Progetto di Intelligenza Artificiale e Laboratorio
Modulo 1: Programmazione logica

Alessio Ballone, Francesco Odierna

1 Prolog: labirinto

1.1 Generazione del labirinto

La generazione del labirinto è stata effettuata tramite uno script Python. Dati in input altezza e larghezza viene generato un file di output che rappresenta la knowledge base del programma Prolog. Questa si compone di quattro tipologie di fatti:

- Numero di righe
- Numero di colonne
- Posizione iniziale: per convenzione è sempre $pos(1, 1)$
- Posizione finale: fornita in input e rappresentata da $finale(pos(x, y))$

I suddetti fatti rispecchiano il modello di knowledge base presentato a lezione. Lo script Python genera i muri in maniera casuale, inoltre è possibile variare la probabilità di generare un muro. Questo aspetto è fondamentale per il testing, in quanto in questo modo è possibile generare labirinti costituiti da un numero di muri arbitrario. Oltre alla dimensione del labirinto possiamo controllare anche quanto questo debba essere fitto manipolando la probabilità di generare un muro.

La posizione dei muri è casuale, difatti non sempre i labirinti generati hanno soluzione, al contrario, se il labirinto è troppo fitto è difficile che ci sia una soluzione.

Infine lo script genera su terminale una riproduzione visiva del labirinto fatta con simboli ASCII. Tramite la visualizzazione è possibile stabilire se il labirinto ha soluzione o meno, ovviamente per i labirinti di piccole dimensioni. In questo modo la fase di testing risulta più immediata.

1.2 Algoritmi implementati

Gli algoritmi Iterative Deepening Search (IDS), Iterative deepening A* (ID A*) e A* Search Algorithm, sono stati sviluppati secondo le strutture note in letteratura. Gli pseudo-codici degli algoritmi sono stati modificati in modo da adattarli al paradigma utilizzato in Prolog, in quanto presentavano dei loop. Prolog infatti non permette l'utilizzo di loop e altri formalismi di programmazione presenti in tutti e tre gli pseudo-codici. L'approccio utilizzato per risolvere questo gap sarà spiegato nelle sezioni dedicate. Inoltre il codice è altamente riutilizzabile in quanto permette di cambiare facilmente euristica e inoltre evita computazioni ripetute.

1.3 Scelte progettuali

La prima scelta progettuale riguarda il costo di una mossa: abbiamo scelto di considerare un costo unitario per tutte le mosse in modo tale da poter fare un confronto accurato nella fase finale di testing. La possibilità di variare il costo delle mosse è legata alla natura del problema, ad esempio nel caso del labirinto è ragionevole pensare che gli spostamenti abbiano tutti lo stesso costo; nel caso del mondo dei blocchi potrebbe essere più sensato diversificare i costi. Siccome le mosse possibili si intendono come la medesima azione in diverse direzioni la scelta più sensata è stata quella di mantenere il costo unitario.

La seconda scelta progettuale, come già accennato, è quella di considerare l'angolo in alto a sinistra come posizione iniziale e l'angolo in basso a destra come posizione finale. Sarebbe stato possibile variare queste posizioni dopo la generazione del labirinto, ma in questo caso sarebbe stato necessario un controllo caso per caso, in quanto i labirinti sono generati casualmente e nel cambio di posizione va controllata la presenza di muri. Inoltre nel caso in cui la posizione scelta non coincida con un muro, potrebbe coincidere con una posizione all'interno di un "recinto". Questo caso porta ad avere un labirinto senza soluzione. Il metodo di generazione dei labirinti e l'alta probabilità di trovarci in un recinto giustificano tale scelta.

1.4 Euristiche implementate

Abbiamo deciso di implementare le seguenti euristiche:

Distanza di Manhattan è la somma dei valori assoluti delle differenze tra $current\_cell.x$ e $goal.x$ e tra $current\_cell.y$ e $goal.y$, che rappresentano rispettivamente la distanza dello stato corrente dal goal rispetto alle coordinate x

e y, rispettivamente:

$$h = |current\_cell.x - goal.x| + |current\_cell.y - goal.y|$$

L'euristica va utilizzata quando è ci consentito spostarci solo in quattro direzioni: destra, sinistra, in alto e in basso.

Distanza Diagonale è il massimo dei valori assoluti delle differenze tra $current\_cell.x$ e $goal.x$ e tra $current\_cell.y$ e $goal.y$, che rappresentano rispettivamente la distanza dello stato corrente dal goal rispetto alle coordinate x e y, rispettivamente:

$$h = max(|current\_cell.x - goal.x|, |current\_cell.y - goal.y|)$$

L'eurisrica va utilizzato quando ci è consentito spostarci in otto direzioni.

1.5 Iterative Deepening Search (IDS)

IDS unisce l'efficienza spaziale della ricerca in profondità e l'efficienza temporale della ricerca in ampiezza (per nodi più vicini alla radice). IDS effettua una ricerca DFS per profondità diverse a partire da un valore iniziale. In ogni chiamata, DFS non può andare oltre la profondità fissata. È importante notare che visitiamo più volte nodi di livello superiore. L'ultimo livello (o profondità massima) viene visitato una volta, il penultimo livello viene visitato due volte e così via. Può sembrare costoso, ma in realtà non lo è poiché in un albero la maggior parte dei nodi si trova nel livello inferiore, quindi non il fatto di visitare i livelli superiori più volte non ha un impatto significativo.

Implementazione La regola da attivare è $ids(sogliaInfinita)$ che permette all'utente di impostare una soglia teoricamente infinita permettendo di lanciare l'algoritmo. Inoltre abbiamo la regola ausiliaria

$$ids\_aux(Profondita, SogliaInfinita, Soluzione)$$

che termina o al raggiungimento della soglia o trovando una soluzione.

