UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica

IPERMINIMIZZAZIONE DI AUTOMI A STATI FINITI DETERMINISTICI

Relatore: Prof. Giovanni PIGHIZZINI

Autore:
Andrea TINELLI
Matricola: 941800

Anno Accademico 2023 - 2024

Questa pagina è stata lasciata intenzionalmente vuota.

Prefazione

Nel campo dell'informatica teorica, uno degli argomenti di maggiore interesse è quello relativo alla teoria degli automi. All'interno di questo campo, sono studiate diverse tipologie di automi, tra i quali figurano gli automi a stati finiti deterministici. Uno dei problemi di maggiore rilevanza che riguarda questo tipo di automi è quello della minimizzazione. Tale problema consiste nel trovare un automa a stati finiti deterministico che riconosca lo stesso linguaggio di un automa di partenza, ma con il minor numero di stati possibile. L'automa che soddisfa tali condizioni è detto minimo e, in generale, è unico. Questo significa che, a meno di isomorfismi, non esiste un automa differente con lo stesso numero di stati che riconosca lo stesso linguaggio. Recentemente è stato introdotto un nuovo problema affine a quello della minimizzazione, chiamato iperminimizzazione di automi a stati finiti deterministici. Questo problema consiste, dato un automa deterministico, nel trovare un automa a stati finiti deterministico, detto iperminimo, con il minor numero di stati possibile che riconosca un linguaggio finitamente differente rispetto al linguaggio riconosciuto dall'automa di partenza. In altri termini, dato un automa iperminimo, un qualsiasi altro automa con un numero minore di stati riconosce un linguaggio infinitamente differente rispetto al linguaggio accettato dall'automa di partenza. Contrariamente al problema della minimizzazione, l'automa iperminimo non è unico. È infatti possibile trovare più automi iperminimi per uno stesso automa di partenza, ciascuno dei quali riconosce un linguaggio diverso finitamente differente rispetto a quello riconosciuto dall'automa di partenza. L'iperminimizzazione risulta essere un problema di grande interesse teorico e pratico in quanto, in alcuni casi, permette di ridurre notevolmente il numero di stati dell'automa di partenza, anche rispetto alla sua versione minimizzata. Il costo di questa riduzione consiste però nell'introduzione di un numero finito di errori, cioè di stringhe su cui la risposta dell'automa iperminimo è sbagliata rispetto a quella che sarebbe stata fornita dall'automa di partenza. Negli ultimi anni la ricerca di algoritmi efficienti per la soluzione del problema dell'iperminimizzazione è stata oggetto di studio. Nel corso del tempo sono emersi principalmente tre algoritmi per la risoluzione del problema, ciascuno dei quali rifinisce le strategie utilizzate migliorando la complessità computazionale dell'algoritmo precedente. Il primo algoritmo presentato in grado di risolvere

il problema dell'iperminimizzazione in tempo polinomiale è stato quello di Badr, Geffert e Shipman [BGS09], con complessità computazionale pari a $\mathcal{O}(n^3)$, dove n è il numero di stati dell'automa di partenza. Successivamente, sono stati proposti due nuovi algoritmi, l'algoritmo di Badr [Bad08], con complessità computazionale $\mathcal{O}(n^2)$, e l'algoritmo di Holzer e Maletti [HM10], con complessità computazionale uguale a $\mathcal{O}(n \log n)$. In questo elaborato vengono indagate le prestazioni di questi algoritmi nel tentativo di stabilire un ponte tra la teoria e la pratica. Gli algoritmi sono stati dunque implementati e sono stati condotti dei test per valutarne le presentazioni. I risultati degli esperimenti svolti mostrano come l'algoritmo di Holzer e Maletti sia il più efficiente tra quelli proposti, in accordo con quanto previsto dalla teoria. Più sorprendentemente invece l'algoritmo di Badr, Geffert e Shipman, nonostante abbia una complessità computazionale maggiore rispetto all'algoritmo di Badr, si comporta in media meglio nella pratica in quanto il caso peggiore si verifica molto di rado.

L'elaborato è organizzato come segue:

- nel Capitolo 1 sono presentate le nozioni preliminari, necessarie alla comprensione dell'elaborato, riguardanti la teoria degli automi;
- nel Capitolo 2 sono esposti il problema dell'iperminimizzazione di automi a stati finiti deterministici ed i diversi algoritmi proposti per la sua risoluzione;
- nel Capitolo 3 sono fornite delle possibili implementazioni degli algoritmi di iperminimizzazione presentati nel capitolo precedente;
- nel Capitolo 4 sono discussi i risultati sperimentali ottenuti dagli esperimenti condotti per la valutazione delle prestazioni degli algoritmi di iperminimizzazione nella pratica.

Ringraziamenti

Prima di tutto, voglio ringraziare il mio relatore, Prof. Giovanni Pighizzini, per la sua guida e il suo supporto durante questo progetto.

Voglio inoltre esprimere la mia gratitudine verso la mia famiglia, in particolare ai miei genitori, per tutto quello che hanno fatto affinchè potessi intraprendere questo percorso e durante il quale mi hanno sempre fornito il loro pieno supporto ed incoraggiamento. Ci tengo anche a ringraziare la mia fidanzata, Beatrice, per essere sempre stata al mio fianco e per avermi stimolato a proseguire e ad impegnarmi al massimo.

Desidero infine manifestare la mia più sincera riconoscenza agli amici che mi hanno accompagnato durante questo percorso accademico. Un primo ringraziamento va ai miei colleghi universitari, la cui collaborazione e il cui supporto sono stati preziosi per il mio sviluppo scolastico e professionale. Allo stesso tempo, vorrei estendere i miei ringraziamenti agli amici di lunga data con i quali, pur non condividendo direttamente il mio percorso universitario, ho passato momenti di svago e leggerezza che mi hanno aiutato a mantenere l'equilibrio e la motivazione.

A tutti loro va il mio più sentito ringraziamento per aver contribuito, ciascuno a suo modo, al compimento di questo importante capitolo della mia vita.

24 ottobre 2024

Indice

Capitolo 1

Nozioni preliminari

La teoria degli automi è uno dei principali e più antichi campi dell'informatica teorica. In questo capitolo ne vengono presentati i concetti fondamentali. Per un'esposizione più completa il lettore è rimandato a [HMU06].

1.1 Alfabeti, parole e linguaggi

Un *alfabeto*, generalmente indicato con la lettera Σ, è un insieme finito di simboli, entità astratte non definite formalmente cui esempi possono essere lettere e cifre.

Una *parola* (o *stringa*) è una sequenza finita di simboli giustapposti, in particolare, una parola su un alfabeto Σ è una sequenza finita di simboli appartenenti a Σ.

La *lunghezza di una parola* w, indicata con $|w|$, è il numero di simboli che la compongono. Caso particolare è la parola vuota, a cui per convenzione si fa riferimento con la lettera ε, composta da zero simboli ($|\varepsilon| = 0$).

Un possibile esempio di alfabeto è $\Sigma = \{0, 1\}$, mentre una possibile parola su Σ è $w = 01$, dove $|w| = 2$.

Si è in grado a questo punto di introdurre il concetto di linguaggio.

