un answer set nel seguente modo:

Sia M un modello classico del programma Π, M è un answer set di Π se e solo se M è un modello minimale del ridotto di Π rispetto a M.

Per concludere, in ASP le regole acquistano la forma:

$h \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m.$

che è corrispondente alla seguente:

$h : - a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m.$

2.3 ASP con variabili

Tutti i casi proposti in esame nelle sezioni precedenti erano *grounded*, ossia istanziati: non presentavano pertanto variabili all'interno delle regole o dei fatti. Una regola si definisce, al contrario, *non ground*, quando contiene variabili non istanziate.

Per esempio, la definizione della relazione "nonno" tra tre enti (nonno, padre, figlio) può costituirsi a partire dai due fatti:

padre (alfredo, giovanni)

padre (giovanni, luca)

che corrispondono a "Alfredo è il padre di Giovanni" e "Giovanni è il padre di Luca". Da qui, si ottiene la regola:

nonno (alfredo, luca) :- padre (alfredo, giovanni), padre (giovanni, luca).

Da questa, è possibile ottenere una generalizzazione sostituendo agli enti "alfredo", "luca" e "giovanni" delle variabili scelte arbitrariamente. I fatti risulteranno quindi:

padre (X, Y)

padre (Y, Z)

da leggersi come "X è il padre di Y" e "Y è il padre di Z". La regola generalizzata corrispondente diventa pertanto:

nonno (X, Z) :- padre (X, Y), padre (Y, Z).

con il significato di "X è nonno di Z se X è padre di Y e Y è padre di Z".

Più formalmente, un atomo non ground è della forma $p(t_1,\ldots,t_n)$, dove p rappresenta un predicato, mentre $t_1,\ldots,t_n$ sono termini. I termini possono essere costanti e variabili,

dove le costanti rappresentano individui noti (ad esempio, luca) mentre le variabili rappresentano individui generici (ad esempio, X). Per convenzione, le costanti iniziano per lettera minuscola, mentre le variabili per lettera maiuscola.

Come riportato in [13], per definizione, dato un programma Π, un termine di Herbrand è un termine base (*ground term*) ovvero un termine che non contiene variabili. Un atomo di Herbrand, invece, è un atomo base (*ground atom*), ovvero un atomo che non contiene variabili. Entrambi prendono il nome da Jacques Herbrand, matematico francese dei primi anni del Novecento.

Similmente, l'*Universo di Herbrand* U_Π è l'insieme di tutti i termini di Herbrand o, in altre parole, di tutte le costanti che appaiono in Π. E la *Base di Herbrand* B_Π è l'insieme di tutti gli atomi di Herbrand, ossia di tutti gli atomi ground possibili che possono essere costruiti dai simboli predicati che appaiono in Π con le costanti di U_Π.

Data una regola *r*, *Ground*(*r*) denota l'insieme di regole ottenute applicando tutte le possibili sostituzioni σ dalle variabili in *r* agli elementi di U_Π. Allo stesso modo, dato un programma Π, l'istanziazione ground *Ground*(Π) di Π è l'insieme $U_{r \in \Pi}$ *Ground* (*r*).

Quindi la valutazione di un programma ASP Π con variabili si basa su due fasi: nella prima, Π è trasformato nel programma *Ground*(Π) e nella seconda sono calcolati i modelli stabili di *Ground*(Π), come descritto nella sezione precedente.

2.4 Soluzione al frame problem

Answer Set Programming può inoltre fornire una valida soluzione al frame problem descritto nel Capitolo 1, ancora più semplice di quella di occlusione del fluente poiché spiegata da una sola regola:

```
r(X, T + 1) :- r(X, T), not r(X, T + 1).
```

che logicamente significa che se r(X) è vero al tempo T, e si può assumere che r(X) rimanga vero al tempo T + 1, allora si può concludere che r(X) rimanga vero. In questo modo tutte le azioni che si presuppone non si modifichino nel compiersi di una determinata azione, vengono semplicemente ignorate.

Capitolo 3

Applicazioni

3.1 Introduzione

Come già affermato nel Capitolo precedente, Answer Set Programming è un linguaggio dichiarativo con l'abilità di fornire specifiche formali pronte per l'esecuzione di problemi anche molto complessi, come i problemi di ottimizzazione. Per questo motivo, i vantaggi chiave di ASP come tecnica di risoluzione sono due: i programmi sono compatti e veloci da scrivere e il programmatore può concentrarsi sulla descrizione del problema piuttosto che sulla progettazione dell'algoritmo di ricerca.

A partire dalla seconda metà degli anni '90, e ancor più negli ultimi anni, sono pertanto stati rilasciati numerosi sistemi ASP. Una delle loro prime applicazioni significanti nel mondo reale è stato lo sviluppo dell'USA-Advisor, un sistema di supporto decisionale per i controllori di volo dello Space Shuttle [14], che venne utilizzato dalla NASA per manovrare la navicella in orbita. Successivamente, si sono susseguite molte altre applicazioni scientifiche, nell'ambito dell'Intelligenza Artificiale [15, 16, 17], dell'Information Integration [18] e nel Knowledge Management [19, 20, 21], in particolare in alcune sue aree, come il Text Classification, l'Information Extraction e l'Ontology Representation and Reasoning. Numerose sono state, inoltre, le applicazioni nel mondo reale e nell'industria [22]: in questa sezione si analizzano dunque alcune di esse, che confermano l'efficienza di ASP come semplice e conciso strumento di problem-solving e di ragionamento su attività avanzate basate sulla conoscenza.

3.2 Team-building nel porto di Gioia Tauro

Questa applicazione [23, 24] si basa su DLV, uno dei sistemi ASP più popolari nel settore del Knowledge Management, creato nel 1997 da un team di ricerca Italo-Austriaco (in cui sono state coinvolte l'Università della Calabria e e la Vienna University of Technology) [25]. Utilizzando la stessa logica di ASP, il sistema DLV supporta un

linguaggio basato su formalismi logici che consentono ai programmi di rappresentare problemi pratici anche in presenza di conoscenze incomplete o contraddittorie.

DLV è stato utilizzato dall'autorità portuale di Gioia Tauro per risolvere un problema di team-building e allocazione intelligente delle risorse. In particolare, il porto di Gioia Tauro è una risorsa strategica nel cuore del Mediterraneo: oltre a essere il più grande porto in Italia per il volume di container trasportati, nonché il sesto nel Mediterraneo, esso si trova nel corridoio che va da Suez a Gibilterra, uno dei più trafficati al mondo. È inoltre un nodo fondamentale per il trashipment, ossia il trasbordo di merci da nave a nave, poiché collega le reti globali e regionali che attraversano il Mediterraneo: navi di ogni genere e dimensione vi transitano quotidianamente, e le merci trasportate devono essere gestite, immagazzinate, eventualmente trattate e in seguito rilasciate verso la loro destinazione finale.

Il particolare dinamismo e la continuità operativa del porto di Gioia Tauro necessitano pertanto di sistemi di ottimizzazione al fine di garantire la qualità dei servizi e il completamento delle operazioni di sbarco e imbarco di merci e container nel modo più efficiente e, soprattutto, nel minor tempo possibile. Questo, naturalmente, implica una continua disponibilità delle risorse tecnologiche di movimentazione dei container e di efficienza nell'impiego delle risorse umane disponibili: ottimizzare, infatti, implica gestire al meglio i flussi, ridurre i costi e migliorare il controllo e l'integrazione dei diversi servizi, incrementando di conseguenza la competitività e l'innovazione dei processi gestionali ed operativi, siano essi relativi alla logistica portuale interna che alla logistica distributiva, che si interfaccia continuamente con compagnie di navigazioni provenienti da tutto il mondo.

Una delle sfide maggiori che ha dovuto intraprendere l'autorità portuale di Gioia Tauro è stata proprio quella del team-building: i team di dipendenti, infatti, devono essere in grado di gestire in modo adeguato il volume di merci e container giornalmente in arrivo, anche quando i dati relativi alle navi che attraccano nel porto (ad esempio: data di arrivo, data di partenza, destinazione, numero e tipo di merci) sono resi noti con poco più di un giorno di arrivo. Non è da dimenticare, inoltre, che gruppi di dipendenti diversi possiedono abilità diverse, e per questo devono essere assegnati a lavori per i quali dispongono delle giuste competenze.

L’allocazione delle risorse umane rappresenta quindi un serio problema di ottimizzazione, poiché necessita di tenere presente varie costrizioni e condizioni, come il fatto che un impiegato non possa lavorare per più di 36 ore alla settimana, che i lavori pesanti o pericolosi debbano necessitare di un ricambio più frequente di personale o, ancora, che debba essere garantita un’equa distribuzione dei compiti da svolgere. Si noti anche che ogni impiegato potrebbe ricoprire più ruoli a seconda delle proprie competenze, e che persistono differenze di ruoli e competenze anche a livello piramidale all’interno del team stesso.

Ognuno dei vincoli sopra citati deve essere soddisfatto quotidianamente e, come si è detto, spesso con margini temporali per l’elaborazione dei team estremamente ridotti. L’impossibilità di assegnare adeguati gruppi di impiegati potrebbe causare ritardi o violazioni del contratto con compagnie di navigazione, con conseguenti sanzioni pecuniarie per la compagnia che gestisce il porto. Da qui si deduce come sia necessario un programma in grado di affrontare e risolvere complessi problemi di ottimizzazione, e quel programma è stato trovato nell’ASP sotto forma del sistema DLV. Giornalmente, una volta che viene acquisita l’informazione relativa all’attracco delle navi, infatti, il DLV produce facilmente una specifica allocazione indicando il numero di impiegati richiesti per ogni competenza. Semplificando il suo kernel, gli input che necessita sono:

- gli impiegati e le loro competenze e abilità;
- gli impiegati assenti;
- gli impiegati esclusi a causa di una decisione di management;
- le statistiche settimanali che specificano, per ogni impiegato, sia il numero di ore lavorate sia l’ultima data di impiego;
- la specificazione di un turno per cui un team deve essere impiegato;
- il numero degli impiegati necessari per una competenza particolare e che copre lo stesso turno di lavoro;
- la quantità totale di ore lavorative a settimana per impiegato.

Impiegati assenti o esclusi, insieme con impiegati che eccedono il numero massimo di ore lavorative settimanali, vengono automaticamente eliminati. In seguito, le soluzioni ammissibili vengono selezionate in base ai vincoli immensi nel programma: con il primo, si eliminano le assegnazioni con un numero errato di impiegati per una qualunque competenza; con il secondo, si evita che un impiegato copra due ruoli nella

stessa competenza; dopodiché, si implementa l'avvicendamento dei ruoli; e, infine, si garantisce un'adeguata distribuzione del carico di lavoro fra tutti i membri dei team.

Poiché viene riportato solo il nucleo del programma in una forma semplificata, si specifica che in realtà sono stati testati e sviluppati molti altri vincoli.

Il sistema completo è dotato di una Graphical User Interface (GUI) sviluppata in Java, ed è in grado sia di costruire nuovi team, sia di completare automaticamente l'allocazione dopo che i ruoli di alcuni impiegati chiave sono stati fissati manualmente. Viceversa, team calcolati al computer possono anche essere modificati manualmente, e il sistema è in grado di verificare se il team modificato manualmente soddisfa ancora i vincoli. In caso di errori, vengono evidenziate le cause e vengono proposti suggerimenti per risolvere il problema. Ad esempio, se non può essere generata una pianificazione, il sistema suggerisce di allentare i vincoli.

La possibilità di modificare in pochi minuti un ragionamento logico complesso, ad esempio cambiando o aggiungendo nuovi vincoli, e di testarlo *in situ* in coordinazione con il cliente, si è rivelata un notevole vantaggio, in cui ASP ha giocato un ruolo chiave e vincente.