Per la natura dell'algoritmo si inizia con una profondità pari a uno fino ad arrivare a zero. La scelta implementativa importante è nella chiamata della DFS, implementata dalla regola

$$cercaSoluzione(Profondita, Soglia, NuovaProfondita, Soluzione)$$

La ricerca DFS arriva al più alla profondità limite dell'iterazione corrente. Se la ricerca fallisce, la regola non fallisce e assegna alla variabile *NuovaProfondità* il valore della profondità corrente, in modo tale da non far fallire IDS e proseguire la ricerca.

Se la regola *cercaSoluzione* ha successo, allora alla variabile *NuovaProfondità* verrà assegnato esattamente il valore della soglia impostato dall'utente così da far terminare IDS con successo e fornire la soluzione. In questo modo la ricerca DFS non viene mai ripetuta sulla medesima profondità e si interrompe non appena viene trovata la soluzione.

La regola

$$ids\_aux(Profondita, Soglia, Soluzione)$$

è ricorsiva, incrementa ogni volta la profondità corrente e chiama *cercaSoluzione*.

Per quanto riguarda la DFS anch'essa si compone di un metodo ausiliario ed è ricorsiva. Ad ogni passo di ricorsione controlla che la soglia corrente non sia arrivata a zero.

Il backtracking è possibile grazie al punto di scelta nella regola applicabile: nel caso un path fallisca viene fatto backtracking su questa regola per provare un nuovo path.

Prestazioni L'algoritmo utilizza in maniera efficiente la memoria, ma non si può dire lo stesso per la complessità temporale, che è pari a $O(b^d)$, dove dove d è la profondità e b è il fattore di ramificazione. La ricerca in questo modo è molto inefficiente in quanto i nodi iniziali o comunque i nodi sul percorso della soluzione vengono riesplorati fino a quando non si raggiunge la profondità della soluzione. Questo aspetto incide negativamente sulle prestazioni e ne influenza l'applicabilità su labirinti molto ampi.

Terminazione La terminazione non è garantita per la natura algoritmica. Per limitare la ricorsione infinita abbiamo leggermente modificato l'algoritmo permettendo di impostare una soglia iniziale che deve essere un numero molto grande. In questo modo l'algoritmo termina in quanto è l'utente ad impostare la soglia massima. Se il labirinto non ha soluzione l'algoritmo continua comunque l'esecuzione fino alla soglia infinita impostata. Ciò è dovuto al fatto che l'algoritmo non ha una conoscenza assoluta e quindi non si rende conto che non esiste soluzione. È possibile che la ricerca DFS fallisca ad una profondità $P < N$, dove N è la soglia infinita, e che a profondità $P + 1 < N$ fallisca nuovamente, ma senza arrivare alla profondità $P + 1$ in quanto il fallimento avviene sempre a profondità P.

1.6 Iterative Deepening A* (ID A*)

È una variante di IDS in cui si introduce una funzione euristica per valutare il costo rimanente per raggiungere l'obiettivo. Poiché si tratta di un algoritmo di ricerca informato, il suo utilizzo della memoria è inferiore rispetto ad A*, ma a differenza della normale IDS si concentra sull'esplorazione dei nodi più promettenti e quindi non va alla stessa profondità ovunque nell'albero di ricerca. ID A* è una versione con memoria limitata di A*. Fa tutto ciò che fa A*, ha le caratteristiche ottimali di A* per trovare il percorso più breve, ma usa meno memoria di A*.

Euristica La soglia non viene aumentata casualmente, ma dipende da ciò che la funzione ricorsiva restituisce come nuova soglia. Durante la ricorsione ogni volta che viene raggiunto un nodo con un valore euristico superiore alla soglia questo non viene ulteriormente esplorato, ma viene memorizzato il valore euristico. Poiché incontriamo molti nodi di questo tipo viene restituito come soglia il minimo di questi valori.

Implementazione La regola da attivare è *ida*() che calcola il valore euristico della posizione di partenza ed inizia la ricerca con un metodo ausiliario. Siccome non è possibile implementare loop, viene fatta una ricorsione che termina al successo della DFS modificata, altrimenti aggiorna la soglia e torna in ricorsione. Per aggiornare la soglia si rende esplicito il fatto inerente alla soglia nella knowledge base. In particolare abbiamo due fatti principali: uno relativo alla soglia corrente ed uno relativo al valore euristico superiore alla soglia e minore tra quelli trovati fino a quel momento. Questa operazione di modifica delle soglie avviene nella DFS modificata (modificata proprio per questo aspetto). La regola principale è

$$dfs\_aux(PosizioneCorrente, Azioni, MosseEseguite, Soglia)$$

che calcola il valore euristico del nodo corrente con

$$calcolaEuristicaNodo(PosizioneCorrente, MosseEseguite)$$

e se minore della soglia corrente continua l'esplorazione, altrimenti aggiorna la soglia minima euristica e continua la ricerca dal successivo nodo da esplorare.

Tutti gli aggiornamenti di soglia avvengono con apposite regole che utilizzano *assert* e *retract* per modificare i due fatti principali nella knowledge base. Il calcolo dell'euristica è realizzato con un'apposita regola. Per questo algoritmo abbiamo scelto di implementare due versioni utilizzando due euristiche.

Prestazioni Come già detto per IDS, anche ID A* gestisce bene la memoria, mentre la complessità temporale è la medesima di IDS. C'è però da fare una puntualizzazione: l'effetto di una funzione euristica è di ridurre la complessità della ricerca da $O(b^d)$ a $O(b'^d)$, dove $b' < b$, riducendo il valore di fattore di ramificazione effettivo. Per quanto detto il confronto effettivo può quindi avvenire con IDS. Purtroppo per come è fatto l'algoritmo non è stato possibile calcolare fattore di ramificazione effettivo, in quanto per farlo sarebbe stato necessario asserire un fatto per ogni nodo generato. Risulta comunque intuitivo capire che per costruzione algoritmica ID A* esplora un minor numero di nodi rispetto IDS.