Definizione 1.1. *Un* linguaggio L *su un alfabeto* Σ *è un insieme di parole su* Σ*, ovvero, un insieme di parole formate dalla giustappozione di simboli appartenenti a* Σ*.*

Due esempi particolari possono essere il linguaggio vuoto $L_\varnothing = \varnothing$ ed il linguaggio composto solo dalla parola vuota $L_\varepsilon = \{\varepsilon\}$, puntualizzando il fatto che $L_\varnothing \neq L_\varepsilon$, infatti $||L_\varnothing|| = 0$ mentre $||L_\varepsilon|| = 1$.

Convenzionalmente, fissato un alfabeto Σ, viene indicato con Σ^* il linguaggio composto da tutte le parole su Σ, compresa la parola vuota.

Sia $\Sigma = \{a, b\}$, allora $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$.

1.2 Automi a stati finiti

In generale, un automa è un modello matematico di una macchina che esegue delle operazioni predefinite.

Una delle principali tipologie di automi è quella degli automi a stati finiti (FSA). Questi, sono sistemi con un numero finito di input, che possono trovarsi in un numero finito di configurazioni interne chiamate stati. In questa categoria rientrano gli automi a stati finiti deterministici (DFA) e gli automi a stati finiti non deterministici (NFA).

Definizione 1.2. *Un* automa a stati finiti deterministico *è una quintupla* $A = (Q, \Sigma, \delta, q_I, F)$ *dove* Q *denota un insieme finito di stati,* Σ *un alfabeto,* $\delta : Q \times \Sigma \to Q$ *la funzione di transizione,* $q_I \in Q$ *lo stato iniziale e* $F \subseteq Q$ *l'insieme degli stati finali.*

La funzione di transizione δ può essere estesa per essere applicata a parole. Si definisce dunque $\hat{\delta} : Q \times \Sigma^* \to Q$ dove $\hat{\delta}(q, w)$ è lo stato in cui l'automa si trova, partendo dallo stato q, dopo aver letto la parola w. Più formalmente, $\hat{\delta}(q, \varepsilon) = q$ e $\hat{\delta}(q, w\sigma) = \delta(\hat{\delta}(q, w), \sigma)$ per ogni $w \in \Sigma^*$ e $\sigma \in \Sigma$.

Una comune rappresentazione della funzione di transizione è quella in forma tabellare, chiamata tabella di transizione, nella quale le righe corrispondono agli stati, le colonne ai possibili simboli in ingresso ed il generico elemento di riga q e colonna σ allo stato $\delta(q, \sigma)$. Lo stato iniziale e gli stati finali sono evidenziati rispettivamente con $\to$ e $*$.

Generalmente, gli automi a stati finiti vengono rappresentanti graficamente attraverso grafi orientati. In tale rappresentazione gli archi corrispondono alle transizioni e i nodi agli stati, tra i quali, lo stato iniziale è evidenziato con un arco in ingresso privo di origine mentre gli stati finali sono evidenziati con un doppio cerchio.

Un esempio di automa a stati finiti deterministico è mostrato in Figura 1, in tale caso, $Q = \{A, B, C, D, E, F, G, H\}$, $\Sigma = \{a, b\}$, $q_I = A$ e $F = \{E, G, H\}$, mentre la funzione di transizione δ è mostrata in Tabella 1.

Definizione 1.3. *Sia* $A = (Q, \Sigma, \delta, q_I, F)$ *un automa a stati finiti deterministico, si definisce il* linguaggio riconosciuto *da* A *come* $L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_I, w) \in F\}$, *ovvero, come il linguaggio composto da tutte le parole* $w \in \Sigma^*$ *che portano dallo stato iniziale* q_I *ad uno stato* $q \in F$.

Gli automi a stati finiti non deterministici sono simili ai DFA, la differenza principale risiede nella funzione di transizione che è definita come $\delta : Q \times \Sigma \to 2^Q$ che ne cambia di conseguenza anche il modo di elaborare gli input. In questo modo, partendo dallo stato iniziale e leggendo una parola w è possibile raggiungere più stati. La parola è accettata se almeno uno degli stati raggiunti è finale.

Data la non rilevanza degli NFA per il proseguo della trattazione, maggiori dettagli, per i quali il lettore è nuovamente rimandato a [HMU06], sono omessi. Si ritiene tuttavia importante puntualizzare come gli automi a stati finiti non deterministici siano equivalenti a quelli deterministici, ovvero, per ogni NFA esiste un DFA che riconosce lo stesso linguaggio e viceversa.

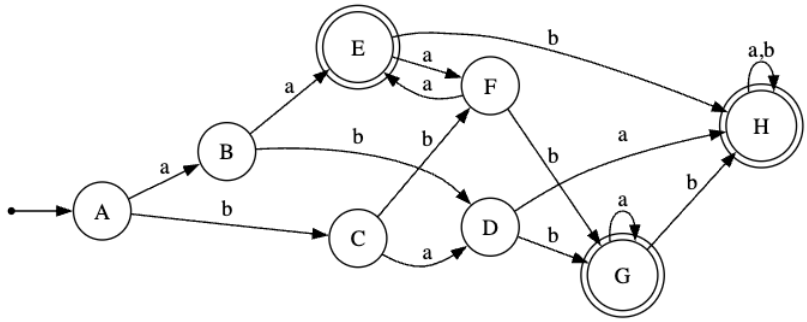


Figura 1: Un esempio di automa a stati finiti deterministico.

	a	b
$\to A$	B	C
B	E	D
C	D	F
D	H	G
$*E$	F	H
F	E	G
$*G$	G	H
$*H$	H	H

Tabella 1: Tabella di transizione dell'automa in Figura 1.

1.3 Minimizzazione di DFA

Uno dei problemi di maggiore interesse nel campo della teoria degli automi è quello riguardante la minimizzazione di automi a stati finiti deterministici. Il problema consiste, dato un automa $A = (Q, \Sigma, \delta, q_I, F)$, nel trovare un automa $A' = (Q', \Sigma, \delta', q'_I, F')$

tale che $L(A') = L(A)$ e $||Q'||$ sia minimo.

Il concetto alla base della minimizzazione è quello di equivalenza tra stati:

Definizione 1.4. *Due stati q_A e q_B sono detti* equivalenti*, denotato con $q_A \equiv q_B$, se $\forall w \in \Sigma^*$, $\hat{\delta}(q_A, w) \in F$ se e solo se $\hat{\delta}(q_B, w) \in F$. Specularmente, due stati q_A e q_B non equivalenti sono detti distinguibili.*

Teorema 1.1. *"$\equiv$" è una relazione di equivalenza, ovvero, è riflessiva, simmetrica e transitiva.*

L'importanza dell'equivalenza tra stati è evidente nel momento in cui si considera che due stati equivalenti, possono essere sostituiti da un singolo stato che si comporti come entrambi. La diretta conseguenza risiede nel fatto che, dato un automa di partenza $A = (Q, \Sigma, \delta, q_I, F)$, siano $q_A, q_B \in Q$ tali che $q_A \equiv q_B$, è possibile costruire un automa $A' = (Q', \Sigma, \delta', q'_I, F')$, con $||Q'|| < ||Q||$ e $L(A') = L(A)$, facendo collassare gli stati q_A e q_B in un singolo stato q', ovvero, definendo $Q' = Q \setminus \{q_A, q_B\} \cup \{q'\}$ e la funzione di transizione δ' in modo che tutte le transizioni in ingresso a q_A e q_B vengano reindirizzate a q' e che $\delta'(q', \sigma) = \delta(q_A, \sigma)$ per ogni $\sigma \in \Sigma$, prestando particolare attenzione, se necessario, anche alla conseguente ridefinizione dello stato iniziale e degli stati finali in A'.