3.3 e-Tourism: il progetto IDUM

Dalla sua nascita nei primi anni Sessanta del secolo scorso fino ad alcuni anni fa, il turismo di massa è rimasto cristallizzato in un'asimmetria informativa, per cui un turista in procinto di partire aveva a disposizione dei mezzi limitati e un numero esiguo di fonti per il reperimento delle informazioni necessarie a pianificare il proprio viaggio. Oggi, tuttavia, il profilo del turista stesso e delle agenzie di viaggi è profondamente cambiato, contribuendo al passaggio da un turismo tradizionale ad un *e-tourism* in cui il binomio tra turismo e tecnologia è inscindibile. Internet si configura, infatti, come il mezzo prioritario per tutte le attività che precedono e seguono un viaggio: la ricerca di informazioni, la comparazione di prezzi e alternative economiche, la comunicazione interattiva, il rilascio di recensioni e opinioni sulle mete, la prenotazione di strutture e pacchetti di viaggio e molte altre ancora.

È in questo contesto che si inserisce IDUM (progetto "IDUM: Internet Diventa Umana"), sistema di e-tourism finanziato dall'amministrazione della Regione Calabria [25, 26, 27]. Il sistema IDUM aiuta sia i dipendenti che i clienti di un'agenzia di viaggi

a trovare la migliore soluzione di viaggio possibile in un breve lasso di tempo: non si tratta pertanto di un sistema chiamato a sostituire l'attività dei tour operators e delle web agencies, ma di un "mediatore", che trova la migliore corrispondenza tra le loro offerte e le richieste dei turisti. La sterminata quantità di informazioni di cui dispone Internet, infatti, apre spazio a infinite possibilità combinatorie, in un'online search che può risultare troppo ampio e dispendioso in termini di tempo per il cliente.

IDUM, come altri portali esistenti, è dotato di un'appropriata interfaccia utente, ma dietro di essa un nucleo intelligente sfrutta la rappresentazione della conoscenza e le tecnologie di ragionamento basate su Answer Set Programming. Il funzionamento del sistema è semplice: le informazioni relative alle offerte turistiche fornite dai tour operator vengono ricevute da IDUM come un insieme di e-mail, ognuna delle quali può contenere testo semplice ma anche file pdf o file immagine rappresentati da volantini che, poiché sono concepiti per essere visionati da lettori umani, possono mescolare al proprio interno, testo, immagini e colori e non avere lo stesso layout. È cruciale che IDUM raccolga anche questi volantini, perché contengono i vari dettagli delle offerte proposte (ad esempio, luogo, alloggio, prezzo, sconti, offerte). Per fare questo, IDUM si serve di HiLeX [58, 59], un sistema avanzato per l'estrazione di informazioni basate su ontologie da documenti semi-strutturati (archivi di documenti, librerie digitali, siti Web, eccetera) e documenti non strutturati (principalmente testi scritti in linguaggio naturale). In pratica, il sistema HiLeX implementa un approccio semantico al problema dell'estrazione delle informazioni basandosi su un modello concettuale semantico di nuova generazione, sfruttando le ontologie come formalismi di rappresentazione della conoscenza, e un modello di rappresentazione generale dei documenti in grado di unificare formati di documenti diversi (html, pdf, doc, ...).

Le e-mail ricevute da IDUM (e il loro contenuto) vengono elaborate automaticamente mediante il sistema HiLeX e i dati estratti sulle offerte turistiche vengono utilizzati per creare un'ontologia, la "*Tourism Ontology*", composta da entità come le seguenti:

```
class Place (description: string).
class TransportMean (description: string).
class TripKind (description: string).
class Customer (firstName: string, lastName: string, birthdate: Date,
    status: string, childNumber: positive integer, job: Job).
```

```
class TouristOfficer (start: Place, destination: Place, kind:
    TripKind, means: TransportMean, cost: positive integer, fromDay:
    Date, toDay: Date, maxDuration: positive integer, deadline: Date,
    uri: string).
relation PlaceOffer (place: Place, period: positive integer).
relation SuggestedPeriod (place: Place, period: positive integer).
relation BadPeriod (place: Place, period: positive integer).
intentional relation Contains (pl1: place, pl2: place) {
    Contains (P1, P2) :- Contains (P1, P3), Contains (P3, P2).
    Contains ('Europe', 'Italy'). }
```

Questa ontologia risultante è dunque analizzata sfruttando un insieme di moduli di ragionamento che combina i dati estratti con le conoscenze relative ai luoghi (ossia, le informazioni geografiche) e agli utenti (le loro preferenze). In base a questi due tipi di informazioni, il sistema riproduce le tipiche deduzioni fatte da un dipendente di un'agenzia di viaggi per selezionare le risposte più appropriate alle esigenze dell'utente.

La Tourism Ontology è stata specificata in collaborazione con lo staff di una vera agenzia turistica. In questo modo, viene modellata qualsiasi delle entità chiave che descrivono il processo di organizzazione e vendita di un vero pacchetto turistico completo. In particolare, la Tourism Ontology modella tutte le informazioni richieste, come informazioni geografiche, tipo di vacanza, mezzi di trasporto, e altre ancora. Nella Figura 1 sopra riportata, per esempio, la classe *Customer* consente di modellare le informazioni personali di ciascun cliente, comprendendo nome, cognome, data di nascita, status, professione e numero di figli. Il tipo di viaggio è invece rappresentato utilizzando la classe *TripKind*: esempi di sue istanze sono safari, vacanze al mare, vacanze sciistiche (da rendersi come *safari*, *sea_holiday*, *skiing_holiday*), eccetera. Allo stesso modo, istanze della classe *TransportMean* possono essere *airplane*, *train*, *ship*, mentre appartengono a *Place* le informazioni geografiche di descrizione del luogo stesso: questa classe è stata popolata sfruttando Geonames, uno dei più grandi database geografici disponibili al pubblico. Inoltre, ogni luogo è associato a un tipo di viaggio tramite la relazione *PlaceOffer* (ad esempio, il Kenya offre safari, la Sicilia offre sia il mare che i giri turistici, le Dolomiti offrono una vacanza sciistica). È importante sottolineare che la parte naturale della gerarchia dei luoghi è facilmente modellabile utilizzando la relazione *Contains*.

Le mere informazioni geografiche sono, a questo punto, arricchite da altre informazioni che vengono solitamente sfruttate dai dipendenti delle agenzie di viaggio per selezionare una possibile destinazione. Ad esempio, si potrebbe suggerire di evitare le vacanze al mare in inverno o le vacanze sciistiche in luoghi montanari sprovvisti di neve durante l'estate. Questi due esempi sono stati codificati per mezzo delle relazioni *SuggestedPeriod* e *BadPeriod*.

Infine, la classe *TouristicOffer* contiene un'istanza per ogni pacchetto vacanze disponibile: le istanze di questa classe vengono aggiunte automaticamente, sfruttando il sistema HiLeX sopra descritto, oppure manualmente dal personale dell'agenzia. HiLeX estrae il periodo e il tipo in cui viene offerto un ipotetico viaggio, la destinazione e i mezzi di trasporto da impiegarsi.

In IDUM è stata inserita anche la ricerca di viaggio personalizzata, funzione necessaria e concepita per semplificare il compito di selezionare i pacchetti vacanza che meglio si adattano alle esigenze del cliente. In uno scenario tipico, quando un possibile turista entra in un'agenzia viaggi, ciò che deve essere compreso chiaramente da entrambi per una corretta selezione di un pacchetto di vacanze che soddisfi le esigenze del cliente è riassunto in quattro istanze: il luogo, il periodo, il mezzo e il budget. Tuttavia, il cliente non specifica direttamente tutte queste informazioni: ad esempio, può chiedere una vacanza al mare a gennaio ma senza saper specificare un luogo preciso, oppure può chiedere un tipo di viaggio che non è possibile svolgere in un dato periodo. In IDUM, le esigenze attuali sono specificate compilando un appropriato modulo di ricerca, in cui devono essere fornite alcune delle informazioni chiave (ossia le quattro sopra citate: luogo, periodo, mezzo e budget).

Ad esempio, si supponga che un cliente specifichi il tipo di vacanza (*TripKind*) e il periodo (*Period*). Si origina quindi una selezione di pacchetti di vacanza in seguito a un modulo (qui semplificato), come proposto nel codice seguente:

```
module (kindAndPeriod) {
 %detect possible and suggested places
 possiblePlace(P) :- askFor(tripKind:K), PlaceOffer(place:P, kind:K).
 suggestPlace(P) :- possiblePlace(P), askFor (period:D),
                    SuggestedPeriod (place:P1, period:D),
                    not BadPeriod (place:P1, period:D).
```

```
%select possible packages
possibleOffer(0):-0:TourisicOffer(destination:P),possiblePlace(P).}
```

La prima regola seleziona tutti i possibili luoghi disponibili (*possiblePlace(P)*), cioè quelli che offrono il tipo di vacanza inserito in input. La seconda, invece, identifica i luoghi da suggerire al cliente (*suggestPlace(P)*), perché offrono sia il tipo di vacanza richiesto, sia la loro disponibilità nel periodo specificato (escludendo, quindi, quelli entro la relazione *BadPeriod*). Infine, la regola rimanente cerca nei pacchetti vacanze disponibili quelli che offrono una festività che corrisponda all'input originale (offerta possibile).

Oltre al modulo della ricerca di viaggio personalizzata, il continuo processo di estrazione svolto da HiLeX popola ininterrottamente l'ontologia con nuove offerte turistiche. Pertanto, il sistema, eseguendo il modulo di ragionamento appositamente concepito, combina le informazioni specificate dal cliente con quelle disponibili nell'ontologia e mostra i pacchetti vacanze che meglio si adattano alle sue esigenze, in una fornitura continuamente aggiornata.

3.4 e-Government

ASP è stata utilizzata anche nel campo dell'e-Government [28], ancora una volta entro il sistema DLV.

In questo caso, il programma utilizza un'ontologia basata sul Thesaurus del Senato della Repubblica Italiana (TE.SE.O., Tesauro Senato per l'Organizzazione dei documenti parlamentari), ossia un sistema di classificazione che conta di quasi 4000 descrittori organizzati in un sistema di relazioni gerarchiche, di correlazione e di affinità, in cui i termini di classificazione sono quelli usati per gli atti parlamentari del Senato dal 1848. Il programma, oltre che sul TE.SE.O., si basa anche su una lista di termini costruita analizzando un corpus di 16.000 documenti della pubblica amministrazione e, infine, su un archivio contenente un elenco completo di vocaboli ordinati in gruppi correlati tra loro per sinonimi o per concetti, inerenti alla terminologia giuridica utilizzata dal Parlamento italiano. Da questi input, il programma ASP è in grado di classificare tutti gli atti giuridici e i decreti emessi dalle autorità pubbliche italiane.

È un utilizzo, dunque, che rientra entro il Knowledge Management, in particolare nel Text Classification.

DLV è stato validato con l'aiuto dei dipendenti amministrativi della Regione Calabria, regione che ha visto la creazione del sistema stesso, e ha oggi una precisione media del 96% dei documenti del mondo reale. Inoltre, il sistema è stato valutato con una f-measure del 92%, ricordando che f-measure si utilizza nel campo del recupero dell'informazione per misurare l'elaborazione del linguaggio naturale, l'accuratezza delle ricerche o, come in questo caso, della classificazione dei documenti.

