UNIVERSITÀ DEGLI STUDI DI PALERMO

SCUOLA POLITECNICA
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Ingegneria Chimica, Gestionale, Informatica e Meccanica

Tesina per il Corso di **INTELLIGENZA ARTIFICIALE**
Docente: **Chiar.mo Prof. Salvatore Gaglio**

STUDIO ED IMPLEMENTAZIONE DELL' OBSERVATION PACK ALGORITHM

Presentata da:

Nicola Ciaco

Relatori:

Chiar.mo Prof. Marco Ortolani

Dott. Pietro Cottone

Anno Accademico
2015\16

Indice

Elenco delle figure

Elenco delle tabelle

Acronimi

ObP	Observation Pack
IIR	Inferenza Induttiva Regolare
P	enunciati premessa
BK	conoscenze di background
H	ipotesi induttiva
MQ	Membership Query
EQ	Equivalence Query
$\mathcal{L}$	linguaggio target
MAT	Minimally Adeguate Teacher
FSA	Finite State Automata
DFA	Deterministic Finite Automata
FSM	Finite State Machine
NFA	Non-deterministic Finite Automata
DT	Discrimination Tree
LCA	least common ancestor

Introduzione

L'oggetto di studio di questo elaborato è un algoritmo utilizzato per l'appredimento di linguaggi regolari: l' **Observation Pack (*ObP*)**. L'apprendimento di linguaggi regolari è collocabile nel più ampio tema dell' inferenza grammaticale — usata in una varietà di campi come pattern recognition, biologia computazionale e elaborazione del linguaggio naturale — ed è il processo di automaticamente inferire una grammatica esaminando delle stringhe di un linguaggio sconosciuto. Il modus operandi degli algoritmi d'inferenza grammaticale o regolare e dell'*ObP* è inquadrabile all'interno dell"apprendimento per induzione e per questo è spesso detta **Inferenza Induttiva Regolare (*IIR*)** (o grammaticale). Allo stato dell'arte l' *ObP* costituisce il secondo algoritmo di riferimento nell'ambito dell'apprendimento di linguaggi regolari. L'algoritmo più performante è invece il più recente TTT algorithm [15]. Si può trovare una presentazione completa dell' *ObP* in [7] e una sua implemetazione nella libreria LearnLib[1] . Corredata a questa tesina vi è anche l'implementazione dell' *ObP* in C++ , codice che è stato integrato in una libreria preesistente. Il lavoro qui esposto si divide in quattro parti. Nel primo capitolo si parlerà dell'inferenza induttiva, e riferendosi alla classificazione proposta in [9], si inquadrerà questo meccanismo nel complesso meccanismo dell'apprendimento. Inoltre, dopo avere messo a confronto l'induzione con la deduzione e l'abduzione verranno passati in rassegna le peculiarità del processo induttivo Nel secondo capitolo si definirà l'inferenza induttiva grammaticale specificando e approfondendo i punti di cui si compone. Poi saranno scandagliati i risultati teorici e i limiti dell'*IIR*. Nel terzo capitolo saranno presentati brevemente i principali algoritmi dell'*IIR* RPNI e EDSM nel red-blue framework e poi nel paradigma dell'Active Learning sarà introdotto e approfondito L* [1] che può essere considerato il capostipite dell'*ObP* Nel quarto capitolo verrà esposto in maniera dettagliata la ratio che muove l'*ObP* le strutture dati e le prestazioni. Inoltre verranno riportate le scelte discostanti dal riferimento principale dell'algoritmo [7] e le motivazioni. Verranno proposti infine i risultati sperimentali per mettere a confronto le prestazioni dell'algoritmo oggetto di studio con L*

[1] http://www.learnlib.de/ Qui *ObP* è menzionato come Discrimination Tree

Capitolo 1

Inferenza Induttiva

Il metodo induttivo o induzione è un procedimento logico per cui dalla constatazione di fatti particolari si risale ad affermazioni o formulazioni generali. Si suole quindi indicare con il termine induzione il passaggio dal *particolare* al *generale*. Con il termine **Inferenza Induttiva** si indica un processo che partendo da degli esempi specifici congettura delle regole generali. L'inferenza induttiva gioca un ruolo fondamentale nel più vasto scenario dell'apprendimento ed in ogni contesto che si prefigge la scoperta di strutture universali. L'applicazione di questo metodo scientifico ,intrinseco agli esseri intelligenti, all'interno delle macchine ha portato alla nascita di diversi filoni di ricerca. Uno dei più rilevanti tratta dei complessi meccanismi che consento ad un uomo di imparare un linguaggio.

1.1 Apprendimento

1.1.1 Definire l'apprendimento

Insieme alla capacità di pianificare cioè elaborare piani, la capacità di apprendere è ritenuta uno dei segni distintivi di un sistema intelligente. Un sistema si può considerare autonomo fintantochè le sue azioni sono determinate dalle esperienze pregresse e dalle percezioni correnti, invece che dal suo progettista (si pensi agli agenti stimolo-risposta). Senza la capacità di apprendimento un sistema non sarà in grado di operare con successo in qualsiasi ambiente ma solo in quelli previsti dal suo progettista. Nonostante la grande importanza dell'apprendimento, una conclusione largamente diffusa è che non sia possibile darne una definizione precisa: si procede invece analizzando gli effetti che l'apprendimento ha eventualmente prodotto.
Due concetti rivestono un ruolo importante nell'apprendimento:

- Il miglioramento delle capacità del sistema che apprende
- L'acquisizione di nuova conoscenza

Simon [14] approfondisce cosa significa migliorare le capacità di un sistema mediante l'apprendimento: *L'apprendimento identifica delle modifiche in un sistema che sono*

adattive, nel senso che consentono al sistema di svolgere lo stesso goal, o goals analoghi, in maniera migliore nel futuro. E' doveroso però osservare che esistono sistemi che migliorano nel tempo senza essere soggetti a nessun processo di apprendimento e che esistono degli scenari in cui non è facile calare la definizione data da Simon.

L'altro fattore che contraddistingue l'apprendimento è l'acquisizione di nuove conoscenze che presuppone a monte una rappresentazione della conoscenza in maniera descrittiva o iconica per potere rappresentare la nuova conoscenza avvenuta mediante l'apprendimento. In quest'ottica *l'apprendimento è creare e modificare rappresentazioni di ciò che è stato sperimentato.* Laddove con sperimentare si intende sia l'informazione proveniente dall'apparato sensoriale dell'agente sia ciò che il sistema recepisce mediante processi interni (ad esempio ripetere più volte una frase tra sè e sè ci consente d'impararla nonostante non è avvenuto nessuno stimolo dai nostri sensi).Da questo punto di vista apprendere significa costruire una rappresentazione della realtà anzichè un miglioramento delle capacità dell'agente, aspetto quest ultimo considerato come una conseguenza.

E' possibile quindi constatare il grado di apprendimento di un sistema misurando i miglioramenti nel portare a compimento un certo job dopo l'apprendimento. (Miglioramenti che implicitamente sono considerati una conseguenza della rappresentazione interna dell'agente della realtà esterna). Si fa presente che in questa caratterizzazione si assume che l'agente abbia un obbiettivo e che tale obbiettivo sia conosciuto all'osservatore che valuta l'apprendimento.

1.1.2 Una possibile schematizzazione

Esistono diverse possibilità di classificare i fattori che influenzano l'apprendimento. Michalski [11] propone una divisione basata sulle caratteristiche del sistema che apprende.
Michalski esegue una prima distinzione in base alla quantità di conoscenze iniziali di cui il sistema è dotato. Ai due estremi della classificazione troviamo le reti neurali artificiali e i sistemi esperti. Nel contesto dei sistemi dotati di scarse conoscenze iniziali le reti neurali sono uno strumento largamente usato: le connessioni dei neuroni che costituiscono il sistema, sono determinate in maniera essenziale dagli esempi presentati e solo marginalmente dai valori iniziali (di solito casuali) delle connessioni. Nella progettazione di un sistema esperto invece una grande quantità di informazione viene fornita al sistema. Un altro approccio, suggerito da Michalski, propone di suddividere i sistemi artificiali che apprendono in base al tipo di manipolazione eseguita dal **learner**(sistema che apprende) sull'informazione proveniente dall'esterno. In ogni processo di apprendimento il *learner* trasforma l'informazione fornita da un *teacher*, o più in generale da un **informant** sorgente di informazione, in una nuova forma che viene poi memorizzata per usi futuri. Questa trasformazione dell'informazione, che fa uso anche delle conoscenze già possedute dal *learner* viene chiamata **inferenza**. Il tipo di trasformazione eseguita determina la strategia di ap-

prendimento di cui il sistema fa uso. Si possono distinguere,seguendo esattamente Michalski in [9] , cinque diverse strategie:

- Apprendimento per *imitazione*
- Apprendimento per *istruzioni*
- Apprendimento per *deduzione*
- Apprendimento per *analogia*
- Apprendimento per *induzione*

Queste strategie sono elencate in ordine crescente di complessità del learning e decrescente di difficoltà del teaching. Michalski attua questa classificazione restrigendo l'ambito di applicabilità al ***concept learning*** una branca del machine learning. Un sistema intelligente deve essere abile nel classificare alcuni oggetti,eventi o comportamenti come equivalenti per raggiungere un determinato goal. Detto in maniera succinta un sistema intelligente deve essere capace di individuare i *concetti*.

Definizione 1.1. Un **concetto** è una classe di equivalenza per cui esiste un metodo operativo che permette di discriminare le istanze come appartenenti o non appartenti al concetto.

Dove le istanze sono le singole entità della classe di equivalenza (del concetto),cioè gli esempi presentati dall'informant. Un *learner* impara un concetto quando ,tramite una procedura effettiva, è in grado di distinguere le entità che appartengono al concetto da quelle che non appartengono.Adesso si prenderanno brevemente in esame le cinque strategie di apprendimento contestualizzandole nel *concept learning*:

Apprendimento per imitazione Questo è il caso estremo in cui il *learner* non deve effettuare alcuna inferenza sulle informazioni che gli provengono dall' *informant*. Infatti questo metodo è anche detto da Michalski impianto diretto di conoscenza (meglio conosciuto ancora come rote learning) proprio perchè il *learner* non deve fare altro che inidicizzare l'informazione per poterla poi recuperare. In questo caso l'*informant* fornirà una descrizione del concetto in input al *learner*. Questa strategia è usata quando uno specifico algoritmo per riconoscere un concetto è implementato su un calcolatore(oppure vi è a disposizione un database di fatti che permette di riconoscere il concetto) . Ad esempio nei primi programmi che giocavano a scacchi si salvavano i risultati dell'esplorazione del grafo di ricerca (in alcuni punti che rappresentano possibili situazioni in una partita) in un albero di gioco in modo che quando una situazione già memorizzata si fosse presentata in una partita reale si potesse risparmiare spazio e tempo di esecuzione.

Apprendimento per istruzioni In questo caso il *learner* acquisisce un concetto da un *teacher*, o da un'altra forma organizzata di informazione, come una publicazione o un libro, ma non copia direttamente in memoria l'informazione

acquisita. Nell'apprendimento per istruzioni le trasformazioni sull'informazione eseguite dal *learner* sono la selezione e la riformulazione a livello sintattico. Il processo di apprendimento può consistere nel selezionare i fatti più importanti e poi trasformarli in una forma più appropriata. Un programma che costruisce una database di fatti e regole sulla base di una conversazione con un utente è un esempio di sistema che apprende per istruzioni.

Apprendimento per deduzione Il *learner* acquisisce un concetto deducendolo dalle conoscenze fornite dall' *informant* insieme a quelle che il sistema già possedeva. Inoltre, questa strategia include ogni processo nel quale la conoscenza appresa è il risultato di una trasformazione che preserva la verità delle informazioni generate dall' *informant* e di ciò che viene inferito. All'interno dell' apprendimento dei concetti, l'apprendimento per deduzione tramite il processo inferenziale trasforma una definizione non adoperabile per discriminare il concetto, in una definizione operativa adatta a questo scopo. Ad esempio dal fatto che una giarra sia un oggetto stabile e trasportabile, si può dedurre che la brocca ha un fondo piatto e un manico.

Apprendimento per analogia Il *learner* acquisisce un nuovo concetto modificando la definizione di un concetto simile già noto. Anzichè formulare una descrizione del concetto ex novo, il sistema adatta una descrizione esistente modificandola appropriatamente per il nuovo scopo. Ad esempio se già si conosce una regola che definisce il concetto di arancia, per imparare il concetto di mandarino si possono notificare le differenze e le similitudini tra arancia e mandarino. L'apprendimento per analogia può essere visto come un incrocio tra l'apprendimento deduttivo e quello induttivo. Attraverso l'inferenza induttiva si possono determinare le caratteristiche generali o le trasformazioni che unificano i concetti confrontati. Poi, attraverso un'inferenza deduttiva si possono derivare le proprietà caratterizzanti possedute dal concetto che deve essere appreso

Apprendimento per induzione In questa strategia il *learner* acquisisce un concetto effettuando inferenza induttiva sui fatti forniti dall' *informant* o in base a delle osservazioni su tali fatti. Esistono due differenti forme di questa strategia:

1. **Apprendimento da esempi**
   Al *learner* partendo da degli esempi specifici (istanze del concetto) ed eventualmente dei controntroesempi induce una descrizione del concetto catturando la struttura generale . Si assume che il concetto esiste e che esiste anche un metodo effettivo per testare l'appartenenza di un'istanza ad un concetto. Il compito del *learner* è determinare una descrizione del concetto analizzando le singole istanze del concetto. Questa strategia è utilizzata nell' *IIR*
2. **Apprendimento per osservazione e scoperta**
   Il *learner* analizza le entità in input e determina che qualche sottoinsieme

di queste entità può essere raggruppato in un singolo concetto.Poichè , diversamente dall'apprendimento da esempi, non c'è un *teacher* che conosce in anticipo i concetti questa strategia è talvota menzionata come *unsupervisioned learning*. Un esempio è il *clustering* cioè il partizionamento di una collezione di oggetti all'interno di gruppi o classi che avviene in maniera gerarchica; l'eredità gioca un ruolo importante: se un entità è riconosciuta appartenere ad un determinato concetto erediterà da esso e dai concetti più in alto nella gerarchia tutte le proprietà . Ad esempio se si apprende che Freddy è un elefante, allora si può,senza vedere Freddy, dire che ha la proboscide e tutte le proprietà degli elefanti e più in generale anche degli erbivori e dei mammiferi.

1.2 Induzione

L'induzione è quel procedimento logico che permette di passare dal particolare all'universale. Questa definizione è troppo semplice e non spiega tutte le componenti in gioco nel processo induttivo. A tal fine si seguirà ancora [9]. Qui le principali componenti induttive sono più precisamente distinte e specificate nel contesto della manipolazione simbolica:

Dati i seguenti elementi di partenza

- Gli **enunciati premessa (*P*)** che comprendono fatti, generalizzazioni intermedie, specifiche osservazioni che forniscono informazioni su oggetti,fenomeni,processi eccetera. Costituiscono l'input del processo inferenziale.
- Le **conoscenze di background (*BK*)** che contengono concetti generali o specifici del dominio, che permettono di interpretare gli enunciati premessa e le regole rilevanti per l'inferenza. Ed includono concetti precedentemente imparati, vincoli del dominio, relazioni di causalità,goals dell'inferenza,e metodi per valutare la bontà di una congettura in base al goal (criterio di preferenza)

si determina alla fine dell'inferenza induttiva

- Una **ipotesi induttiva (*H*)** che implica gli enunciati premessa nel contesto delle conoscenze di background ed è l'ipotesi migliore in base al criterio di preferenza.

Si dice che H implica fortemente P nel contesto di BK se usando BK e l'inferenza deduttiva P è una conseguenza logica di H. Schematizzando si ottiene l'equazione

$$H \vee BK \implies P \tag{1.1}$$

che è vera con tutte le possibili *interpretazioni*. In contrasto H implica debolmente gli enunciati premessa nel contesto delle BK se usando le BK e l'inferenza deduttiva P è solo una conseguenza plausibile ma non una conseguenza logica. Michalski fornisce un esempio di quanto appena detto:

Enunciati premessa
Aristotele era greco
Socrate era greco
Platone era greco

Conoscenze di background
Socrate, Aristotele e Platone erano filosofi
Sono vissuti nell'antichità
I greci sono persone
I filosofi sono persone
Criterio di preferenza: Si preferiscono le ipotesi più corte e più utili per decidere la nazionalità dei filosofi

Le **ipotesi induttive** sono:

1. I filosofi che hanno vissuto nell'antichità erano greci
2. Tutti i filosofi sono greci
3. Tutte le persone sono greche

L'ipotesi da preferire,in base al criterio di preferenza, è la 2, perchè è più breve della 1 e più specifica della 3; consente a differenza della 1 di determinare la nazionalità di tutti i filosofi. Si può di mostrare che questa ipotesi induttiva è un'ipotesi forte,poichè P risulta essere una conseguenza logica di H e delle BK.
Supponiamo di aggiungere alla premessa gli enunciati Locke era inglese e Hume era inglese e di modificare le BK aggiungendo il fatto che sia Locke che Hume erano filosofi. In questo caso una ipotesi induttiva forte potrebbe essere che tutti i filosofi erano greci, con l'eccezione di Locke e Hume. Mentre una ipotesi induttiva debole potrebbe essere che alcuni filosofi erano greci. Dal fatto che Platone era un filosofo e sulla base di questa nuova ipotesi debole non consegue che Platone era greco, consegue solo che c'è la possibilità che Platone fosse greco.
Senza pretesa di esaustività si accenna ad altri tipi di inferenza presenti nel pensiero logico con lo scopo di fare emergere le peculiarità dell'inferenza induttiva. L'inferenza sta alla base dell'apprendimento. Seguendo [10] l'apprendimento si può sintetizzare in $apprendimento = inferenza + memorizzazione$ (definizione leggermente diversa da quella data in 1.1.1) quindi una completa teoria dell'apprendimento deve includere una completa teoria dell'inferenza [10]. Viene innanzitutto generalizzata l'equazione (1.1) valida solo per l'induzione ottenendo:

$$\mathrm{Q} \vee BK \models \mathrm{C} \tag{1.2}$$

detta **equazione fondamentale** per l'inferenza. Poi Michalsky effettua una prima suddivisione tra i metodi d'inferenza:

1. conclusivi
2. contingenti

Nel secondo caso nell' equazione (1.2) C è solo una plausibile,parziale,probabilistica conseguenza logica delle BK e di Q. Nell'inferenza conclusiva invece la conseguenza logica è garantita. Le proprietà dell'inferenza induttiva sono confrontate con quelle dell'inferenza deduttiva ed emerge che sono duali come si vede in figura 1.1 La

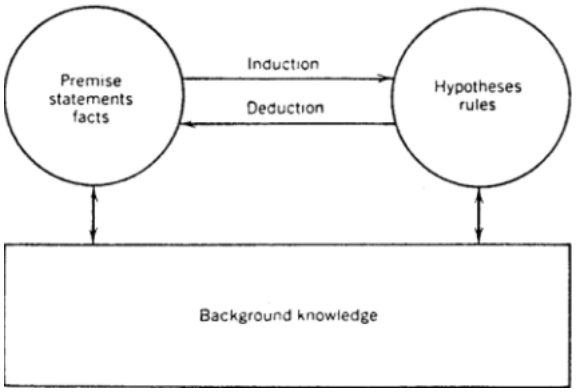


Figura 1.1: Relazione tra deduzione e induzione

relazione logica (1.2) succintamente cattura la relazione tra i due tipi d'inferenza. L'inferenza deduttiva deriva logicamente C date BK e Q. L'inferenza induttiva invece va ad ipotizzare Q date BK e C. La deduzione è il processo di determinare una conseguenza logica a partire da una conoscenza data, ed è *truth-preserving* (C deve essere vero se BK e Q sono vere). In contrasto l'induzione sta ipotizzando un Q che insieme con BK implica l'input C, ed è *false-preserving* (se C è falso allora anche Q deve essere falso. Cioè se l'input in ingresso è falso anche le ipotesi congetturate saranno false). La deduzione contingente invece suona come debole in quanto è debolmente *true-preserving* cioè produce conseguenze che possono essere vere in alcune situazioni e false in altre. Analogamente l'induzione contingente è debolmente *false-preserving*.
In [10] l'inferenza viene considerata come un processo che prende in Input un enunciato e tramite le BK già possedute (ed eventualmente la conoscenza dei criteri di preferenza per il goal che permette di restringere tutte le possibili ipotesi tra le quali scegliere) fornisce un enunciato in Output .In quest'ottica le proprietà dell'induzione sono messe in risalto dal confronto con quelle della deduzione e dell'abduzione fornendo per ciascuno di essi degli esempi chiarificatori.