Terminazione A differenza di IDS, ID A* in labirinti privi di soluzione termina. Per lo stesso concetto espresso per la terminazione di IDS, ID A* continua ad iterare fino al valore di MAX_INT impostato di default. Non appena l'algoritmo aggiorna la soglia con questo valore infinito, allora il labirinto è privo di soluzione. Se la soluzione esiste, questa viene sicuramente trovata.

1.7 A* Search Algorithm

A* è un algoritmo di ricerca informato, il che significa che è formulato in termini di grafi pesati: a partire da uno specifico nodo iniziale di un grafo cerca un percorso di costo minore verso il nodo obiettivo. Per fare ciò mantiene un albero di percorsi originati dal nodo iniziale ed estendendo tali percorsi di un livello di profondità alla volta fino a quando non viene soddisfatto il goal. Di volta in volta A* deve determinare quale dei suoi percorsi estendere. La decisione è presa in base al costo del percorso corrente e una stima euristica del costo richiesto per estendere il percorso fino al nodo goal.

Euristica L'euristica viene calcolata per ogni nodo e, a differenza di ID A*, il valore euristico è un'informazione che è memorizzata nel nodo. Per garantire una maggiore efficienza a parità di valori di $f(n)$ si sceglie il nodo con valore euristico $h(n)$ minore. Nel nodo abbiamo il valore $f(n)$ e il valore euristico $h(n)$.

Implementazione La regola principale è $aStar()$. Quando termina, oltre a mostrare la soluzione, fornisce le informazioni relative all'esplorazione compiuta, quali costo della ricerca, fattore di ramificazione effettivo e profondità della soluzione trovata. Questa regola utilizza la consueta regola ausiliaria che effettua la vera e propria esplorazione a partire dal nodo iniziale.

La struttura dell'algoritmo è simile alla BFS presentata a lezione, difatti è composta dalla regola

$$generaMosse(Posizione\&AzioniFinOra,$$
$$AzioniPossibili, MosseEseguite, ListaPossibiliMosse)$$

che però abbiamo modificato rispetto alla versione presentata a lezione, in quanto occorreva introdurre nel nodo le informazioni del costo euristico.

In particolare, il calcolo del valore $g(n)$ viene effettuato contando il numero di azioni fino a quel nodo, mentre il calcolo di $h(n)$ viene effettuato da una regola ad hoc.

Una scelta implementativa importante riguarda la scelta del nodo candidato all'esplorazione. Ad ogni iterazione vengono generati nuovi nodi da esplorare che vengono aggiunti ad una lista; dopo molte iterazioni questa lista assume grandi dimensioni e abbiamo dovuto scegliere come trattarla. Le soluzioni esplorate sono state due:

- Soluzione 1: mantenere la lista ordinata, quindi utilizzare ad ogni iterazione un sorter che riordinava la lista inserendo in coda i nuovi nodi creati. Tuttavia questa implementazione si è rivelata molto inefficiente, in quanto dopo innumerevoli iterazioni l'operazione di ordinamento andava ad impiegare gran parte del tempo di esecuzione.

- Soluzione 2: la lista non è ordinata e aggiungiamo in coda ogni volta i nuovi nodi trovati. Ad ogni iterazione si scorre la lista cercando il nodo migliore. Questa scelta ha avuto un forte impatto sulle prestazioni. Anche se questa operazione comporta l'intero scorrimento della lista, in generale non impatta sul tempo di ricerca. L'operazione di scelta del nodo migliore appena descritta viene eseguita dalla regola:

  $$ordinaMosse(ListaNuoveMosse, PosizioneFinale,$$
  $$MosseRimanenti, MosseConMossaMiglioreInCima)$$

  che rimuove il nodo migliore dalla lista *ListaNuoveMosse* e lo inserisce in cima alla lista, cosicché la ricorsione di A* continui sul nodo migliore.

Per ragioni di efficienza abbiamo scelto la Soluzione 2.

Prestazioni Possiamo individuare due aspetti principali che hanno fortemente inciso sulle prestazioni:

1. Criterio di scelta del nodo da esplorare. Sebbene ci siano delle regole di libreria che permettono di cercare gli elementi all'interno della lista abbiamo preferito implementare una regola ad hoc, ossia:

   $$cercaMossaMigliore(ListaMosseConEuristica, MossaMigliore)$$

   che scorre l'intera lista e confronta a coppie di nodi. Il confronto avviene tra il nodo migliore trovato fino a quel momento ed il nodo che si sta scorrendo. In questo modo la scansione viene eseguita una sola volta in maniera molto rapida. Inoltre la scelta di rimuovere la mossa dalla lista ed inserirla in cima ha impattato fortemente le prestazioni. Utilizzando la regola discritta, rispetto alla regola di libreria, si ottiene una maggiore efficienza.

2. Ottimizzazione dell'utilizzo della memoria. Nelle prime implementazioni A* non terminava a causa di un elevato utilizzo di memoria, anche per istanze medie del labirinto. Eseguendo un debug approfondito e monitorando il comportamento dello stack, abbiamo notato che erano presenti moltissimi punti di scelta dove non necessari. Per tale motivo abbiamo utilizzato pesantemente il *cut*. Abbiamo notato che l'interprete Prolog lascia aperto ogni punto di scelta anche se la regola ha avuto successo, non sapendo che dopo non esistono altre regole con cui fare matching. Questo aspetto ha notevolmente alleggerito lo stack. Insieme ad altri accorgimenti sulle ridondanze delle liste siamo riusciti ad ottimizzare al meglio questo algoritmo.

Le soluzioni appena illustrate sono state applicate anche a IDS e ID A*, tuttavia con A* abbiamo notato una maggiore differenza e un notevole impatto sulle prestazioni.

Terminazione L'algoritmo termina sempre, che la soluzione esista o meno. Questo è possibile perché l'algoritmo sa quando ha esplorato tutto il labirinto, quindi ha abbastanza informazioni per riconoscere le istanze prive di soluzione.