3.5 e-Medicine

In quest'applicazione, riportata in [29], l'efficienza di ASP è stata utilizzata in particolare nella regione del Veneto, nell'area di Asolo, dall'ente locale per i servizi sanitari ULSS (unità locale socio-sanitaria) n.8. Gli analisti dell'ULSS sono oggi aiutati dal programma ASP nella navigazione e nella ricerca di documenti riguardanti patologie specifiche, pazienti che vivono in un dato luogo o che hanno ricevuto una data assistenza sanitaria, e altre informazioni. Il programma è infatti in grado di classificare automaticamente case history e documenti contenenti diagnosi cliniche, soprattutto nell'ambito delle analisi epidemiologiche.

I dati reperiti come input vengono dall'ICD9-CM (Classificazione internazionale delle malattie), un sistema di classificazione in cui i traumatismi e le malattie sono ordinati in gruppi per finalità statistiche, e in cui termini medici, interventi chirurgici, problemi di salute generale, trattamenti di ricovero e procedure diagnostiche sono espressi in codici alfa-numerici, per una generalizzazione a livello internazionale. Altri dati vengono anche dal MESH (Medical Subject Headings), vocabolario e sistema di metadati statunitense consultabile gratuitamente e la cui traduzione è stata curata dall'Istituto Superiore di Sanità.

3.6 Assemblaggio e routing di veicoli presso Mercedes-Benz

Nell'aprile 2018, tre ricercatori dell'Università di Potsdam, Martin Gebser, Philipp Obermeier e Torsten Schaub, hanno pubblicato l'articolo "*Routing Driverless Transport Vehicles in Car Assembly with Answer Set Programming*" ("Instradamento dei veicoli di trasporto senza conducente nell'assemblaggio di automobili con Answer Set

Programming") [22], in cui supportavano l'utilizzo di ASP nel contesto del montaggio e dello stoccaggio di auto presso la Mercedes-Benz Ludwigsfelde GmbH, il produttore su larga scala di veicoli commerciali Mercedes-Benz stanziato a Ludwigsfelde, in Brandeburgo. I componenti principali delle moderne strutture di produzione e magazzino di Mercedes-Benz, come di altre case automobilistiche, sono infatti i sistemi automatici di stoccaggio e riparazione, soprattutto quelli a guida automatica: sia che siano nei magazzini, nelle miniere o negli impianti di produzione, questi veicoli automatizzati svolgono infatti un ruolo chiave, ma sono manualmente programmati da ingegneri umani per eseguire i loro compiti specifici. Per questo motivo, pertanto, il problema di ottimizzazione preso in esame dai tre ricercatori riguarda l'impostazione di percorsi in modo tale che un insieme di compiti di trasporto sia realizzato nel modo più efficace e meno dispendioso possibile. L'approccio dichiarativo di ASP consente qui di gestire uno scenario che copre l'intero ciclo di produzione nello stabilimento automobilistico, senza sostituire completamente l'attività degli ingegneri umani, ma assistendoli soprattutto nella creazione di percorsi da prendere periodicamente dai veicoli a guida automatica. Inoltre, altri scenari di più piccole dimensioni, come la riassegnazione delle attività o il reinstradamento del veicolo in caso di circostanze impreviste, possono essere virtualmente gestiti in tempo reale.

Prima dell'introduzione di un programma basato su ASP, erano già utilizzati dei preesistenti sistemi di controllo di basso livello e manualmente codificati che già si occupavano della navigazione dei veicoli senza conducente lungo percorsi programmati, evitandone collisioni anche in circostanze impreviste. Si tratta di del software di sistema di controllo opensource openTCS, sviluppato da Fraunhofer IML. Tuttavia, mentre openTCS offre una piattaforma generica per il controllo automatizzato dei veicoli, non è orientato a fornire un solutore per compiti combinatori difficili come l'ottimizzazione multiobiettivo dei percorsi, e non è neanche in grado di riassegnare rapidamente le attività in caso di guasti o di reagire facilmente ai cambiamenti dei requisiti o degli obiettivi di produzione. Inoltre, è caratterizzato da un'assoluta mancanza di portabilità tra diversi impianti di produzione e layout di fabbrica, dunque rende impossibile utilizzare lo stesso programma di coordinazione e ottimizzazione tra i percorsi di veicoli automatizzati da uno stabilimento e a un altro, anche di proprietà della stessa Mercedes-Benz. Con openTCS, nuovi dati e nuovi vincoli devono essere inseriti con una tediosa

riconfigurazione manuale di tutto il sistema e, naturalmente, con conseguente dispendio di tempo e personale. Non solo: non è possibile trarre conclusioni sull'efficacia o addirittura sull'ottimalità dei nuovi percorsi o, in linea ancora più generale, dei percorsi di veicoli pre-programmati e trasportati in un nuovo impianto.

La mancanza di tolleranza all'elaborazione, in generale, non porta solo a spese elevate, ma anche a una gestione poco flessibile dell'insieme dei veicoli a guida automatizzata.

Il vantaggio del sistema di controllo openTCS, tuttavia, è che si tratta di un software estensibile e che consente l'integrazione di componenti personalizzati: ed è proprio qui che si inserisce l'approccio dei tre ricercatori dell'Università di Potsdam al routing dei veicoli gestito con ASP, da attuarsi con CLINGO (versione 5.2.2.) [30], una parte del progetto POTASSCO (Potsdam Answer Set Solving Collection) per l'Answer Set Programming.

L'approccio dichiarativo basato su Answer Set Programming comprende, infatti, numerosi vantaggi, tra i quali:

- una formalizzazione dei problemi trasparente ed eseguibile;
- la dimostrabile condizione ottimale dei percorsi elaborati;
- la tolleranza dell'elaborazione verso diversi layout di fabbrica e obiettivi di produzione.

In questo modo, il nuovo programma mira a un'ottimizzazione dei trasporti: si vuole pertanto garantire che tutti i componenti dell'automobile da assemblare siano prelevati correttamente dai luoghi di stoccaggio e portati nella posizione designata accanto alla linea di assemblaggio ed esattamente nel momento idoneo all'installazione. Contemporaneamente, possono occorrere numerose attività: ad esempio, un veicolo può prima fermarsi in qualche luogo di stoccaggio per caricare il materiale di produzione, quindi passare a una stazione di assemblaggio che necessita del materiale, e da lì caricare il materiale rimasto in un impianto di riciclaggio. Il piano del suo moto prevede quindi tre soste in tre punti distinti, tra le quali il veicolo deve scegliere un percorso senza essere bloccato da altri, e dove devono essere necessariamente inclusi anche gli spazi di parcheggio e spostamento per consentire ai veicoli di farsi spazio a vicenda. In particolare, è da ricordare anche che è necessario che il processo di produzione regolare

venga eseguito in modo periodico, così che le attività di spostamento e trasporto possano, e debbano, essere eseguite ripetutamente a intervalli fissi.

Per dimostrare la validità dall'approccio empirico operato dai tre ricercatori di Potsdam, si mettono in relazione i risultati ottenuti con quelli derivati dallo scheduler predefinito di openTCS, che implementa una semplice procedura round-robin (ossia, di alternanza) per assegnare compiti e scegliere i percorsi più brevi per i veicoli. Sono stati definiti diciotto casi d'uso basati su un layout di fabbrica con venticinque postazioni e trentacinque connessioni, che ricalcavano le aree di stoccaggio, le linee di assemblaggio e i percorsi di trasporto presso la fabbrica di automobili di Mercedes-Benz Ludwigsfelde GmbH. Tali casi d'uso sono stati poi eseguiti in simulazione per testare l'intera funzionalità di un sistema di controllo, e sono raggruppati, in base ai principali obiettivi del test, in cinque categorie:

A. comunicazione e feedback;
B. assegnazione ed esecuzione delle attività;
C. gestione del routing e del traffico;
D. condizioni speciali e guasti;
E. ciclo di produzione completo.

Mentre i casi di test nei primi quattro gruppi si concentrano su particolari scenari di piccole dimensioni, ovvero fino a tre compiti e tre veicoli, il caso d'uso nell'ultima categoria (E) emula un ciclo di produzione completo di circa venti minuti in tempo reale, coinvolgendo un totale di quarantanove attività, da dividersi in dieci attività di trasporto e trentanove secondarie, ed eseguite da quattro veicoli a guida automatica.

Per il calcolo dei percorsi ottimali con CLINGO, i tre ricercatori hanno rappresentato gli scenari dei casi d'uso in termini di fatti, ossia considerando l'assegnazione e l'ordine delle attività, il loro completamento e il routing dei veicoli: poco peso è stato dato al controllo del sistema piuttosto che, dunque, sull'assegnazione delle attività e sull'instradamento. La Figura 3 mostra come i risultati ottenuti utilizzando ASP siano ottimamente provvisori in tutti i diciotto casi d'uso e si dimostrino vincenti rispetto all'approccio con il sistema di controllo openTCS. In particolare, in questo caso, i runtimes di CLINGO, supportati da un hardware dotato di CPU Intel i7-6700 a 3,40 GHz, variavano da pochi secondi a un massimo di dieci secondi per ciascuno dei diciassette piccoli scenari nei primi quattro gruppi (A, B, C, D).

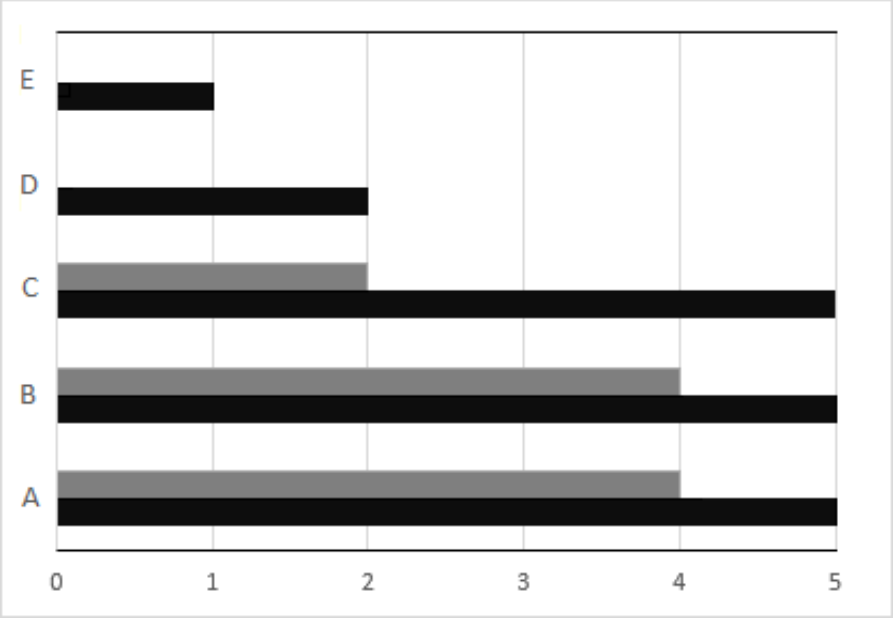

Figura 3. Casi d'uso risolti mediante ASP (in nero) e mediante lo scheduler predefinito di openTCS (in grigio).

Mentre le piccole istanze dei primi quattro gruppi derivano da casi d'uso creati principalmente per testare la reazione di un sistema di controllo ad una situazione isolata, lo scenario in E mira a gestire tutte le attività di trasporto ricorrenti all'interno di ogni ciclo di produzione completo: si tratta dunque di un'operazione complessa finora eseguita soltanto da ingegneri umani e prima che un processo di produzione inizi o venga ripreso con una configurazione aggiornata. In effetti, l'ottimizzazione delle rotte per i veicoli a guida automatica durante l'intero ciclo di produzione è computazionalmente impegnativa, anche per un sistema come CLINGO che è orientato verso la risoluzione di problemi combinatori complessi: basti menzionare che ci sono volute circa cinque ore per calcolare una soluzione ottimale. Il suo makespan (il tempo trascorso dal rilascio di un compito, all'istante di completamento dell'ultimo pezzo in produzione nell'ultima macchina) di duecentoventicinque cicli di clock equivale a cinque volte il numero di secondi in tempo reale, circa venti minuti, condiviso tra quattro veicoli i cui percorsi calcolati coinvolgono trentanove incroci e centoquarantuno sovrapposizioni in totale. Benché cinque ore di tempo di calcolo non possano certo essere garantite da un sistema di controllo che deve reagire agli incidenti in tempo reale, il risultato è comunque incoraggiante, poiché testimonia che i percorsi per i compiti di trasporto ricorrenti in processi di produzione realistici possono essere programmati in modo automatico da un

sistema dichiarativo come Answer Set Programming, e dare luogo a condizioni efficienti e ottimali che non possono essere stabilite dai soli ingegneri umani.