1. DEDUZIONE tabella 1.1 Riferendosi all'equazione fondamentale (1.2) l'Input sta per Q e l'Output sta per C. L'Input consiste in enunciato che afferma l'appartenenza di un elemento a ad X. Le BK sono costituite da un enunciato

che assegna una certà proprietà q agli elementi dell'insieme X, e da una regola logica detta *regola di specializzazione universale*. L'inferenza consiste solo nell'applicazione di tale regola che essendo una tautologia [1] fa si che il risultato dell'inferenza deduttiva assuma pure valore logico vero. Questo è un esempio di inferenza deduttiva conclusiva dato che l'Output è sempre una conseguenza logica dell'Input e delle *BK*

Input	$a \in X$	a è un elemento di X
BK	$\forall x \in X, q(x)$	Tutti gli elementi di X hanno la proprietà q.
	$\forall x \in X, q(x) \implies (a \in X \implies q(a))$	Se tutti gli elementi di X hanno la proprietà q, allora ogni elemento di x, e quindi anche a, deve avere la proprietà q
Output	$q(a)$	a ha la proprietà q

Tabella 1.1: Deduzione

2. INDUZIONE tabella 1.2 Riferendosi all'equazione fondamentale (1.2) l'Input è la conseguenza C e l'Output è Q (l'ipotesi).Si può dimostrare che l'Input è conseguenza logica dell'Output (l'ipotesi) e delle *BK* quindi dato che l'equazione fondamentale 1.2 è rispettata l'inferenza è conclusiva (forte). Infatti nel processo inferenziale è *false-preserving* se l'Input fosse falso (a non ha la proprietà q) allora l'Output avrebbe dovuto essere pure falso. Da rimarcare è che l'Output dell'inferenza induttiva (sia che sia conclusiva che contingente) non ha un valore di verità sempre vero ma può essere vero o falso (anche se l'Input e le *BK* sono vere) da cui deriva il termine ipotesi per connotare l'Output.Essa si basa sull'assunzione che determinate regolarità osservate in un fenomeno continueranno a manifestarsi nella stessa forma anche in futuro e quindi generalizza ciò che vero per alcune istanze ad un insieme più grande. Invece nell'inferenza deduttiva conclusiva è garantito logicamente che l'Output assuma valore di verità vero se l'Input e le *BK* sono pure vere perchè ciò che vero in generale resta vero in un caso specifico contemplato dalla regola generale.

   Nell'esempio riportato le conoscenze di *BK* sono le stesse della deduzione. Tuttavia l'Output(l'ipotesi) è ottenuto tracciando all'indietro la *regola di specializzazione universale*. Quindi l'inferenza consiste nel supporre l'implicazione presente nella regola di specializzazione valida anche nel verso opposto, perciò

[1] E' un enunciato che ha sempre valore logico vero

si dice che l'induzione è una regola d'inferenza all'indietro a la deduzione una regola d'inferenza in avanti.

Input	$q(a)$	a ha la proprietà q
BK	$a \in X$	a è un elemento dell'insieme X.
	$\forall x \in X, q(x) \implies (a \in X \implies q(a))$	Se tutti gli elementi di X hanno la proprietà q, allora ogni elemento di x, e quindi anche a, deve avere la proprietà q
Output	$\forall x \in X, q(x)$	Tutti gli elementi di X hanno la proprietà q

Tabella 1.2: Induzione

3. ABDUZIONE tabella 1.3 In riferimento all'equazione (1.2) l'Output è Q e l'Input è C.Si può dimostrare che l'Input è conseguenza logica dell'Output (l'ipotesi) e delle BK quindi dato che l'equazione fondamentale 1.2 è rispettata l'inferenza è conclusiva (forte).Come nel caso dell'induzione l'inferenza abduttiva conclusiva è *false-preserving*. Come nell'induzione l'Output è solo un ipotesi e quindi il suo valore di verità è incerto e c'è solo una probabilità che sia vero. L'abduzione, come l'induzione, non contiene in sé la sua validità logica e deve essere confermata per via empirica. Nell'abduzione come nell'induzione la regola implicativa di specializzazione universale viene tracciata all'indietro. Tuttavia c'è un'importante differenza infatti nell'induzione la regola implicativa nelle BK costituisce una tautologia mentre nel caso dell'abduzione rappresenta una verità solo nel dominio di conoscenza e non una verità universale.

   Nell'esempio specifico si assume che un elemento a gode della proprietà q. Le BK consistono in un unico enunciato, che esprime il fatto che tutti gli elementi di un certo insieme X hanno la proprietà q. L'inferenza abduttiva produce in Output un enunciato che asserisce l'appartenenza di a ad X. Intuitivamente tutti gli elementi che appartengono ad un insieme X hanno una proprietà; dall'input si ha che un elemento a ha quella proprietà; siccome tutti gli elementi appartenenti all'insieme X possiedono quella stessa proprietà si suppone che a appartiene all'insieme X

Input	$q(a)$	a ha la proprietà q
BK	$\forall x, x \in X \implies q(x)$	Se x è un elemento di X allora x ha la proprietà q.
Output	$a \in X$	a è un elemento di X

Tabella 1.3: Abduzione

1.2.1 Metodologia di ricerca induttiva

Si introduce brevemente,seguendo ancora [9] il *learning da esempi* induttivo di un concetto come un problema di ricerca in uno spazio. L'algoritmo inferenziale induttivo riceve in ingresso degli esempi (ed eventualmente anche controesempi) di membri del concetto target(specifiche istanze) sottoinsieme dello **spazio delle istanze** che costituisce l'insieme di tutte le possibili istanze osservabili. Lo **spazio dei concetti** costituisce invece l'insieme di tutti i possibili concetti (tutte le possibili soluzioni). I concetti quasi sempre necessitano di una descrizione, un linguaggio che formalmente consente di definire operativamente un concetto e per questo si parla in maniera interscambiabile di **spazio delle descrizioni**. Un concetto è consistente se accetta alcuni esempi positivi e rifiuta tutti quelli negativi; è completo invece quando accetta tutti gli esempi positivi. La macchina inferenziale induttiva ha lo scopo di selezionare un'ipotesi dallo **spazio delle ipotesi** che sia consistente e completa con gli esempi visti. Lo spazio delle ipotesi è un sottoinsieme dello spazio dei concetti. All'aumentare degli esempi visti lo spazio delle ipotesi si riduce, tuttavia le ipotesi valide possono comunque essere numerose e spesso è necessario utilizzare dei criteri di preferenza per scegliere l'ipotesi corrente. E' necessario anche definire dei criteri di terminazione per sancire la ricerca conclusa. In sintesi il *concept learning induttivo* può essere descritto come una ricerca euristica nello spazio delle descrizioni della migliore ipotesi tra tutte quelle consistenti e complete rispetto agli esempi forniti.

Capitolo 2

Algoritmi di apprendimento

2.1 Active Learning

L' *Active Learning* è un caso speciale di *semi-supervisionato machine learning*[1] in cui un algoritmo di *learning* può interagire con l'utente o qualche sorgente d'informazione per ottenere informazioni significative come ad esempio l'etichetta di un'istanza. Nel contesto dell' *IIR* l' *active learning* non esplora il reticolo costruito a partire dal PTA o dall' APTA come fanno gli algoritmi presentati in [METTI RIFERIMENTO RPNI EDSM] ma si basa su una stretta interazione tra il *learner* e il ***teacher*** talvolta detto ***Oracolo*** o ***informant***. Non vi è più l'esplorazione dello spazio di ricerca creato a partire dal nodo iniziale (PTA o APTA) costruito a sua volta dagli esempi iniziali ma è il *learner* che sceglie gli esempi e inoltre il *teacher* può selezionare attentamente i controesempi significativi: proprio per queste ragioni spesso il numero di esempi per apprendere un concetto e in generale il tempo di esecuzione del processo di apprendimento è minore nell' *active learning* rispetto agli approcci basati sull'esplorazione del reticolo. L'*active learning* nasce per ragioni teoriche come ad esempio per dimostrare che non è possibile apprendere in maniera efficiente alcune classi di linguaggi con algoritmi given-data in cui gli esempi sono imposti: è sufficiente dimostrare che il numero di query non può essere polinomiale nell' *active learning* (nel cui contesto si può scegliere quali esempi vedere). Ma è anche applicabile in numerosi contesti pratici come la Robotica in cui un agente può costruire una mappa usando l'interazione tra i sensori e l'ambiente come *Oracolo* o nella modellazione dell'acquisizione dei linguaggi naturali dove attribuire la figura del *teacher* al genitore risulta naturale. Qui si esaminerà l' *active learning* nell' *IIR*. L* è senz altro il più noto algoritmo in letteratura di *active learning* applicato ai linguaggi regolari. Il più recente e performante *ObP* sarà introdotto nel capitolo 3

[1] Nel semi-supervisionato learning una piccola quantità di dati è etichettata e la restante, la maggioranza, è senza etichetta.

2.1.1 Active learning nell' Inferenza Induttiva Regolare

Il paradigma dell' *active learning* si basa sull'esistenza di un *Oracolo* che conosce **linguaggio target ($\mathcal{L}$)** e può rispondere solo a certi tipi di interrogativi del *learner*. L' *Oracolo* può trovarsi in una situazione in cui più risposte valide sono possibili e in questo caso si deve assumere che non viene rispettata nessuna distribuzione di probabilità nelle risposte date ma che queste sono casuali pertanto nell'analisi dell'algoritmo si deve assumere il caso peggiore cioè un *Oracolo* avverso (nell' algoritmo adoperato nell' *IIR*, il table-filling descritto nella sottosezione 2.2.3, l'*Oracolo* non garantisce di ritornare la ***witness*** cioè il controesempio più breve). I principali tipi di interrogativi possibili a cui si può sottoporre un *Oracolo* sono:

- **Membership Query (*MQ*)** Una membership query è effettuata proponendo una stringa all'*Oracolo*, che risponde YES se la stringa appartiene a $\mathcal{L}$ e NO se la stringa non appartiene:

$$MQ : \Sigma^* \rightarrow \{\text{YES,NO}\}$$

- **Equivalence Query (*EQ*)** (forte) Un'equivalence query (forte) è effettuata proponendo un DFA ipotesi all'*Oracolo* che risponde YES se il DFA ipotesi è equivalente al DFA target altrimenti ritorna una stringa(*witness*) appartenente alla differenza simmetrica tra $\mathcal{L}$ e $\mathcal{L}$(DFA):

$$EQ : DFA \rightarrow \{\text{YES}\} \cup \Sigma^*$$

- ***WEQ*** (debole) Un'equivalence query (debole) è effettuata proponendo un DFA ipotesi all'*Oracolo* che risponde YES se il DFA ipotesi è equivalente al DFA target altrimenti ritorna NO :

$$WEQ : DFA \rightarrow \{\text{YES,NO}\}$$

- ***SSQ*** Una subset query è effettuata proponendo un DFA all'*Oracolo* che risponde YES se $\mathcal{L}$(DFA) è un sottoinsieme di $\mathcal{L}$ altrimenti ritorna una stringa appartenente a $\mathcal{L}$(DFA) che non appartiene ad $\mathcal{L}$:

$$SSQ : DFA \rightarrow \{\text{YES}\} \cup \Sigma^*$$

I seguenti risultati e definizioni sono in [2].Si da la seguente definizione preliminare:

Definizione 2.1. Chiamiamo ρ un'esecuzione del *learner* A. Chiamiamo $\langle r_1, r_2, \ldots r_m \rangle$ la sequenza di risposta alle query $\langle q_1, q_2, \ldots q_m \rangle$ che l'*Oracolo* fa durante l'esecuzione ρ . Si dice che **A** è **polinomialmente limitato** se esiste un polinomio a due variabili p() che dato qualsiasi formalismo L descrivente $\mathcal{L}$ e in qualsiasi esecuzione ρ, e a qualsiasi *query point*(indica il momento in cui avviene una specifica query) k dell'esecuzione, denotando il tempo di esecuzione prima di quel punto con t_k, si ha:

- $k \leq p(\|L\|, \max\{|r_i| : i < k\})$

- $|q_k| \leq p(\|L\|, \max\{|r_i| : i < k\})$
- $t_k \in \mathcal{O}(p(\|L\|, \max\{|r_i| : i < k\}))$

Informalmente significa che in qualsiasi *query point* k di qualunque esecuzione, al momento precedente l'effettuazione della query q_k, si ha che il numero di query fatte, il tempo di esecuzione e la dimensione della prossima query (q_k) sono tutte limitate da un polinomio p dipendente dalla dimensione del target e dalla lunghezza del più lungo controesempio ritornato dall'*Oracolo* fino a quel punto
La seguente definizione stabilisce quando una classe di linguaggi è efficientemente identificabile *in the limit* da un algoritmo di *learning*.

Definizione 2.2. Una classe di linguaggi $\mathfrak{L}$ è **polinomialmente *identificabile in the limit con query*** fissati i tipi di query possibili se esiste un **polinomialmente limitato** *learner* **A** che dato il formalismo descrivente qualsiasi linguaggio L in $\mathfrak{L}$, identifica L in the limit, cioè ritorna L' equivalente ad L e termina.

Adesso ci si chiede se la classe dei linguaggi regolari è polinomialmente identificabile in the limit tramite qualche algoritmo di apprendimento secondo la definizione 2.2. E' importante sottolineare che la risposta a questa domanda dipende anche dalla classe cui appartiene l'*Oracolo* cioè dal tipo di interrogativi che è possibile rivolgergli. A tal proposito si hanno i seguenti risultati :

Teorema 2.1. *La classe dei linguaggi regolari non è polinomialmente identificabile in the limit da un numero polinomiale di MQ, WEQ e SSQ*

Quindi come conseguenza del teorema 2.1 la classe dei linguaggi regolari non è polinomialmente identificabile in the limit neanche sottoponendo all'*Oracolo* esclusivamente *MQ* .
Un'ulteriore risultato è il seguente:

Teorema 2.2. *La classe dei linguaggi regolari: DFA non è polinomialmente identificabile in the limit da un numero polinomiale di EQ (forti)*

Si rimanda alla sezione 2.2 per le condizioni di polinomiale identificabilità in the limit dei linguaggi regolari

2.2 L*

L* è il più noto algoritmo di *active learning* nell'ambito dell' *IIR* e garantisce di emettere in output il DFA minimo (o uno ad esso isomorfo) accettante $\mathcal{L}$. Detto n il numero degli stati del DFA target minimo ed m la lunghezza del più lungo controesempio ritornato dall' *Oracolo* durante l'inferenza, il costo computazionale di L* sarà limitato da una funzione polinomiale di n ed m. In L* il *teacher* appartiene alla classe dei **Minimally Adeguate Teacher (*MAT*)** in grado di rispondere ad *EQ* e *MQ*. Questi risultati ,che consentono di dire che i linguaggi regolari sono polinomialmente identificabili in the limit (definizione 2.2), sono stati conseguiti da Dana Angluin [1] e succintamente riportati nel seguente teorema:

Teorema 2.3. *Dato un MAT presentante un linguaggio regolare sconosciuto U, il Learner L* termina restituendo in output un automa finito isomorfo al DFA minimo accettante il linguaggio target U. Inoltre, se n è il numero di stati del DFA minimo accettante U e m è un limite superiore della lunghezza di ogni controesempio ritornato dal Teacher, allora il costo totale di esecuzione di L* è limitato da un polinomio in n ed m*

L* viene presentato all'interno del *red–blue framework* [INSERISCI RIFERIMENTO] che in algoritmi come *EDSM* [INSERISCI RIFERIMENTO] consente di diminuire i *merges*. In L* l'adozione di questo *framework* malgrado non comporti un vantaggio computazionale consente un'esposizione più chiara.

2.2.1 Tabella di Osservazione

Una **tabella di osservazione** è una struttura dati che rappresenta il DFA ipotesi congetturato al passo corrente. Al suo interno sono codificati gli esiti delle *MQ* richieste al *teacher*.

Definizione (Tabella di Osservazione)**.** La *tabella di osservazione* è una tripla $\langle STA, EXP, OT\rangle$, dove:

- STA = RED $\cup$ BLUE. STA è un insieme finito di stringhe definite su Σ che rappresentano gli stati. STA è **prefix–closed**
  RED $\in \Sigma^*$ è un insieme finito di stati
  BLUE = $\{ua \notin \text{RED} : u \in \text{RED}\}$ è l'insieme dei successori degli stati RED che non sono RED. Rappresentano le transizioni.
- EXP $\in \Sigma^*$ è l'insieme degli esperimenti. E' **suffix–closed**
- OT : STA $\times$ EXP $\rightarrow$ {0,1,*} è una funziona così definita:

$$OT[u][e] = \begin{cases} 1 & \text{se } ue \in \mathcal{L} \\ 0 & \text{se } ue \notin \mathcal{L} \\ * & \text{altrimenti} \end{cases}$$

Dalla tabella di osservazione si costruisce una nuova ipotesi e la si sottopone al'*teacher*. Se l'ipotesi non è equivalente al *DFA target* il *teacher* torna un controesempio che sarà usato dal *learner* per *splittare* gli stati e modificare la tabella di osservazione per ottenere una nuova ipotesi cofacente al controesempio. Le *MQ* permettono di riempire i buchi generati dall'introduzione di nuovi prefissi dal controesempio. A partire dalla tabella di osservazione è possibile costruire un DFA ipotesi sole se questa gode di tre proprietà:

Completezza

La completezza garantisce che non ci siano comportamenti parzialmente (o totalmente) sconosciuti per prefissi presenti all'interno della tabella.

Definizione (Tabella completa)**.** Una tabella è completa se non ha *buchi*. Un *buco* in una tabella di osservazione è una coppia (u,e) tale che $OT[u][e] = *$.

L'eventuale incompletezza può essere eliminata mediante *MQ* al *teacher*.

Chiusura

La chiusura (algoritmo 4) assicura che ogni possibile stato raggiunto con una transizione sia presente tra gli stati finali dell'automa. Dato un elemento $s \in$ STA e gli n esperimenti $e \in$ EXP si indica con row(s) la riga in OT indicizzata da s cioè $row(s) = OT[s][e_1] \cdot OT[s][e_2] \cdot \ldots OT[s][e_n]$. Gli stati dell'automa sono un sottoinsieme degli stati RED, quando una transizione da uno stato RED porta ad uno stato BLUE si deve trovare uno stato RED equivalente a quello BLUE (vedasi algoritmo 1) (almeno secondo i suffissi trovati fino a quel momento) in modo che la transizione arrivi in questo stato (che è presente nell'automa ipotesi perchè è uno stato RED a differenza di quello BLUE)

Definizione (Tabella chiusa)**.** Una tabella è **chiusa** se $\forall u \in$ BLUE, $\exists s$ $\in$ RED : row(u) = row(s)

Se la tabella di osservazione non fosse chiusa è possibile renderla tale mediante una (o più) **promozione**, cioè l'inserimento di **u** nei RED e $u \cdot \Sigma$ nei BLUE

Consistenza

La consistenza (algoritmo 5) impedisce situazioni di indeterminismo nel DFA, nella fattispecie che da uno stato dell'ipotesi per uno stesso simbolo dell'alfabeto si giunga in stati di arrivo diversi. Questa situazione è resa possibile dal fatto che l'algoritmo non impedisce di avere due stati RED s_1 ed s_2 tali che $\text{row}(s_1) = \text{row}(s_2)$.

Definizione (Tabella consistente)**.** Una tabella di osservazione è **consistente** se $\forall s_1, s_2 \in \text{RED} : \text{row}(s_1) = \text{row}(s_2) \implies \forall a \in \Sigma, \text{row}(s_1 a) = \text{row}(s_2 a)$

La definizione sopra significa che affinchè vi sia consistenza ogni coppia di stati equivalenti RED cioè di stati in RED con righe uguali deve restare equivalente in STA aggiungendo qualsiasi simbolo dell'alfabeto. Se la tabella di osservazione non fosse consistente è possibile renderla tale ampliando l'insieme EXP con la stringa composta dal suffisso e dall'esperimento che hanno generato l'inconsistenza. Ciò assicura che i due stati s_1 ed s_2 che prima erano equivalenti (e che quindi rappresentavano un unico stato nell'ipotesi) adesso non lo sono più perchè $\text{row}(s_1) \neq \text{row}(s_2)$ e quindi sarà aggiunto un nuovo stato all'insieme RED cioè un nuovo stato all'ipotesi.