Il Teorema 1.1 permette di estendere il risultato sopra riportato ad un numero arbitrario di stati equivalenti. Più formalmente, siano $q_1, q_2, \ldots, q_n \in Q$ con $n \in \mathbb{N}$ tali che $q_1 \equiv q_2 \equiv \cdots \equiv q_n$, allora è possibile collassare gli stati $q_1, q_2, \ldots, q_n$ in un singolo stato q' mantenendo il linguaggio riconosciuto dall'automa inalterato.

Grazie all'idea di equivalenza tra stati, è possibile definire l'equivalenza tra automi: siano $A_1 = (Q_1, \Sigma, \delta_1, q_{I_1}, F_1)$ e $A_2 = (Q_2, \Sigma, \delta_2, q_{I_2}, F_2)$ due DFA, A_1 è detto *equivalente* ad A_2, denotato con $A_1 \equiv A_2$, se $q_{I_1} \equiv q_{I_2}$. È banale osservare che, se $A_1 \equiv A_2$, questi accettano gli stessi input, pertanto $L(A_1) = L(A_2)$.

Ricordando inoltre che una classe di equivalenza è un sottoinsieme della partizione P di un insieme S determinata da una relazione di equivalenza R su S [Ros18], si introduce il concetto di *insieme quoziente*, denotato da S/R, come l'insieme delle classi di equivalenza di R su S.

Teorema 1.2. *Sia $A = (Q, \Sigma, \delta, q_I, F)$ un automa a stati finiti deterministico, A è detto* minimo *(o* minimizzato*) se non esiste un automa $A' = (Q', \Sigma, \delta', q'_I, F')$ tale che $A \equiv A'$ e $||Q'|| < ||Q||$.*

Queste nozioni permettono di fornire quella che è la soluzione al problema della minimizzazione di automi a stati finiti deterministici.

Dato un automa $A = (Q, \Sigma, \delta, q_I, F)$, l'automa $A' = (Q', \Sigma, \delta', q'_I, F')$ ottenuto, dapprima rimuovendo gli stati irraggiungibili in A, e successivamente $\forall Q_i \in Q/\equiv$ collassando gli stati in Q_i in un singolo stato $q_i \in Q_i$ scelto arbitrariamente, è equivalente ad A e $||Q'||$ è minimo. A' corrisponde quindi all'automa A minimizzato.

Teorema 1.3. *L'automa minimo A' ottenuto come descritto sopra è unico, questo significa che a meno di isomorfismi, non esiste un automa A'' tale che $A'' \equiv A'$ e $A'' \neq A'$.*

Prendendo come esempio l'automa in Figura 1, si procede alla sua minimizzazione seguendo la procedura sopra descritta. L'automa risulta essere privo di stati irraggiungibili, pertanto si procede da subito con il calcolo del partizionamento di Q in classi di equivalenza, che risulta essere $Q/\equiv = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{G, H\}\}$. Si passa quindi al collasso degli stati, in particolare, gli unici stati equivalenti risultano essere G e H che vengono sostituiti da un unico stato G, ottenendo l'automa in Figura 2.

In ultima istanza, si sottolinea come sia ben noto un algoritmo per la minimizzazione di automi a stati finiti deterministici, l'algoritmo di Hopcroft [Hop71], che permette, partendo da un DFA in ingresso, di ottenere l'automa minimo in tempo $\mathcal{O}(n \log n)$.

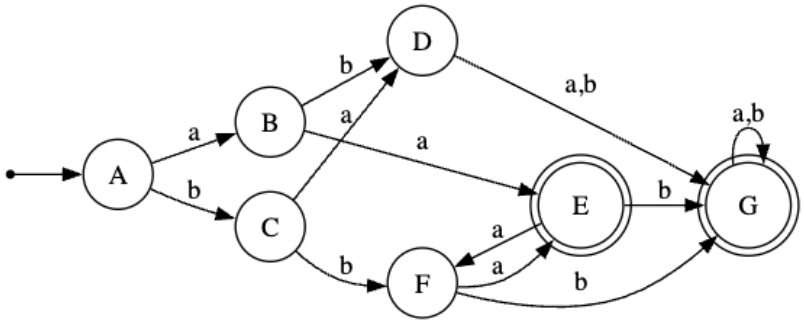


Figura 2: L'automa in Figura 1 minimizzato.

Capitolo 2

Iperminimizzazione di DFA

2.1 Introduzione

Recentemente è stato introdotto un nuovo problema, affine a quello della minimizzazione, chiamato iperminimizzazione di automi a stati finiti deterministici [BGS09].

Il problema dell'iperminimizzazione può essere visto come un'estensione del problema della minimizzazione: dove la minimizzazione permette di ridurre al minimo il numero di stati di un DFA mantenendo inalterato il linguaggio riconosciuto, l'imperminimizzazione prevede un'ulteriore riduzione al minimo numero di stati mantenendo invece finita la differenza simmetrica tra il linguaggio riconosciuto dall'automa originale e quello riconosciuto dall'automa ottenuto.

Definizione 2.1. *Due linguaggi L_1 e L_2 sono detti* finitamente equivalenti *(o* f-equivalenti*), denotato con $L_1 \sim L_2$, se $L_1 \Delta L_2$ è un insieme finito, dove $L_1 \Delta L_2$ denota la differenza simmetrica tra i linguaggi.*

Più formalmente dunque, dato un DFA $A = (Q, \Sigma, \delta, q_I, F)$, l'iperminimizzazione consiste nel trovare un automa $A' = (Q', \Sigma, \delta', q'_I, F')$ tale che $L(A') \sim L(A)$ e $||Q'||$ sia minimo.

2.2 Classi di quasi-equivalenza

Analogamente al concetto classico di minimizzazione, in cui è di centrale rilevanza l'equivalenza tra stati, è necessario introdurre la nozione di quasi-equivalenza tra stati per poter affrontare il problema dell'iperminimizzazione.

Definizione 2.2. *Due stati q_A e q_B sono detti* quasi-equivalenti*, denotato con $q_A \sim q_B$, se $\exists k \in \mathbb{N}$, con $k \geq 0$, tale che $\forall w \in \Sigma^*$ di lunghezza $|w| \geq k$, $\hat{\delta}(q_A, w) \in F$ se e solo se $\hat{\delta}(q_B, w) \in F$.*

Teorema 2.1. *"$\sim$" è una relazione di equivalenza, ovvero, è riflessiva, simmetrica e transitiva.*

Si introduce inoltre un importante risultato relativo alla quasi-equivalenza tra stati:

Lemma 2.1. $q_A \sim q_B \Leftrightarrow \forall \sigma \in \Sigma, \delta(q_A, \sigma) \sim \delta(q_B, \sigma)$

Risulta evidente dunque come, grazie al Teorema 2.1, sia possibile partizionare l'insieme degli stati Q di un automa a stati finiti deterministico $A = (Q, \Sigma, \delta, q_I, F)$ in classi di quasi equivalenza e costruire dunque l'insieme quoziente $Q/\sim$.

Nell'automa $A = (Q, \Sigma, \delta, q_I, F)$ riportato in Figura 2, ad esempio, partizionando l'insieme degli stati Q in classi di quasi-equivalenza, si ottiene l'insieme quoziente $Q/\sim = \{\{A\}, \{C\}, \{D, G\}, \{F, B\}, \{E\}\}$.