1.8 Sperimentazione Prolog

In questa sezione illustreremo i risultati ottenuti in fase di sperimentazione. Abbiamo testato labirinti di diverse dimensioni così da avere un idea delle prestazioni al variare delle dimensioni. 10×10, 30×30, 60×60, 100×100, 500×500.

Per ogni configurazione abbiamo generato quattro tipi di labirinti:

- Senza muri
- Basso numero di muri. Il numero di muri è pari al 20% del totale
- Numero medio di muri: 33% del totale.
- Alto numero di muri: 50% del totale

Non è stato possibile andare oltre in quanto la generazione casuale dei labirinti porta ad istanze prive di soluzione. In totale sono stati geneati 15 labirinti con soluzione. Inoltre per ogni istanza dei quindici labirinti abbiamo generato un clone ponendo manualmente dei muri davanti alla posizione finale così da rendere l'istanza clone priva di soluzione. In questo modo è possibile stimare il tempo che l'algoritmo impiega a stabilire l'assenza di una soluzione.

Un esempio di file dove memorizziamo il labirinto è $60x60AS\_1770.pl$: la prima parte indica la dimensione, A indica Alta presenza di muri (le altre sono M media, B bassa), S indica Soluzione (N non soluzione), il numero finale 1770 indica la quantità di muri.

Considerando anche le istanze prive di muri arriviamo ad un totale di 35 labirinti. Abbiamo testato tutti gli algoritmi per un totale di 113 test, ognuno eseguito due volte così da avere una stima più robusta.

I parametri registrati in ogni misurazione sono i seguenti:

- Memoria utilizzata dal processo
- Profondità raggiunta inferenze prodotte
- Tempo di esecuzione

L'algoritmo viene testato per un massimo di cinque minuti così da poter condurre un maggior numero di test. Tutte le misurazioni sono state eseguite sullo stesso calcolatore senza alterare i parametri di default di Prolog per l'assegnazione delle risorse (50% CPU e 1 Gb memoria). Il processore utilizzato è un Intel Celeron N3050 1.60GHz Dual-Core (Average CPU Mark: 879 su *cpubenchmark.net*). Di seguito i commenti sui risultati ottenuti.

Un'osservazione che riguarda tutti i test è la seguente: nei labirinti con pochi muri, dove abbiamo più alternative, le performance sono leggermente peggiori. Ciò è dovuto al fatto che l'algoritmo esplorerà diverse strade. Sui labirinti densi di muri si hanno risultati migliori in quanto la presenza di muri taglia fuori molte delle alternative. La stessa considerazione vale per il costo della ricerca, il fattore di ramificazione effettivo e la memoria utilizzata.

IDS I labirinti testati hanno dimensione 10×10 e 30×30. L'istanza peggiore è il labirinto senza muri in quanto IDS esplora gli interi alberi fino alla profondità della soluzione.

All'aumentare della percentuale di muri l'algoritmo migliora passando da oltre $180s$ e quasi $500,000$ inferenze (caso 10×10 senza muri) ad appena $44ms$ e poco più di $100,000$ inferenze (caso 10×10 50% di muri).

Per quasi la totalità delle misurazioni delle istanze senza soluzione l'algoritmo non termina entro il limite temporale.

Per tutte le istanze 30×30 con soluzione l'algoritmo non termina entro il limite temporale.

In tutte le misurazioni la memoria non ha mai superato i $10Mb$. Sono state svolte un totale di 14 misurazioni.

ID A* Per quanto riguarda la sperimentazione di ID A* abbiamo suddiviso in due parti le misurazioni: una in cui utilizziamo l'euristica di Manhattan che consente quattro mosse e una in cui utilizzamo l'euristica diagonale che consente otto mosse. A parità di labirinto le soluzioni ottenute sono diverse e quindi non confrontabili.

Euristica di Manhattan I labirinti testati arrivando ad una dimensione massima di 60×60 in quanto oltre questa dimensione l'algoritmo non termina entro la soglia temporale.

A differenza di IDS le prestazioni migliori si ottengono con le istanze prive di muri, cosa prevedibile data la guida euristica. Inoltre l'algoritmo trova facilmente la soluzione se questa si trova alla prima soglia impostata, cioè sul percorso più breve. In quasi tutti i test sulle istanze prive di soluzione l'algoritmo non termina entro il limite temporale. In totale sono stati svolti 21 test.

Euristica Diagonale Anche in questo caso ci siamo fermati ad una dimensione massima del labirinto pari a 60×60. L'algoritmo non termina per le istanze prive di soluzione. Rispetto all'euristica di Manhattan l'algoritmo non termina su labitinti di dimensione 30×30. In totale sono stati effettuati 21 test.

Considerazioni generali Per il problema del labirinto ID A* non risulta particolarmente efficiente, anche utilizzando euristiche diverse, in quanto è pensato per istanze precise e circoscritte. I risultati prodotti evidenziano un'elevata complessità temporale rispetto alla complessità spaziale.

A* Come per ID A* abbiamo testato entrambe le euristiche. A differenza degli algoritmi precedenti abbiamo raccolto due dati in più: costo della ricerca e fattore di ramificazione effettivo. Questi due dati ci hanno permesso di fare ulteriori considerazioni sulle sperimentazioni eseguite.

Euristica di Manhattan Questa implementazione risulta la migliore di tutte in termini di performance, anche su labirinti di grandi dimensioni. Il labirinto più grande testato ha una dimensione pari a 1000×1000 con il 30% di muri. La soluzione è stata trovata in $180s$ a fronte di un occupazione di memoria pari a $10Mb$, un costo della ricerca pari a 23179 nodi e un fattore di ramificazione effettivo pari a 116. Inoltre l'algoritmo termina anche con istanze prive di soluzione, ad eccezione delle istanze 500×500.