2.2.2 L'algoritmo

Funzionamento

La *ratio* che ispira L* é il teorema *Myhill-Nerode* (sezione A.1). Nella tabella di osservazione le righe RED,in realtà un sottoinsieme delle stringhe RED, corrispondono

Algoritmo 1 LSTAR-BUILDAUTOMATON

Input: a closed and complete observation table ⟨STA,EXP,OT⟩
Output: DFA $\langle \Sigma, Q, q_\epsilon, F_A, F_R, \delta \rangle$

$Q \leftarrow \{q_u : u \in \text{RED} \wedge \forall v < u \, \text{row}(v) \neq \text{row}(u)\}$
▷ le stringhe più corte sono minori e per stringhe della stessa lunghezza si intendono minori quelle che lessicograficamente vengono prima
$F_{\mathbb{A}} \leftarrow \{q_u \in Q : \text{OT}[u][\epsilon] = 1\}$
$F_{\mathbb{R}} \leftarrow \{q_u \in Q : \text{OT}[u][\epsilon] = 0\}$
for $q_u \in Q$ **do**
    for $a \in \Sigma$ **do** $\delta(q_u, a) \leftarrow q_w \in Q : \text{row}(ua) = \text{row}(w)$
end for
return $\langle \Sigma, Q, q_\epsilon, F_A, F_R, \delta \rangle$

Algoritmo 2 LSTAR

Input: –
Output: DFA $\mathcal{A}$

LSTAR-INITIALISE
repeat
    while ⟨STA,EXP,OT⟩ *is not closed or not consistent* **do**
        if ⟨STA,EXP,OT⟩ *is not closed* **then**
            ⟨STA,EXP,OT⟩ ← LSTAR-CLOSE(⟨STA,EXP,OT⟩)
        if ⟨STA,EXP,OT⟩ *is not consistent* **then**
            ⟨STA,EXP,OT⟩ ← LSTAR-CONSISTENT(⟨STA,EXP,OT⟩)
    end while
    Answer ← EQ(⟨STA,EXP,OT⟩)
    if Answer ≠ YES **then**
        ⟨STA,EXP,OT⟩ ← LSTAR-USEEQ(⟨STA,EXP,OT⟩ , Answer)
until Answer = YES
return LSTAR-BUILDAUTOMATON(⟨STA,EXP,OT⟩)

Algoritmo 3 LSTAR-INITIALISE

Input: –
Output: ⟨STA,EXP,OT⟩

RED ← $\{q_\epsilon\}$
BLUE ← $\{q_a : a \in \Sigma\}$
EXP ← $\{\epsilon\}$
OT$[\epsilon][\epsilon]$ ← MQ(ϵ)
for $a \in \Sigma$ **do** OT$[a][\epsilon]$ ← MQ(a)
return ⟨STA,EXP,OT⟩

Algoritmo 4 LSTAR-CLOSE

Input: ⟨STA,EXP,OT⟩
Output: ⟨STA,EXP,OT⟩ updated
1: **for** $s \in$ BLUE *such that* $\forall u \in$ RED $\text{row}(s) \neq \text{row}(u)$ **do**
▷ Non per $\forall s$ ma per uno solo quindi la tabella in output può ancora essere non chiusa
2: RED $\leftarrow$ RED $\cup \{s\}$
3: BLUE $\leftarrow$ BLUE $\setminus \{s\}$
4: **for** $a \in \Sigma$ **do** BLUE $\leftarrow$ BLUE $\cup \{s \cdot a\}$
5: **for** $u, e \in \Sigma^*$ *such that* OT$[u][e]$ *is a hole* **do** OT$[u][e] \leftarrow$ MQ(ue)
6: **end for**
7: **return** ⟨STA,EXP,OT⟩

Algoritmo 5 LSTAR-CONSISTENT

Input: ⟨STA,EXP,OT⟩
Output: ⟨STA,EXP,OT⟩ updated
1: find $s_1, s_2 \in$ RED, $a \in \Sigma$ and $e \in$ EXP *such that* $\text{row}(s_1) = \text{row}(s_2)$ and
2: OT$[s_1 \cdot a][e] \neq$ OT$[s_2 \cdot a][e]$
▷ se $s_1 a$ ed $s_2 a$ differiscono per più di un esperimento basta considerarne uno
3: EXP $\leftarrow$ EXP $\cup \{a \cdot e\}$
4: **for** $u, e \in \Sigma^*$ *such that* OT$[u][e]$ *is a hole* **do** OT$[u][e] \leftarrow$ MQ(ue)
5: **return** ⟨STA,EXP,OT⟩

Algoritmo 6 LSTAR-USEEQ

Input: ⟨STA,EXP,OT⟩
Output: ⟨STA,EXP,OT⟩ updated
1: **for** $p \in$ PREF(Answer) **do** ▷ Anche Answer fa parte dei prefissi
2: RED $\leftarrow$ RED $\cup \{p\}$ ▷ Se un pref. è già in OT renderlo RED se non lo è
3: **for** $a \in \Sigma : pa \notin$ PREF(Answer) **do** BLUE $\leftarrow$ BLUE $\cup \{pa\}$
4: **end for**
5: **for** $u, e \in \Sigma^*$ *such that* OT$[u][e]$ *is a hole* **do** OT$[u][e] \leftarrow$ MQ(ue)
6: **return** ⟨STA,EXP,OT⟩

agli stati del DFA ipotesi e le colonne, l'insieme EXP, corrispondono alle stringhe rappresentanti i suffissi che distinguono coppie di stati distinti dell'ipotesi. I singoli stati sono etichettati dalle stringhe che portano dallo stato iniziale allo stato stesso. Lo stato iniziale è etichettato dalla stringa ϵ. Per ogni stato l'etichetta della colonna (un esperimento) indica lo stato che potrebbe essere raggiunto dallo stato dopo la lettura della stringa corrispondente all'etichetta dell'esperimento. Due stati sono considerati equivalenti se hanno le righe uguali nella tabella.
L'algoritmo inizia costruendo una tabella di osservazione corrispondente all'automa universale (algoritmo 3) Una volta resa la tabella completa,chiusa e consistente viene estratta l'ipotesi corrispondente e viene effettuta un'*EQ* dell'ipotesi al *teacher*. In caso di risposta affermativa cioè di equivalenza il processo termina in quanto il *learner* ha identificato un automa uguale o equivalente al *target* (L* inferisce il DFA minimo, invece il *target* potrebbe non essere un DFA minimo). In caso contrario sarà ritornato un controesempio che sarà usato per modificare la tabella di osservazione e quindi formulare una nuova ipotesi. L'algoritmo 2 chiarifica e approfondisce i passaggi summenzionati.

Correttezza

Per verificare che L* è corretto è sufficiente dimostare che termina dato che la terminazione con un *MAT* assicura l'equivalenza dell'ipotesi col target. A tal fine si riporta preliminarmente il seguente teorema ([1]):

Teorema 2.4. *Se una tabella di osservazione è completa,chiusa e consistente, allora l'ipotesi H (quella inferita dall'algoritmo 1) è consistente con la funzione OT. Qualsiasi altra ipotesi consistente con OT ma non equivalente ad H deve avere più stati.*

Nel teorema 2.4 con ipotesi consistente con la funzione OT si intende che $\forall u \in$ STA e $\forall e \in \text{EXP}, ue \in \mathcal{L} \iff \text{OT}[u][e] = 1$
Il risultato del teorema 2.4 è che qualunque DFA consistente con OT o è isomorfico all'ipotesi inferita da L* o contiene almeno uno o più stati. Quindi ogni ipotesi H fatta da L* è sempre il minimo DFA consistente con OT.
Un altro risultato che ci torna utile è:

Lemma 1. *Detto n il numero di differenti valori di $row(s)$ per $\forall\ s\ \in RED$ in una tabella di osservazione. Qualsiasi H consistente con OT deve avere almeno n stati*

Si indica con n il numero di stati del DFA minimo di $\mathcal{L}$. E' facile dimostrare che il numero di valori distinti di $row(s)$per$s$in RED è incrementato monotonicamente fino ad un massimo di n durante l'esecuzione di L*. Infatti sia la chiusura che la consistenza introducono un nuovo stato RED. Se la tabella fosse già chiusa e consistente il controesempio t tornato dal *teacher* comunque garantisce che un nuovo stato RED venga aggiunto alla tabella di osservazione: se T_u è il DFA *target* minimo (ovviamente consistente con OT) e il controesempio t ci permette di dedurre che H e T_u non sono equivalenti dal teorema 2.4 sappiamo che H ha al massimo $n-1$ stati.

Inoltre L* classificherà il controesempio t allo stesso modo di T_u e quindi il nuovo DFA ipotesi che otterrà *H'* sarà non equivalente con *H* (per via di t) e inoltre sarà consistente con OT, quindi dal teorema 2.4 si deduce che *H'* deve avere almeno n stati (almeno 1 stato in più di *H*).
Questo dimostra che ad ogni passo del ciclo più esterno di L* almeno uno stato RED distinto deve essere aggiunto sempre alla tabella di osservazione. Quando ci saranno n stati RED distinti L* troverà il DFA *target* minimo consistente con OT infatti il DFA *target* minimo è sempre consistente con OT[2] e l'ipotesi creata da L* è sempre il DFA minimo per il teorema 2.4 e dal lemma 1 si ha che l'ipotesi creata da L* deve avere almeno n stati. Essendo il DFA ipotesi un DFA ipotesi minimo e con n stati e consistente con T non può essere che uguale o isomorfo col DFA *target minimo*. Quindi L* dovrà costruire nel caso peggiore $n-1$ ipotesi errate prima di trovare l'ipotesi corretta e quindi il numero di *EQ* è al massimo n (perchè l'ipotesi corretta comunque va sottoposta al *teacher*)

Complessità computazionale

Come detto in precedenza la complessità computazionale di L* è limitata da un polinomio dipendente dal numero di stati del DFA minimo identificante $\mathcal{L}$ e dalla lunghezza del controesempio più lungo ritornato dal *teacher*.In [1] si trova una dimostrazione dettagliata di quanto detto sopra. In questa sede si analizzano dei parametri oggettivi nella valutazione del costo computazionale cioè il numero di *MQ* e di *EQ*. In quest'analisi si terrà conto anche di k un ulteriore parametro che rappresenta la dimensione dell'alfabeto. Il numero di *EQ* sarà limitato da n (sottosezione correttezza 2.2.2) come dimostrato in precedenza. Il numero di *MQ* è invece limitato dalla dimensione della tabella di osservazione. Il numero di elementi in EXP non può eccedere n , in quanto l'insieme EXP viene incrementato di un elemento quando la tabella di osservazione è inconsistente (algoritmo 5) e l'inconsistenza può presentarsi al più $n-1$ volte perchè ogni volta viene aggiunto un nuovo nodo RED distinto (con n nodi RED distinti L* termina) (la dimensione di EXP è al più n e non $n-1$ perchè EXP inizialmente contiene λ).EXP rappresenta il numero di colonne della tabella di osservazione, adesso si calcola il numero di righe della stessa. Il numero di stati RED non può eccedere $n+m(n-1)$ perchè gli stati RED sono aggiunti quando si scopre che la tabella non è chiusa e quando il *teacher* torna un controesempio. La non chiusura può accadere al più $n-1$ volte ed ogni volta aggiunge uno stato RED, e ci possono essere massimo $n-1$ controesempi ognuno dei quali può causare l'aggiunta di al più m stati RED (numero dei prefissi se il controesempio è lungo m). Il numero degli stati BLUE è al più $k(n+m(n-1))$ perchè i BLUE sono ottenuti concatenando tutti i simboli dell'alfabeto agli stati RED. Quindi la dimensione della

[2]Ma se sia il *target* che *H* sono sempre consistenti con OT come si fa a trovare un controesempio? La risposta è semplice: da OT viene creata un' ipotesi *H* consistente all'OT ma nell'ipotesi possono essere *parsate* anche altre stringhe non contemplate in OT da cui può derivare la non equivalenza con il *target*

tabella sarà righe $*$ colonne cioè (RED + BLUE) $*$ EXP quindi si ha:

$$(k+1)(n+m(n-1))\,n = \mathcal{O}(kmn^2)$$

che è il numero di *MQ* totali.

2.2.3 Il teacher

Il teacher di L* essendo un *MAT* è chiamato a rispondere a due tipi di query: *MQ* ed *EQ*. Si suppone che esso abbia a disposizione il DFA che identifica $\mathcal{L}$ quindi è immediato rispondere a una *MQ*. Il *teacher* deve anche vagliare l'equivalenza del target con l'ipotesi fornitagli dal *learner* e in caso di inequivalenza deve tornare un controesempio. A tal fine il *table-filling algorithm* [12] risulta essere un buon algoritmo (il più performante con complessità quasi lineare atto solo a testare l'equivalenza e tornare una witness quindi senza consentire anche la minimizzazione è [5])

Table-filling

Il *table-filliling* [12] è un algoritmo in grado di individuare ricorsivamente tutti gli stati tra loro distinti, alla fine dell'esecuzione le coppie di stati non marcati come tali saranno coppie di stati equivalenti. Per questo motivo l'algoritmo di table-filling è utilizzato anche nella minimizzazione di DFA dove gli stati trovati equivalenti saranno fusi in un unico stato.
Per stati distinti si intende stati per cui esiste almeno una stringa che partendo da quei due stati (e non dallo stato iniziale) giunge in una coppia di stati di arrivo composta da uno stato accettante e da uno stato rigettante.
Inizialmente si distinguono le coppie di stati che non sono equivalenti cioè gli stati distinti dalla stringa vuota cioè quelle coppie di stati formate da uno stato accettante e da uno rigettante. Al passo successivo si procede esaminando tutte le coppie di stati che momentaneamente l'algoritmo considera equivalenti (che non ha marcato come distinti nei passi precedenti): se per un simbolo dell'alfabeto s da quella coppia di stati di partenza si arriva a una coppia di stati distinti (già marcati dall'algoritmo nei passi precedenti) anche la coppia di stati di partenza va marcata come distinti perchè se gli stati di arrivo sono distinti vuol dire che esiste un suffisso w che li distingue quindi gli stati di partenza saranno distinti dalla stringa sw. Questa procedura va ripetuta ed ha termine quando al passo corrente l'algoritmo non ha trovato nessuna nuova coppia di stati distinti.Inoltre come attesta il seguente teorema:

Teorema. *Se due stati non sono marcati come distinti dall'algoritmo di table-filling, allora questi stati sono equivalenti. [6]*

Identificati gli stati equivalenti è possibile passare alla minimizzazione tramite il merge degli stati.
Per testare l'equivalenza di due DFA si manda in esecuzione il *table-filling* sul DFA costituito dall'unione dei due DFA di cui verificare l'equivalenza. Se al termine dell'esecuzione i due stati iniziali risultano equivalenti i due DFA di partenza saranno

equivalenti perchè non esiste nessuna stringa che distingue i due stati iniziali e quindi i due linguaggi dei due DFA sono identici. Per ottimizzare l'esecuzione è possibile interrompere l'esecuzione non appena viene individuato che i due stati iniziali sono distinti.
Per abilitare il *teacher* a ritornare, in caso di inequivalenza, una *witness* è necessario modificare leggermente il *table-filling*. Anzichè limitarsi nel marcare le coppie di stati non equivalenti è necessario anche memorizzare il simbolo dell'alfabeto che ha causato l'inequivalenza. Inoltre bisogna marcare con la stringa vuota o con un marcatore speciale le coppie di stati distinti in fase d'inizializzazione . Quando l'algoritmo termina è possibile creare il controesempio, partendo dalla coppia di stati iniziali, percorrendo la struttura dati usata durante il *table-filling* con l'ausilio delle funzioni di transizione dei due DFA e del marcatore memorizzato fino a quando non viene trovata una coppia di stati contrassegnata con la stringa vuota (cioè uno stato accettante e l'altro no). E' garantito che il controesempio venga sempre individuato ma non vi è la garanzia che ad essere scovato sia quello di lunghezza minima.

Una versione più efficiente

Il *table-filling* ha una complessità polinomiale rispetto ad n cioè alla somma del numero degli stati dei DFA di cui si vuole testare l'equivalenza. Si avranno $n * \frac{n-1}{2}$ coppie distinte di stati che verranno tutte considerate ad ogni visita della tabella (la struttura dati mantenuta dall'algoritmo) quindi $\mathcal{O}(n^2)$. Nel caso peggiore una sola coppia verrà scoperta essere distinta e le coppie sono tutte distinte quindi la tabella verrà esaminata al più $\mathcal{O}(n^2)$ volte. Moltiplicando le due complessità si ottiene $\mathcal{O}(n^4)$ che è il costo computazionale nel caso peggiore.
E' possibile migliorare la complessità computazionale a $\mathcal{O}(n^2)$ memorizzando per ogni coppia di stati (i, j) una lista di dipendenza costituita da tutte quelle coppie (x,y) che tramite un singolo simbolo dell'alfabeto k arrivano in (i, j) cioè $\hat{\delta}(x, k) = i$ e $\hat{\delta}(y, k) = j$. Si memorizzano in una coda tutte le coppie di stati inizialmente distinte. Si estrae una coppia dalla coda (che quindi è distinta) e tutte le coppie che da essa dipendono sono marcate come distinte nella tabella e sono aggiunte in fondo alla coda. L'algoritmo ripete questi passi finchè la coda è vuota.

Capitolo 3

Observation Pack

Esistono molte varianti dell'algoritmo L* originariamente presentato da Angluin. L'algoritmo *ObP* presentato da Falk Howar in [7] –presentato per le Mealy Machines ma comunque applicabile ai DFA – si basa su alcune di queste varianti. Tecnicamente *ObP* combina l'idea di usare un **Discrimination Tree (*DT*)** per i linguaggi regolari [8] con una versione localizzata della tabella di osservazione [13]. Inoltre viene utilizzata una tecnica più efficiente di gestione del controesempio rispetto ad L*. Quindi l'*ObP* è in stretta correlazione con L*, e molti dei concetti introdotti nel capitolo 2 rimangono validi. Anche *ObP*, come L* è un algoritmo di *active learning* nell'ambito dell'*IIR* che garantisce di emmettere in output il **Deterministic Finite Automata (*DFA*)** minimo (o uno ad esso isomorfo) accettante $\mathcal{L}$. Anche in questo caso detto n il numero degli stati del DFA target minimo ed m la lunghezza del più lungo controesempio ritornato dal *teacher* durante l'inferenza, il numero di *MQ* sarà limitato da una funzione polinomiale di n ed m e il numero di *EQ* linearmente da n. Nonostante L* sia il più noto algoritmo di *active learning* l'*ObP* ottiene prestazioni migliori dato che, come si evince dal lavoro sperimentale, permette di diminuire il numero delle *MQ*. In *ObP* il teacher appartiene ancora alla classe dei *MAT*, in grado di rispondere ad *EQ* e *MQ*.

3.1 Fondamenta teoriche

Si parla di *active learning* perchè il *learner* attivamente può interrogare il *teacher* sull'appartenenza o meno di alcune stringhe ad $\mathcal{L}$, in contrapposizione al *passive learning* in cui le stringhe sono date a priori come in *EDSM* ad esempio. Il *learner* da un certo punto di vista si trova ad affrontare un problema di *classificazione* cioè deve assegnare alcune stringhe ad un determinato stato di *H*. L'ipotesi *H* ottenuta induttivamente sarà consistente con l'etichettatura degli esempi sottoposti al *teacher* fino al momento della creazione di *H* ma produrrà una generallizzazione perchè in *H* è possibile effettuare il *parsing* di stringhe mai sottoposte al *teacher*. Detto T il *DFA* target,la *classificazione* delle stringhe e la costruzione dell'ipotesi si basa sui risultati del teorema Myhill-Nerode (sezione A.1) che consente di:

1. Trovare un insieme $Sp \subset \Sigma^*$ di **prefissi**, detti **short prefix o access sequence**, in cui ogni short prefix è una stringa reppresentativa per una classe di equivalenza sulla relazione d'equivalenza $\simeq_{\lambda_T}$

2. Trovare un insieme $V \subset \Sigma^*$ di **suffissi** che è sufficiente a realizzare la relazione di Nerode su Sp, cioè tale che $s \not\simeq_{\lambda^T} s'$ implica $\lambda^T(sv) \neq \lambda^T(s'v)$ per $s, s' \in Sp$ e qualche $v \in V$

Il teorema di Myhill-Nerode asserisce che un linguaggio $L(T) = \mathcal{L}$ è regolare se e solo se $\simeq_{\lambda^T}$ ha un numero finito di classi d'equivalenza. $\simeq_{\lambda^T}$ agisce su due stringhe x e y che sono in relazione se non esiste nessuna stringa z tale che xz e yz, esattamente una delle due appartiene ad $\mathcal{L}$: quindi nella classe d'equivalenza ci saranno stringhe non distinguibili da nessuna altra stringa e che quindi rappresentano un unico stato dell'automa T. L'*ObP* trova dei prefissi in cui ogni singolo prefisso è una stringa. contenuta in una specifica classe d'equivalenza (e vi è un prefisso rappresentativo, detto short prefix, per ogni classe d'equivalenza). Essendo T un *DFA*, L(T) è regolare, quindi il numero di classi d'equivalenza è finito e sarà certamente possibile trovare un insieme di short prefix Sp . L'esistenza dell'insieme di suffissi V in 2 è garantita dal fatto che se esistono almeno due classi di equivalenza deve esistere almeno un suffisso che distingue le due classi d'equivalenza (che altrimenti sarebbero un'unica classe d'equivalenza). Quindi $|V|$ è limitato dall'indice di $\simeq_{\lambda^T}$ cioè dal numero di stati del *DFA* target.

3.2 Costruzione dell'ipotesi

L'*ObP* mantiene due strutture dati rappresentative dell'ipotesi: il DT e un insieme di componenti.

3.2.1 Tabella di osservazione localizzata

A differenza di L* in cui vi è un'unica tabella d'osservazione rappresentativa dell'ipotesi, in *ObP* vi è una tabella di osservazione di dimensioni ridotte per ogni stato trovato fino a quel momento denotata come **componente**:

Definizione (Componente)**.** Un componente C è una quadrupla $\langle U, u_0, V, OT \rangle$ dove:

$U \subset \Sigma^*$ è un insieme finito di prefissi
$u_0 \in U$ è l'unico short prefix del componente
$V \subset \Sigma^*$ è un insieme di suffissi $v_1, \ldots, v_k$
$OT : U \times V \to \{0, 1, *\}$ è una funzione così definita:

$$OT[u][v] = \begin{cases} 1 & \text{se } uv \in \mathcal{L} \\ 0 & \text{se } uv \notin \mathcal{L} \\ * & \text{altrimenti} \end{cases}$$

Sia $u \in U$ e sia $|V| = n$, si indica con row(u) la riga in OT indicizzata da u cioè $row(u) = OT[u][v_1] \cdot OT[u][v_2] \cdot \ldots OT[u][v_n]$. Il componente è individuato dall'access sequence e quindi spesso lo si indica con C_{u_0}

Un componente approssima[1] la relazione di Nerode. Ogni componente rappresenta una classe di equivalenza: tutti i prefissi del componente fanno parte della stessa classe di equivalenza cioè sono equivalenti (secondo OT) in base ai suffissi di quel componente. I valori di output di OT sono ricavati tramite delle *MQ*. L'access sequence non è altro che un prefisso, possibilmente il più breve ma non necessariamente, rappresentativo del componente. Nell'ipotesi H ad ogni componente corrisponde uno stato. I prefissi di un componente sono tutte quelle stringhe che terminano nello stato dell'ipotesi rappresentato da quel componente. L'insieme di tutti i componenti è indicato con C_I, invece con Sp(C_I) o semplicemente Sp si indica l'insieme delle access sequences di tutte le componenti. **Sp è prefix-closed**.