Equivalentemente a quanto visto per l'equivalenza tra stati, è possibile estendere il concetto di quasi-equivalenza tra stati agli automi: due automi $A_1 = (Q_1, \Sigma, \delta_1, q_{I_1}, F_1)$ e $A_2 = (Q_2, \Sigma, \delta_2, q_{I_2}, F_2)$ sono detti *quasi-equivalenti*, denotato con $A_1 \sim A_2$, se $q_{I_1} \sim q_{I_2}$. Si osservi come, se $A_1 \sim A_2$, allora $L(A_1) \sim L(A_2)$.

Grazie al concetto di quasi-equivalenza tra automi, è ora possibile fornire la definizione formale di automa a stati finiti deterministico iperminimo.

Definizione 2.3. *Sia $A = (Q, \Sigma, \delta, q_I, F)$ un automa a stati finiti deterministico, A è detto* iperminimo *(o* iperminimizzato*) se non esiste un automa $A' = (Q', \Sigma, \delta', q'_I, F')$ tale che $A \sim A'$ e $||Q'|| < ||Q||$.*

2.3 Stati preambolo e kernel

Dato un automa a stati finiti deterministico $A = (Q, \Sigma, \delta, q_I, F)$, è importante, come ulteriore passo verso la risoluzione del problema dell'iperminimizzazione, distinguere le diverse tipologie di stati che possono essere presenti in A.

È possibile partizionare l'insieme degli stati Q di A in tre parti: l'insieme degli stati irraggiungibili $U(A)$, l'insieme degli stati preambolo $P(A)$ e l'insieme degli stati kernel $K(A)$.

Sorvolando sulla definizione formale degli stati irraggiungibili, ben nota e non di centrale importanza per la trattazione, viene fornita di seguito la definizione degli stati preambolo e kernel.

Definizione 2.4. *Uno stato $q \in Q$ è detto* stato preambolo *e dunque $q \in P(A)$, se esiste almeno una parola in ingresso $w \in \Sigma^*$ tale che $\hat{\delta}(q_I, w) = q$, ma il numero di tali stringhe è finito. In altri termini, q è raggiungibile solo da un numero finito di input a partire dallo stato iniziale.*

Definizione 2.5. *Uno stato $q \in Q$ è detto* stato kernel *e dunque $q \in K(A)$, se esistono infinite parole, tutte differenti tra loro, in ingresso $w \in \Sigma^*$ tali che $\hat{\delta}(q_I, w) = q$. In altri termini, q è raggiungibile da un numero infinito di input a partire dallo stato iniziale.*

Lemma 2.2. *Uno stato $q_A \in Q$ è in $K(A)$ se e solo se può essere raggiunto dallo stato iniziale q_I attraverso un cammino passante per uno stato $q_B \in Q$ tale che esiste un ciclo in A che inizia e termina in q_B.*

Nell'automa A riportato in Figura 2, ad esempio, è possibile osservare come gli stati A, B, C e D siano stati preambolo mentre gli stati F, G ed E siano stati kernel, o ancora, come $P(A) = \{A, B, C, D\}$ e $K(A) = \{F, G, E\}$.

Si sottolinea un'importante differenza nel parallelismo tra automa minimo ed iperminimo: l'automa iperminimo non è necessariamente unico. Si possono trovare esempi in cui un automa minimo A' con n stati può essere sostituito da diversi automi iperminimi differenti, tutti con il medesimo numero di stati $m \leq n$, ciascuno dei quali accetta un diverso linguaggio L f-equivalente a $L(A')$. Tali automi possono tuttavia differire solo per le transizioni in uscita da uno stato preambolo e in ingresso ad uno stato kernel, per i valori accettati dagli stati preambolo, o per gli stati iniziali. In particolare, i kernel di due automi iperminimi quasi-equivalenti sono isomorfi secondo la definizione classica mentre i preamboli sono isomorfi, eccetto per i valori accettati.

Definizione 2.6 (Isomorfismo tra kernel)**.** *Siano $A_1 = (Q_1, \Sigma, \delta_1, q_{I_1}, F_1)$ e $A_2 = (Q_2, \Sigma, \delta_2, q_{I_2}, F_2)$ due DFA, A_1 e A_2 hanno kernel isomorfi, denotato $A_1 \sim_K A_2$, se esiste una corrispondenza biunivoca $h : K(A_1) \to K(A_2)$ tale che:*

(a) $\forall q \in K(A_1)$ e $\forall \sigma \in \Sigma$, $h(\delta_1(q, \sigma)) = \delta_2(h(q), \sigma)$;

(b) $\forall q \in K(A_1)$, $h(q) \in F_2$ se e solo se $q \in F_1$;

Teorema 2.2. *Siano A_1 e A_2 due automi minimi quasi-equivalenti, allora i loro kernel sono isomorfi ($A_1 \sim_K A_2$).*

Definizione 2.7 (Isomorfismo tra preamboli)**.** *Siano $A_1 = (Q_1, \Sigma, \delta_1, q_{I_1}, F_1)$ e $A_2 = (Q_2, \Sigma, \delta_2, q_{I_2}, F_2)$ due DFA, A_1 e A_2 hanno preamboli isomorfi, denotato $A_1 \sim_P A_2$, se esiste una corrispondenza biunivoca $h : P(A_1) \to P(A_2)$ tale che $\forall q_A, q_B \in P(A_1)$ e $\forall \sigma \in \Sigma$ tali che $\delta_1(q_A, \sigma) = q_B$ si ha che $\delta_2(h(q_A), \sigma) = h(q_B)$.*

Teorema 2.3. *Siano A_1 e A_2 due automi iperminimi quasi-equivalenti, allora i loro preamboli sono isomorfi ($A_1 \sim_P A_2$).*

2.4 Strategia generale

Grazie alle nozioni presentate nelle Sezioni 2.2 e 2.3, è possibile presentare una strategia generale per l'iperminimizzazione di automi a stati finiti deterministici.

Dato un automa $A = (Q, \Sigma, \delta, q_I, F)$, la procedura generale per l'iperminimizzazione consiste nel:

1. ottenere l'automa minimo $A' = (Q', \Sigma, \delta', q'_I, F')$ minimizzando classicamente A;
2. identificare l'insieme degli stati preambolo $P(A')$ e degli stati kernel $K(A')$;
3. determinare le classi di quasi-equivalenza su Q' e costruire l'insieme $Q'/\sim$;
4. $\forall Q_i \in Q'/\sim$ tale che $||Q_i|| > 1$ ed $\exists q_{i_1} \in Q_i$ tale che $q_{i_1} \in P(A')$:
   (a) se $\exists q_{i_2} \in Q_i$ tale che $q_{i_2} \in K(A')$, $\forall q \in Q_i \cap P(A')$ si collassa q in q_{i_2},
   (b) altrimenti, tutti gli stati in $Q_i \setminus \{q_{i_1}\}$ sono collassati in q_{i_1}.

L'automa A'' così ottenuto è l'automa iperminimo.