Analizzando ora le barre grigie mostrate in Figura 3, si evince che lo scheduler predefinito fornito con openTCS non riesce a trovare una soluzione fattibile in otto dei diciotto casi d'uso, ossia quasi la metà di quelli proposti, il che è dovuto al suo approccio *greedy* di assegnare un'attività e bloccare le posizioni lungo il percorso più breve di un veicolo, e di farlo uno alla volta. Ad esempio, tale approccio è destinato a fallire in uno scenario di esempio in cui due veicoli devono necessariamente visitare un nodo per completare qualsiasi loro attività secondaria, e che blocca il rispettivo altro veicolo fuori dal nodo, impedendogli di completare da solo il suo compito di trasporto entro i termini stabiliti.

Questa osservazione, insieme al fatto che CLINGO può gestire tempestivamente i piccoli scenari dei casi d'uso nei primi quattro gruppi, motiva la sua integrazione in openTCS come componente responsabile dell'assegnazione delle attività e del routing dei veicoli.

Inoltre, è interessante citare che gli studiosi hanno verificato che CLINGO richiede meno di 1 GB di RAM per rappresentare durate e makespan di un ciclo di produzione completo presso la fabbrica di automobili della Mercedes-Benz Ludwigsfelde GmbH. A CLINGO possono tuttavia anche essere integrate eventuali estensioni che potrebbero gestire intervalli di tempo ancora più grandi e in modo ancora più compatto.

Anche in questa applicazione, dunque, l'approccio tollerante dichiarativo e di elaborazione al problem solving combinatorio di Answer Set Programming si è dichiarato innovativo e sufficientemente efficiente anche per soddisfare i requisiti dei processi di produzione di scala industriale di uno stabilimento automobilistico.

3.7 Il Nurse Scheduling Problem

Il Nurse Scheduling Problem (NSP, "Problema di pianificazione delle infermiere") consiste nel generare un programma di lavoro e di giorni di riposo per gli infermieri che lavorano nelle unità ospedaliere, in base a determinati vincoli pratici. In particolare, il programma determina gli incarichi di turno degli infermieri entro una finestra di tempo prestabilita e soddisfando i requisiti imposti dal Regolamento degli ospedali. Trovare un'adeguata soluzione a questo problema è fondamentale per garantire l'elevato livello

di qualità dell'assistenza sanitaria, per migliorare il reclutamento di personale qualificato e il grado di soddisfazione degli infermieri.

Numerosi sono stati, negli anni precedenti, i diversi approcci per risolvere diverse varianti dell'NSP; tuttavia, essi non sono direttamente confrontabili tra loro, poiché i requisiti di solito dipendono dalla specifica politica degli ospedali. Essendo l'NSP un complesso problema combinatorio, invece, si presta in modo eccellente all'applicazione di formalismi logici come Answer Set Programming, che con la sua sintassi semplice e la sua semantica intuitiva, può fornire una codifica ottimale.

ASP viene per la prima volta utilizzato nella risoluzione dell'NSP nell'articolo "*Nurse Scheduling via Answer Set Programming*" [31]: è il sistema ASP CLINGO, già presentato nella sezione precedente entro lo stabilimento automobilistico Mercedes-Benz Ludwigsfelde GmbH, a produrre ancora una volta risultati positivi, conducendo un'analisi sperimentale mediante dati reali forniti da un ospedale italiano. È stato utilizzato anche il sistema WASP [32], sviluppato presso l'Università della Calabria, ma le sue prestazioni sono risultate più lente di CLINGO in tutte le istanze testate.

Innanzitutto, l'NSP prevede la generazione di orari per gli infermieri consistenti in giorni di lavoro e di riposo per un periodo di tempo predeterminato, che nell'analisi è stato fissato di un anno. Inoltre, gli orari devono soddisfare una serie di requisiti:

- *Requisiti ospedalieri.* Sono stati considerati tre diversi turni: il mattino (dalle sette alle quattordici), il pomeriggio (dalle quattordici alle ventuno), e la notte (dalle ventuno alle sette del mattino). Al fine di garantire il miglior programma di assistenza per i pazienti, ogni turno è stato inoltre associato a un numero minimo e massimo di infermieri che devono essere presenti nell'ospedale.
- *Requisiti degli infermieri.* Questi requisiti si esplicano nella ricerca di un equo carico di lavoro tra gli infermieri: pertanto, viene imposto un limite al numero minimo e massimo di ore di lavoro all'anno. Inoltre, sono previsti requisiti aggiuntivi per garantire un periodo di riposo adeguato e che comprende: trenta giorni di ferie pagate legalmente garantiti; il fatto che l'orario di partenza di un turno sia di almeno ventiquattro ore dopo l'ora di inizio dal turno precedente; e la presenza di almeno due giorni di riposo per ogni infermiere ogni quattordici giorni. È da sottolineare, inoltre, che dopo due notti di lavoro consecutive c'è un giorno speciale di riposo che non è incluso nei giorni di riposo sopra citati.

- *Requisiti di equilibrio.* Infine, il numero di volte che un infermiere può essere assegnato ai turni mattutini, pomeridiani e notturni è fisso. Tuttavia, sono valide anche le pianificazioni in cui questo numero è variabile entro un intervallo di valori.

Nel sistema ASP utilizzato per questo problema di ottimizzazione, gli answer set corrispondono alle soluzioni dell'NSP.

Nella codifica ASP, le istanze del predicato *assign(N, S, D)* vengono utilizzate per memorizzare l'assegnazione del turno S per un infermiere N in un giorno specifico D. Le istanze del predicato *shift(S, H)* sono invece utilizzate per rappresentare i turni, dove S è un turno e H è il numero di ore di lavoro associate al turno stesso.

Altre istanze considerate sono: *shift("1-mor",7), shift("2-aft", 7), shift("3-nig", 10), shift("4-specres", 0), shift("5-rest", 0), shift("-vac", 0)*; in cui le ultime tre istanze sono utilizzate per i giorni di riposo degli infermieri. Poiché l'ordine lessicografico delle stringhe non è sempre garantito dai sistemi attuali, nella codifica testata sono stati utilizzati solo gli interi, ad es. *shift ("1-mor", 7)* è stato sostituito da *shift (1,7).*

Dopo aver redatto i vari vincoli, l'analisi empirica è stata condotta su dati reali forniti da un ospedale italiano utilizzato come riferimento: la pianificazione è stata creata per una finestra temporale di un anno e il numero massimo di ore all'anno è stato impostato su 1692, mentre il numero minimo su 1687. Il numero di infermieri che lavoravano presso l'unità ospedaliera considerata era quarantuno, da dividersi nel numero di quelli che lavoravano durante la mattina e il pomeriggio, che variava da sei a nove, e in quello degli infermieri impiegati nel turno notturno, che andava da quattro a sette.

Sono stati considerati quindici giorni di vacanza scelti dalle infermiere, in base alle ferie nell'anno selezionato (2015), mentre gli altri quindici giorni di ferie sono stati assegnati in base alle esigenze dell'ospedale. Il numero desiderato di mattinate e pomeriggi di lavoro era pari a settantotto, mentre il numero desiderato di notti di lavoro era sessanta.

Sono state testate due varianti della codifica. La prima, segnalata come ENC1, considerava i requisiti di equilibrio nella modalità fissa, mentre ENC2 nella modalità variabile. Gli esperimenti sono stati eseguiti su un Intel Xeon 2.4 GHz, con tempo limitato a un'ora e memoria a 15 GB.

Per quanto riguarda la codifica ENC1, CLINGO ha calcolato una pianificazione in 12 minuti con un picco di utilizzo della memoria di 300 MB. Le prestazioni eseguite sulla

codifica ENC2 sono state ancora migliori: la soluzione ottimale è stata trovata in 6 minuti con un picco di utilizzo della memoria di 224 MB. Questo poiché CLINGO adotta una particolare strategia in presenza di vincoli deboli: essi vengono prima considerati come vincoli rigidi e quindi allentati quando non possono essere soddisfatti.

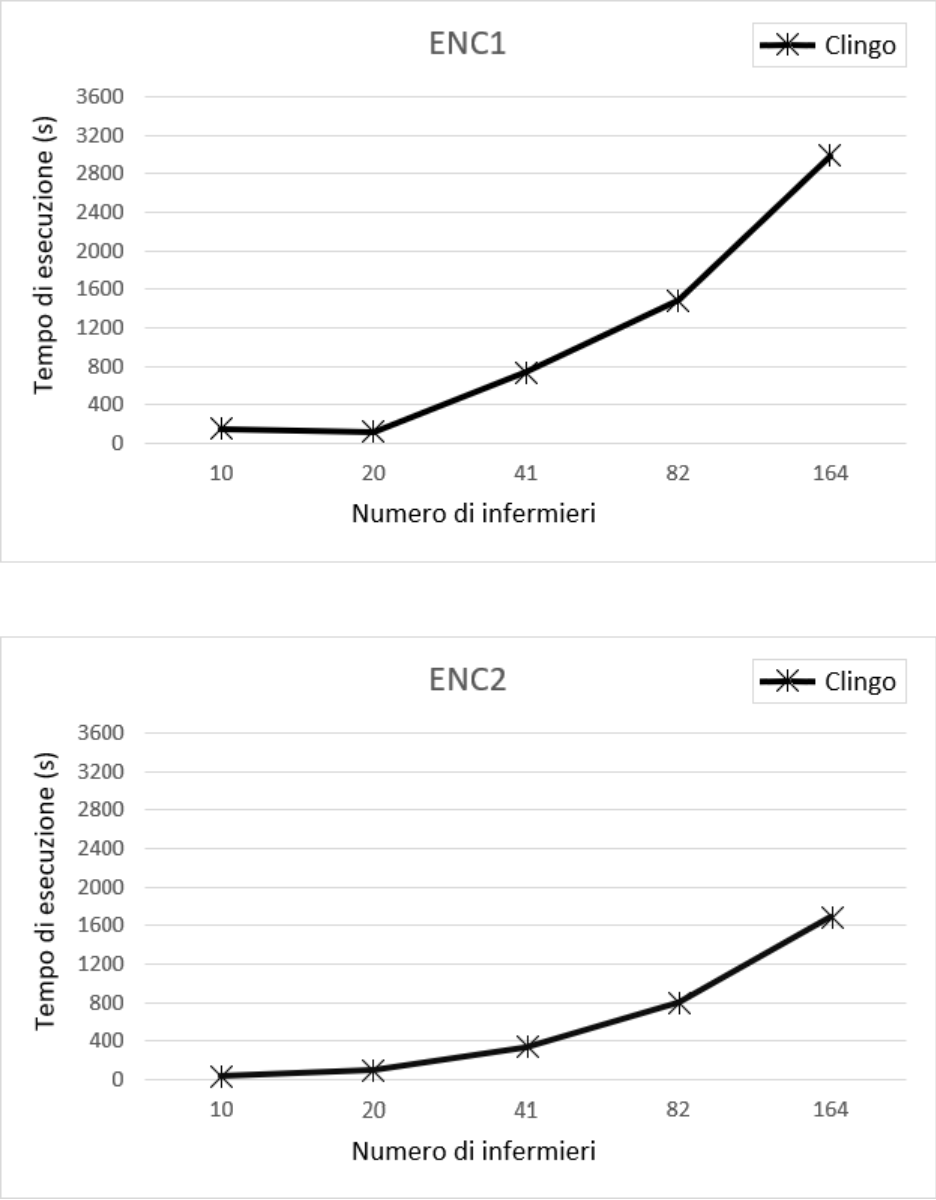

Figura 4. Analisi di scalabilità con 10, 20, 41, 82 e 164 infermieri.