Completezza

La completezza garantisce che non ci siano comportamenti parzialmente (o totalmente) sconosciuti per prefissi presenti all'interno di una componente.

Definizione (Componente completo)**.** Un componente è completo se non ha *buchi*. Un *buco* in un componente è una coppia (u,v) tale che $OT[u][v] = *$.

C_I è completo se tutti i componenti sono completi. L'eventuale incompletezza può essere eliminata mediante *MQ* al *teacher*.

Chiusura

La proprietà di chiusura di C_I garantisce che tutti i prefissi di un componente sono equivalenti secondo la relazione di Nerode approssimata.

Definizione (Componente chiuso)**.** Un componente $\langle U, u_0, V, OT\rangle$ è chiuso se row(u) = row(u_0) per $\forall u \in U$.

C_I è chiuso se tutti i componenti sono chiusi.

La proprietà di chiusura su C_I consente l'applicazione del teorema di Myhill-Nerode sulla relazione $\simeq_{OT}$. La relazione di Nerode è soltanto approssimata in quanto la funzione OT non è definita su Σ^* ma su un dominio più ristretto. Tuttavia, se le proprietà di chiusura e completezza su C_I sono mantenute il teorema Myhill-Nerode è ancora valido e come visto [INSERIRE RIFERIMENTO MIHILL NERODE] assicura la costruzione di un *DFA* la cui *funzione di output* è quella su cui si basa la

[1] Si parla di approssimazione perchè può accadere che alcuni prefissi che attualmente fanno parte dello stesso componente in futuro facciano parte di componenti diverse. Ciò è dovuto all'aggiunta di nuovi suffissi non ancora esaminati fino a quel momento. In ultima analisi ciò è dovuto al fatto che OT ha un dominio ristretto ad $U \times V$ anzichè Σ^*

relazione di Nerode utilizzata che nella fattispecie è OT. Un ipotesi H consistente con OT può essere costruita come nell'algoritmo 7. L'eventuale non chiusura di un componente può essere eliminata con uno *split* , che consiste nel dividere un componente in due componenti oppure uno stato in due stati (un componente corrisponde ad uno stato).

Algoritmo 7 OBP-BUILDAUTOMATON

Input: a closed and complete components set C_I
Output: DFA $\langle \Sigma, Q, q_\epsilon, F_A, F_R, \delta \rangle$
1: $Q \leftarrow \{q_u : \forall u \in \mathrm{Sp}(C_I)\}$
2: $q_\epsilon \leftarrow$ Component with short prefix ϵ
▷ Stato iniziale corrispondente al componente con short prefix ϵ
3: $F_{\mathbb{A}} \leftarrow \{q_u \in Q : \mathrm{OT}[u][\epsilon] = 1\}$
4: $F_{\mathbb{R}} \leftarrow \{q_u \in Q : \mathrm{OT}[u][\epsilon] = 0\}$
5: **for** $q_u \in Q$ **do**
6: **for** $a \in \Sigma$ **do** $\delta(q_u, a) \leftarrow q_w \in Q : \mathrm{row}(ua) = \mathrm{row}(w))$
▷ Da q_u per a si va nello stato q_w se ua è nel componente con short prefix w
7: **end for**
8: **return** $\langle \Sigma, Q, q_\epsilon, F_A, F_R, \delta \rangle$

3.2.2 Discrimination tree

Definizione (Discrimination tree)**.** Un *discrimination tree* è un albero binario con radice definito come $DT = \langle N, n_0, E, \tau, L \rangle$ dove:

N è un insieme finito di nodi
$n_0 \in N$ è la radice dell'albero
$E \subseteq N \times N \times \mathbb{B}$ è l'insieme finito di archi[2]
$\tau : N \to \Sigma^*$ assegna le etichette ai nodi
$L \subseteq N$ è l'insieme di foglie

Un DT è in stretta connessione con l'ipotesi come mostrato in figura 3.1. C'è una corrispondenza biunivoca tra foglie nel DT e stati in H. Come si può vedere in figura 3.1 alcune delle transizioni in H sono disegnate in *bold* perchè corrispondono ai prefissi che fungono da short prefix: nella fattispecie si ha $Sp = \{\epsilon, a, ab, b\}$. Le foglie sono etichettate con gli short prefix contenuti in Sp. I nodi interni invece sono etichettati con i suffissi $\in V$ di C_I che consentono di *discriminare* le foglie cioè stati diversi del *DFA*. Il DT assicura l'esistenza di un discriminatore per ogni coppia di foglie (e quindi di short prefix e di stati) diverse, il **least common ancestor (*LCA*)**. L' LCA di due foglie a e b è incontrato nei rispettivi percorsi dai

[2]Si specifica per ogni arco se il secondo nodo è figlio destro o sinistro del primo nodo (rispettivamente 1 e 0)

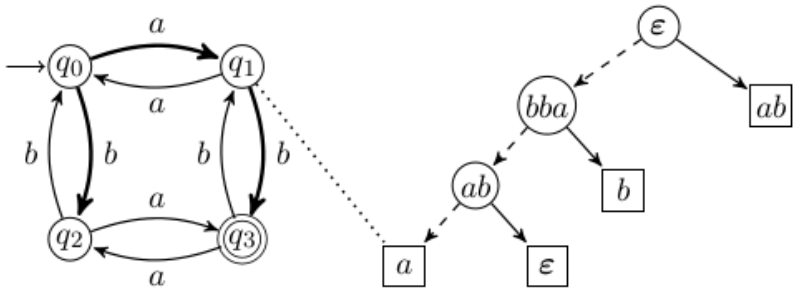


Figura 3.1: DFA ipotesi e possibile Discrimination Tree corrispondente

nodi verso la radice e rappresenta ,nei percorsi dalla radice verso a e b, il nodo in cui questi percorsi divergono come evidenziato in figura 3.2.
Un'operazione fondamentale è il *sift* di una stringa $x \in \Sigma^*$ in DT che consente di *affondare* x all'interno dell'albero. Sia q un nodo interno etichettato con v ed A il *DFA* target, il sifting di x procede,partendo dalla radice, nel sottoalbero sinistro o nel sottoalbero destro di q a seconda del valore di λ^A(xv) (se è 0 si va nel sottoalbero sinistro). Questa procedura è ripetuta finchè una foglia è raggiunta. Ad esempio in figura 3.1 il *sift* della stringa aba —,che ha access sequence b, e che in H termina nello stato q_2 — tramite la valutazione nel *DFA* target A di $\lambda^A(\epsilon \cdot aba)$ che da 0 (per questo si va a sinistra) e di $\lambda^A(bba \cdot aba)$ produce 1 (per questo si va a destra) arriva alla foglia con etichetta b (non casualmente, infatti b e aba sono transizioni che nell'ipotesi giungono nello stesso stato e quindi sono nella stessa classe di equivalenza).Il *sift* è descritto in dettaglio in [INSERIRE RIFERIMENTO PSEUDOCODICE SIFT] e in [INSERIRE RIFERIMENTO sottosezione DOVE SI PARLA DEL SIFT].

3.2.3 Observation Pack

Definizione (Observation Pack)**.** Un Observation Pack è una tupla $\langle C_I, DT \rangle$

Da cui deriva il nome dell'algoritmo. Un Observation Pack è chiuso e completo se C_I è chiuso e completo.

3.3 Gestione del controesempio

3.3.1 Classificazione

Un controesempio è una stringa w $\in \mathcal{L} \oplus L(H)$ che viene ritornato dal *teacher* quando H e il target differiscono. w va sfruttato dal *learner* in qualche modo per produrre una nuova ipotesi H. Esistono essenzialmente due modi per farlo:

1. *Metodi Suffix-based.* Aggiungono uno o più suffissi del controesempio all'insieme di suffissi dell'algoritmo provocando la non chiusura.

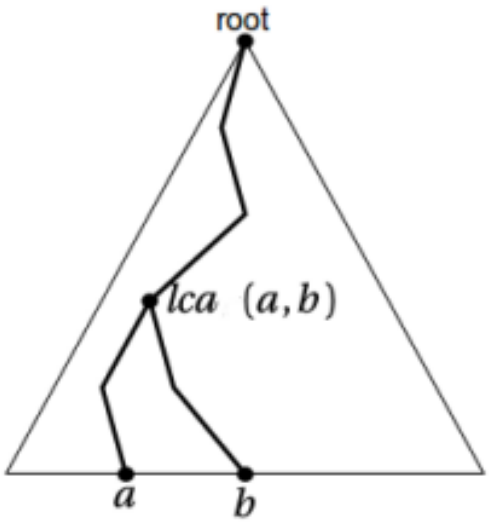


Figura 3.2: LCA di due nodi in un discrimination tree

2. *Metodi Prefix-based.* Aggiungono uno o più prefissi di un controesempio all'insieme di prefissi dell'algoritmo causando l'inconsistenza delle osservazioni, cioè una situazione di indeterminismo in H.

ObP usa la prima strategia.Inoltre un'ulteriore classificazione sui metodi di gestione del controesempio viene effettuata in base a (1) se tutti o solo qualche suffisso (prefisso nel caso dei metodi *prefix-based*) del controesempio sono impiegati e (2) questi suffissi (prefissi) sono applicati a tutti o solo a qualche prefisso (nell'accezione di prefisso rappresentante uno stato dell'ipotesi). Esistono diverse versioni di *ObP* (e in generale anche per i metodi *suffix-based*) in base a quest'ulteriore classificazione:

- **AllGlobally**.
  Si aggiungono tutti i suffissi del controesempio all'insieme di suffissi V di ogni componente in C_I.

- **OneGlobally**
  Si individua e si aggiunge un singolo suffisso del controesempio e lo si aggiunge all'insieme di suffissi V di ogni componente in C_I.

- **OneLocally**
  Si individua e si aggiunge un singolo suffisso del controesempio ad un ben preciso componente.

In tutti e tre i casi l'aggiunta del suffisso (o dei suffissi) porterà alla non chiusura di almeno un componente e al conseguente *split* che produrrà un nuovo stato nell'ipotesi. In AllGlobally e OneGlobally è possibile far diventare l'insieme V globale dato che è uguale per ogni componente. Nella strategia OneLocally l'insieme V di suffissi differirà da componente a componente. Su come sia possibile individuare un singolo

suffisso dal controesempio e l'esatto componente a cui bisogna aggiungere questo suffisso si rimanda [INSERIRE RIFERIMENTO TEOREMA RIVEST-SHAPIRE GESTIONE CONTROESEMPIO]. Una classificazione simile è possibile per i metodi *prefix-based*. La politica di gestione del controesempio è una differenza rilevante tra L* e *ObP*. L* gestisce il controesempio in maniera poco sofisticata: usa un metodo *prefix-based* in cui tutti i prefissi del controesempio sono aggiunti alla tabella di osservazione:quindi è una strategia AllGlobally dato che tutti i prefissi del controesempio sono usati in combinazione con un insieme di suffissi globale. Questa strategia causa l'inconsistenza di alcuni prefissi e per risolverla si aggiunge un suffisso che a sua volta causa una non chiusura e una promozione (un prefisso BLUE diventa RED). Lo svantaggio principale di questa strategia è che vengono aggiunti dei prefissi improduttivi che fanno aumentare il numero di *MQ*. Invece con una strategia che aggiunge un solo suffisso (o un solo prefisso) è necessario fare delle *MQ* per trovare il suffisso in questione dal controesempio ma il numero complessivo di *MQ* risulterà sempre minore complessivamente ad una strategia AllGlobally come quella usata da L* (qui si fa riferimento alla versione originale di L* in [1]. Esiste la variante OneLocally di L* in [8] che determina un unico prefisso del controesempio che genera inconsistenza, che viene risolta con l'aggiunta di un suffisso che causa uno *split* nella loro versione del discrimination tree).

3.3.2 Decomposizione del controesempio

Il seguente risultato fondamentale [13] garantisce che dato qualsiasi controesempio esiste sempre un suffisso che discrimina due prefissi nello stesso componente e ne causa di conseguenza lo *split*:

Teorema 3.1 (Decomposizione del controesempio)**.** *Sia H un'ipotesi, A il target, e $w \in \Sigma^+$ un controesempio ,cioè $\lambda^A(w) \neq \lambda^H(w)$. Allora esiste una decomposizione $\langle u, a, v\rangle \in \Sigma^* \times \Sigma \times \Sigma^*$ tale che $w = u{\cdot}a{\cdot}v$ e $\lambda^A(\lfloor u \rfloor_H a \cdot v) \neq \lambda^A(\lfloor ua \rfloor_H \cdot v)$.*

Innanzitutto bisogna precisare che il controesempio non può essere mai ϵ perchè $\lambda^A(\epsilon) = \lambda^H(\epsilon)$ che segue da $\epsilon \in Sp$ e dall'invariante (I2) [INSERIRE RIFERIMENTO INVARIANTE] e ciò giustifica l'assunzione $w \in \Sigma^+$ che si fa nel teorema 3.1. Si osservi che u e $\lfloor u \rfloor_H$ sono stringhe che terminano nello stesso stato di H, lo stesso dicasi allora per ua e $\lfloor u \rfloor_H \cdot a$. Inoltre ua e $\lfloor ua \rfloor_H$ terminano nello stesso stato di H da cui segue transitivamente che pure $\lfloor u \rfloor_H \cdot a$ e $\lfloor ua \rfloor_H$ terminano nello stesso stato dell'ipotesi H (e quindi appartengono anche allo stesso componente dato che c'è corrispondenza biunivoca tra stati dell'ipotesi e componenti). Quanto detto è schematizzato nella figura 3.3. Ma il teorema 3.1 afferma che $\lambda^A(\lfloor u \rfloor_H a{\cdot}v) \neq \lambda^A(\lfloor ua \rfloor_H{\cdot}v)$ il che significa che $\lfloor u \rfloor_H \cdot a$ e $\lfloor ua \rfloor_H$ — che sono due stringhe che in H terminano nello stesso stato — nel target A terminano in due stati diversi. Quindi si è trovato il suffisso v che discrimina i due prefissi $\lfloor u \rfloor_H \cdot a$ e $\lfloor ua \rfloor_H$ e in più si sa che i due prefissi sono anche nello stesso componente.

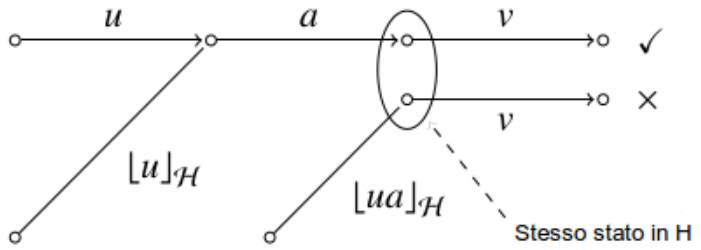


Figura 3.3: Sfruttare il controesempio

Decomposizione del controesempio riformulata

In [16] è descritto un framework che permette di riformulare il teorema 3.1 in mod da facilitarne la comprensione della correttezza ed anche l'implementazione. Alcune funzioni si trovano nell'appendice B e qui sono date per scontate.
Sia dato un controesempio $w \in \Sigma^+$ tale che $|w| = m$. Dato che $\pi_H(w, m) \in Sp$ (si evince dalla definizione di w considerando che il controesempio è lungo m), dall'invariante (I2) [INSERIRE RIFERIMENTO INVARIANTE INVARIANTE] si ha $\lambda^A(\pi_H(w, m)) = \lambda^H(w)$ quindi $\alpha(m) = 1$. Inoltre dato che π_H(w,0) = w e w è un controesempio si ha $\lambda^A(\pi_H(w, 0)) \neq \lambda^H(w)$, quindi $\alpha(0) = 0$. Allora si può ricondurre il teorema 3.1 nel trovare un undice i tale che $\alpha(i) \neq \alpha(i + 1)$. Siccome $\alpha(0) = 0$ e $\alpha(1) = 1$ sicuramente ci sarà un indice i in cui il valore di α passa da 0 ad 1 e ciò dimostra la correttezza e l'esistenza del suffisso. Il teorema 3.1 può essere riformulato così:

Teorema 3.2 (Decomposizione del controesempio riformulata)**.** *Una politica di analisi del controesempio suffix-based può essere riformulata come il problema di, data una funzione* $\alpha : [0, m + 1) \to \mathbb{B}$ *con* $\alpha(0) = 0$ *e* $\alpha(1) = 1$*, trovare un indice i,* $0 \leq i < m$*, soddisfacente* $\alpha(i) \neq \alpha(i + 1)$

Una volta trovato un indice i siffatto la decomposizione è u=$w_{[0,i)}$, a=w_i, v=$w_{[i+1,m)}$

3.3.3 Metodi di decomposizione del controesempio

Esistono diverse strategie per la ricerca del suffisso (o per i metodi *prefix-based*) all'interno del controesempio. I parametri da tenere in considerazione nella valutazione di tali strategie sono il numero di *MQ* effettuate e la dimensione del suffisso trovato. Infatti per valutare $\alpha(i)$ è necessaria una *MQ* quindi l'impiego di un'euristica ao di una politica che porta velocemente alla scoperta dell'indice i e quindi del suffisso permette di risparmiare il numero di *MQ* da effettuare al *teacher*. La dimensione dei suffissi[3] è anch esso un parametro molto importante in quanto il suffisso trovato verrà aggiunto all'insieme dei suffissi dei componenti o del componente rispettivamente in OneGlobally e OneLocally, e anche al *DT* (vedasi algoritmo []INSERIRE

[3]All'interno di un controesempio possono esistere più suffissi discriminatori, cioè molteplici indici i che permettono una decomposizione

RIFERIMENTO SPLIT]). Quando si navigherà il DT, per esempio durante un sift, verranno fatte delle MQ in cui parte della stringa è composta dall'etichetta di un nodo interno costituita da un suffisso; quindi il costo della MQ crescerà linearmente con la dimensione del suffisso.
La strategia più semplice è quella che effettua una ricerca lineare in ordine discendente cioè partendo da $i = m - 1$ che termina quando un valore i tale che $\alpha i = 0$ è incontrato. Questo metodo assicura di trovare il suffisso più breve ma nel caso peggiore richiede m-1 MQ. Anche se un costo lineare alla dimensione del controesempio sembra accettabile negli scenari reali non sempre lo è. In un contesto reale spesso capita di non avere la possibilità di effettuare un EQ che vanno sostituite con un certo numero di MQ nell'ambito del *PAC-learning*. In questo scenario il controesempio tornato è una stringa di lunghezza non-ottimale che può avere una lunghezza significativa. Esistono dei metodi che permettono di diminuire il numero di MQ anche se spesso bisogna rinunciare alla lunghezza minima del controesempio ed accontentarsi di una lunghezza ottima.

Algoritmo 8: Binary-Search

Input: A counterexample w with $|w| = m$
Output: Index i : $\alpha(i) \neq \alpha(i+1)$

```
low ← 0
high ← m
while high − low > 1 do
    mid ← ⌊(low+high)/2⌋
    if α(mid) = 0 then
        low ← mid
    else
        high ← mid
    end if
end while
return low
```

Algoritmo 9: Exponential-Search

Input: A counterexample w with $|w| = m$
Output: Index i : $\alpha(i) \neq \alpha(i+1)$

```
low←0, high←m, ofs←1, found←false
while high − ofs > 0 and ¬found do
    if α(high − ofs) = 0 then
        low ← high − ofs
        found ← true
    else
        high ← high − ofs
        ofs ← 2 · ofs
    end if
end while
return Binary-Search(α, low, high)
```

Binary search

Il metodo descritto nell'algoritmo [INSERIRE RIFERIMENTO PSEUDOCODICE RIVEST-SHAPIRE] è suggerito in [13]. Esso impiega la ricerca binaria per trovare una decomposizione valida. Il numero di MQ necessarie è $\lceil log_2(m) \rceil$ sempre, cioè non esiste un caso migliore ma il numero di MQ è fisso perchè è necessario testare il valore di $\alpha()$ per due indici i contigui e ciò avviene solo alla fine della ricerca. La dimensione del suffisso trovata può anche essere molto più grande di quella minima.

Exponential Search

Il metodo descritto qui e tutti quelli a seguire sono descritti in [16]. La ricerca binaria ha lo svantaggio evidente che può essere tornato un relativamente lungo controesempio: se il primo valore per *mid* testato è tale che $\alpha(mid) = 1$ il suffisso

risultante sarà di lunghezza almeno $\lceil m/2 \rceil$[4]. Con *exponential search* si testano $\alpha(m-2^0), \alpha(m-2^1), \alpha(m-2^2)$ eccetera, finchè non si trova un intervallo [l,h) per cui $\alpha(l) = 0 e \alpha(h) = 1$ ed allora si chiamerà il metodo che usa la ricerca binaria descritto in algoritmo [INSERIRE RIFERIMENTO RICERCA BINARIA] sui due indici l ed h. Nel caso peggiore(*eponential search* non riesce a restringere l'intervallo cioè l resta 0 ed h resta m) questo metodo richiede $2\lfloor log_2(m) \rfloor$ *MQ* , $\lfloor log_2(m) \rfloor$ per *exponential search* e altrettante per la ricerca binaria. In pratica l'algoritmo termina molto prima e nel caso migliore ($\alpha(m-1) = 0$) si effettua una singola *MQ*. Per come funziona quest algoritmo favorisce il ritrovamento di suffissi più brevi rispetto alla ricerca binaria.