Teorema 2.4 (Caratterizzazione di un DFA iperminimo)**.** *Un automa a stati finiti deterministico $A = (Q, \Sigma, \delta, q_I, F)$ è iperminimo se e solo se:*

(a) A è minimo

(b) non esiste una coppia di stati $q_A, q_B \in Q$ tali che $q_A \neq q_B$, $q_A \sim q_B$ e almeno uno dei due sia in $P(A)$

Si osservi che il primo passo della procedura, ovvero la minimizzazione dell'automa, permette di garantire che l'automa minimo ottenuto A', sia privo di stati irraggiungibili ($U(A') = \emptyset$) e che quindi nei passaggi successivi della strategia l'insieme degli stati possa essere partizionato esclusivamente in due parti, gli stati preambolo e gli stati kernel ($Q' = P(A') \cup K(A')$).

Prendendo ancora una volta come esempio l'automa $A = (Q, \Sigma, \delta, q_I, F)$ in Figura 1, si procede alla sua iperminimizzazione seguendo la strategia generale appena descritta. La minimizzazione dell'automa, discussa in precedenza, porta all'automa minimo $A' = (Q', \Sigma, \delta', q'_I, F')$ in Figura 2. Ricordando ora che $P(A') = \{A, B, C, D\}$, $K(A') = \{F, G, E\}$ e che $Q'/\sim = \{\{A\}, \{C\}, \{D, G\}, \{F, B\}, \{E\}\}$, si procede con il collasso degli stati:

– essendo $D \sim G$, $D \in P(A')$ e $G \in K(A')$, si collassa D in G

– essendo $F \sim B$, $B \in P(A')$ e $F \in K(A')$, si collassa B in F

ottenendo l'automa A'' iperminimo in Figura 3.

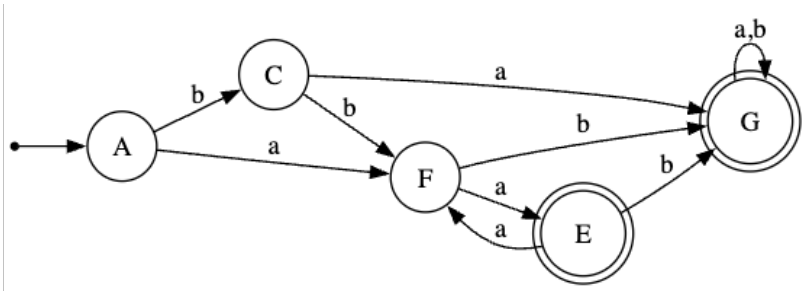


Figura 3: L'automa in Figura 2 iperminimizzato.

2.5 Algoritmi

Nel corso degli anni sono emersi diversi algoritmi per l'iperminimizzazione di automi a stati finiti deterministici, ciascuno della quali, cambiando e/o raffinando le metodologie di identificazione degli stati preambolo, degli stati kernel e delle classi di quasi-equivalenza, e come queste siano manipolate, ha portato a differenti risultati, migliorando la complessità temporale dei precendenti. Tutti gli algoritmi presentati di seguito condividono lo schema ad alto livello riportato nella Sezione 2.4 per poi specializzarsi, in maniera differente, sulle strategie utilizzate per portare a termine le singole operazioni.

Si porta all'attenzione del lettore come, da qui in seguito, detto $A = (Q, \Sigma, \delta, q_I, F)$ l'automa in ingresso ad un algoritmo di iperminimizzazione, si utilizzerà n per fare riferimento al numero di stati $||Q||$ ed m per il numero di simboli in ingresso $||\Sigma||$.

2.5.1 Algoritmo di Badr, Geffert e Shipman

L'algoritmo di Badr, Geffert e Shipman [BGS09] è stato il primo algoritmo proposto per l'iperminimizzazione in grado, dato un automa $A = (Q, \Sigma, \delta, q_I, F)$, di ottenere l'automa iperminimo $A'' = (Q'', \Sigma, \delta'', q_I'', F'')$ in tempo polinomiale.

Come primo passo, l'algoritmo minimizza l'automa in ingresso A ottenendo l'automa minimo $A' = (Q', \Sigma, \delta', q_I', F')$ attraverso l'algoritmo di Hopcroft.

Successivamente, viene costruita una matrice E di dimensione $||Q'|| \times ||Q'||$, utilizzata nel passo successivo per semplificare la fase di identificazione degli insiemi $P(A')$ e

$K(A')$, tale che:

$$e_{i,j} = \begin{cases} 1 & \text{se } \exists w \in \Sigma^* \setminus \{\varepsilon\} \text{ tale che } \hat{\delta}(q_i, w) = q_j \\ 0 & \text{altrimenti} \end{cases}$$

In altri termini, poiché che l'algoritmo verifica l'esistenza di un cammino di lunghezza maggiore di 1 tra ogni coppia di stati $q_i, q_j \in Q'$, trattando l'automa come un grafo, il problema può essere ricondotto al calcolo della chiusura transitiva di quest'ultimo. Sono noti diversi algoritmi per lo svolgimento di tale compito [DFI08]. Un possibile approccio classico potrebbe essere quello di utilizzare l'algoritmo di Floyd-Warshall, che richiede tempo $\mathcal{O}(n^3)$, tuttavia, in questo caso specifico, è possibile sfruttare una ricerca in ampiezza (BFS) partendo da ogni stato $q_i \in Q'$ per ottenere gli stati raggiungibili da q_i e popolare la corrispondente riga nella matrice E, ottenendo così un tempo di esecuzione $\mathcal{O}(n^2 \cdot m)$.

Ricordando ancora una volta che grazie alla minimizzazione nell'automa A' non sono presenti stati irraggiungibili, è ora possibile distinguere in Q' gli stati kernel da quelli preambolo, si costruisce a questo scopo un vettore K di dimensione $||Q'||$ tale che $k_i = 1$ se e solo se $q_i \in K(A')$, altrimenti $k_i = 0$. Grazie al Lemma 2.2 la costruzione del vettore K si riduce nel determinare se per ogni stato $q_i \in Q'$, esista uno stato $q_j \in Q'$ tale che $e_{1,j} = e_{j,j} = e_{j,i} = 1$.

È banale dunque osservare come la costruzione del vettore K sia eseguita in tempo $\mathcal{O}(n)$ essendo che per ogni stato $q_i \in Q'$ la verifica di appartenenza a $K(A')$ richiede un tempo costante $\mathcal{O}(1)$.

L'algoritmo procede con la determinazione delle classi di quasi-equivalenza, a questo scopo, viene creato un vettore R di dimensione $||Q'||$ che conterrà in posizione i l'indice j della classe di quasi-equivalenza $Q_j \in Q/\sim$ a cui lo stato q_i appartiene.

Inizialmente, R viene inizializzato in modo tale che rappresenti la partizione di Q' composta esclusivamente da singoletti $Q' = \{q_1\} \cup \{q_2\} \cup \cdots \cup \{q_{||Q'||}\}$, ovvero, $\forall i \in \{1, \ldots, ||Q'||\}, r_i = i$. La costruzione avviene iterativamente, si cercano coppie di stati $q_i \in Q_i$ e $q_j \in Q_j$, con $i \neq j$, tali che $\forall \sigma \in \Sigma$ si abbia che $\delta'(q_i, \sigma)$ e $\delta'(q_j, \sigma)$ appartengano alla stessa classe di quasi-equivalenza $Q_l \in Q'/\sim$. Se tale coppia di stati viene trovata, seguendo quanto riportato nel Lemma 2.1, significa che $q_i \sim q_j$ pertanto si uniscono le classi Q_i e Q_j in un'unica classe e si aggiorna il vettore R di conseguenza. Il processo termina quando non vengono trovate nuove coppie di stati che soddisfino i requisiti, il che genera al più $||Q'|| - 1$ iterazioni.