I tempi di esecuzione sono ottimi e superano i $60s$ solo con istanze la cui dimensione è superiore a 500×500. Con labirinti privi di muri A* trova subito la soluzione, così come in labirinti con un numero di muri alto (50%). Per questa implementazione abbiamo svolto 30 test.

Euristica Diagonale Tale euristica non è performante come la precedente. L'istanza limite testata è il labirinto 500×500. L'algoritmo non termina con le istanze prive di soluzione.

Una considerazione da fare riguarda il costo della ricerca: anche se tutte le soluzioni trovate sono a profondità minore il costo della ricerca è molto alto in quanto si esplorano più nodi a causa del numero di mosse più elevato rispetto all'euristica di Manhattan. Ciò limita l'utilizzo dell'euristica diagonale in quanto la moltitudine di mosse possibili impatta fortemente sulle prestazioni e ciò lo si deduce anche dagli elevati fattori di ramificazione misurati.

In generale, per avere delle ottime prestazioni l'algoritmo necessita di istanze con un'alta percentuale di muri (oltre il 60%), cosa difficile da testare in quanto la probabilità di avere labirinti con la posizione finale "murata" è alta.

In generale A* risulta più performante di ID A* con euristica diagonale. In totale sono stati svolti 27 test.

Considerazioni generali A* risulta il miglior risolutore per il problema del labirinto. La nostra implementazione usa in maniera efficiente la memoria, infatti a fronte della knowledge base per labirinto 1000×1000 (7Mb) vengono utilizzati solo 10Mb di memoria. Ciò è dovuto alle scelte implementative descritte nella sezione dedicata.

2 ASP: orario scolastico

Per il problema scelto abbiamo sviluppato un insieme di vincoli in ASP in grado di trovare una soluzione valida. In più punti abbiamo fatto diverse scelte implementative per cercare di diversificare la soluzione e renderla il più possibile corretta e realistica. La presentazione del nostro lavoro si divide in quattro parti fondamentali: la prima riguardo i vincoli semplici, la seconda riguardo l'uso degli aggregati, la terza dove vengono formulati i vincoli che compongono la soluzione e l'ultima parte riguarda ulteriori vincoli per la correttezza ed efficacia della soluzione.

2.1 Dominio di riferimento

Fatti essenziali Nella modellazione del dominio evidenziamo sei fatti essenziali:

- Aule e laboratori. Per scelta implementativa abbiamo considerato come unico fatto $aula(NomeAula)$ assegnandogli undici nomi di colori.
- Docenti. Abbiamo docenti intesi come $docente(NomeDocente)$ a cui abbiamo dato nomi propri di persona.
- Insegnamenti. Abbiamo dieci insegnamenti e quattro insegnamenti extra da noi scelti $insegnamento(NomeInsegnamento)$.
- Ore della giornata. Queste possono essere intese in due modi: siccome sono presenti classi con attività extrascolastiche (nel nostro caso quattro), abbiamo inteso la giornata formata da dieci ore, ma solo le prime sei sono considerate come ore dedicate all'insegnamento delle materie ordinarie.
- Giorni della settimana. Questi vanno dal lunedì al venerdì.
- Classi. Intese come $1A$, $1B$, $2A$, $2B$, $3A$, $3B$. Per indicare che le classi della sezione A sono a tempo prolungato abbiamo modellato l'apposito fatto $tempo\_prolungato(NomeClasse)$

Fatti composti I fatti composti riguardano la relazione tra l'aula e l'insegnamento che è modellata con il seguente fatto:

$$aula\_riservata(NomeAula, NomeInsegnamento)$$

le scelte sono state puramente casuali e siccome non abbiamo distinto tra aule e laboratori abbiamo deciso di mettere a disposizione anche le aule per

le attività extrascolastiche. Abbiamo fatto in modo di assegnare le aule considerando le materie più affini (es. l'aula di scienze è la stessa di chimica, che è un'attività extrascolastica). Il secondo fatto composto riguarda le relazioni tra i docenti e gli insegnamenti, anche queste le abbiamo modellate in maniera puramente arbitraria creando il seguente fatto:

$$istruisce(NomeDocente, NomeInsegnamento)$$

Per quanto riguarda le attività extrascolastiche abbiamo deciso che ognuna di esse venisse insegnata da un docente diverso, e che quest'ultimo insegnasse la materia più inerente all'insegnamento extra. In questo modo si identificano anche i docenti che si occupano delle attività extrascolastiche.

2.2 Aggregati

Nel nostro sistema usiamo gli aggregati per definire quattro diverse tipologie di vincoli.

- Numero di ore per insegnamento. Uno dei vincoli fornito dal testo riguarda l'esatto numero di ore per cui ogni materia deve essere insegnata. Per fare ciò abbiamo introdotto il vincolo

  $$ora\_riservata(NomeInsegnamento, Classe, Giorno, Ora)$$

  all'interno di un aggregato per ogni materia, ponendo come estremi dell'aggregato il numero di ore richiesto. Con questo vincolo si mette in relazione l'insegnamento con la classe, il giorno e l'ora. Mettendo la classe come valore variabile fuori dall'aggregato si garantisce l'esatto numero di insegnamenti per ogni classe.

- Numero di ore giornaliere per insegnamento. Questo aggregato serve per rendere la soluzione più realistica ed impedisce che nel medesimo giorno sia insegnata la stessa materia. Abbiamo scelto di impostare il range da zero a due ore massimo.

- Insegnante unico per materia e per classe. Anche questo aggregato ottimizza la soluzione rendendola realistica. Impone che l'insegnate per una classe sia lo stesso per il medesimo insegnamento. In questo modo si impedisce che più docenti insegnino la stessa materia alla stessa classe nelle regolari ore (es. lettere viene insegnato quattro ore da Marco, tre a Andrea e tre da Francesca alla stessa classe). Per sviluppare questo vincolo abbiamo introdotto

  $$assegnazione(Docente, Insegnamento, Classe)$$

proprio per evidenziare il legame tra la classe, l'insegnamento ed il docente.