È stata anche eseguita un'analisi sulla scalabilità della codifica, considerando diversi numeri di infermieri (Figura 4): in particolare, 10, 20, 41, 82 e 164. Per ogni test, è stato ridimensionato proporzionalmente il numero di infermieri che lavoravano durante ogni turno, mentre altri requisiti non sono stati modificati. Per quanto riguarda la codifica ENC1, tutte le istanze sono state risolte entro il tempo assegnato: sono stati necessari meno di 3 minuti per calcolare il programma con 10 infermieri e circa 100 MB di consumo di memoria. Risultati simili sono stati ottenuti per calcolare il programma con 20 infermieri. Per quanto riguarda la programmazione con 82 e 164 infermieri, la soluzione è stata calcolata entro 25 e 50 minuti, rispettivamente, mentre l'utilizzo della memoria è stato rispettivamente di 500 e 990 MB.

Anche i risultati ottenuti con la codifica ENC2 sono molto positivi: l'assegnazione ottimale è stata trovata entro 60 minuti e 1024 MB per tutti i casi di test considerati, con un picco di 28 minuti e 838 MB di memoria utilizzata quando sono considerati 164 infermieri.

3.8 Riparazione di errori o incoerenze nei censimenti

Nella maggior parte dei paesi, ogni cinque o dieci anni viene tenuto un censimento completo della popolazione, che viene aggiornato costantemente e rappresenta la fonte più fondamentale di informazioni sul paese stesso. I suoi dati forniscono preziose informazioni sulle condizioni economiche, sociali e demografiche e sulle tendenze che vi si verificano, che forniscono un ritratto statistico del paese e dei suoi abitanti. Il censimento è l'unica fonte affidabile di dati dettagliati per piccoli gruppi di popolazione, come le categorie industriali e occupazionali molto specifiche, e per piccole aree geografiche come, per esempio, un quartiere di una città. I dati del censimento possono essere utilizzati in molti modi: fornendo i confini per i distretti elettorali federali, aiutando le imprese a selezionare nuovi siti produttivi, analizzando i mercati per prodotti e servizi, sviluppando politiche e programmi per l'occupazione e altro ancora.

I dati sensibili sono raccolti per mezzo di milioni di questionari, ognuno dei quali include i dettagli delle persone che vivono insieme nella stessa casa.

Ogni famiglia compila infatti un questionario, che include i dettagli del capofamiglia insieme alle altre persone che vivono in casa. I moduli vengono raccolti e quindi analizzati dalle agenzie statistiche al fine di modificare e aggiornare le statistiche relative

alla popolazione. Tuttavia, i questionari potrebbero essere incompleti o potrebbero contenere informazioni incoerenti; addirittura, una parte significativa di essi presenta problemi di questo tipo, che se non risolti possono alterare gravemente le statistiche aggiornate.

Con milioni di questionari da elaborare ogni volta, è auspicabile utilizzare metodologie per rilevare e risolvere automaticamente gli errori. Questa fase di pulizia è detta "imputazione" e si svolge prima che i dati dei moduli vengano inviati agli statistici da analizzare, al fine di eliminare problemi di coerenza, mancate risposte o errori. Naturalmente, è importante eseguire la fase di imputazione senza alterare la validità statistica dei dati raccolti.

Tutte le metodologie di imputazione attualmente utilizzate dalle agenzie di censimento (tra tutte, la più popolare è la metodologia "*Fellegi and Holt*", FH) si basano su principi statistici che non hanno una semantica chiara rispetto alla complessità dei vari errori che possono verificarsi, in modo che il loro comportamento mentre si imputano i dati è spesso imprevedibile.

Più recentemente, la New Imputation Methodology (NIM) è stata introdotta come miglioramento rispetto a FH. Sia in FH che NIM le regole di modifica sono scritte in modo dichiarativo e vengono utilizzate per verificare la coerenza di un questionario domestico; il controllo di coerenza viene solitamente implementato con regole di un database relazionale. Quando un questionario non riesce a soddisfare le regole di modifica, le informazioni necessarie per la fase di imputazione sono prese in prestito da un questionario simile che ha superato le regole di modifica (che è chiamato *modulo di donazione*).

FH prima determina il numero minimo di attributi e quindi esegue l'imputazione, possibilmente ricercando donatori. NIM, invece, cerca inizialmente i donatori che corrispondono al record fallito sul maggior numero possibile dei valori degli attributi coinvolti. Secondo gli esperimenti, la modifica dell'ordine di queste operazioni consente al NIM di risolvere in modo efficiente problemi di imputazione più grandi e più complessi.

Entrambi questi due metodi non hanno una semantica dichiarativa. In particolare, differiscono leggermente nella fase di imputazione, rendendo talvolta impossibile capire cosa viene riparato e perché. Inoltre, sono computazionalmente costosi, poiché entrambi

si basano intrinsecamente sui donatori, che devono essere trovati nell'ampio database dell'intero censimento.

Ancora una volta, ricercatori dall'Università della Calabria, insieme all'Università di Manchester, hanno proposto un sistema basato su Answer Set Programming [33] per fornire una semantica dichiarativa ben fondata e chiara ai questionari e al problema dell'imputazione, da attuarsi con il già citato DLV, sistema che, inoltre, essendo liberamente disponibile sul web, può essere utilizzato per ottenere un'implementazione immediata e continua. Questo tipo di approccio ha due chiari vantaggi rispetto a FH, NIM e altri metodi derivati. Prima di tutto, è dotato di una semantica dichiarativa formale, che specifica in modo inequivocabile il significato e il comportamento del compito di imputazione; questa semantica non è in contrasto con il comportamento delle procedure FH e NIM, tuttavia, i vari algoritmi attualmente utilizzati producono risultati diversi che sono difficili da interpretare e confrontare data l'assenza di una semantica dichiarativa. In secondo luogo, si basa su un approccio altamente modulare: ciascun questionario viene elaborato e corretto da solo, indipendentemente da tutti gli altri dati del censimento. Questa proprietà di modularità consente un elevato grado di parallelismo (ossia, diversi questionari possono essere elaborati in parallelo su macchine diverse) e garantisce un carico di calcolo molto più elevato. Infatti, anche se la correzione di un questionario può richiedere un tempo esponenziale, questo tempo è esponenziale nella dimensione di un singolo questionario e nel numero di regole di modifica.

Nella pratica, durante il periodico censimento della popolazione, la struttura dei questionari è simile in ogni paese: un singolo modulo è associato a ciascuna famiglia, con domande su ognuno dei suoi membri che, a fini statistici, è identificato dalla sua relazione con il membro di riferimento. A questo viene assegnato il numero "1", mentre agli altri viene assegnato un numero crescente intero. Un esempio semplificato di classificazione delle persone che vivono in una famiglia privata in *relazione* alla persona di riferimento della famiglia è il seguente: Coniuge; Compagno; Bambino; Genitore; Altro parente della persona di riferimento, coniuge o partner; Non-parente della persona di riferimento, coniuge o partner.

Ci sono anche altri attributi *strutturali* (da specificare obbligatoriamente per ciascun membro della famiglia) considerati come parte di un questionario di base, ossia: *Sesso*

(maschio, femmina); *Età* (come numero intero); *Stato civile* (single, sposato, divorziato o vedovo).

Un modulo di censimento che viene compilato dai membri di una famiglia contiene quindi, per ciascun membro, le domande per almeno ciascuno degli attributi di cui sopra.

I dati da inserire paiono quindi semplici, ma è comune trovare errori, dati mancanti o dati incoerenti: ad esempio, è possibile che una persona dichiari di essere il coniuge della persona di riferimento, ma allo stesso tempo abbia dimenticato di dichiarare uno stato civile, o abbia dichiarato di essere single o di avere sei anni. Prima che i dati dei questionari vengano inviati agli statistici da analizzare, viene eseguita la fase di pulizia, che però deve essere eseguita senza alterazione della validità statistica dei dati raccolti. Ad esempio, se il coniuge ha dichiarato di avere sei anni e se ci sono altri argomenti per far valere che egli/ella è in realtà il coniuge della persona di riferimento (ad esempio, egli/ella può aver anche dichiarato di essere sposato), allora la sua età deve essere modificata per rendere coerenti i dati: qualsiasi età, ad esempio, maggiore di sedici anni andrebbe bene se si tiene conto solo della validità assoluta dei dati. Tuttavia, l'età dovrebbe essere modificata in un'età in accordo con l'età media del coniuge nelle sue condizioni e non con un valore che potrebbe alterare le statistiche. In questo caso, sarebbe più sensato cambiare l'età a, per esempio, trentasei anni, piuttosto che a novantasei anni (ci sono pochissime persone di quell'età, e questo potrebbe alterare le statistiche degli anziani) o a sedici anni (ci sono pochissime persone sposate a quell'età). Altre correzioni possono essere fatte in modo deterministico (nel senso che c'è un solo cambiamento valido, che sia anche un cambiamento statisticamente valido): per esempio per un coniuge che ha dimenticato di specificare lo stato civile, e per il quale affermare che egli/ella è sposato/a non introduce altre incongruenze.

Per questi motivi, una serie di regole di modifica viene introdotta dalle agenzie statistiche per ogni classe di questionari, al fine di capire se un questionario è coerente e per guidare la riparazione di quelli incoerenti. Il processo di produzione automatica di un insieme di questionari statisticamente coerenti da quelli grezzi è chiamato *imputazione*. Con il sistema basato su Answer Set Programming, il problema dell'imputazione viene risolto in un contesto proposizionale, utilizzando donatori presi da questionari corretti e secondo un criterio di cambiamento minimo.

Da questo progetto è stata avviata un'attività di sperimentazione con i dati del censimento reale forniti dall'Agenzia Statistica Italiana (ISTAT) (inclusa la loro strategia basata sui donatori per modificare i questionari errati) e al fine di confrontare i risultati di questo approccio ASP con le metodologie di imputazione già consolidate e basate sulle statistiche: i risultati degli esperimenti dimostrano, ancora una volta, la fattibilità e la validità di ASP, anche in questo ambito.

3.9 Robotica

L'efficienza di Answer Set Programming come paradigma di ragionamento e di rappresentazione della conoscenza è stata sfruttata anche nell'ambito della pianificazione robotica, in particolare nell'integrazione tra essa e l'intelligenza artificiale. La ricerca è andata spaziando in vari domini robotici che implicavano la navigazione e la manipolazione di più robot in un ambiente dinamico con presenza di esseri umani.

Alcuni esempi di applicazioni ASP includono pertanto robot mobili che rassettano in modo collaborativo una casa [34, 35, 36] che producono un determinato numero di ordini in una fabbrica [37] e che guidano i clienti verso le loro destinazioni in un centro commerciale; codifiche ASP sono state anche utilizzate in un robot mobile che raccoglie e recapita la posta tra gli uffici [38], in uno che naviga tra uffici per localizzare visivamente gli oggetti [39] e in uno che serve bevande [40]. Infine, anche all'interno di manipolatori mobili che riorganizzano più oggetti su una superficie ingombra [41] o che preparano un tavolo da cucina [42].