Risulta immediato stabilire che il tempo richiesto per l'inizializzazione del vettore R sia lineare rispetto al numero di stati dell'automa e che pertanto il tempo necessario a

tale operazione sia $\mathcal{O}(n)$. Necessita invece un'analisi più approfondita il calcolo della complessità temporale relativa alla costruzione di R: scandire le coppie di stati in Q' richiede un tempo $\mathcal{O}(n^2)$, mentre la verifica della quasi-equivalenza per ciascuna coppia richiede un tempo $\mathcal{O}(m)$, essendo inoltre il processo iterato al più $||Q'|| - 1$ volte, la complessità temporale totale risulta essere $\mathcal{O}(n^3 \cdot m)$.

Infine, per ottenere l'automa A'' iperminimo, considerando l'insieme delle classi di quasi-equivalenza rappresentato dal vettore R, si esaminano iterativamente tutti i $Q_i \in Q'/\sim$ collassando ove possibile gli stati seguendo quanto descritto nel punto 4 della strategia generale riportata nella Sezione 2.4.

Il tempo complessivo richiesto dall'algoritmo per la costruzione dell'automa iperminimo A'' risulta quindi essere $\mathcal{O}(n^3 \cdot m)$.

Supponendo, a titolo di esempio, di stare iperminimizzando l'automa A' in Figura 2, la matrice E costruita dall'algoritmo risulterebbe essere:

$$E = \begin{array}{c} \begin{matrix} A & B & C & D & E & F & G \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \\ G \end{matrix} \end{array}$$

mentre i vettori K dei kernel e R delle classi di quasi-equivalenza risulterebbero essere:

$$K = \underset{A}{\begin{bmatrix}0\end{bmatrix}} \underset{B}{\begin{bmatrix}0\end{bmatrix}} \underset{C}{\begin{bmatrix}0\end{bmatrix}} \underset{D}{\begin{bmatrix}0\end{bmatrix}} \underset{E}{\begin{bmatrix}1\end{bmatrix}} \underset{F}{\begin{bmatrix}1\end{bmatrix}} \underset{G}{\begin{bmatrix}1\end{bmatrix}}, \quad R = \underset{A}{\begin{bmatrix}A\end{bmatrix}} \underset{B}{\begin{bmatrix}F\end{bmatrix}} \underset{C}{\begin{bmatrix}C\end{bmatrix}} \underset{D}{\begin{bmatrix}D\end{bmatrix}} \underset{E}{\begin{bmatrix}E\end{bmatrix}} \underset{F}{\begin{bmatrix}F\end{bmatrix}} \underset{G}{\begin{bmatrix}D\end{bmatrix}}$$

2.5.2 Algoritmo di Badr

L'algoritmo proposto da Andrew Badr [Bad08] migliora la complessità temporale dell'algoritmo di Badr, Geffert e Shipman cambiando la strategia utilizzata per identificare le classi di quasi-equivalenza.

Equivalentemente all'algoritmo precedente, l'algoritmo di Badr come prima operazione minimizza l'automa in ingresso A ottenendo l'automa minimo $A' = (Q', \Sigma, \delta', q'_I, F')$.

Il passo successivo prevede di identificare le classi di quasi-equivalenza. A questo scopo, si introduce una nuova operazione tra automi a stati finiti deterministici e un importante risultato ad essa associato.

Definizione 2.8. *Siano* $A_1 = (Q_1, \Sigma, \delta_1, q_{I_1}, F_1)$ *e* $A_2 = (Q_2, \Sigma, \delta_2, q_{I_2}, F_2)$ *due DFA, si definisce lo* XOR cross product *tra* A_1 *e* A_2*, denotato con* $A_1 \otimes A_2$*, come l'automa* $A^{\otimes} = (Q^{\otimes}, \Sigma, \delta^{\otimes}, q_I^{\otimes}, F^{\otimes})$ *tale che:*

1. $Q^{\otimes} = \{(q_1, q_2) : q_1 \in Q_1 \wedge q_2 \in Q_2\}$
2. $\forall q_1 \in Q_1, \forall q_2 \in Q_2, \forall \sigma \in \Sigma : \delta^{\otimes}((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$
3. $q_I^{\otimes} = (q_{I_1}, q_{I_2})$
4. $F^{\otimes} = \{(q_1, q_2) : (q_1 \in F_1) \otimes (q_2 \in F_2)\}$*, con* $\otimes$ *operazione di XOR*

Un esempio di XOR cross product è quello tra l'automa in Figura 2 e se stesso, il cui risultato è riportato in Figura 4 e di cui, per una maggiore chiarezza, si riporta la tabella di transizione in Tabella 2.

Definizione 2.9. *Sia* $A = (Q, \Sigma, \delta, q_I, F)$ *un automa a stati finiti deterministico, si estende la nozione di linguaggio riconosciuto da* A *ad un generico stato* $q \in Q$*, denotato con* $L(q)$ *e definito* linguaggio indotto *da* q*, come l'insieme* $L(q) = \{w \in \Sigma^* : \delta(q, w) \in F\}$.

Teorema 2.5. *Siano* $A = (Q, \Sigma, \delta, q_I, F)$ *un automa a stati finiti deterministico minimo e* $A^{\otimes} = (Q^{\otimes}, \Sigma, \delta^{\otimes}, q_I^{\otimes}, F^{\otimes})$ *l'automa risultante dallo XOR cross product di* A *con se stesso, allora:* $\forall w \in \Sigma^*$ *e* $\forall (q_A, q_B) \in Q^{\otimes}$ *si ha* $w \in L((q_A, q_B)) \Leftrightarrow \delta^{\otimes}((q_A, q_B), w) \in F^{\otimes} \Leftrightarrow (\delta(q_A, w) \in F) \otimes (\delta(q_B, w) \in F)$*. In altri termini, il linguaggio* $L((q_A, q_B))$ *di ogni stato nel contesto* $A^{\otimes}$*, è uguale al linguaggio* $L(q_A) \Delta L(q_B)$ *nel contesto* A.

Corollario 2.5.1. *Siccome* A *è minimo e dunque non esistono due stati* $q_A, q_B \in Q$*, con* $q_A \neq q_B$*, tali che inducano il medesimo linguaggio, l'insieme degli stati* $S = \{(q, q) : q \in Q\}$ *è esattamente l'insieme degli stati in* $A^{\otimes}$ *che inducono* $L_{\emptyset}$.

Corollario 2.5.2. *Il linguaggio indotto da uno stato* $(q_A, q_B) \in Q^{\otimes}$ *è finito se e solo se* $q_A \sim q_B$*, pertanto, l'insieme degli stati in* $A^{\otimes}$ *che inducono un linguaggio finito è la relazione di quasi-equivalenza sugli stati in* A.

L'idea alla base dell'algoritmo di Badr è quella di utilizzare l'automa $A^{\otimes}$ prodotto dallo XOR cross product tra l'automa minimo A' e se stesso, la cui costruzione richiede tempo $\mathcal{O}(n^2)$, per computare $Q'/\sim$. In particolare, l'algoritmo utilizza il sottoinsieme $R \subset Q^{\otimes}$ di tutti gli stati che inducono un linguaggio finito in $A^{\otimes}$ per calcolare le classi di quasi-equivalenza su Q'.