- Aula occupata. Questo aggregato è il primo fatto fondamentale che compone la soluzione. Grazie ai vincoli definiti in precedenza, che sono il fatto composto dell'aula riservata e il primo aggregato dell'ora riservata, definiamo:

$$aula\_occupata(Aula, Classe, Giorno, Ora)$$

  Ponendo gli estremi dell'aggregato ad uno si impone l'unicità del fatto.

2.3 Vincoli che compongono la soluzione

- Docente occupato. È il secondo vincolo che compone la soluzione ed è formato dal fatto composto *ora riservata* e dall'aggregato *assegnazione*. In questo modo si mette in relazione in docente e l'insegnamento assegnato alla relativa classe con un giorno e l'ora. Il vincolo ha la seguente forma:

$$docente\_occupato(Docente, Insegnamento, Classe, Giorno, Ora)$$

- Lezione. La soluzione finale si compone di questo predicato che è l'ultimo che abbiamo creato, proprio perché il calendario scolastico non è altro che l'insieme delle lezioni. Mettendo in relazione i due precedenti vincoli si può definire la lezione come l'unione di tutti i fatti essenziali quali: aula, docente, insegnamento, classe, giorno e ora. Il vincolo ha la seguente forma:

$$lezione(Aula, Docente, Insegnamento, Classe, Giorno, Ora)$$

2.4 Integrity constraints

Distinguiamo tra due categorie:

- Vincoli essenziali. I vincoli essenziali per una corretta soluzione sono i seguenti:

  1. Si impone che ci sia lezione nelle prime sei ore di ogni giorno per ogni classe.
  2. Si specifica che le classi non a tempo prolungato frequentino solo le prime sei ore.

3. Si specifica che due insegnamenti diversi non possono essere insegnati nel medesimo posto classe e ora; è un vincolo aggiuntivo che serviva per l'ora riservata.
4. Si specifica che il docente non può insegnare contemporaneamente in due classi.
5. I seguenti vincoli riguardano l'aula occupata. Si specifica che fissando il giorno, l'ora e l'aula, allora la classe è unica e, viceversa, fissando il giorno, l'ora e la classe, allora l'aula è unica.

- Ottimizzazione della soluzione. Per rendere la soluzione il più realistica possibile abbiamo introdotto tre ulteriori vincoli:

1. Le ore delle attività extrascolastiche sono considerate subito dopo le ore standard. In questo modo si evita che una soluzione preveda le prime sei ore di lezione e successivamente la nona e decima, di fatto lasciando due ore buca. Accorgimento non essenziale, ma realistico per una scuola media.
2. Questo vincolo riguarda solo l'insegnamento di lettere, che è l'unico ad avere due aule dedicate. Ciò impone che una classe segua una lezione di lettere in un'aula e nell'ora successiva non debba spostarsi in una seconda aula, così come per l'insegnante.
3. Se un insegnamento per una classe in un determinato giorno è presente due volte, allora deve essere svolto in due ore consecutive. In questo modo si risolve il fatto che nel medesimo giorno una classe segua un determinato insegnamento alla prima e all'ultima ora, ad esempio. Questo vincolo permette di ottimizzare la soluzione.

In Figura 1 troviamo raffigurato uno schema dei vincoli per comprendere meglio la struttura gerarchica della composizione dei vincoli principali.

2.5 Risultati

Nonostante i vincoli aggiuntivi implementati il numero di modelli generati risulta molto alto in quanto il possibile numero di combinazioni di elementi del dominio è molto elevato.

Nelle figure dalla 2 alla 12 sono riportati gli orari generati per ogni aula.