Queste applicazioni robotiche implicano, da un lato, un ragionamento basato su logiche di alto livello (ad esempio pianificazione e ragionamento ipotetico) e *problem solving* dichiarativo (come la coordinazione della squadra o il percorso da seguire) su domini discreti. D'altra parte, implicano controlli di basso livello e verifiche basate sulla probabilità (per esempio, controlli di collisione, controlli di raggiungibilità, esistenza di traiettorie) in spazi continui. L'ASP fornisce linguaggi espressivi e solutori efficienti per rappresentare formalmente domini robotici di alto livello e risolvere problemi rilevanti nella robotica cognitiva. In questo modo, ASP fornisce una buona piattaforma per il ragionamento ibrido, in cui il ragionamento logico discreto è strettamente integrato con il ragionamento continuo probabilistico.

Nella robotica possono sussistere due tipi di applicazioni: la prima, in cui i robot hanno una conoscenza completa e una piena osservabilità del mondo circostante (e che corrisponde a una pianificazione classica) e la seconda, di pianificazione condizionale, in cui essi hanno una conoscenza incompleta e una osservabilità parziale del mondo. Per semplicità di esempio in merito degli argomenti trattati, si propone il primo caso.

3.9.1. Pianificazione classica

Un problema di pianificazione è tradizionalmente caratterizzato da una descrizione del dominio di pianificazione, una descrizione dello stato iniziale e una descrizione dell'obiettivo. Pertanto, la sua soluzione è un piano la cui esecuzione garantisce il raggiungimento dell'obiettivo partendo dallo stato iniziale. Nella pianificazione classica,

- gli stati sono intesi come stati del mondo;
- le azioni sono intese come azioni deterministiche, ossia ognuna è interamente causata da eventi precedenti;
- i domini sono tali per cui le condizioni preliminari e gli effetti di tutte le azioni sono completamente determinati dallo stato attuale del mondo;
- lo stato iniziale è completamente specificato;
- un piano è una sequenza finita di azioni.

ASP fornisce linguaggi espressivi per rappresentare domini di pianificazione e solutori efficienti per elaborare piani.

Figura 5. Modello con due robot di pulizia.

Si consideri, ad esempio, il dominio di pulizia [35, 36] in cui più robot stanno riordinando una casa (Figura 5). Come si è visto nella sezione 1.2.1, a proposito del frame problem (ossia degli stati di cose che non subiscono modifiche ma che, al contrario, permangono quando occorre un'azione) si suppone che da un dato istante di tempo a un altro, i mobili rimangano fermi e le loro posizioni siano note a priori ai robot. Altri oggetti (ad esempio, libri, cuscini, piatti) sono invece mobili. Questo dominio può essere descritto usando un'azione costante, come *move(R, L, T)* che denota il movimento di un robot *R* in una posizione *L* nella casa alla fase temporale *T*, e costanti fluenti, come *at(R, L, T)* che denotano che il robot *R* è nella posizione *L* nella fase temporale *T*.

Di seguito si propongono dunque alcune parti di una descrizione di dominio in Answer Set Programming:

:- move(R, L, T), at(R, L, T).

evidenzia che un robot R non può andare in una posizione *L* se il robot è già in *L*;

at(R, L, T + 1) :- move(R, L, T).

identifica che il robot *R* si trova nella posizione *L* dopo che il robot ha raggiunto *L*;

at(O, L, T) :- at(R, L, T), holding(R, O, T).

mediante l'inserimento di un oggetto *o*, codifica la descrizione per cui se un robot *R* è in una posizione *L* e tiene (reso con "*holding*") un oggetto *O*, allora anche la posizione dell'oggetto *O* è *L*. In altre parole, la posizione dell'oggetto cambia qualora cambi la posizione del robot;

:- move(R, L, T), pick(R, O, T)

evidenzia che un robot *R* può muoversi ("*move*") e raccogliere ("*pick*") oggetti contemporaneamente.

Con una tale descrizione di dominio, con un insieme di fatti che descrivono lo stato iniziale completo e con una serie di vincoli per garantire che gli obiettivi siano soddisfatti in qualche momento, è possibile calcolare un piano (o più piani) utilizzando un solutore ASP.

Tuttavia, a volte i piani calcolati non possono essere eseguiti nel mondo reale a causa di controlli di fattibilità di basso livello che non sono considerati nella descrizione del dominio.

Figura 6. Modello con due robot di pulizia le cui traiettorie prevedono collisioni nel mondo reale.

Si considerino, per esempio, i robot nella Figura 6: secondo il piano originariamente calcolato, il robot A (cerchiato di blu) dovrebbe andare verso il letto (seguendo il percorso indicato dalla freccia blu) e poi il robot B (all'interno del cerchio nero) dovrebbe procedere verso il rubinetto della cucina (seguendo la freccia nera). Tuttavia, il passaggio tra la camera da letto e la cucina diventa troppo stretto perché il robot B passi, poiché è parzialmente bloccato dal robot A; pertanto, non vi è una traiettoria che il robot B possa seguire per raggiungere il lato destro del tavolo, e quindi questo movimento non è eseguibile nel mondo reale.

In questo scenario di housekeeping, l'esistenza di una traiettoria senza collisioni può essere verificata e trovata da un solutore ASP come CLINGO, che consente di eseguire controlli di fattibilità di questo tipo mediante calcoli esterni che includono, ad esempio, metodi di pianificazione del moto probabilistico.

Capitolo 4

Caso di studio: assegnamento dipendenti-progetti

4.1 Introduzione

In questo capitolo si propone un'applicazione affine a quelle analizzate nella sezione precedente e che prevede la creazione di un programma basato su Answer Set Programming per la risoluzione di un problema computazionale complesso: l'assegnazione di progetti ai dipendenti di un'azienda. L'ambito in cui si inserisce il lavoro è quello del Knowledge Management, in particolare del team building, come era già stato presentato in 3.2. riguardo al porto di Gioia Tauro.

In ogni azienda, ciascuno dei dipendenti svolge determinate attività sulla base delle proprie competenze e per raggiungere ben precisi obiettivi; talvolta è anche necessario che vengano coinvolti altri colleghi per poterli portare a termine. In questi casi si creano dunque dei team di lavoro che possono essere formati da un minimo di due persone e per i quali sussistono dei vincoli, dettati sia dalla natura del progetto stesso sia dalle competenze dei dipendenti o dalla compatibilità dei due o più colleghi. Un problema simile rientra nell'ambito dell'allocazione intelligente di risorse umane e di soddisfazione dei vincoli, e quindi ben si presta a una risoluzione tramite Answer Set Programming.

Mediante l'utilizzo dei software CLINGO (versione 5.3.0) e DLV (versione del 17 dicembre 2012), in questo capitolo si esporranno prima gli input e gli output del programma, poi i vincoli che devono essere rispettati, infine alcuni esempi ed esperimenti con scalabilità progressiva, inserendo un numero di progetti e di dipendenti sempre maggiore.

4.2 Input e output

Come si è analizzato nel Capitolo 1 (sezione 1.3), Answer Set Programming si basa sulla sintassi simile a quella della logica del primo ordine. All'interno di questa applicazione, si avranno dunque le seguenti funzioni:

dipendente(D).

progetto(P).

Il primo identifica i nomi o i numeri identificativi dei dipendenti: per esempio, *dipendente(marco)* o *dipendente(matricola_43483).*

Il secondo, invece, identifica i nomi dei progetti: *progetto(p_1), progetto(p_2)* e via discorrendo.

Occorrono anche i seguenti predicati:

skillDipendente(D,S).

skillProgetto(P,S).

incompatibili(D_1,D_2).

Con il primo si definiscono le competenze o qualità di ogni dipendente: se, per esempio, il dipendente con matricola 43483 avesse capacità informatiche, come la conoscenza del linguaggio di scripting JavaScript, il predicato risultante sarebbe *skillDipendente(matricola_43483, javascript).* Vicendevolmente, se il dipendente 43484 disponesse di competenze nel linguaggio di programmazione C#, si descriverebbe con *skillDipendente(matricola_43484, csharp).*

Il secondo predicato identifica i requisiti di competenza di cui necessita un progetto. Un esempio a tal proposito è *skillProgetto(p_1, javascript).*

Infine, l'ultimo occorre per mettere in relazione due dipendenti che sono incompatibili tra loro, per ragioni personali o lavorative: *incompatibili(matricola_43483, matricola_43484).*

Il risultato sarà caricato come output nel predicato *assegna(D, P)*, che indica che il dipendente D è assegnato al progetto P.

4.3 Vincoli ed encoding

I vincoli da rispettare sono i seguenti:

1. ogni dipendente deve essere in esattamente un progetto;
2. ogni progetto deve avere almeno due dipendenti;
3. posto che ogni dipendente ha delle qualifiche specifiche (*skill*), e che ogni progetto richiede un certo numero di *skill*, per ogni progetto ci deve essere almeno un dipendente con quello *skill*;
4. due dipendenti possono essere tra loro incompatibili, e dunque non possono stare entro lo stesso progetto.

L’encoding è basato sulla metodologia Guess&Check che prevede due fasi: nella prima si generano tutte le possibili soluzioni (*guess*) e nella seconda si escludono quelle che violano i vincoli (*check*). La fase di guess è definita dalle seguenti regole:

assegna(D,P) :– dipendente(D), progetto(P), not nonassegna(D, P).

che reca il significato di assegnare un dipendente D a un progetto P se si ha un dipendente D e un progetto P e se D non è non assegnato a P; e

nonassegna(D,P) :– dipendente(D), progetto(P), not assegna(D, P).

con il significato di non assegnare un dipendente D a un progetto P se si ha un dipendente D e un progetto P e se D non è assegnato a P.

Mediante la seguente codifica:

:– assegna (D, P_1), assegna(D, P_2), P_1 != P_2.

Si pone che lo stesso dipendente non può essere assegnato a due progetti. La stringa è infatti da leggersi nel seguente modo: “se si assegna il dipendente D al progetto P_1 e se si assegna il dipendente D al progetto P_2, allora P_1 e P_2 non possono essere diversi”.

Dopodiché, la codifica del primo vincolo vuole che, dato un dipendente e un progetto, non è possibile che un dipendente non sia assegnato a un progetto, e si rende con:

assegnato(D) :- assegna (D,P), progetto(P), dipendente (D).

:- not assegnato(D), dipendente (D).

Per quanto riguarda il secondo vincolo (“ogni progetto deve avere almeno due dipendenti”) si inserisce un ulteriore predicato, detto “predicato di lavoro” (“*almenoDue*”), che specifica la condizione che un progetto abbia almeno due dipendenti:

almenoDue(P) :- progetto(P), assegna(D_1, P), assegna(D_2, P), D_1 != D_2.

:- progetto(P), not almenoDue (P).

In particolare, la seconda di queste due regole specifica che per ogni progetto non è possibile che a esso non siano associati almeno due dipendenti.

Il quarto vincolo (“per ogni progetto ci deve essere almeno un dipendente con quello skill”) necessita dell’inserimento del predicato “*skillEffettivoProgetto*”, che identifica lo skill proprio del progetto, da quello che risulta dall’intersezione tra lo skill del progetto e quello del dipendente a esso assegnato.

skillEffettivoProgetto (P,S) :- progetto(P), assegna(D,P), skillDipendente(D,S).

:- progetto(P), skillProgetto(P,S), not skillEffettivoProgetto(P,S).

La prima regola rende la condizione per cui, dato un progetto e il dipendente assegnato a esso, si identificano gli skill dei dipendenti assegnati al progetto. La seconda, invece, ha il significato di evidenziare che, sempre dato un progetto e uno skill richiesto, non è possibile che quello skill non sia a esso assegnato.

L'ultimo vincolo, infine, si rende con la seguente regola:

:– assegna(D_1, P), assegna(D_2, P), incompatibili (D_1, D_2).