A questo scopo si calcola l'insieme degli stati che inducono il linguaggio vuoto S in $A^{\otimes}$ ed il suo insieme complementare S^c. Si osservi come, dato un automa a stati finiti deterministico generico $M = (Q_M, \Sigma_M, \delta_M, q_{I_M}, F_M)$, l'insieme degli stati che inducono un linguaggio vuoto in M è per definizione $S_M = \{q \in Q_M \ t.c. \ \forall w \in \Sigma_M^*, \ \delta_M(q, w) \notin F_M\}$ che in $A^{\otimes}$, per il Corollario 2.5.1, corrisponde all'insieme $S =$

$\{(q,q) : q \in Q'\}$. È banale dunque osservare come la costruzione di S e S^c richiede un tempo $\mathcal{O}(n)$.

	a	b
$\rightarrow (A,A)$	(B,B)	(C,C)
(A,B)	(B,E)	(C,D)
(A,C)	(B,D)	(C,F)
(A,D)	(B,G)	(C,G)
(A,E)	(B,F)	(C,G)
(A,F)	(B,E)	(C,G)
(A,G)	(B,G)	(C,G)
(B,A)	(E,B)	(D,C)
(B,B)	(E,E)	(D,D)
(B,C)	(E,D)	(D,F)
(B,D)	(E,G)	(D,G)
(B,E)	(E,F)	(D,G)
(B,F)	(E,E)	(D,G)
(B,G)	(E,G)	(D,G)
(D,B)	(G,E)	(G,D)
(D,C)	(G,D)	(G,F)
(D,D)	(G,G)	(G,G)
(C,A)	(D,B)	(F,C)
(C,B)	(D,E)	(F,D)
(C,C)	(D,D)	(F,F)
(C,D)	(D,G)	(F,G)
(C,E)	(D,F)	(F,G)
(C,F)	(D,E)	(F,G)
(C,G)	(D,G)	(F,G)
(D,A)	(G,B)	(G,C)

	a	b
(D,E)	(G,F)	(G,G)
(D,F)	(G,E)	(G,G)
(D,G)	(G,G)	(G,G)
(E,A)	(F,B)	(G,C)
(E,B)	(F,E)	(G,D)
(E,C)	(F,D)	(G,F)
(E,D)	(F,G)	(G,G)
$*(E,E)$	(F,F)	(G,G)
(E,F)	(F,E)	(G,G)
$*(E,G)$	(F,G)	(G,G)
(F,A)	(E,B)	(G,C)
(F,B)	(E,E)	(G,D)
(F,C)	(E,D)	(G,F)
(F,D)	(E,G)	(G,G)
(F,E)	(E,F)	(G,G)
(F,F)	(E,E)	(G,G)
(F,G)	(E,G)	(G,G)
(G,A)	(G,B)	(G,C)
(G,B)	(G,E)	(G,D)
(G,C)	(G,D)	(G,F)
(G,D)	(G,G)	(G,G)
$*(G,E)$	(G,F)	(G,G)
(G,F)	(G,E)	(G,G)
$*(G,G)$	(G,G)	(G,G)

Tabella 2: Tabella di transizione dell'automa in Figura 4.

Una volta calcolati S e S^c, l'algoritmo procede istanziando per ogni stato $q \in Q^{\otimes}$ due insiemi $T_q^{\rightarrow} = \emptyset$ e $T_q^{\leftarrow} = \emptyset$ che conterranno rispettivamente gli stati raggiungibili da q e gli stati che possono raggiungere q. La costruzione di $T_q^{\rightarrow}$ e $T_q^{\leftarrow}$ avviene iterativamente: per ogni stato $q \in S^c$ e per ogni $\sigma \in \Sigma$, sia $q' = \delta^{\otimes}(q,\sigma)$, allora $T_q^{\rightarrow} = T_q^{\rightarrow} \cup \{(q',\sigma)\}$ e $T_{q'}^{\leftarrow} = T_{q'}^{\leftarrow} \cup \{(q,\sigma)\}$.

In seguito, viene costruito l'insieme R, inizializzato come $R = \emptyset$, iterativamente: fino a quando $S \neq \emptyset$, si estrae uno stato $q \in S$ e lo si aggiunge ad R, si rimuove q da S

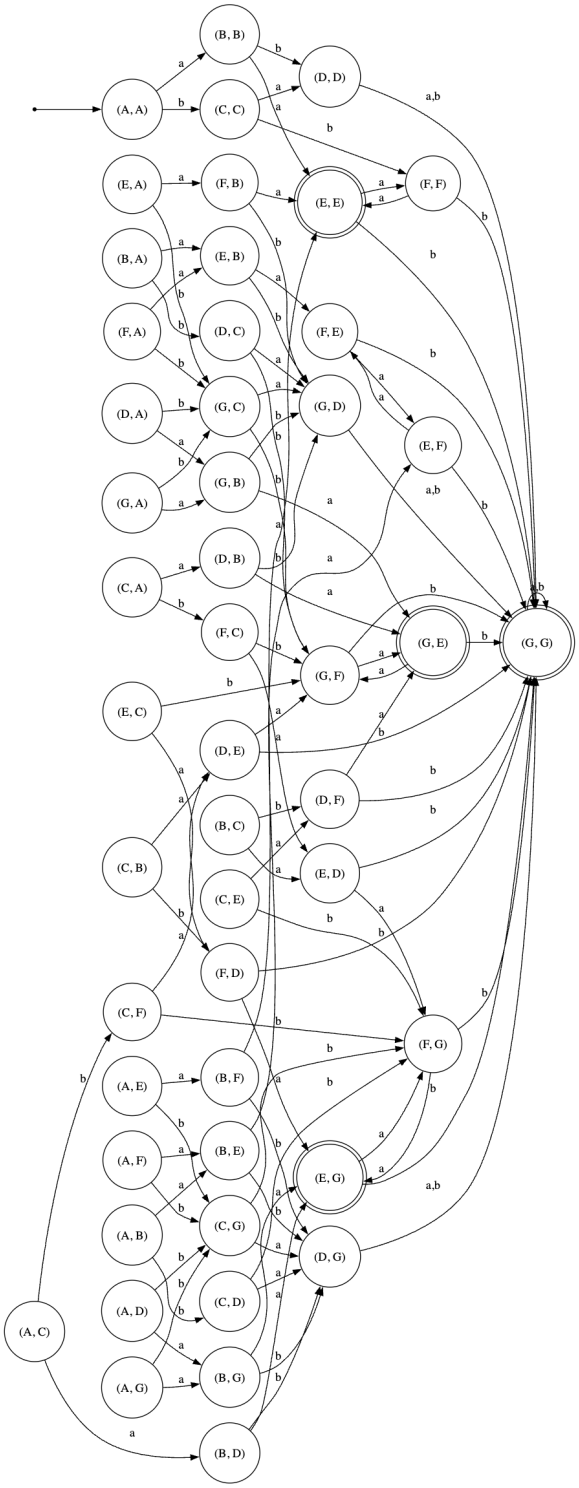


Figura 4: L'automa in Figura 2 in XOR cross product con se stesso.

e $\forall(q', \sigma) \in T_q^{\leftarrow}$ si rimuove (q, σ) da $T_{q'}^{\rightarrow}$ e se, in seguito a tale rimozione, $T_{q'}^{\rightarrow} = \emptyset$, si aggiunge q' a S.