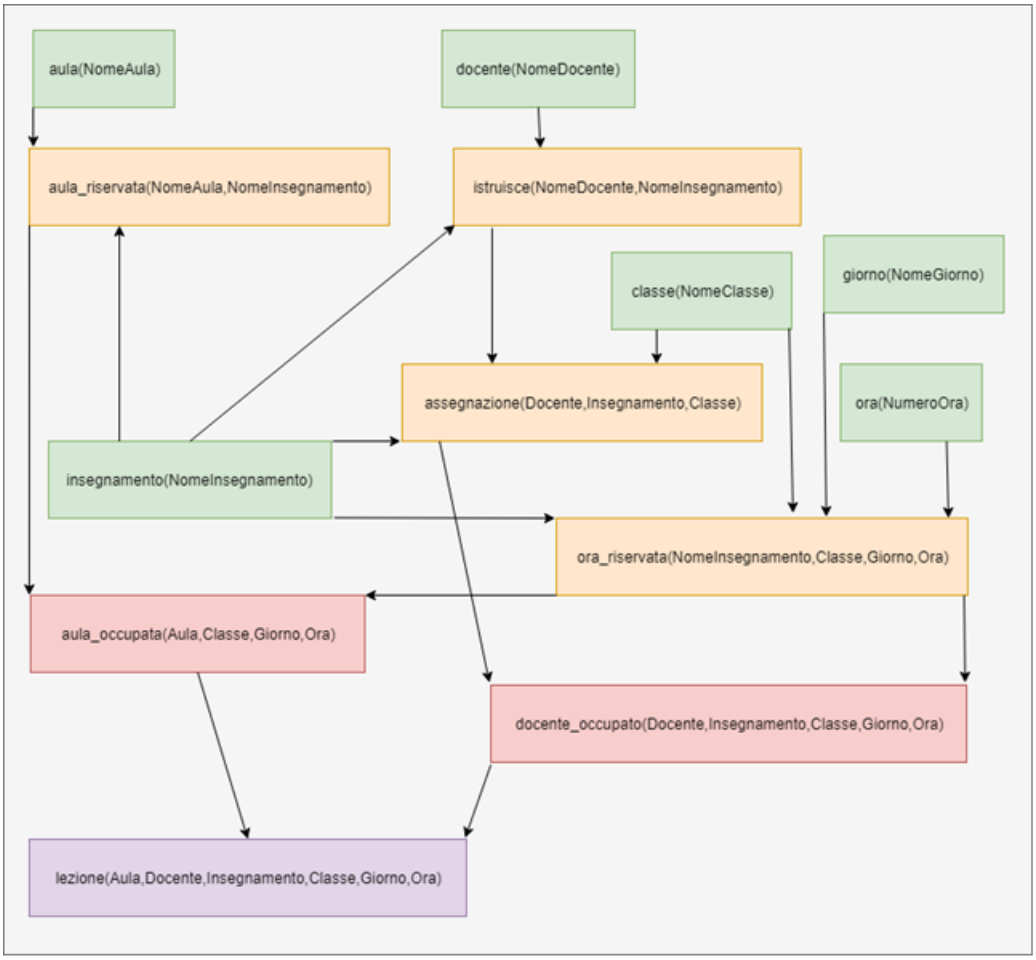

Figura 1: Schema dei vincoli

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1	Classe:b2 Materia: tecnologia Docente: erica				
2	Classe:b2 Materia: tecnologia Docente: erica				
3		Classe: a1 Materia: tecnologia Docente: erica			
4		Classe: a1 Materia: tecnologia Docente: erica			
5	Classe:a2 Materia: tecnologia Docente: erica		Classe: a3 Materia: tecnologia Docente: erica		
6	Classe:a2 Materia: tecnologia Docente: erica		Classe: a3 Materia: tecnologia Docente: erica		
7					
8					
9					
10					

Figura 2: Aula Azzurra

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1	Classe:b3 Materia: spagnolo Docente: luigi	Classe: a1 Materia: spagnolo Docente: luigi	Classe: a3 Materia: portoghese Docente: luigi	Classe: a2 Materia: portoghese Docente: luigi	
2	Classe:b3 Materia: spagnolo Docente: luigi	Classe: a1 Materia: spagnolo Docente: luigi	Classe: a3 Materia: portoghese Docente: luigi	Classe: a2 Materia: portoghese Docente: luigi	
3			Classe: a3 Materia: spagnolo Docente: luigi		
4			Classe: a3 Materia: spagnolo Docente: luigi		
5			Classe: a1 Materia: portoghese Docente: luigi	Classe: b1 Materia: spagnolo Docente: luigi	
6			Classe: a1 Materia: portoghese Docente: luigi	Classe: b1 Materia: spagnolo Docente: luigi	
7		Classe: a2 Materia: spagnolo Docente: luigi			
8		Classe: a2 Materia: spagnolo Docente: luigi			
9					
10					

Figura 3: Aula Arancione

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1	Classe:a3 Materia: musica Docente: rebecca	Classe: b3 Materia: musica Docente: rebecca			
2	Classe:a3 Materia: musica Docente: rebecca	Classe: b3 Materia: musica Docente: rebecca			
3					
4					
5		Classe: a1 Materia: musica Docente: rebecca	Classe: b2 Materia: musica Docente: rebecca		
6		Classe: a1 Materia: musica Docente: rebecca	Classe: b2 Materia: musica Docente: rebecca		
7					
8					
9					
10					

Figura 4: Aula Bianca

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1	Classe:b1 Materia: scienze Docente: martina	Classe: a2 Materia: scienze Docente: marco	Classe: a1 Materia: chimica Docente: marco	Classe: a1 Materia: chimica Docente: marco	
2	Classe:b1 Materia: scienze Docente: martina	Classe: a2 Materia: scienze Docente: marco	Classe: a1 Materia: chimica Docente: marco	Classe: a1 Materia: chimica Docente: marco	
3	Classe:a3 Materia: chimica Docente: martina	Classe: b3 Materia: scienze Docente: marco		Classe: a3 Materia: chimica Docente: martina	
4	Classe:a3 Materia: chimica Docente: martina	Classe: b3 Materia: scienze Docente: marco		Classe: a3 Materia: chimica Docente: martina	
5		Classe: b2 Materia: scienze Docente: martina		Classe: a2 Materia: chimica Docente: marco	
6		Classe: b2 Materia: scienze Docente: martina		Classe: a2 Materia: chimica Docente: marco	
7					
8					
9	Classe:a3 Materia: scienze Docente: martina				
10	Classe:a3 Materia: scienze Docente: martina				

Figura 5: Aula Blu

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1		Classe: b1 Materia: matematica Docente: martina	Classe: a2 Materia: matematica Docente: martina	Classe: b2 Materia: matematica Docente: martina	
2		Classe: b1 Materia: matematica Docente: martina	Classe: a2 Materia: matematica Docente: martina	Classe: b2 Materia: matematica Docente: martina	
3	Classe:a1 Materia: matematica Docente: marco	Classe: a3 Materia: matematica Docente: martina	Classe: b3 Materia: matematica Docente: martina	Classe: a1 Materia: geometria Docente: marco	
4	Classe:a1 Materia: matematica Docente: marco	Classe: a3 Materia: matematica Docente: martina	Classe: b3 Materia: matematica Docente: martina	Classe: a1 Materia: geometria Docente: marco	
5	Classe:b1 Materia: matematica Docente: martina			Classe: a3 Materia: matematica Docente: martina	
6	Classe:b1 Materia: matematica Docente: martina			Classe: a3 Materia: matematica Docente: martina	
7	Classe:a2 Materia: matematica Docente: martina	Classe: a1 Materia: matematica Docente: marco			
8	Classe:a2 Materia: matematica Docente: martina	Classe: a1 Materia: matematica Docente: marco			
9				Classe: a2 Materia: geometria Docente: martina	
10				Classe: a2 Materia: geometria Docente: martina	