In cui, naturalmente, D_1 e D_2 sono i due dipendenti che non possono lavorare nello stesso progetto.

L'encoding totale risultante per la risoluzione del problema, dunque, posti anche gli output specificati in 4.2, sarà:

```
assegna(D,P) :- dipendente(D), progetto(P), not nonassegna(D,P).
nonassegna(D,P) :- dipendente(D), progetto(P), not assegna(D,P).

assegnato(D) :- assegna(D,P), progetto(P), dipendente(D).
:- not assegnato(D), dipendente(D).

:- assegna(D,P1), assegna(D,P2), P1 != P2.

almenoDue(P):- progetto(P), assegna(D1, P), assegna(D2, P), D1 != D2.
:- progetto(P), not almenoDue(P).

skillEffettivoProgetto(P,S) :- progetto(P), assegna(D,P),
skillDipendente(D,S).
:- progetto(P), skillProgetto(P,S), not skillEffettivoProgetto(P,S).

:- assegna(D1, P), assegna(D2,P), incompatibili(D1,D2).
```

4.4 Esperimenti

Abbiamo compiuto un'analisi di scalabilità su un numero crescente di dipendenti e di progetti mediante degli esperimenti eseguiti su un computer Windows 10 a 64 bit, con processore Intel® Core™ i5-7200U 2.50GHz e con 4 GB di Memoria RAM.

È stato pertanto utilizzato Windows PowerShell, l'interfaccia a riga di comando (ossia, un'interfaccia utente caratterizzata da un'interazione di tipo testuale tra utente ed

elaboratore) distribuita da Microsoft, su cui sono stati fatti girare trentanove test: quattro per un'azienda di piccole dimensioni, con 10 dipendenti e 2, 3, 4 e 5 progetti, e i restanti trenta per aziende di medie e grosse dimensioni, ossia con 20, 30, 40, 50 e 60 dipendenti e rispettivamente un numero crescente da 2 a 8 di progetti.

Come già affermato nell'Introduzione (4.1) il problema computazionale è stato affrontato mediante due solver già presentati nel Capitolo 3: CLINGO e DLV.

Per misurare il tempo impiegato da CLINGO per giungere alla soluzione delle assegnazioni Dipendente-Progetto, è stato inserito il comando:

```
Measure-Command {start-process clingo.exe -ArgumentList ("encoding.asp", "test-10-dipendenti-4-progetti.asp") -Wait}
```

Nell'encoding di DLV, invece, poiché il programma computa tutti gli answer set inseriti, è stato necessario inserire due ulteriori opzioni: "*-n=1*" e "*-filter=assegna*". Quest'ultima, in particolare, mostra soltanto il predicato che viene espressamente richiesto, in questo caso il predicato *assegna(D, P)*:

```
Measure-Command {start-process dlv.exe -ArgumentList ("encoding.asp", "test-10-dipendenti-4-progetti.asp", "-n=1", "-filter=assegna") -Wait}
```

I risultati, espressi in secondi, sono visibili nella Tabella 3, di seguito:

Test numero	Dipendenti	Progetti	CLINGO (in secondi)	DLV (in secondi)
1	10	2	1,160	1,531
2	10	3	1,158	1,208
3	10	4	1,165	1,186
4	10	5	1,211	1,171
5	20	2	1,260	1,176
6	20	3	1,168	1,177
7	20	4	1,243	1,120
8	20	5	1,359	1,169
9	20	6	1,125	1,167
10	20	7	1,124	1,221

11	20	8	1,163	1,204
12	30	2	1,154	1,183
13	30	3	1,176	1,167
14	30	4	1,120	1,282
15	30	5	1,195	1,125
16	30	6	1,144	1,169
17	30	7	1,168	1,128
18	30	8	1,153	1,238
19	40	2	1,135	1,193
20	40	3	1,234	1,183
21	40	4	1,077	1,177
22	40	5	1,222	1,175
23	40	6	1,130	1,125
24	40	7	1,178	2,18
25	40	8	1,140	2,176
26	50	2	1,205	1,166
27	50	3	1,173	1,175
28	50	4	1,180	1,248
29	50	5	1,164	2,171
30	50	6	1,184	2,142
31	50	7	1,171	2,170
32	50	8	1,145	3,165
33	60	2	1,143	1,184
34	60	3	1,172	1,134
35	60	4	1,136	2,171
36	60	5	1,195	2,142
37	60	6	1,177	3,190
38	60	7	2,184	3,252
39	60	8	2,195	4,173

Tabella 3. Il tempo, in secondi, impiegato da ognuno dei due solver.

Poiché questi due solver usano diverse tecniche euristiche, i risultati in termini di performance sono differenti tra loro. In questo caso, si può notare che, in media, CLINGO risolve il problema computazionale in tempistiche migliori rispetto a DLV, soprattutto quando il numero dei progetti sale a 7 e 8, fino al massimo di differenza nell'ultimo test, risolto da CLINGO in 2 secondi e 195 millisecondi, e da DLV in 4 secondi e 173 secondi.

Il confronto tra i due programmi è ben visibile nelle Figure 7 e 8:

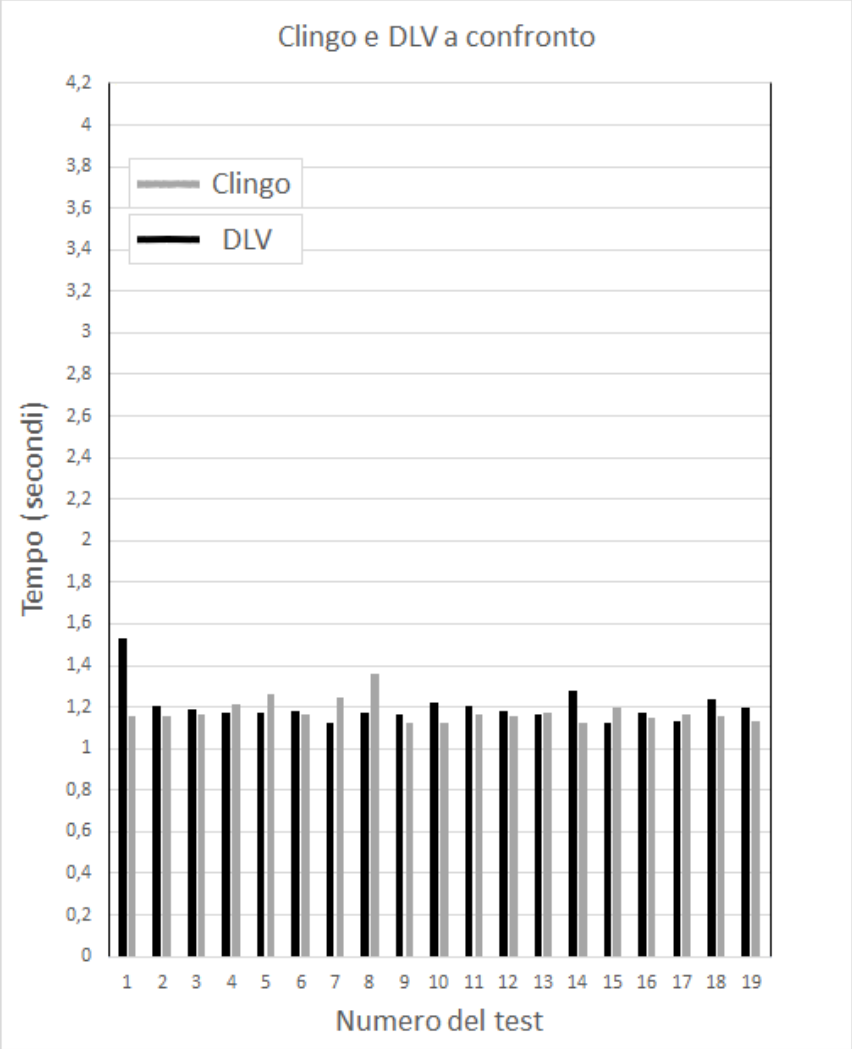

Figura 7. Confronto del tempo impiegato dai due solver per i primi 19 test.

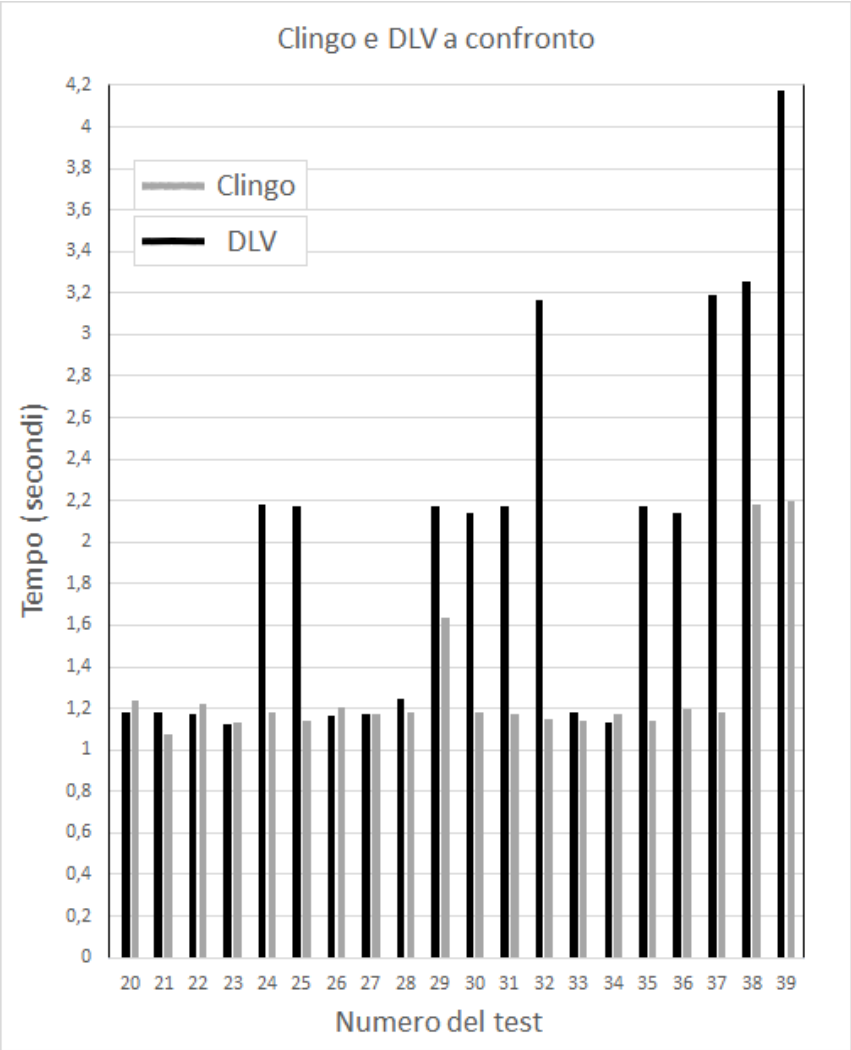

Figura 8. Confronto del tempo impiegato dai due solver per i restanti 20 test.

Ora si propone nel dettaglio un esempio di assegnazione, analizzando uno degli esempi più semplici, ossia quello del test con 10 dipendenti e 5 progetti.

Nell'encoding, i dipendenti erano stati identificati da numeri di matricola come quelli proposti negli esempi della sezione 4.2. Solo due di questi erano stati resi incompatibili (le matricole 62491 e 62488).

Nell'istanza considerata, tutti gli *skill* dei progetti, quindi i predicati *skillProgetto(P,S).*, hanno a che fare con il campo dell'informatica, essendo linguaggi di programmazione: si avranno, pertanto, C++, Ruby, Ada, Perl, COBOL, BASIC.