Partition Search

Exponential search può terminare velocemente e individuare suffissi molto brevi ma nel caso che anche poche posizioni di $\alpha(m-2^i) = 1$ può essere svantaggiosa per via della rapida (anche per i piccolo),esponenziale, crescita dell'intervallo. Un approccio più bilanciato è quello di patizionare α in $\lceil log_2(m) \rceil$ intervalli, ognuno di lunghezza $\lfloor \frac{m}{log_2 m} \rfloor$. Poi i valori testati saranno $\alpha(m-s), \alpha(m-2s)$ eccetera, finchè un intervallo [l,h) soddisfacente $\alpha(l) = 0$ e $\alpha(h) = 1$ è trovato. Quest intervallo sarà poi sottoposto al metodo di ricerca binaria per trovare un indice i in esso. In *exponential search* a causa del passo esponenziale questo intervallo poteva risultare molto grande, in *partition search* è di dimensione s. Quindi il numero di *MQ* da effettuare nel caso peggiore con *partition search* è $log_2(s) = log_2(\lfloor \frac{m}{log_2 m} \rfloor)$ per via della ricerca binaria da eseguire sull'intervallo trovato più $\lfloor log_2(m) \rfloor$ (il numero di partizioni) *MQ* per trovare l'intervallo. Quindi sono necessarie $\mathcal{O}(log_2(m))$ *MQ*. *Partition Search* è presentato in dettaglio nell'algoritmo [INSERIRE RIFERIMENTO ALGORITMO]. Si osservi come il costo della ricerca binaria sia fisso (cioè è presente anche nel caso migliore) e dipendente da m. Quindi ci si aspetta che questo metodo funzioni meglio per controesempi non troppo grandi (m piccolo).

Eager Search

Eager Search è una variante del metodo di ricerca binaria. Quest ultimo richiede sempre — cioè non c'è un caso migliore o medio ma il numero di *MQ* è sempre lo stesso — $log_2(m)$ *MQ*. Come accennato in precedenza il motivo è che il solo valore $\alpha(i)$ da solo non è sufficiente ma è necessaro testare anche $\alpha(i+1)$[5]. La soluzione proposta in *Eager Search* è di testare ogni volta il valore di α sia per i che per i+1 e vagliare se differiscono o detto in maniera più succinta che il valore di β(vedasi[INSERIRE RIFERIMENTO IN APPENDICE DI BETA]) sia uguale ad 1. Nel caso peggiore siccome ogni valutazione di β richiede 2 *MQ* e il numero di

[4]La funzione $\alpha()$ non è necessariamente monotona quindi anche se $\alpha(mid) = 1$ ci può essere un indice $i > m$ per il quale $\alpha(i) = 0$ e quindi esserci un suffisso più breve

[5]perchè dal teorema [INSERIRE RIFERIMENTO DECOMPOSIZIONE CONTROESEMPIO RIFORMULATA] si deve trovare un indice i per cui $\alpha(i) \neq \alpha(i+1)$

valutazioni da fare è lo stesso della ricerca binaria (nel caso peggiore) sono necessarie $2log_2(m)$ *MQ*. Tuttavia nel caso migliore solo 2 *MQ* sono sufficienti e la ricerca può terminare molto prima di quella binaria. Questa strategia è affetta dallo stesso problema della ricerca binaria per quanto riguarda la dimensione dei suffissi tuttavia è possibile utilizzare *Eager Search* in luogo della ricerca binaria sia in *exponential search* che in *partition search*.

Algoritmo 10: Partition-Search

Input: A counterexample w with $|w| = m$
Output: Index i : $\alpha(i) \neq \alpha(i+1)$

```
step ← ⌊m / log2(m)⌋ low ← 0, high ← m
found ← false
while high−step>low and ¬found do
    if α(high − step) = 0 then
        low ← high − step
        found ← true
        break
    else
        high ← high − step
    end if
end while
return Binary-Search(α, low, high)
```

Algoritmo 11: Eager-Search

Input: A counterexample w with $|w| = m$
Output: Index i : $\beta(i) = 1$

```
low ← 0, high ← m − 1
while high > low do
    mid ← ⌊(low+high)/2⌋
    if β(mid) = 1 then
        return mid
    else if β(mid) = 0 then
        low ← mid + 1
    else
        high ← mid − 1
    end if
end while
return low
```

3.4 L'algoritmo

3.4.1 Funzionamento

ObP nella fase d'inizializzazione crea il componente corrispondente allo stato iniziale $C_\epsilon = \langle \Sigma \cup \{\epsilon\}, \epsilon, \epsilon, \emptyset \rangle$ ed il DT è costituito solo dalla radice quindi $DT = \langle \{n_\epsilon\}, n_\epsilon, \emptyset, \tau(n_\epsilon) = \epsilon, \{n_\epsilon\} \rangle$ come si può vedere nell'algoritmo [INSERIRE RIFERIMENTO ALGORITMO OBSERVATION PACK]. Dopodichè si chiama la funzione closePack() (vedi algoritmo [INSERIRE RIFERIMENTO ALGORITMO CLOSEPACK]) che ha lo scopo di rendere chiusi e completi C_I e modificare il DT in modo da rappresentare la stessa ipotesi rappresentata da C_I. Si completano i C_I e poi si ricerca un componente C_u (questo in generale e non solo nella fase di inizializzazione dove l'unico componente è C_ϵ) e un prefisso $u' \in C_u$ per cui row$(u) \neq$ row(u'). Si seleziona il suffisso v per cui $OT[u][v] \neq OT[u'][v]$ e si divide C_u chiamando la funzione *split* (algoritmo [INSERIRE RIFERIMENTO SPLIT]) che genererà un nuovo componente $C_{u'}$ e lo ritornerà a closePack(). In split() alcuni dei prefissi ,diciamo x,facenti parte di C_u ed esattamente quelli per cui $OT[x][v] \neq OT[u][v]$ vengono fatto migrare in $C_{u'}$ che nel frattempo è stato creato(cioè eliminati dal componente C_u ed immessi in $C_{u'}$). Si procede poi a modificare di conseguenza anche il DT: si individua il nodo foglia con etichetta u e lo si splitta nel senso che questo nodo foglia diventa un nodo interno con etichetta v (il suffisso, cioè il discriminatore) ed i suoi figli saranno due nodi foglia uno con etichetta u' e l'altro con etichetta u come si

può apprezzare in figura[6] 3.5b (si stabilisce qual è il figlio sinistro e quale il destro in base al risultato della *MQ* $\lambda^A(uv)$ nel target A, se questa fa zero u è il figlio sinistro del nuovo nodo con etichetta v altrimenti è il figlio destro). Si puntualizza che lo *split* non necessita di *MQ* aggiuntive e che l'insieme V del nuovo componente creato con lo *splitting* è lo stesso dell'insieme V del componente da cui ha origine. A questo punto split() ritorna il componente in questione a closePack() che memorizza il componente trovato $C_{u'}$ in W[7]. Finchè W non è vuoto si deve estrarre ed eliminare da W un componente , nella fattispecie $C_{u'}$, e concatenare l'access sequence del componente u' ad ogni simbolo di Σ e per ogni stringa ottenuta effettuare il *sift* (algortimo [INSERIRE RIFERIMENTO SIFT]). Questo serve per assicurare in ogni caso la possibilità di costruire una nuova ipotesi H in modo che per ogni nuovo componente aggiunto per ogni simbolo dell'alfabeto sia ben definito lo stato che viene raggiunto. Adesso si descrive in dettaglio come avviene il sift di una stringa x nel DT: si parte dalla radice e si effettua la query nel target A $\lambda^A(\tau(n_0) \cdot x)$ se da esito 0 si procede verso il sottoalbero sinistro altrimenti verso quello destro. Si continua in questa maniera finchè due situazioni possono presentarsi:

- Un nodo foglia $n_{x'}$ è incontrato. Significa che la stringa x in H porta nello stato rappresentato dallo foglia in cui si è arrivati: detto x' lo short prefix di questa foglia si ha che $x \in [x']_{\simeq_{OT}}$. Quindi si dovrà anche provvedere ad aggiungere x all'insieme U del componente $C_{x'}$. L'assegnazione della stringa come nuovo prefisso a un componente non è necessariamente definitiva perchè successivamente può accadere che la scoperta di un nuovo suffisso [INSERIRE RIFERIMENTO GESTIONE DEL CONTROESEMPIO] renda non chiuso un componente e per assicurare la chiusura si debba effettuare uno split che divide il componente in due parti e causando così possibilmente anche lo split di alcuni prefissi.

- Un nodo interno n_z con un solo figlio e con etichetta z è incontrato ma $\lambda^A(xz)$ suggerisce di andare verso il percorso dove n_z non ha figli. In questo caso deve avvenire la creazione di una nuova foglia n_x con etichetta x come figlio del nodo n_z. Conseguentemente deve essere aggiunto C_x a C_I. L'insieme di suffissi V di C_x sarà inizializzato in OneLocally con i suffissi trovati lungo il percorso nel DT per arrivare alla nuova foglia e con OneGlobally e AllGlobally con i suffissi globali dell'algoritmo.

Nel caso in cui sift() torna un nuovo componente a closePack() quest ultimo va aggiunto in W. Si ripete il procedimento finchè W è non diventa vuoto. Si ripete questo procedimento finchè C_I è completo (la non completezza è generata dal *sifting* e anche l'eventuale non chiusura tranne alla prima iterazione dove è causata dal suffisso trovato nella decomposizione del controesempio). Si procede poi alla generazione dell'ipotesi H come visto in [INSERIRE RIFERIMENTO ObP BUILAUTOMATON]

[6] Per coerenza con il discorso fatto si consideri $v = v$, $b = u$,$b' = u'$

[7] (solo quando closePack() viene chiamato per la prima volta in assoluto ,immediatamente dopo la fase di inizializzazione, a W deve essere aggiunto anche C_ϵ (per esempio una coda)

e si effettua un *EQ* cui il *teacher* può rispondere dato che si assume che appartenga alla classe dei *MAT*. Se il target A ed *H* sono equivalenti l'algoritmo termina garntendo che *H* sia il DFA minimo e l'equivalenza di A e H altrimenti verrà tornato un controesempio che verrà trovato per trovare un suffisso discriminante due prefissi appartenenti allo stesso componente (e con uno dei prefissi ,diciamo x, : $x \in Sp$). Se si usa OneLocally si aggiunge il suffisso all'insieme V di C_x, con OneGlobally si aggiunge il suffisso all'insieme V che sarà globale per tutte le componenti. Con la strategia AllGlobally non è necessario effettuare la decomposizione del controesempio in quanto si aggiungeranno tutti i suffissi del controesempio all'insieme di suffissi V globale e comune a tutte le componenti ma è chiaramente una strategia controproducente. L'aggiunta del suffisso v garantisce la non chiusura ad ogni " generazione " e quindi l'aggiunta di almeno un nuovo componente, e quindi stato, mediante lo *split*.

Algoritmo 12 OBP-SIFT

Input: a $DT = \langle N, n_0, E, \tau, L\rangle$, C_I , new prefix $u \in \Sigma^*$
Output: A new component or "OK"

1: $n = n_0$
2: **while** $n \notin L$ **do**
3: $v \leftarrow \tau(n)$
4: $o \leftarrow MQ(uv)$ ▷ $MQ(uv) = \lambda^A(uv)$
5: **if** $\exists(n, n', o) \in E$ **then** ▷ Se n ha un figlio nella direzione indicata da o
6: $n \leftarrow n'$
7: **else** ▷ Il nodo n non ha figli nella direzione indicata da o
8: Create new node n_u
9: $N \leftarrow N \cup \{n_u\}, \quad E \leftarrow E \cup \{(n, n_u, o)\}$
10: $\tau(n_u) = u$
11: Create component C_u
▷ Aggiungi a C_u lo short prefix u e i suffissi secondo la strategia usata
12: **return** C_u
13: **end while**
▷ Se si è qui significa che si è arrivati a una foglia
14: $u' = \tau(n)$ ▷ Si prende lo short prefix del nodo n
15: add u to $C_{u'}$
▷ aggiungere il prefisso al componente solo se il prefisso non è già presente
16: **return** "OK'

Inoltre a differenza di L* in *ObP* è possibile sfruttare più volte lo stesso controesempio. In L* tutti i prefissi del controesempio compreso il controesempio stesso vengono aggiunti alla tabella di osservazione e la nuova ipotesi generata è consistente con essi e quindi alla successiva " generazione " il controesempio precedente non è più riutilizzabile perchè difatti non è più in $\mathcal{L} \oplus$ L(H). In *ObP* invece viene aggiunto un solo suffisso del controesempio e quindi potrebbe accadere che alla successiva " generazione " sia ancora un controesempio utlizzabile per trovare un altro suffisso

che produra la non chiusura. Ciò può fare risparmiare molte *EQ* al costo di una *MQ* aggiuntiva per ogni " generazione " (perchè è necessario testare se il controesempio è ancora tale).

Algoritmo 13 OBP-SPLIT

Input: a $DT = \langle N, n_0, E, \tau, L\rangle$, C_I , a component $C_{u_0} = \langle U, u_0, V, OT\rangle$, a prefix $u \in U$, a suffix $v \in V : OT[u_0][v]) \neq OT[u][v]$
Output: A new component
1: Create component $C_u = \langle \emptyset, u, V, \emptyset\rangle$
2: **for** $u' \in U$ **do**
3: **if** $OT[u_0][V]) \neq OT[u'][v])$ **then**
4: Transfer u' from C_{u_0} to C_u ▷ Trasferire significa eliminare da C_{u_0}
5: **end for**
▷ Adesso a seguire le modifiche da apportare al discrimination tree
6: Let $n \in L$ where $\tau(n) = u_0$ ▷ Seleziona la foglia con etichetta u_0
7: $\tau(n) = v$ ▷ Modifica l'etichetta del nodo con quella del suffisso discriminante
8: Create new node n_u
9: Create new node n_{u_0}
10: $N \leftarrow N \cup \{n_u, n_{u_0}\}$
11: $\tau(n_u) = u$
12: $\tau(n_{u_0}) = u_0$
13: $E \leftarrow E \cup \{(n, n_u, OT[u][v])\} \cup \{(n, n_{u_0}, OT[u_0][v])\}$
14: **return** C_u

3.4.2 Correttezza

La correttezza dell' *ObP* scaturisce dal mantenimento di tre ***invarianti d'apprendimento*** :

(I1) $u \neq u' \in Sp$ corrispondono a stati differenti nel *DFA* target A cioè $A[u] \neq A[u']$.

(I2) Ogni stato q in H è accettante se e solo se "parsando" $\lfloor q \rfloor_H$ nel target A si termina pure in uno stato accettante. In simboli: $\forall q \in Q^H : q \in F^H_{\mathbb{A}} \Leftrightarrow f_{Sp}(q) \in F^A_{\mathbb{A}}$

(I3) Transizioni in H puntano allo stato corretto nel target, se quest ultimo è già stato scoperto dal *learner*: $\forall q \in Q^H, a \in \Sigma$ si ha $\delta^A(f_{Sp}(q), a) \in A[Sp] \implies f_{Sp}(\delta^H(q, a)) = \delta^A(f_{Sp}(q), a)$

E' evidente che la scoperta di nuovi short prefix in Sp mantenendo la condizione (I1) porterà alla scoperta di tutti gli stati nel target A (dato che sono finiti per il teorema di [Myill NERODE INSERIRE RIFERIMENTO]). Le condizioni (I2) ed

Algoritmo 14 OBP-CLOSEPACK

Input: a observation pack $\langle DT, C_I \rangle$
Output: ipotesi H

$W \leftarrow \emptyset$ (only in first call ever $W \leftarrow \{C_\epsilon\}$)
while C_I is unclosed or incomplete **do**
  Complete C_I
  Let $C_{u_0} = \langle U, u_0, V, OT \rangle$ with $u \in U, v \in V : OT[u_0][v] \neq OT[u][v]$
  $C_u = \text{OBP-SPLIT}(DT, C_I, C_{u_0}, u, v)$
  $W \leftarrow W \cup \{C_u\}$
  while $W \neq \emptyset$ **do**
    $C_u \leftarrow poll(W)$
  ▷ Estrai un componente secondo una qualche politica ad esempio FIFO
    for $a \in \Sigma$ **do**
      $C = \text{OBP-SIFT}(DT, C_I, ua)$
      if $C \neq$ "OK" **then**
        $W \leftarrow W \cup C$
    end for
  end while
end while
$H = \text{OBP-BUILDAUTOMATON}(C_I)$
return H

(I3) garantiscono che f_{Sp} è un isomorfismo .In L* vi è una violazione della condizione (I1) perchè può accadere che più di uno short prefix (più di uno stato RED) rappresentino lo stesso stato nel target. Il requisito di consistenza tuttavia garantisce che uno qualsiasi di questi può essere scelto come rappresentativo dello stato target.

Si dimostra che le tre *invarianti* sono possedute dall'algoritmo *ObP*:

(I1) Siano u e $u' \in Sp$ e $u \neq u'$, allora essendo in due componenti diversi esiste un suffisso v che distingue u e u' cioè tale che $\lambda^A(uv) = OT[u][v] \neq \lambda^A(u'v) = OT[u'][v]$. Quindi si ha anche $\lambda^A_{A[u]}(v) \neq \lambda^A_{A[u']}(v)$ e quindi $A[u] \neq A[u']$

(I2) Sia $u = \lfloor q \rfloor_H$ l'access sequence di uno stato $q \in Q^H$. Si ha allora che $q \in F^H_{\mathbb{A}} \iff$[8] $OT[u][\epsilon] = 1$. Dato che $OT[u][\epsilon] = \lambda^A(u \cdot \epsilon) = \lambda^A_{A[u]}(\epsilon)$ si può concludere che $q \in F^H_{\mathbb{A}} : u = \lfloor q \rfloor_H \Leftrightarrow A[u]$[9] $\in F^A_{\mathbb{A}}$

[8] Nel caso della strategia OneGlobally è lampante capire che in ogni componente ci sarà sempre il suffisso ϵ essendo questo già presente in fase d'inizializzazione. Utilizzando OneLocally accade lo stesso perchè i suffissi da aggiungere ad un nuovo stato (rappresentato da una access sequence e da un componente) sono quelli incontrati durante il *sifting* dell'access sequence nel discrimination tree e quindi si passa sempre dalla radice che contiene il suffisso ϵ

[9] è uguale a $f_{Sp}(q)$

Algoritmo 15 OBSERVATION PACK

Input: alfabeto Σ
Output: minimal DFA H : L(H)=$\mathcal{L}$

1: $C_\epsilon = \langle \Sigma \cup \{\epsilon\}, \epsilon, \epsilon, \emptyset \rangle$
2: $C_I = C_\epsilon$
3: $DT = \langle \{n_\epsilon\}, n_\epsilon, \emptyset, \tau(n_\epsilon) = \epsilon, \{n_\epsilon\} \rangle$
4: **loop**
5: H = OBP-CLOSEPACK(DT, C_I)
6: w = EQ(H)
7: **if** w = "OK" **then**
8: **return** H
9: **end if**
10: **if** AllGlobally **then**
11: Add all suffixes of w to V of all components in C_I
12: **continue**
13: **end if**
14: Split w with a decomposition method in uav : $\lambda^A(\underbrace{\lfloor u \rfloor_H a}_{p} v) \neq \lambda^A(\underbrace{\lfloor ua \rfloor_H}_{u'} v)$ tale che $p \in C_{u'}$
15: **if** OneGlobally **then**
16: Add v to V of all components $\in C_I$
17: **end if**
18: **if** OneLocally **then**
19: Add v to V of $C_{u'}$
20: **end if**
21: **end loop**

(I3) Sia dato uno stato $q \in Q^H$ tale che $u = \lfloor q \rfloor_H$, il successore di q per il simbolo $a \in \Sigma$ è determinato con il *sifting* di ua nel DT. Sia lo stato con access sequence u' il risultato di questa operazione di *sifting*, si definisce allora $\delta^H(q, a) = H[u']$. Se però nel target A questa transizione è errata cioè $\delta^A(A[u], a) \neq A[u']$, si sa per certo che il successore reale sul simbolo a $A[ua]$ non è ancora stato scoperto, quindi (I3) è preservata: si noti che *siftando* ua nell'albero, arrivando ad u' si escludono definitivamente gli altri stati scoperti fino a quel punto $A[Sp \backslash \{u'\}]$ come possibili successori sul simbolo a di $A[u]$.

Ciò dimostra la correttezza dell' *ObP* ma tramite quanto detto sopra si dimostra solo che *H* ed A sono isomorfi senza assumere la canonicità per *H*. Infatti in generale A non è un *DFA* minimo e secondo quanto detto finora non si può dedurre che *H* sia minimo ma solo isomorfo ad A. Inoltre a causa del maccanismo di gestione del controesempio non vi è la garanzia che l'insieme di suffissi sia *suffix-closed* dato che viene aggiunto solo un suffisso del controesempio. Quanto detto causa anche la generazione di ipotesi intermedie non minime. Tuttavia applicando l'euristica di utilizzare più volte lo stesso controesempio finchè da esso non è più possibile estrarre un suffisso discriminante è garantito che l'ipotesi finale è un *DFA* canonico [INSERIRE RIFERIMENTO INTRODUCTION TO ACTIVE AUTOMATA LEARNING FROM A PRACTICE PROSPECTIVE]. Esiste un'ottimizzazione che tramite il concetto di *semantic suffix closdness* [INSERIRE RIFERIMENTO MESSO PRIMA] assicura che anche le ipotesi intermedie prodotte siano canoniche ed il meccanismo che permette di scoprire dei nuovi suffissi da quelli esistenti in modo da ottenere la *semantic suffix closdness* permette di risparmiare anche delle *EQ* e *MQ* perchè permette di scoprire nuovi suffissi discriminanti e nuovi stati senza costi aggiuntivi in termini di *EQ* ed *MQ* (vi è però il tempo di esecuzione aggiuntivo dell'algoritmo per garantire la *semantic suffix closdness*).