Figura 6: Aula Gialla

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1	Classe:a1 Materia: religione Docente: francesca		Classe: b2 Materia: religione Docente: francesca		
2					
3	Classe:b3 Materia: religione Docente: francesca	Classe: b1 Materia: religione Docente: francesca	Classe: a2 Materia: religione Docente: francesca		
4					
5		Classe: a3 Materia: religione Docente: francesca			
6					
7					
8					
9					
10					

Figura 7: Aula Lilla

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1					
2					
3	Classe:a2 Materia: arte Docente: federica		Classe: b1 Materia: arte Docente: federica		
4	Classe:a2 Materia: arte Docente: federica		Classe: b1 Materia: arte Docente: federica		
5	Classe:b2 Materia: arte Docente: federica		Classe: b3 Materia: arte Docente: federica		
6	Classe:b2 Materia: arte Docente: federica		Classe: b3 Materia: arte Docente: federica		
7					
8					
9					
10					

Figura 8: Aula Marrone

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1		Classe: a3 Materia: educazioneFisica Docente: federico		Classe: b3 Materia: educazioneFisica Docente: federico	
2		Classe: a3 Materia: educazioneFisica Docente: federico		Classe: b3 Materia: educazioneFisica Docente: federico	
3			Classe: a1 Materia: educazioneFisica Docente: federico	Classe: b2 Materia: educazioneFisica Docente: federico	
4			Classe: a1 Materia: educazioneFisica Docente: federico	Classe: b2 Materia: educazioneFisica Docente: federico	
5		Classe: a2 Materia: educazioneFisica Docente: federico	Classe: b1 Materia: educazioneFisica Docente: federico		
6		Classe: a2 Materia: educazioneFisica Docente: federico	Classe: b1 Materia: educazioneFisica Docente: federico		
7					
8					
9					
10					

Figura 9: Aula Nera

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1			Classe: b1 Materia: lettere Docente: alice		
2			Classe: b1 Materia: lettere Docente: alice		
3	Classe:b2 Materia: lettere Docente: andrea	Classe: a2 Materia: lettere Docente: alice		Classe: b3 Materia: lettere Docente: andrea	
4	Classe:b2 Materia: lettere Docente: andrea	Classe: a2 Materia: lettere Docente: alice		Classe: b3 Materia: lettere Docente: andrea	
5	Classe:b3 Materia: lettere Docente: andrea	Classe: b3 Materia: lettere Docente: andrea	Classe: a2 Materia: lettere Docente: alice	Classe: b2 Materia: lettere Docente: andrea	
6	Classe:b3 Materia: lettere Docente: andrea	Classe: b3 Materia: lettere Docente: andrea	Classe: a2 Materia: lettere Docente: alice	Classe: b2 Materia: lettere Docente: andrea	
7			Classe: a1 Materia: lettere Docente: alice		
8			Classe: a1 Materia: lettere Docente: alice		
9					
10					

Figura 10: Aula Rossa

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1	Classe:a2 Materia: lettere Docente: alice		Classe: b3 Materia: lettere Docente: andrea	Classe: b1 Materia: lettere Docente: alice	
2	Classe:a2 Materia: lettere Docente: alice		Classe: b3 Materia: lettere Docente: andrea	Classe: b1 Materia: lettere Docente: alice	
3	Classe:b1 Materia: lettere Docente: alice	Classe: b2 Materia: lettere Docente: andrea	Classe: b2 Materia: lettere Docente: andrea	Classe: a2 Materia: lettere Docente: alice	
4	Classe:b1 Materia: lettere Docente: alice	Classe: b2 Materia: lettere Docente: andrea	Classe: b2 Materia: lettere Docente: andrea	Classe: a2 Materia: lettere Docente: alice	
5	Classe:a1 Materia: lettere Docente: alice	Classe: b1 Materia: lettere Docente: alice		Classe: a1 Materia: lettere Docente: alice	
6	Classe:a1 Materia: lettere Docente: alice	Classe: b1 Materia: lettere Docente: alice		Classe: a1 Materia: lettere Docente: alice	
7	Classe:a3 Materia: lettere Docente: andrea	Classe: a3 Materia: lettere Docente: andrea	Classe: a3 Materia: lettere Docente: andrea	Classe: a3 Materia: lettere Docente: andrea	
8	Classe:a3 Materia: lettere Docente: andrea	Classe: a3 Materia: lettere Docente: andrea	Classe: a3 Materia: lettere Docente: andrea	Classe: a3 Materia: lettere Docente: andrea	
9		Classe: a1 Materia: lettere Docente: alice			
10		Classe: a1 Materia: lettere Docente: alice			

Figura 11: Aula Verde

	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì
1		Classe: b2 Materia: inglese Docente: lorenzo		Classe: a3 Materia: tedesco Docente: lorenzo	
2	Classe:a1 Materia: inglese Docente: lorenzo	Classe: b2 Materia: inglese Docente: lorenzo	Classe: b2 Materia: inglese Docente: lorenzo	Classe: a3 Materia: tedesco Docente: lorenzo	
3				Classe: b1 Materia: inglese Docente: lorenzo	
4	Classe:b3 Materia: inglese Docente: lorenzo	Classe: b1 Materia: inglese Docente: lorenzo	Classe: a2 Materia: inglese Docente: lorenzo	Classe: b1 Materia: inglese Docente: lorenzo	
5	Classe:a3 Materia: inglese Docente: lorenzo			Classe: b3 Materia: inglese Docente: lorenzo	
6	Classe:a3 Materia: inglese Docente: lorenzo	Classe: a3 Materia: inglese Docente: lorenzo		Classe: b3 Materia: inglese Docente: lorenzo	
7				Classe: a2 Materia: inglese Docente: lorenzo	
8				Classe: a2 Materia: inglese Docente: lorenzo	
9		Classe: a2 Materia: tedesco Docente: lorenzo	Classe: a1 Materia: inglese Docente: lorenzo		
10		Classe: a2 Materia: tedesco Docente: lorenzo	Classe: a1 Materia: inglese Docente: lorenzo		

Figura 12: Aula Viola