Per quanto riguarda i dipendenti, è importante precisare che non avevano tutti lo stesso numero di *skill*: il dipendente 62494, per esempio, aveva nove qualifiche (nella fattispecie, in: Ruby, C, BASIC, Fortran, Objective-C, C#, Visual Basic, Ada), mentre il dipendente 62489 ne aveva solo quattro (in: Ruby, COBOL, Perl e Obectivec).

Il risultato è visibile inserendo il comando per CLINGO, che, come affermato precedentemente, non necessita delle opzioni "*-n=1*" e "*-filter=assegna*", ma dell'aggiunta "*#show assegna/2.*" nell'encoding.

Oppure inserendo il comando DLV:

```
./dlv.exe encoding.asp .\test-10-dipendenti-5-progetti.asp -n=1 -filter=assegna
```

Prendendo in considerazione, a titolo di esempio, quello ottenuto dal solver DLV, si può osservare l'istanziazione di tutte le variabili del predicato *assegna(D, P)*, o, in altre parole, il risultato del problema di ottimizzazione proposto.

```
{assegna(matricola_62488,p_1), assegna(matricola_62489,p_1),
assegna(matricola_62494,p_2), assegna(matricola_62495,p_2),
assegna(matricola_62487,p_3), assegna(matricola_62490,p_3),
assegna(matricola_62492,p_4), assegna(matricola_62496,p_4),
assegna(matricola_62491,p_5), assegna(matricola_62493,p_5)}
```

Si può subito notare, per esempio, come questo answer set supporti il vincolo per cui ogni dipendente deve essere in un progetto, e quello per cui ogni progetto deve avere almeno due dipendenti. I dipendenti incompatibili, inoltre, non risultano mai nello stesso progetto: 62491 fa parte del progetto numero 5, mentre 62488 lavora nel numero 1.

Il planning di assegnazione risulta, come ci si aspettava, completato in modo ottimale.

Bibliografia

1. Mangione Corrado e Bozzi Silvio: *Storia della logica. Tomo 1*, CUEM, 2004
2. Francesco Berto: *Logica da zero a Gödel*. Pubblicato da Laterza, 2007.
3. L. Console, E. Lamma, P. Mello, M. Milano: *Programmazione Logica e Prolog*. Pubblicato da UTET editore, 2009
4. Lemmon Edward J.: *Elementi di logica*. Pubblicato da Laterza, 2008
5. Frank van Harmelen, Vladimir Lifschitz e Bruce W. Porter: *Handbook of Knowledge Representation*. In Foundations of Artificial Intelligence, volume 3. Pubblicato da Elsevier, 2008.
6. Michael Gelfond e Vladimir Lifschitz: *Classical Negation in Logic Programs and Disjunctive Databases*. Pubblicato da New Generation Computing, volume 9, numeri 3 e 4, pagine 365-386, 1991.
7. John McCarthy e Patrick J. Hayes: *Some Philosophical Problems from the Standpoint of Artificial Intelligence*. In Machine Intelligence 4, pagine 463-502. Pubblicato da Edinburgh University Press, 1969.
8. Sandewall Erik: *Cognitive Robotics Logic and its Metatheory: Features and Fluents Revisited*. Pubblicato da Electronic Transactions on Artificial Intelligence, volume 123, numero 1 e 2, pagine 271-273, 2000.
9. Sandewall Erik: *Review: M. Shanahan, Solving the Frame Problem*. Pubblicato da Artificial Intelligence, volume 123, numeri 1 e 3, pagine 271-273, 2000.
10. Victor W. Marek e Miroslaw Truszczynski: *Stable models and an alternative logic programming paradigm*. Pubblicato in Computing Research Repository, volume cs.LO/9809032, 1998.
11. Alain Colmerauer e Philippe Roussel: *The Birth of Prolog*. In History of Programming Languages Conference, pagine 37-52. Pubblicato da Association for Computing Machinery, 1993.
12. Martin Brain e Marina de Vos: *Answer Set Programming - a Domain in Need of Explanation: Position Paper*. In European Conference on Artificial Intelligence, pagine 37-48. Pubblicato da University of Patras, 2008.
13. Cuteri Bernardo, Dodaro Carmine, Ricca Francesco e Schüller Peter: *Constraints, Lazy Constraints, or Propagators in ASP Solving: An Empirical Analysis*. Pubblicato

in Theory and Practice of Logic Programming, volume 17, numeri 5 e 6, pagine 780-799, 2017.

14. Marcello Balduccini, Michael Gelfond, Richard Watson e Monica Nogueira: *The USA-Advisor: A Case Study in Answer Set Planning*. In International Conference of Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, volume 2173, pagine 439-442. Pubblicato da Springer, 2001.
15. Chitta Baral e Cenk Uyan: *Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming*. In International Conference of Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, volume 2173, pagine 186-199. Pubblicato da Springer, 2001.
16. Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub e Miroslaw Truszczynski: *The First Answer Set Programming System Competition*. In International Conference of Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, volume 4483, pagine 3-17. Pubblicato da Springer, 2007.
17. Chitta Baral e Michael Gelfond: *Reasoning Agents in Dynamic Domains*. In Logic-Based Artificial Intelligence, volume 597, pagine 257-279. Pubblicato da Springer, 2000.
18. Nicola Leone, Gianluigi Greco, Giovambattista Ianni, Vincenzino Lio, Giorgio Terracina, Thomas Eiter, Wolfgang Faber, Michael Fink, Georg Gottlob, Riccardo Rosati, Domenico Lembo, Maurizio Lenzerini, Marco Ruzzi, Edyta Kalka, Bartosz Nowicki, Witold Staniszkis: *The INFOMIX system for advanced integration of incomplete and inconsistent data*. In International Conference on Management of Data, pagine 915-917. Pubblicato da Association for Computing Machinery, 2005.
19. Chitta Baral: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Pubblicato da Cambridge University Press, 2010.
20. Victor A. Bardadym: *Computer-Aided School and University Timetabling: The New Wave*. In First International Conference, Practice and Theory of Automated Timetabling, Lecture Notes in Computer Science, volume 1153, pagine 22-45. Pubblicato da Springer, 1995.
21. Giovanni Grasso, Salvatore Iiritano, Nicola Leone e Francesco Ricca: *Some DLV Applications for Knowledge Management*. In International Conference on Logic

Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, volume 5753, pagine 591-597. Pubblicato da Springer, 2009.

22. Martin Gebser, Philipp Obermeier, Torsten Schaub, Michel Ratsch-Heitmann e Mario Runge: *Routing Driverless Transport Vehicles in Car Assembly with Answer Set Programming*. Pubblicato in Theory and Practice of Logic Programming, volume 18, numeri 3 e 4, pagine 520-534, 2018.
23. Giovanni Grasso, Salvatore Iiritano, Nicola Leone, Vincenzino Lio, Francesco Ricca, Francesco Scalise: *An ASP-Based System for Team-Building in the Gioia-Tauro Seaport*. In International Symposium of Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, volume 5937, pagine 40-42. Pubblicato da Springer, 2010.
24. Giovanni Grasso, Nicola Leone, Marco Manna e Francesco Ricca: *ASP at Work: Spin-off and Applications of the DLV System*. In Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday, Lecture Notes in Computer Science, volume 6565, pagine 432-451. Pubblicato da Springer, 2011.
25. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, Francesco Scarcello: *The DLV system for knowledge representation and reasoning*. Pubblicato da Association for Computing Machinery in Transactions on Computer Systems, volume 7, numero 3, pagine 499-562, 2006.
26. Giovanni Grasso, Nicola Leone e Francesco Ricca: *Answer Set Programming: Language, Applications and Development Tools*. In International Conference on Web Reasoning and Rule Systems, Lecture Notes in Computer Science, volume 7994, pagine 19-34. Pubblicato da Springer, 2013.
27. Francesco Ricca, Antonella Dimasi, Giovanni Grasso, Salvatore Maria Ielpa, Salvatore Iiritano, Marco Manna e Nicola Leone: *A Logic-Based System for e-Tourism*. Pubblicato in Fundamentae Informaticae, volume 105, numeri 1 e 2, pagine 25-55, 2010.
28. Chiara Cumbo, Salvatore Iiritano e Pasquale Rullo: *OLEX – A Reasoning-Based Text Classifier*. In European Workshop on Logics in Artificial Intelligence, Logics in Artificial Intelligence, volume 3229, pagine 722-725. Pubblicato da Springer, 2004.

29. Piero A. Bonatti, Francesco Calimeri, Nicola Leone, Francesco Ricca: *Answer Set Programming*. In A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, Lecture Notes in Computer Science, volume 6125, pagine 159-182. Pubblicato da Springer, 2010.
30. Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub e Marius Thomas Schneider: *Potassco: The Potsdam Answer Set Solving Collection*. Pubblicato in: Artificial Intelligence, volume 24, numero 2, pagine 107-124, 2011.
31. Carmine Dodaro e Marco Maratea: *Nurse Scheduling via Answer Set Programming*. In International Conference on Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, volume 10377, pagine 301-307. Pubblicato da Springer, 2017.
32. Mario Alviano, Carmine Dodaro, Nicola Leone e Francesco Ricca: *Advances in WASP*. In International Conference of Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, volume 9345, pagine 40-54. Pubblicato da Springer, 2015.
33. Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri e Francesco Scarcello: *Census Data Repair: a Challenging Application of Disjunctive Logic Programming*. In International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, Lecture Notes in Computer Science, volume 2250, pagine 561-578. Pubblicato da Springer, 2001.
34. Esra Erdem, Erdi Aker e Volkan Patoglu: *Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution*. Pubblicato in Intelligent Service Robotics, volume 5, numero 4, pagine 275-291, 2012.
35. Esra Erdem e Volkan Patoglu: *Applications of ASP in Robotics*. Pubblicato in Künstliche Intelligenz, volume 32, numeri 2 e 3, pagine 143-149, 2018.
36. Esra Erdem, Michael Gelfond e Nicola Leone: *Applications of Answer Set Programming*. Pubblicato in Artificial Intelligence, volume 37, numero 3, pagine 53-68, 2016.
37. Esra Erdem, Volkan Patoglu, Zeynep Gozen Saribatur, Peter Schüller e Tansel Uras: Finding optimal plans for multiple teams of robots through a mediator: A logic-based

approach. Pubblicato in Theory and Practice of Logic Programming, volume 13, numeri 4 e 5, pagine 831-846, 2013.

38. Benjamin Andres, David Rajaratnam, Orkunt Sabuncu e Torsten Schaub: *Integrating ASP into ROS for Reasoning in Robots*. In International Conference of Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, volume 9345, pagine 69-82. Pubblicato da Springer, 2015.
39. Shiqi Zhang, Mohan Sridharan e Jeremy L. Wyatt: *Mixed Logical Inference and Probabilistic Planning for Robots in Unreliable Worlds*. Pubblicato in Institute of Electrical and Electronics Engineers, volume 31, numero 3, pagine 699-713, 2015.
40. Kai Chen, Fangkai Yang e Xiaoping Chen: *Planning with Task-Oriented Knowledge Acquisition for a Service Robot*. In International Joint Conference on Artificial Intelligence, pagine 812-818. Pubblicato da International Joint Conference on Artificial Intelligence Press, 2016.
41. Giray Havur, Guchan Ozbilgin, Esra Erdem e Volkan Patoglu: *Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach*. In International Conference on Robotics and Automation, pagine 445-452. Pubblicato da Institute of Electrical and Electronics Engineers, 2014.
42. Ibrahim Faruk Yalciner, Ahmed Nouman, Volkan Patoglu ed Esra Erdem: *Hybrid conditional planning using answer set programming*. Pubblicato in Theory and Practice of Logic Programming, volume 17, numeri 5 e 6, pagine 1027-1047, 2017.