3.4.3 Complessità computazionale

Il tempo di esecuzione degli algoritmi di *active learning* è quasi sempre un polinomio di grado piccolo dipendente dalla dimensione dell'alfabeto k, il numero di stati del *DFA* canonico equivalente ad $\mathcal{L}$ n , e la lunghezza del controesempio m con cui si indica la lunghezza del più lungo controesempio tornato dal *teacher*. Questo tempo di esecuzione è quasi sempre dominato dal tempo speso nell'effettuare *MQ* ed *EQ* [10] e quindi contare quest ultime può essere sufficiente per analizzare le prestazioni dell'algoritmo. In quest'ottica in [INSERIRE RIFERIMENTO HOWAR] si trova un'analisi dell' *ObP* nelle sue varie forme per le Mealy machines confrontando i risultati con L*. Questi risultati vengono qui applicati ai *DFA* e riportati nella tabella [INSERIRE RIFERIMENTO TABELLA]. Dato che per tutti gli algoritmi il numero di *EQ* è limitato da $\mathcal{O}(n)$ l'analisi delle *EQ* viene omessa. Infine si noti come l'analisi venga effettuata nel caso peggiore.

[10] Talvolta si conta anche il numero di simboli contenuto in tutte le stringhe per cui si effettua una *MQ* per pesare il costo di una *MQ* dato che è lineare con la lunghezza

Per tutti gli approcci il numero di suffissi necessari per distinguere tutti gli stati è limitato da n dato che ogni aggiunta di un suffisso determina almeno uno *split* e l'individuazione di almeno un nuovo stato (stati che sono al massimo n). Il numero di prefissi in tutte le componenti per *ObP* è al più nk perchè al più si hanno n componenti (n stati), se da ogni stato si hanno k transizioni il numero di transizioni totali sarà nk (i prefissi di uno stato sono le transizioni che finiscono in quello stato). La dimensione della tabella di osservazione o meglio la somma delle dimensioni di tutte le tabelle di osservazione di ogni componente è allora $nk \cdot n$ (numero prefissi · numero suffissi). Per AllGlobally si aggiungono tutti gli m suffissi di ogni controesempio, e lo si fa n volte quindi si avranno nm suffissi con una tabella di dimensione $nk \cdot nm$. Per OneLocally e OneGlobally vi è da aggiungere il numero di query fatte durante la ricerca del suffisso che nel caso della ricerca binaria è $log_2(m)$ effettuate n volte. Per la dimensione della tabella di L* si faccia riferimento alla sezione [INSERIRE RIFERIMENTO SEZIONE COMPLESSITA' COMPUTAZIONALE L*].

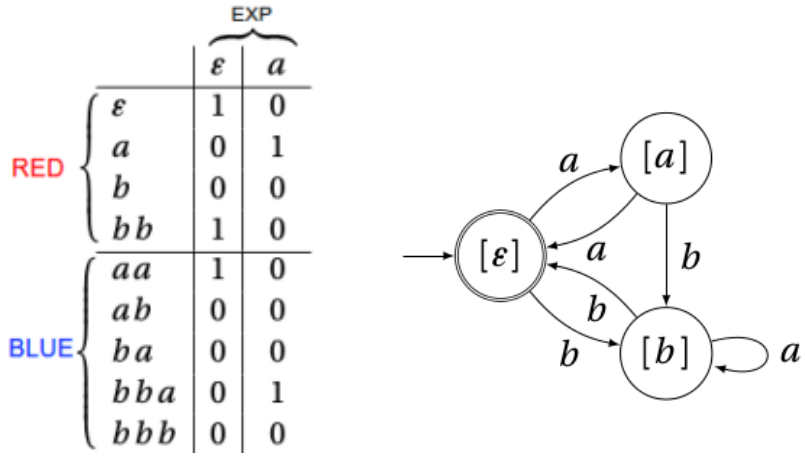
Algoritmo	Dimensione tabella	Membership queries
L*	$(k+1) \cdot (n + m(n-1)) \cdot n$	$\mathcal{O}(n^2km)$
AllGlobally	$nk \cdot nm$	$\mathcal{O}(n^2km)$
OneLocally	$nk \cdot n$	$\mathcal{O}(n^2k)$
OneGlobally	$nk \cdot n$	$\mathcal{O}(n^2k)$

Tabella 3.1: Complessità delle membership queries per le differenti varianti di ObP

Tuttavia i risultati evidenziati in tabella [INSERIRE RIFERIMENTO TABELLA] ed anche quelli omessi per le *EQ* costituiscono solo un limite superiore, il lavoro sperimentale in [INSERIRE RIFERIMENTO HOWAR] mette meglio in evidenza le differenze tra i vari algoritmi. OneLocally è l'algoritmo con il minore numero di *MQ* dato che il suffisso discriminante è aggiunto solo ad un componente e quindi sarà necessario completare solo la tebella di osservazione di questo componente ,e nonostante OneLocally e OneGlobally abbiano lo stesso ordine di grandezza per il numero di *MQ* fatte al crescere dei parametri la differenza in termini di *MQ* è notevole. In OneLocally tuttavia come detto in [INSERIRE RIFERIMENTO TESI DOVE SI PARLA DEL NUMERO DELLE EQUIVALENCE QUERY] il numero delle *EQ* aumenta drasticamente rispetto a OneGlobally anche se in numero pur sempre limitato dal numero di stati del *DFA* minimo equivalente ad $\mathcal{L}$ [INSERIRE RIFERIMENTO PUNTO TESI DOVE HO SPIEGATO QUESTO]. L* effettua un numero di *EQ* quasi sempre minore (seppur di poco) di OneGlobally ma un numero di *MQ* sempre molto maggiore sia a OneLocally che a OneGlobally. AllGlobally consente di effettuare un numero di *EQ* quasi sempre leggermente inferiore anche ad L* ma il numero di *MQ* è sempre molto elevato anche rispetto ad L*.

3.4.4 Discrimination tree vs Tabella di Osservazione localizzata

In figura 3.4a vi è la rappresentazione della tabella di osservazione per l'algoritmo



(a) (b)

Figura 3.4: DFA ipotesi e corrispondente tabella di osservazione in L*

L* per il *DFA* in figura 3.4b. I prefissi RED distinti tra di loro $\{\epsilon, a, b\}$ (cioè con row() distinte tra di loro) sono gli short prefix delle classi di equivalenza che hanno una corrispondenza con gli stati del *DFA*. I prefissi BLUE sono invece delle stringhe appartenenti ad una delle classi di equivalenza (un prefisso BLUE x termina nello stato rappresentato da uno stato RED r ed appartiene alla sua classe di equivalenza se row(r)=row(x)). Il *DT* è una rappresentazione della tabella di osservazione scevra da ridondanze come si può apprezzare in figura 3.5a. Infatti a parte lo short prefix superfluo bb, nella tabella di osservazione di L* non tutti i suffissi sono necessari per distinguere le varie classi di equivalenza (gli stati RED). Ad esempio il singolo suffisso ϵ da solo è sufficiente per distinguere $[\epsilon]$ dalle altre classi di equivalenza. Lo scopo di un *DT* è di eliminare queste ridondanze ed organizzare le osservazioni delle passate *MQ* in una maniera efficiente in modo da consentire efficientemente il *sifting* di una stringa. Nella tabella 3.2 vi è l'insieme di componenti di *ObP* corrispondente alla tabella di osservazione per OneGlobally anche se è solo un esempio e molti dei prefissi presenti in L* potrebbero non essere presenti in *ObP*.

C_ϵ	ϵ	a
ϵ	1	0
aa	1	0
bb	1	0

C_a	ϵ	a
a	0	1
bba	0	1

C_b	ϵ	a
b	0	0
ab	0	0
ba	0	0
bbb	0	0

Tabella 3.2: Insieme di componenti

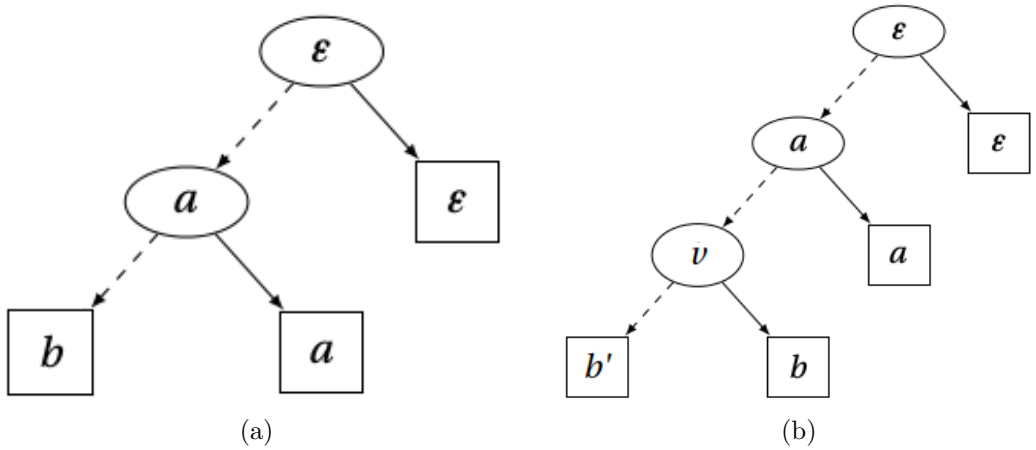


Figura 3.5: Discrimination Tree e split di uno stato

Sia il DT che C_I sono rappresentativi dell'ipotesi e quindi vi deve essere in ogni momento coerenza anche tra DT e C_I oltre che con l'ipotesi. Infatti per ogni foglia nell'albero c'è un corrispondente componente in C_I, l'etichetta della foglia corrisponde all'access sequence di quel componente. Entrambe le strutture dati sono rappresentative dell'ipotesi. A questo punto ci si potrebbe chiedere perchè utilizzare sia un DT che C_I per rappresentare l'ipotesi. Le considerazioni da fare differiscono tra OneLocally e OneGlobally. Quando si fa il sift di una stringa che termina in un nodo foglia si aggiungerà quella stringa al componente corrispondente a quel nodo foglia come detto in [INSERIRE RIFERIMENTO FUNZIONAMENTO].Tuttavia in OneGlobally l'insieme V di suffissi è globale e non coincide con i suffissi (che costituiscono un sottoinsieme di tutti i suffissi globali V),etichette dei nodi interni, incontrati durante la navigazione del DT per arrivare alla foglia summenzionata. Quindi vi è l'evenienza che tramite il *sift* di una stringa venga introdotta un'ulteriore non chiusura (ulteriore perchè sicuramente la non chiusura viene prodotta in prima istanza dal suffisso estrapolato dal controesempio) e conseguente *split*. Se non avessimo l'insieme di componenti sarebbe arduo esaminare la non chiusura nello scenario appena descritto da cui l'esigenza di C_I. Si noti inoltre che quanto appena descritto è il motivo per cui il numero di EQ non coincide ma è minore del numero degli stati del DFA minimo equivalente ad $\mathcal{L}$; infatti ad ogni " generazione " si possono avere più di uno *split* (ogni *split* consente di determinare un nuovo stato ed avvicinarci più velocemente alla soluzione) proprio per il motivo appena descritto. In OneLocally invece i suffissi di un componente C_x coincidono esattamente con i suffissi incontrati durante il *sifting* di x nel DT. Quando si effettua il sifting di un'altra stringa che tramite il DT si scopre essere in $[x]_{\simeq_{OT}}$, dato che giunge nel nodo foglia che ha $\tau(n_x) = x$, i suoi suffissi sono esattamente quelli incontrati durante la navigazione del DT e quindi non può essere introdotta non chiusura. Quindi in OneLocally si ha esattamente uno *split* ad ogni " generazione " ed il numero di EQ coincide con il numero di stati del DFA target minimizzato. In realtà non è

così per via dell'ottimizzazione descritta in [INSERIRE RIFERIMENTO SOTTOSEZIONE DOVE SI PARLA RIUTILIZZO CONTROESEMPIO CHE MI PARE SIA FUNZIONAMENTO] che consente di riutilizzare più volte lo stesso controesempio e risparmiare delle EQ, ma comunque il numero di EQ in OneLocally sarà maggiore del numero di EQ in OneGlobally. Va detto che il controllo sulla non chiusura che si effettua in closePack[INSERIRE RIFERIMENTO closePacK() ED IN PARTICOLARE IL NUMERO DI RIGA DOVE SI FA IL CONTROLLO DELLA NON CHIUSURA] può essere eliminato in OneLocally(il controllo va fatto solo la prima volta in assoluto che closePack() è chiamato, per le restanti generazioni la non chiusura si verificherà solo per via del suffisso del controesempio). Riassumendo in OneLocally si potrebbe fare a meno dell'insieme di componenti ed utilizzare esclusivamente il DT. Tuttavia utilizzare anche l'insieme di componenti in ogni caso comporta dei vantaggi prestazionali per alcune operazioni da fare inObP. La costruzione di H prevede di trovare lo short prefix associato ad una dato prefisso (ottenuto concatenando un altro short prefix con un simbolo di Σ), quest'operazione è più efficiente tramite C_I dato che nel DT comporta il *sifting* del prefisso e quindi un numero di MQ nel caso medio logaritmico da moltiplicare per la dimensione media dei suffissi incontrati. E ancora nella decomposizione del controesempio vi è l'esigenza di trovare dato un prefisso, lo si chiami p,il corrispondente short prefix di p ma può accadere che p non sia mai stato visto o che facendone il *sift* nel DT non termini in una foglia già esistente ma invece provochi la creazione di un nuovo nodo foglia. Quindi non riusciamo a trovare lo short prefix associato a p senza l'ausilio di C_I, anche se in realtà è pur sempre possibile effettuare il " parsing " della stringa x in H e tornare lo short prefix associato allo stato cui si arriva in H. In realtà lo sforzo computazionale richiesto per mantenere un'ipotesi anche tramite l'insieme di componenti potrebbe essere maggiore dei vantaggi prestazionali ottenuti rispetto ad un'implementazione che non fa uso che del DT per rappresentare H nel caso di OneLocally. Si ribadisce invece che nel caso di OneGlobally (e anche AllGlobally) ,per i motivi suddetti, C_I è necessario per testare la chiusura e garantire un corretto funzionamento.

3.4.5 Teacher

Tutte le considerazioni effettuate nella sottosezione [INSERIRE RIFERIMENTO TEACHER L*] inerenti il *teacher* di L* rimangono valide anche per l'ObP e ad esse si rimanda.

3.5 Scelte Progettuali

Si è implementato in C++11 l'algoritmo ObP utilizzando la versione OneGlobally ed interfacciandosi con una libreria preesistente. L'esposizione e lo pseudocodice riportato nella sezione [INSERIRE RIFERIMENTO SEZIONE ALGORITMO] per l'ObP fanno riferimento a Howar [INSERIRE RIFERIMENTO HOWAR] tranne

qualche piccola modifica. Diversamente alcune delle scelte progettuali per il codice si discostano in parte da quanto esposto in Howar. Qui saranno illustrate le differenze principali e le ragioni da cui tali scelte sono scaturite.
In prima analisi viene effettuata una preliminare minimizzazione del *DFA* target al fine di permettere un'eventuale[11] velocizzazione del *testing* dell'equivalenza tra H ed il *DFA* target, dato che le *performances* dell'algoritmo d'equivalenza *table-filling* dipendono in maniera quadratica dal numero di stati.
E' stata inserita l'ottimizzazione che permette di sfruttare più volte lo stesso controesempio finchè non lo è più. Come detto ciò consente di ridurre il numero di *EQ*. L'ottimizzazione è presente anche nell'implementazionde dell'*ObP* nella **LearnLib**[12].
Non è stata implementata invece l'ottimizzazione *semantic suffix closdness*(vedasi [INSERIRE RIFERIMENTO DOVE SI PARLA DI SEMANTIC SUFFIX CLOSDNESS]) che consente di ottenere delle ipotesi intermedie minime (che consentono di velocizzare le *EQ*) e di ridurre eventualmente il numero di *EQ*. L'ottimizzazione è invece presente nell'implementazione dell'*ObP* nella **LearnLib**.
Si è scelto di completare le componenti nel momento stesso che l'incompletezza è introdotta dato che i punti dove avviene sono ben circostanziati:

- dopo l'individuazione del suffisso discriminante nell'algoritmo [INSERIRE RIFERIMENTO OBSERVATION PACK E RIGA CODICE dove c'è l'if DI ONEGLOBALLY] e la sua aggiunta all'insieme globale di suffissi che avviene in una nuova funzione chiamata UPDATE-FROM-COUNTEREXAMPLE. Inoltre in UPDATE-FROM-COUNTEREXAMPLE dato che — in corrispondenza dei prefissi discriminati dal suffisso tornato dal metodo di decomposizione del controesempio — si conosce già l'esito della *MQ* ,perchè già effettuate nel metodo di decomposizione del controesempio, si risparmiano 2 *MQ*. Detto n il numero di stati del *DFA* target minimo ciò consente di risparmiare fino a un massimo di $2 \cdot n$ *MQ*.

- in OBP-SIFT() [INSERIRE RIFERIMENTO ALGORITMO SIFT]. Sia se durante il *sifting* il prefisso termini in un componente esistente oppure in uno nuovo è necessarrio effettuare delle *MQ* per completare il componente esistente oppure il nuovo componente rispettivamente sull'insieme di suffissi globale. In questo modo durante il *sifting* sarà possibile evitare di eseguire nuovamente le *MQ* effettuate durante la navigazione del *DT* con quel dato prefisso per tutti i suffissi incontrati durante il *sifting*. Resta inteso che utilizzando la strategia OneGlobally non tutti i suffissi contenuti nell'insieme di suffissi globale saranno incontrati e per i restanti alla fine sarà necessario effettuare una *MQ* esplicita per assicurare la chiusura del componente.

OBT-SPLIT non introduce incompletezza. La summenzionata gestione della completezza oltre a consentire di risparmiare delle *MQ* implica anche una migliore efficienza

[11] perchè il *DFA* può essere già minimo

[12] `www.learnlib.de` in cui si può trovare sotto la nomenclatura *Discrimination Tree* un'implementazione in Java della'algoritmo *ObP* di Howar

dell'algoritmo dato che l' algoritmo [INSERIRE RIFERIMETO OBT-CLOSEPACK alla riga dove c'è il while che controlla incompletezza] OBT-CLOSEPACK deve controllare se C_I è completo e ciò comportava un controllo su tutte le componenti.
Un'altra modifica significativa rispetto a OBT-CLOSEPACK originale di algoritmo [INSERIRE RIFERIMENTO CLOSE PACK] sta nell'individuazione del suffisso che determina la non chiusura alla riga [INSERIRE RIGA 4 CLOSE PACK]. Infatti il metodo di decomposizione del controesempio consente di determinare quali sono i due prefissi ed il suffisso che determina la non chiusura. Quindi ogni volta che viene chiamato OBT-CLOSEPACK non è necessario spendere del tempo di esecuzione nel cercare dove non si verifica la non chiusura. Questo è vero ad ogni prima chiamata di OBT-CLOSEPACK, può poi accadere che venga generata ulteriore non chiusura all'interno dell'algoritmo OBT-CLOSEPACK che in questo caso va individuata. C'è da precisare che quanto detto non è valido per la prima chiamata in assoluto di OBT-CLOSEPACK dato che la prima chiamata in assoluto non è preceduta da una chiamata al metodo di gestione del controesempio (che come detto permette di individuare esattamente in quale componente e per quale coppia di prefissi e quale suffisso accade la non chiusura). Quanto detto è riassunto sotto forma di pseudocodice nell'algoritmo [INSERIRE RIFERIMENTO MY-OBT-CLOSEPACK].

Esistono due versioni del metodo di decomposizione del controesempio BINARY-SEARCH di cui una è uguale a quella descritta in algoritmo [INSERIRE RIFERIMENTO BINARY SEARCH] ,che è usata dagli altri metodi di decomposizione del controesempio, e l'altra invece consente di scovare la decomposizione del controesempio *on the fly* cioè durante l'esecuzione dell'algoritmo. L'algoritmo è descritto in MY-BINARY SEARCH[INSERIRE RIFERIMENTO MY-BINARY-SEARCH. Nell'appendice [INSERIRE RIFERIMENTO APPENDICE] sono inseriti i dettagli implementativi più significativi

Appendice A

Preliminari

Lo scopo di quest'appendice è di stabilire una comune sintassi e semantica per concetti che sono rilevanti in tutta la tesi. Le definizioni e le notazioni qui introdotte sono essenziali per la maggior parte dei capitoli e quindi andrebbe letta.
L'appendice è divisa concettualmente in tre parti. Nella prima parte si introdurranno questioni puramente matematiche mentre nella seconda parte si definiranno grammatiche e linguaggi, infine nell'ultima parte si tratteranno in dettaglio gli automi a stati finiti.

A.1 Notazione matematica

L'obiettivo di questa sezione è di introdurre i concetti matematici propedeutici per questa tesi. Senza dubbio una conoscenza matematica di base è necessaria e chiaramente non può essere introdotto ogni singolo elementare concetto

A.1.1 Insiemi

Con $\mathbb{N}$ si indica l'insieme di numeri naturali interi non negativi incluso 0 (cioè $\mathbb{N} = 0, 1, 2, \ldots$). L'insieme di interi positivi è denotato da $\mathbb{N}^+$. Si definisce con $\mathbb{B} = 0, 1$ l'insieme di valori booleani dove 0 è associato al valore logico *falso* ed 1 al valore logico *vero*.
Dato un generico insieme X, $|X|$ denota la sua cardinalità, cioè il numero di elementi che contiene.

A.1.2 Relazione d'equivalenza

Una riflessiva ,simmetrica e transitiva relazione binaria $\approx \subseteq X \times X$ su un insieme X è detta una ***relazione d'equivalenza***. Dato un insieme X ed un elemento $x \in X$ si denota con $[x]_{\approx} = \{x' \in X \mid x \approx x'\}$ la *classe di equivalenza* di x(rispetto alla relazione d'equivalenza $\approx$).
Una relazione d'equivalenza $\approx$ su un insieme X si dice che *satura* un sottoinsieme $X' \subseteq X$ se e solo se X' è l'unione di qualche classe d'equivalenza di $\approx$. In simboli si

ha:

$$X' = \bigcup_{x \in X'} [x]_{\approx} \ ,$$

e ogni classe di equivalenza $[x]_{\approx}$ di $\approx$ o è un sottoinsieme o è disgiunta da X'.
Il ***quoziente*** (o insieme quoziente) di X rispetto a una relazione d'equivalenza $\approx$ è definito come l'insieme di tutte le classi d'equivalenza, ed è indicato da $X/\approx = \{[x]_{\approx} \mid x \in X\}$. L'***indice di una relazione d'equivalenza*** $\approx$ è definito come il numero di classi d'equivalenza, cioè è uguale a $|X/\approx|$. Una *partizione* di un insieme X è un insieme P i cui elementi, detti *blocchi* ed indicati con C, sono sottoinsiemi (disgiunti e non vuoti) dell'insieme X tali che:

1. se $C \in P$ allora $C \neq \emptyset$
2. se $C_1, C_2 \in P$ e $C_1 \neq C_2$ allora $C_1 \cap C_2 \neq \emptyset$
3. se $a \in X$ allora esiste $C \in P$ tale che $a \in C$ (è un altro modo di dire che l'unione di tutti gli insiemi C deve formare X)

Il *quoziente* di X forma una *partizione* di X

A.2 Linguaggi e grammatiche

A.2.1 Alfabeto, stringhe e linguaggi

Alfabeto

Di definisce L'***alfabeto*** Σ un qualsiasi insieme finito e non vuoto di simboli.

Stringhe

Una ***stringa*** è definita come una sequenza di simboli presi da un alfabeto. Cioè una stringa s definita su Σ è una sequenza $s = a_1 \dots a_n$ tale che $a_i \in \Sigma$.
$|s|$ denota la lunghezza della stringa s.
La *stringa vuota* è indicata con ϵ e $|\epsilon| = 0$.
Con Σ^* si denota l'insieme di tutte le possibili stringhe ottenibili sull'alfabeto Σ. Inoltre con Σ^+ si denota l'insieme $\Sigma^* - \{\epsilon\}$
I singoli simboli costituenti una stringa $w \in \Sigma^*$ sono indicati con w_i con $0 \leq i \leq |w|$ quindi $w = w_0 w_1 \dots w_{|w|-1}$. Per qualche intero nell'intervallo $I \subseteq [0, |w|], w_I$ è la stringa risultante prendendo solo le posizioni in w corrispondenti agli indici in I . Quindi $w_{[0,k)}$ è il prefisso di w di lunghezza k, e $w_{[k,|w|)}$ è il suffisso di w che inizia all'indice k(compreso). Si osservi che $w_{[0,0)} = w_{[|w|,|w|)} = \epsilon$ e $w_{[0,|w|)} = w$.
Un ***prefisso*** di una stringa $w \in \Sigma^*$ è una stringa $u \in \Sigma^*$ tale che esiste una stringa $v \in \Sigma^*$ soddisfacente $w = uv$. L'insieme di tutti i prefissi di una stringa w si indica con Pref(w)
Un ***suffisso*** di una stringa $w \in \Sigma^*$ è una stringa $v \in \Sigma^*$ tale che esiste una stringa

$u \in \Sigma^*$ soddisfacente $w = uv$. L'insieme di tutti i suffissi di una stringa w si indica con Suff(w)

Linguaggi

Un linguaggio L è un qualsiasi sottoinsieme di stringhe di Σ^*
Si definisce l'insieme dei prefissi di un linguaggio L:

$$\text{Pref}(L) = \bigcup_{w \in L} \text{Pref}(w)$$

e l'insieme dei suffissi di L:

$$\text{Suff}(L) = \bigcup_{w \in L} \text{Suff}(w)$$

Un linguaggio L è detto essere ***prefix-closed*** se e solo se $\text{Pref}(L) = L$. Informalmente questa proprietà di un linguaggio L viene sfruttata per indicare che qualunque stringa appartenente ad L si prende, qualunque suo prefisso deve ancora appartene ad L. Analogamente un linguaggio L è ***suffix-closed*** se e solo se $\text{Suff}(L) = L$.

La ***differenza simmetrica*** di due linguaggi L_1 e L_2 denotata con $L_1 \oplus L_2$ è tale che:

$$L_1 \oplus L_2 = \{x \in \Sigma^* : (x \in L_1 \wedge x \notin L_2) \vee (x \notin L_1 \wedge x \in L_2)$$

Un linguaggio può essere identificato mediante due tipi di descrizioni:

1. **Descrizione generativa**
   Consiste nell'utilizzare un formalismo denominato **grammatica generativa** ,introdotto da Noam Chomsky, che consiste in una serie di simboli e regole mediante le quali è possibile generare tutte e sole le stringhe del linguaggio.

2. **Descrizione descrittiva-identificativa**
   Il linguaggio è identificato o tramite un'enumerazione delle stringhe che vi appartengono o tramite una descrizione che cattura le caratteristiche delle sentenze costituenti il linguaggio,ad esempio le espressioni regolari. Un altro sistema formale identificativo sono gli **automi** .

I linguaggi possono essere classificati in base ai due tipi di descrizioni. Infatti è possibile delineare una tassonomia di grammatiche cui si farà corrispondere una classe di linguaggi. Ad ogni classe di grammatiche corrisponderà una classe di linguaggi (tutti quelli che quella classe di grammatiche è in grado di generare). E' possibile effettuare un'analoga corrispondenza tra classi di automi e classi di linguaggi, e quindi anche tra la gerarchia di automi e quella di grammatiche. Si rimanda alla sottosezione A.2.2 per una formalizzazione di questa gerarchia.

A.2.2 Grammatiche

Definizione (Grammatica generativa di Chomsky)**.** Una ***grammatica generativa di Chomsky*** è una quadrupla:

$$G = (\Sigma, V, S, P)$$

dove:
Σ è l'alfabeto, detto insieme di simboli terminali
V è l'insieme di simboli non terminali
S è il simbolo iniziale ed appartiene a V
P è l'insieme delle produzioni costituiti da una testa Ψ (il lato sinistro della produzione) e da una coda Ω aventi in generale questa forma:

$$\Psi \to \Omega \text{ con } \Psi \in (\Sigma \cup V)^* V (\Sigma \cup V)^* \text{ e } \Omega \in (\Sigma \cup V)^*$$

Si effettua una classificazione delle classi di grammatiche (e dei linguaggi ad esse associate) imponendo delle restrizioni sulle regole di produzione [3]:

- **Grammatiche di tipo zero - Unresticted** E' la classe di grammatiche più in alto nella gerarchia e per la quale non vi sono regole di restrizione da applicare alle produzioni. Sono in grado di generare la classe di **linguaggi ricorsivamente enumerabili**. Mediante l'approccio identificativo l'automa che riconosce ed accetta questi linguaggi è la **Macchina di Turing**

- **Grammatiche di tipo uno - Context Sensitive** Le regole di produzione sono così definite:
  $$\alpha_1 A \alpha_2 \to \alpha_1 \Omega \alpha_2 \text{ con } \alpha_1, \alpha_2, \Omega \in (\Sigma \cup V)^* \text{ e } A \in V$$
  La classe di linguaggi che queste grammatiche sono in grado di generare è detta **context-sensitive**. Gli automi in grado di riconoscere ed identificare questi linguaggi sono detti **Linear Bounded Automata**

- **Grammatiche di tipo due - Context Free** Le regole di produzione sono così definite:
  $$A \to \Omega \text{ con } \Omega \in (\Sigma \cup V)^* \text{ e } A \in V$$
  La classe di linguaggi che queste grammatiche sono in grado di generare è detta **context-free**. Gli automi in grado di riconoscere ed identificare questi linguaggi sono detti **Push Down Automata**

- **Grammatiche di tipo tre - Regular** Le regole di produzione sono così definite:
  $$A \to \alpha B \text{ oppure } A \to B\alpha \text{ con } A \in V, B \in (V \cup \{\epsilon\}) \text{ e } \alpha \in \Sigma^+$$
  I linguaggi che queste grammatiche generano sono detti **linguaggi regolari**. Gli automi in grado di riconoscere ed identificare questi linguaggi sono detti **Finite State Automata (*FSA*)**.

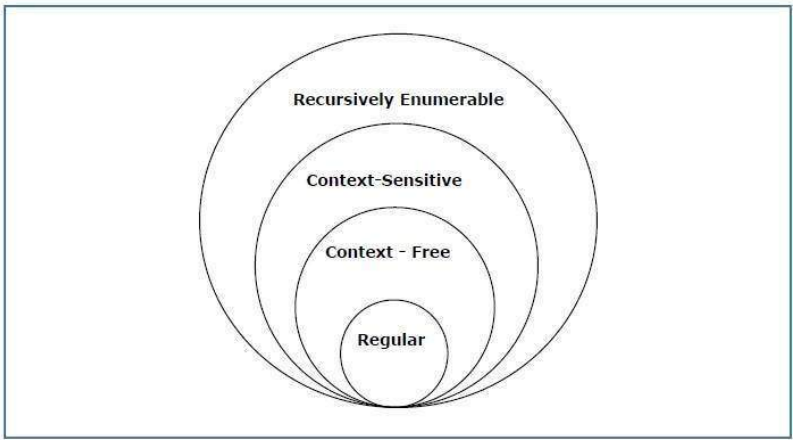


Figura A.1: Gerarchia di linguaggi secondo Chomsky

Le diverse classi di linguaggi, e quindi anche di grammatiche ed automi, si includono propriamente in maniera gerarchica come in figura A.1 . Ad una grammatica si può associare un unico linguaggio. Adesso si definirà come avvenie tale associazione:

Definizione. Il linguaggio generato da una grammatica $\mathcal{G}$ è l'insieme di tutte le stringhe che possono essere derivate a partire dal simbolo iniziale S:

$$L(\mathcal{G}) = \{x \in \Sigma^* : S \overset{\mathcal{G}}{\Rightarrow} x\}$$

Infine è rilevante notare che come detto una grammatica genera un unico linguaggio ma un linguaggio può essere generato da molteplici grammatiche.

A.3 Automi a stati finiti

Nella sottosezione A.2.2 sono stati introdotti gli automi e una loro classificazione. Qui ci si concentrerà sullo studio dei *FSA* in relazione ai linguaggi regolari, la classe dei linguaggi a cui questa tesi è rivolta. Come visto gli *FSA* sono un caso speciale di *macchina di Turing* e più nello specifico un caso speciale di **Finite State Machine (*FSM*)** che in questa sede non interessa definire. Qui basterà dire che una *FSM* è un *transiction system* costituita da un insieme finito di stati dove ogni transizione è innescata da un'azione tra un insieme finito di azioni (di solito denotato da Σ). Esistono diversi tipi di *FSM* come le *Mealy Machines* e i *FSA*. Tra i *FSA* si annoverano i **Non-deterministic Finite Automata (*NFA*)** strettamente correlati ai *DFA* su cui si focalizzerà l'attenzione.

Definizione A.1 (Automa a stati finiti deterministico)**.** Un *DFA* A è una quintupla:

$$A = \left\langle \Sigma, Q^A, q_\epsilon^A, \delta^A, \mathbb{F}_{\mathbb{A}}^A \right\rangle$$

dove:
Σ è un alfabeto

Q^A è un insieme finito di stati
$q^A_\epsilon \in Q^A$ talvolta indicato come q^A_λ è lo stato iniziale
$\delta^A : Q^A \times \Sigma \to Q^A$ è la funzione di transizione
$\mathbb{F}^A_\mathbb{A} \subseteq Q^A$ è l'insieme degli stati accettanti

Inoltre nel corso della trattazione seguendo [4, p. 72] in alcuni casi è conveniente utilizzare anche un'altra definizione per i *DFA* uguale a quella appena data ma comprendente anche un nuovo insieme $\mathbb{F}^A_\mathbb{R} \subseteq Q^A$ che è l'insieme degli stati rigettanti. Quando si parlerà di *DFA* si farà sempre riferimento alla prima definizione,quella classica, a meno che non è specificato o l'utilizzo della seconda definizione risulta tacitamente evidente dall'utilizzo dell'insieme $\mathbb{F}^A_\mathbb{R}$. Si indica con $\|A\| = |Q^A|$. Inoltre in molti frangenti è conveniente utilizzare (questo è un discorso che esula dalla definizione di un *DFA*) una versione estesa della funzione di transizione a una stringa anzichè ad un solo simbolo dell'alfabeto. Si definisce allora $\hat{\delta}^A : Q^A \times \Sigma^* \to Q^A$ definendo induttivamente $\hat{\delta}^A(q, \epsilon) = q$ e $\hat{\delta}^A(q, aw) = \hat{\delta}^A(\delta^A(q, a), w)$ per $q \in Q^A$ e $aw \in \Sigma^+$ e $w \in \Sigma^*$. Per la funzione di transizione estesa nel corso della tesi sarà anche usata interscambiabilmente la definizione $A[w] = \hat{\delta}^A(q^A_\epsilon, w)$. Inoltre si estende quest'ultima notazione agli insiemi di stringhe: $W \subseteq \Sigma^* \mid A[W] = \{A[w] : w \in W\}$.

Una stringa x è detta essere accettata da un *DFA* A se e solo se $\hat{\delta}^A(q_\epsilon, x) = q'$ tale che $q' \in \mathbb{F}^A_\mathbb{A}$ che significa che usando la funzione di transizione estesa $\hat{\delta}^A$ a partire dallo stato iniziale è possibile arrivare ad uno stato accettante. Il linguaggio individuato da un *DFA* A è allora:

$$L(A) = \{x \in \Sigma^* : \hat{\delta}^A(q_\epsilon, x) \in \mathbb{F}^A_\mathbb{A}\}$$

Una relazione analoga a quanto visto tra linguaggi e grammatiche sussiste tra linguaggi e *DFA*: un *DFA* induce un solo linguaggio regolare, ma ad un linguaggio regolare corrispondono più *DFA*.
In molti contesti è utile rifersi alla *funzione di output* di un *DFA* A, λ^A come:

$$\lambda^A : \Sigma^* \to \mathbb{B}, \quad \forall w \in \Sigma^* \quad \lambda^A(w) = \begin{cases} 1 & \text{se } w \in L(A) \\ 0 & \text{altrimenti} \end{cases}$$

La *funzione di output* è la *funzione caratteristica* di L(A).Inoltre λ^A appena definita sopra può essere vista come un caso particolare, per $q = q^A_\epsilon$, di $\lambda^A_q(w)$ che assume valore 1 se $\hat{\delta}^A(q, w) \in \mathbb{F}^A_\mathbb{A}$. Ancora, due *DFA* ,A e A' , sono **equivalenti** denotato da $A \cong A'$, se loro hanno la stessa funzione di output, cioè se $\lambda^A = \lambda^{A'}$, cioè se indivividuano lo stesso linguaggio. Il concetto di equivalenza è rilevante anche tra gli stati dello stesso *DFA* ed informalmente due stati sono equivalenti se non esiste nessuna stringa che li distingue, cioè che partendo da quei due stati porta a stati di arrivo che sono uno accettante e l'altro no.

Definizione (Stati equivalenti)**.** Detto A essere un *DFA*, e q e p $\in Q^A$ stati di A. q e p sono *equivalenti* ,denotato da $q \equiv p$, se $\lambda^A_q = \lambda^A_p$

Una stretta correlazione tra l'equivalenza di stati e l'equivalenza di *DFA* è il seguente risultato: $A \cong A' \Leftrightarrow q_\epsilon^A \equiv p_\epsilon^{A'}$. Un altro concetto importante è quello di *isomorfismo* tra due DFA.

Definizione (Isomorfismo di *DFA*)**.** Detti A ed A$'$ due *DFA* definiti su Σ. A ed A$'$ sono detti isomorfici se esiste un isomorfismo $f : Q^A \to Q^{A'}$, cioè una funzione soddisfacente le seguenti condizioni:

1. $f(q_\epsilon^A) = q_\epsilon^{A'}$
2. $\forall q \in Q^A : q \in F_{\mathbb{A}}^A \Leftrightarrow f(q) \in F_{\mathbb{A}}^{A'}$
3. $\forall q \in Q^A, a \in \Sigma : f(\delta^A(q, a)) = \delta^{A'}(f(q), a)$

L'isomorfismo è un requisito più forte dell'equivalenza: *DFA* isomorfi sono pure equivalenti, ma in generale non è vero il contrario. Quindi dato un linguaggio L esistono più *DFA* in grado di riconoscerlo cioè con la stessa *funzione di output* λ. Tra questi di particolare interesse sono quelli con il minor numero di stati. Un *DFA* A è detto **minimo** se qualunque altro *DFA* A$'$ tale che $A \cong A'$ (con la stessa *funzione di output*) soddisfa $|Q^{A'}| \geq |Q^A|$. Ovviamente in un *DFA* minimo nè stati irragiungibili ne stati equivalenti possono essere presenti perchè potrebbero essere eliminati senza cambiare la funzione di output λ. Inoltre il DFA minimo è sempre unico a meno di una possibile rinomina degli stati. Per motivi storici esiste anche la definizione di *DFA* **canonico** che è intercambiabile con quella di *DFA* minimo ma entrambe indicano lo stesso ente matematico e sono del tutto equivalenti. Formalizzando

Definizione (DFA minimo/canonico)**.** Detto A essere un *DFA* su Σ. A è detto canonico se le seguenti condizioni sono verificate:

1. Tutti gli stati sono raggiungibili: $A[\Sigma^*] = Q^A$
2. Tutti gli stati sono a coppie separabili[1]: $\forall q \neq p \in Q^A : \exists w \in \Sigma^* : \lambda_q^A(w) \neq \lambda_p^A(w)$

Per ogni DFA esiste sempre ,ed è unico (a meno delle etichette degli stati) un *DFA* equivalente che è canonico.

A.3.1 FSA particolari

Alcuni *DFA* ed *NFA* sono particolarmente significativi e ricorreranno spesso nell'ambito di questa tesi. Inoltre è necessario conoscere per il proseguio della trattazione qual è la differenza principale tra *DFA* ed *NFA*. Un *NFA* è detto non deterministico perchè la sua funzione di transizione può avere più di una transizione per un dato simbolo dell'alfabeto (ed inoltre vi possono essere transizioni anche in corrispondenza di ϵ). Inoltre in un *NFA* non vi è necessariamente per ogni stato una transizione in corrispondenza di ogni simbolo dell'alfabeto.

[1] due stati sono separabili o distinti se non sono equivalenti

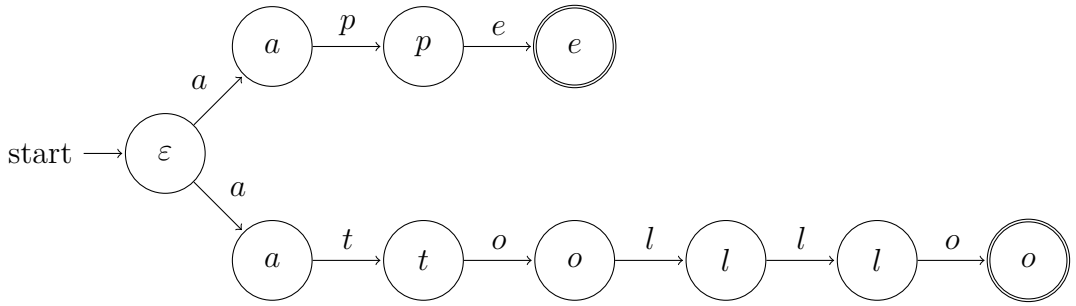


Figura A.2: *MCA(I_+)* per $I_+ = \{\text{ape,atollo}\}$

Maximal Canonical Automaton

Definizione A.2 (Maximal Canonical Automaton)**.** Detto $I_+ = \{x_1, \cdots, x_N\}$ un insieme di esempi positivi, si definisce ***Maximal Canonical Automaton rispetto ad*** I_+ e si denota con ***MCA(I_+)*** un *NFA* costituito da una quintupla $\langle \Sigma, Q, q_\epsilon, \delta, \mathbb{F}_\mathbb{A} \rangle$ dove:

Σ è l'alfabeto su cui è definito I_+
$Q = \{q_u^i : u \in \text{Pref}(x_i) \wedge u \neq \epsilon\} \cup \{q_\epsilon\}, 1 \leq i \leq N$
$q_\epsilon = \{q_\epsilon\}$
$\delta(q_u^i, a) = \{q_{ua}^i : ua \in \text{Pref}(x_i)\}, \forall a \in \Sigma, 1 \leq i \leq N$
$\delta(q_\epsilon, a) = \{q_a^i : a \in \text{Pref}(x_i)\}, \forall a \in \Sigma, 1 \leq i \leq N$
$1 \leq i \leq N, q_{x_i}^i \in \mathbb{F}_\mathbb{A}$
se $\epsilon \in I_+$ aggiungere q_ϵ ad $\mathbb{F}_\mathbb{A}$

Un *MCA(I_+)* per ogni stringa di I_+ ha un percorso dedicato a partire dallo stato iniziale. Si noti che nella definizione data di δ accade che per molti simboli di Σ non c'è la corrispondente transizione per un dato stato. Inoltre se in I_+ sono presenti stringhe che hanno lo stesso simbolo iniziale si avrà indeterminismo sullo stato iniziale q_ϵ. Quindi in generale il *MCA(I_+)* è un *NFA*.Un esempio è dato in figura A.2

Automa quoziente

Sia A un *DFA* su Σ e sia $\approx \subseteq Q^A \times Q^A$ una relazione d'equivalenza sull'insieme Q^A soddisfacente le seguenti due condizioni:

(i) $\approx$ satura $\mathbb{F}_\mathbb{A}^A$

(ii) $\forall q, p \in Q^A : q \approx p \Rightarrow (\forall a \in \Sigma : \delta^A(q, a) \approx \delta^A(p, a))$

Allora da A tramite $\approx$ è possibile ricavare il ***DFA quoziente*** $A/\approx$ così definito:

1. Σ è lo stesso di A

2. $Q^{A/\approx} = Q^A/\approx$
3. $q_\epsilon^{A/\approx} = [q_\epsilon^A]_\approx$
4. $\mathbb{F}_\mathbb{A}^{A/\approx} = \{[q]_\approx : q \in \mathbb{F}_\mathbb{A}^A\}$
5. $\delta^{A/\approx}([q]_\approx, a) = [\delta^A(q,a)]_\approx \quad \forall q \in Q^A, a \in \Sigma$

L'automa quoziente $A/\equiv$,dove la relazione d'equivalenza utilizzata utilizzata è quella di equivalenza tra gli stati, corrisponde al *DFA* canonico. Quindi banalizzando si può concludere dicendo che l'automa quoziente di un *DFA* è ciò che si ottiene fondendo insieme alcuni stati del *DFA* di partenza in base a una relazione d'equivalenza. Quando la relazione usata è quella di stati equivalenti si ottiene il DFA minimo: quindi $Q^{A/\approx}$ sarà una partizione in cui in ogni blocco (sottoinsieme) ci saranno stati equivalenti tra loro (in uno specifico blocco). Ogni blocco è una classe d'equivalenza.

Prefix Tree Acceptor

Si è visto che l'automa quoziente che si ottiene usando la relazione di equivalenza degli stati su un *DFA* A è il *DFA* minimo. Analogamente è possibile ottenere il ***Prefix Tree Acceptor di*** I_+ indicato con $\boldsymbol{PTA}(I_+)$ applicando la definizione di automa quoziente al $MCA(I_+)$[2] con la relazione: *stati che identificano lo stesso prefisso*. Quindi verrà effettuata la fusione degli stati che condividono lo stesso prefisso. Si definisce la relazione d'equivalenza come:

$$p \approx q \Leftrightarrow \text{ Prefix}(p) = \text{ Prefix}(q)$$

allora:

$$MCA(I_+)/\approx = \ PTA(I_+)$$

In figura A.3 il *PTA* ricavato a partite dal *MCA* di figura A.2 .

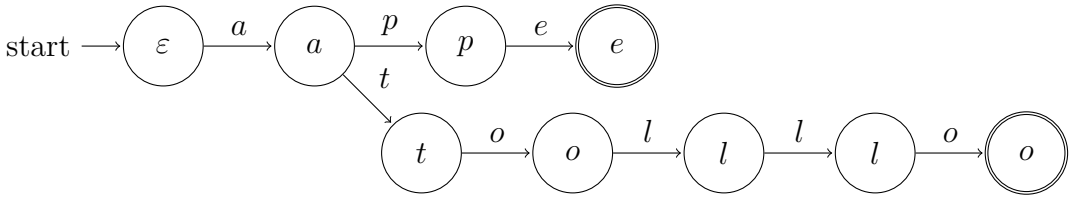


Figura A.3: *PTA*(I_+) per $I_+ = \{$ape,atollo$\}$

Automa Universale

Si indica con ***UA l'automa universale*** che accetta tutte le stringhe definite su Σ. Si ha $L(UA) = \Sigma^*$. Ha un unico stato,che è accettante, con un *self-loop* per ogni simbolo dell'alfabeto.

[2]Anche se tecnicamente può essere un *NFA* la definizione di automa quoziente può essere applicata comunque

A.3.2 Funzioni di output regolari

In una sottosezione di A.3.1 si è visto che è possibile ottenere il *DFA* canonico minimo di un *DFA* A tramite $A/\equiv$ (l'automa quoziente sulla relazione di stati equivalenti). In questa sezione invece si vedrà come ottenere il *DFA* canonico minimo non a partire da un *DFA* preesistente, ma semplicemente sfruttando le proprietà di una funzione di output $\lambda : \Sigma^* \to \mathbb{B}$ che è la funzione caratteristica di qualche linguaggio regolare.

Relazione di Nerode

Si possono caratterizzare le *funzioni di output regolare* come la classe di funzioni $\lambda : \Sigma^* \to \mathbb{B}$ per cui un *DFA* con quella *funzione di output* esiste[3]. Il famoso teorema *Myhill-Nerode* [INSERIRE RIFERIMENTO] fornisce una caratterizzazione alternative delle *funzioni di output regolari* , che non fa affidamento sulla nozione di *DFA*. Come primo step si definisce la *relazione di Nerode* [INSERIRE RIFERIMENTO] sulle stringhe che definisce un'equivalenza sulle stringhe secondo λ:

Definizione A.3 (Relazione di Nerode)**.** Sia $\lambda : \Sigma^* \to \mathbb{B}$ una funzione di output a due valori arbitraria definita su Σ. Due stringhe $u, u' \in \Sigma^*$ sono equivalenti secondo $\simeq_\lambda$[4] denotato da $u \simeq_\lambda u'$ se e solo se:

$$\forall v \in \Sigma^* \quad \lambda(uv) = \lambda(u'v)$$

dove $\simeq_\lambda \subseteq \Sigma^* \times \Sigma^*$ è una relazione binaria detta *relazione di Nerode* o *congruenza di Nerode* che definisce l'equivalenza tra stringhe secondo λ

Teorema Myhill-Nerode

La *relazione di Nerode* $\simeq_\lambda$ può essere vista come l'equivalente a livello di stringhe della relazione di equivalenza $\equiv_A \subseteq Q^A \times Q^A$, sugli stati di un *DFA* A. Dovrebbe essere osservato, tuttavia , che $\simeq_\lambda$ può essere definito per *funzioni di output* arbitrarie, non solo regolari. Il teorema *Myhill-Nerode* [INSERIRE RIFERIMENTO] fornisce una caratterizzazione delle *funzioni di output regolari* basata su $\simeq_\lambda$:

Teorema A.1 (Teorema Myhill-Nerode o di caratterizzazione)**.** *Sia $\lambda : \Sigma^* \to \mathbb{B}$ una funzione di output a due valori. λ è regolare se e solo se la relazione di Nerode $\simeq_\lambda$ ha indice finito.*

Una dimostrazione del teorema si trova in [INSERIRE RIFERIMENTO PROVA TEOREMA]. L'implicazione in uno dei due versi del teorema dice che se $\simeq_\lambda$ ha indice finito, allora λ è *una funzione di output regolare* cioè esiste un *DFA* A con $\lambda^A = \lambda$. Questo *DFA* $A = \langle \Sigma, Q^A, q_\epsilon^A, \delta^A, F_{\mathbb{A}}^A \rangle$ è definito come:

- Σ è il dominio di λ

[3] L'insieme delle funzione di output regolari identifica la classe dei linguaggi regolari

[4] Si noti che per denotare l'equivalenza non si è usato il simbolo $\equiv$ (equivalenza tra stati) perchè qui si parla di equivalenza tra stringhe

- $Q^A = \Sigma^*/\simeq_\lambda$
- $q_\epsilon^A = [\epsilon]_{\simeq_\lambda}$
- $F_{\mathbb{A}}^A = \{[u]_{\simeq_\lambda} |\ \lambda(u) = 1\}$
- $\delta^A([u]_{\simeq_\lambda}, a) = [u{\cdot}a]_{\simeq_\lambda}$

A è il *DFA* minimo corrispondente al linguaggio identificato da λ.Si osservi come la costruzione del *DFA* A è molto simile alla costruzione del *DFA* minimo usando la relazione di equivalenza sugli stati $\equiv$ usando l'automa quoziente a partire da un *DFA*. Questo approccio alla costruzione degli automi è fondamentale nell'*active learning*.

Appendice B

Preliminari e implementazione dell'Observation Pack

Qui si presentano delle ulteriori notazioni inerenti prevalentemente il capitolo 3 e vengono svelati alcuni dettagli implementativi dell'*ObP*.

B.1 Notazione specifica per l'ObP

In questo sezione vi è la delineazione di un'ulteriore notazione utilizzata principalmente nel capitolo 3 in congiunzione a quella introdotta in A, ma che essendo specifica dell'*ObP* viene presentata in quest'appendice. Inoltre vi è anche la presentazione della notazione utilizzata per presentare un framework introdotto in [16] che facilità la compresione della correttezza e l'implentazione del metodo di gestione del controesempio in *ObP*.

B.1.1 Definizioni

L'*ObP* mantiene un insieme Sp,*prefix-closed*, di **short prefix** detti anche **access sequence** ,che sono prefissi . Ogni short prefix identifica unicamente (cioè short prefix diversi identificano stati diversi in H e nel target) gli stati sia nel target A che nell'ipotesi H. Ogni stato $q \in Q^H$ unicamente corrisponde ad una stringa (lo short prefix) $u \in Sp$, ed è assicurato che $H[u] = q$. u è detta l'access sequence di q (in H), ed è denotata da $\lfloor q \rfloor_H$. Alternativamente quanto detto può essere formulato come $\forall q \in Q^H : \ H[\lfloor q \rfloor_H] = q$. Lo stato iniziale q_ϵ^H è lo stato con access sequence ϵ.
Si estende questa notazione a stringhe arbitrarie $w \in \Sigma^* : \lfloor w \rfloor_H = \lfloor H[w] \rfloor_H$ che significa che w raggiunge uno stato in H e questo stato ha un access sequence u cui w si associa. Quindi la funzione $\lfloor \cdot \rfloor_H : \Sigma^* \to Sp$ trasforma stringhe in access sequences.
Uno short prefix $u \in Sp$ corrisponde ad uno stato in A, cioè $A[u]$. Ci si riferisce ad $A[Sp]$ come gli stati scoperti (dal *learner*) di A. Gli short prefixes quindi stabiliscono una funzione f_{Sp} che collega stati nell'ipotesi e stati scoperti nel *DFA* target A come

segue:

$$f_{Sp} : Q^H \to Q^A, f_{Sp}(q) = A[\lfloor q \rfloor_H]$$

B.1.2 Definizioni per il framework

Prefix Transformation

Prefix transformation è una procedura che consente di trasformare un prefisso di un controesempio $w \in \Sigma^+$ in un access sequence in Sp.

Definizione (Prefix Transformation)**.** Prefix transformation rispetto ad H, π_H , è definita come segue:

$$\pi_H : \Sigma^* \times \mathbb{N} \to \Sigma^* \ , \ \pi_H(w, i) = \lfloor w_{[0,i)} \rfloor_H \cdot w_{[i,|w|)}$$

Si osservi che, $\pi_H(w, 0) = w$ e $\pi_H(w, |w|) = \lfloor w \rfloor_H \in Sp$.

Altre definizioni

Sia $w \in \Sigma^+$[1] un controesempio che differenzia il target A da H. Sia $m = |w|$ ed i un indice $0 \leq i \leq m$ allora si definisce la funzione α come:

$$\alpha : [0, m+1) \to \mathbb{B},\ \alpha(i) = \begin{cases} 1 & \text{se } \lambda^A(\pi_H(w, i)) = \lambda^H(w) \\ 0 & \text{altrimenti} \end{cases}$$

Dalla funzione α può essere ricavato anche la definizione della funzione β:

$$\beta : [0, m) \to \{0, 1, 2\},\ \beta(i) = \alpha(i) + \alpha(i+1)$$

B.2 Dettagli implementativi dell' ObP

Vengono ora riportati alcuni dettagli implementativi ritenuti più significativi senza pretesa di esaustività. Per la memorizzazione della funzione OT [INSERIRE RIFERIMENTO OT] di un componente si è utilizzata una map che tipicamente è nella forma chiave-valore dove nella fattispecie la chiave è una stringa (ottenuta dalla concatenazione di un prefisso con un suffisso) ed il valore è l'esito delle *MQ* nel target per quella stringa. Qui si è utilizzato per il valore un doppio campo oltre all'esito della *MQ* costituito da un contatore del numero di volte che quella stringa viene a formarsi in un dato componente (una stessa stringa può essere formata da coppie prefisso-suffisso diverse). Questo è dovuto al fatto che quando si effettua uno split di un componente alcuni dei prefissi del componente migrano nei prefissi del nuovo componente. Oltre all'insieme dei prefissi del componente splittato (che va decrementato) si può modificare anche il dominio della funzione OT restringendolo. Nel codice si è scelto di eliminare la concatenazione del prefisso che migra nel nuovo componente con l'insieme di suffissi del componente splittato, restringendo il dominio di OT. Tuttavia per garantire la correttezza è necessario appurare se

quella concatenazione di quel prefisso con un dato suffisso si venga a formare anche tramite altri prefissi perchè in questo caso l'eliminazione non deve avvenire ed è per questo motivo che si usa un contatore per ogni stringa (per ogni concatenazione) che va incrementato ad ogni inserimento di una stringa (anche una già esistente) e decrementato nel caso suddetto. Questa scelta è stata fatta nell'ottica di consentire velocemente di appurare se un componente è chiuso e in generale consentire una ricerca molto veloce di un prefisso in un componente. L'alternativa sarebbe stata quella di non utilizzare il contatore e non eliminare mai la stringa formata dalla concatenazione di un prefisso e di un suffisso ma solo il prefisso dal componente splittato. Ciò porterebbe ad una crescita del dominio di ricerca per la funzione OT che degraderebbe le prestazioni della ricerca di un prefisso in un componente (operazione effettuata sovente) anche se potrebbe comportare una diminuizione del numero delle *MQ* e l'eliminazione dell'*overhead* per tenere aggiornato il contatore: questa prospettata diminuizione delle *MQ* con questa seconda scelta è però possibile soltanto se quando si completano le componenti, ad esempio nella funzione *update_from_counterexample()* prima di effettuare una *MQ* su una stringa x si controlli se per x non si conosca già l'esito della *MQ* perchè contenuto già nel dominio della funzione OT[2] anche se nel caso vi fossero molti *miss* questa politica potrebbe essere addirittura controproducente. Anche quest'ulteriore strategia non è stata implementata ritenendo poco probabile un *hit* (per implementarla basta decommentare un if in update_from_counterexample() e aggiungerne un altro nella funzione *sift*). Se si fossero fatte delle scelte opposte a quelle fatte e appena descritte senz altro si sarebbe potuto abbassare ulteriormente il numero di *MQ* ma si è ritenuto che il gioco non vale la candela cioè che il prezzo da pagare in termini di tempo di esecuzione per ottenere ciò è maggiore del beneficio ottenuto.
Si sottolinea che diversamente dal metodo OBP-SPLIT (algoritmo [INSERIRE RIFERIMENTO ALGORITMO]) non viene ritornato il nuovo componente ottenuto perchè il chiamante OBP-CLOSEPACK ne è già a conoscenza , quindi il metodo non ritorna niente.
Infine si prende in esame l'implementazione del discrimination tree. Per quest ultimo si è scelto di usare un *vector* di nodi e un insieme di archi. Per ogni nodo si memorizza l'etichetta e se è accettante o meno. Gli archi sono un *vector* di array bidimensionali. L' indice di un nodo nel *vector* di nodi è usato per accedere alla posizione nel *vector* di archi che contiene gli indici dei nodi figli (nell'array contenuto nel *vector* di nodi nella posizione individuata dall'indice del nodo). L'utilizzo di un insieme di nodi e di archi è tipico di un grafo piuttosto che di un albero binario. Si è scelta ugualmente questa implementazione essenzialmente per due ragioni:

- La *Standard Template Library* non mette a disposizione nessuna struttura dati per modellare una albero binario. Esistono delle librerie esterne che mettono a disposizioni un albero n-ario ma l'overhead per gestire n figli ed altre operazioni inutili ai fini dell'*ObP* ha fatto propendere per il declinare il loro utilizzo. Un'altra possibilità sarebbe stata quella d'implementare un albero

[2]più è grande il dominio e maggiore sarà la probabilità di ottenere un *hit* per x

binario *ad hoc* nella maniera classica cioè tramite i puntatori ma essendo le prestazioni simili a quelle della soluzione adottata e descritta sopra non lo si è fatto. Inoltre si è supposto che chiamare un oggetto di una classe esterna (quella dell'eventuale implementazione dell'albero binario), dato che va fatto molte volte, sarebbe divenuto il costo preponderante. Il vantaggio principale nell'usare l'implementazione di un albero binario con i puntatori per il discrimination tree sta nel risparmio di memoria circa doppia ma comunque sempre lineare nella soluzione proposta, e che non avviene mai la riallocazione (operazione che accade quando le dimensioni del vettore superano la capacità dello stesso).

- Utilizzare un vettore indicizzato. Questa soluzione è da scartare perchè se il discrimination tree non è bilanciato e presenta un ramo molto più lungo degli altri sarebbe necessario rendere il vettore molto grande. Il vettore può crescere dinamicamente ed è molto più probabile con un vettore indicizzato superare la capacità totale del vettore (se l'inserimento del nodo avviene nello stesso ramo ciò avviene molto velocemente) che verrebbe quindi riallocato e ricopiato frequentemente degradando le prestazioni.

Bibliografia

[1] Dana Angluin. «Learning Regular Sets from Queries and Counterexamples». In: *Inf. Comput.* 75.2 (nov. 1987), pp. 87–106. ISSN: 0890-5401. DOI: 10.1016/0890-5401(87)90052-6.

[2] Dana Angluin. «Negative results for equivalence queries». In: *Machine Learning Journal* 5 (giu. 1990), pp. 121–150. ISSN: 0885–6125. DOI: 10.1007/BF00116034.

[3] Noam Chomsky. «On Certain Formal Properties of Grammars». In: *Information and Computation* II (giu. 1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6.

[4] Colin De La Higuera. *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0521763169 9780521763165.

[5] J.E. Hopcroft e R.M. Karp. *A Linear Algorithm for Testing Equivalence of Finite Automata*. Rapp. tecn. Cornell Unisersity, dic. 1971. URL: http://hdl.handle.net/1813/5958.

[6] J.E. Hopcroft, R Motwani e J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2ª ed. Pearson/Addison Wesley, 2007, p. 153. ISBN: 9780201441246.

[7] Falk M. Howar. «Active learning of interface programs». Tesi di dott. Technischen Universitat Dortmund, 2012, pp. 9–23. DOI: 2003/29486.

[8] Michael J. Kearns e Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0-262-11193-4.

[9] Ryszard Si Michalski. «Concept learning». In: *Encyclopedia of Artificial Intelligence*. A cura di Stuart C. Shapiro. Vol. I. Wiley–Interscience Publications, gen. 1986, pp. 185–194.

[10] Ryszard Si Michalski. «Inferential theory of learning: Developing fondations for multistrategy learning». In: *Machine Learning: a Multistrategy Approach*. Vol. IV. Morgan Kauffman Inc., 1993.

[11] Ryszard Si Michalski. «Understanding the nature of learning: Issue and research direction». In: *Machine Learning, an Artificial Intelligence Approach*. A cura di S. Ryszard Michalski, G. Jaime Carbonell e M. Tom Mitchell. Vol. II. Morgan Kaufmann, 1986. ISBN: 0-934613-00-1.

[12] Daphne Norton. «Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP». Tesi di laurea mag. Rochester Institute of Technology, mar. 2009, pp. 19–31. URL: http://scholarworks.rit.edu/theses/6939/.

[13] Ronald L. Rivest e Robert E. Schapire. «Inference of Finite Automata Using Homing Sequences». In: *Information and Computation* 103 (apr. 1993), pp. 299–347. ISSN: 0890-5401. DOI: 10.1006/inco.1993.1021.

[14] Herbert A. Simon. «Why should machine learn?» In: *Machine Learning, an Artificial Intelligence Approach*. A cura di Mitchell, Michalski e Carbonel. Vol. I. Tioga Publishing Company, 1983, pp. 25–37.

[15] Bernhard Steffen, Falk M. Howar e Malte Isberner. «The TTT Algorithm: A Redundancy Free Approach to Active Automata Learning». In: *Runtime Verification*. A cura di Bonakdarpour e Smolka. Vol. 8734. Lecture Notes in Computer Science. Springer, 2014, pp. 307–322. ISBN: 978-3-319-11163-6. DOI: 10.1007/978-3-319-11164-3.

[16] Bernhard Steffen e Malte Isberner. «An Abstract Framework for Counterexample Analysis in Active Automata Learning». In: a cura di A. Clark, M. Kanazawa e R. Yoshinaka. Vol. 34. Proceedings of the International Conference on Grammatical Inference, ICGI 2014. Kyoto, Japan, set. 2014, pp. 79–93. URL: http://jmlr.org/proceedings/papers/v34/isberner14a.pdf.