Teorema 2.6. *L'algoritmo appena descritto aggiunge uno stato all'insieme R se e solo se induce un linguaggio finito in $A^{\otimes}$.*

Al termine di questo processo, grazie al risultato esposto nel Teorema 2.6, per la cui dimostrazione il lettore è rimandato a [Bad08], l'insieme R conterrà come da obiettivo tutti gli stati che inducono un linguaggio finito in $A^{\otimes}$ e sarà possibile costruire $Q'/\sim$.

Essendo il numero di transizioni lineare rispetto al numero di stati, il tempo necessario alla costruzione di $T_q^{\rightarrow}$ e $T_q^{\leftarrow}$ per tutti gli stati corrisponde complessivamente a $\mathcal{O}(||Q^{\otimes}||)$. Inoltre, l'aggiunta di ogni stato ad R richiede un tempo costante $\mathcal{O}(1)$ più un tempo trascurabile per ciascuno stato da cui si raggiunge quello corrente, pertanto la costruzione di R richiede anch'essa un tempo lineare $\mathcal{O}(||Q^{\otimes}||)$. Ricordando che $A^{\otimes}$ è lo XOR cross product di A' con se stesso, la complessità temporale di questo passaggio è $\mathcal{O}(n^2)$.

La costruzione di $Q'/\sim$ avviene ora semplicemente, utilizzando una struttura dati QuickFind U con bilanciamento [DFI08]. La struttura viene inizializzata per rappresentare la partizione di Q' composta esclusivamente da singoletti, quindi, per ogni stato $(q_A, q_B) \in R$, siano Q_i e Q_j rispettivamente i risultati delle operazioni di $find(q_A)$ e $find(q_B)$ in U, se $Q_i \neq Q_j$, allora si uniscono gli insiemi Q_i e Q_j tramite l'operazione $union(Q_i, Q_j)$ in U. Per il Corollario 2.5.2, al termine dell'operazione, U rappresenterà $Q'/\sim$.

Dato l'utilizzo di una struttura dati QuickFind con bilanciamento, il costo della singola operazione di $find$ è costante, mentre il costo della singola operazione di *union* è lineare, pertanto poiché per la costruzione di $Q'/\sim$ vengono eseguite al più $n-1$ operazioni di *union* e si itera su tutti gli stati in R, la complessità temporale di questo passaggio è $\mathcal{O}(n^2)$.

Il calcolo degli insieme $K(A')$ e $P(A')$ avviene in tempo $\mathcal{O}(n^2)$. L'idea è simile a quella dell'algoritmo di Badr, Geffert e Shipman, ma in questo caso ne viene cambiata la messa in pratica. Si inizializza l'insieme $K = \emptyset$ che conterrà gli stati kernel e si procede iterativamente: per ogni stato $q \in Q'$ si ottiene l'insieme degli stati raggiungibili da q in A' con un cammino di lunghezza maggiore di 1, se tale insieme contiene q allora si aggiunge q a K. Al termine del processo, K conterrà tutti gli stati kernel e l'insieme $P = P(A')$ sarà dato da $Q' \setminus K$.

Infine, l'automa iperminimo A'' viene costruito: per ogni classe di quasi-equivalenza $Q_i \in U$, si ottiene l'insieme P_{Q_i} degli stati preambolo in Q_i e l'insieme K_{Q_i} degli stati kernel in Q_i tramite intersezione di Q_i rispettivamente con P e K. Si procede dunque

al collasso di tutti gli stati in P_{Q_i} in uno stato $q_K \in K_{Q_i}$, se K_{Q_i} è non vuoto, o al collasso in $q_P \in P_{Q_i}$ degli stati in $P_{Q_i} \setminus \{q_P\}$ altrimenti.

Utilizzando ancora una volta come esempio l'automa A' in Figura 2, l'automa $A^{\otimes}$ risultante dallo XOR cross product tra A' e se stesso è stato mostrato essere quello descritto in Tabella 2. L'insieme S degli stati che inducono un linguaggio finito in $A^{\otimes}$ ne segue essere $S = \{(A,A),(B,B),(C,C),(D,D),(E,E),(F,F),(G,G)\}$, da cui l'insieme R degli stati che inducono un linguaggio finito in A' costruito dall'algoritmo è $R = S \cup \{(B,F),(F,B),(D,G),(G,D)\}$. L'insieme delle classi di quasi-equivalenza rappresentato da U risulta correttamente essere $\{\{A\},\{B,F\},\{C\},\{D,G\},\{E\}\}$.

2.5.3 Algoritmo di Holzer e Maletti

L'ultimo algoritmo presentato è quello proposto da Holzer e Maletti [HM10], questo migliora i risultati ottenuti in precedenza infatti, introducendo una nuova stategia per l'identificazione degli stati kernel e per la costruzione delle classi di equivalenza, riduce la complessità temporale a $\mathcal{O}(n \log n)$.

Siano $A = (Q, \Sigma, \delta, q_I, F)$ un automa a stati finiti deterministico e $q_A, q_B \in Q$ due stati, si definisce formalmente una funzione *merge* utilizzata in seguito dall'algoritmo per il collasso degli stati come $merge(\delta, q_I, q_A, q_B) = (\delta', q'_I)$ tale che $\forall q' \in Q$ e $\forall \sigma \in \Sigma$:

$$\delta'(q',\sigma) = \begin{cases} q_B & \text{se } \delta(q',\sigma) = q_A \\ \delta(q',\sigma) & \text{altrimenti} \end{cases} \text{ e } q'_I = \begin{cases} q_B & \text{se } q_I = q_A \\ q_I & \text{altrimenti} \end{cases}$$

L'algoritmo inizia seguendo quella che è la linea generale degli algoritmi precedenti: minimizza l'automa in ingresso ottenendo l'automa minimo $A' = (Q', \Sigma, \delta', q'_I, F')$.

Definizione 2.10. *Sia $A = (Q, \Sigma, \delta, q_I, F)$ un automa a stati finiti deterministico, uno stato $q \in Q$ è detto* stato centrale *se $\exists w \in \Sigma^* \setminus \{\varepsilon\}$ tale che $\delta(q, w) = q$.*

Il passo successivo prevede di identificare l'insieme $K(A')$ degli stati kernel in A'. L'idea alla base della strategia utilizzata dall'algoritmo è quella di identificare l'insieme C degli stati centrali in A' attraverso una versione dell'algoritmo di Tarjan [DFI08] per l'identificazione delle componenti fortemente connesse semplificata in quanto, essendo A' minimo, tutti gli stati in Q' sono raggiungibili da q'_I. Questo avviene semplicemente in quanto, per definizione, gli stati centrali sono parte di una componente fortemente connessa di almeno due stati o hanno un ciclo su loro stessi. Una volta costruito l'insieme C una Depth-First Search [DFI08] viene utilizzata per identificare tutti gli stati raggiungibili da uno stato centrale, questi stati sono gli stati kernel per il Lemma 2.2.

L'algoritmo passa ora ad occuparsi della costruzione delle classi di quasi-equivalenza. Per ottenere una migliore complessità temporale rispetto agli algoritmi precedenti, la strategia utilizzata prevede di evitare i confronti tra coppie, che migliora la complessità di un fattore $\mathcal{O}(n)$, e di collassare gli stati con una metodologia specifica che riduce un fattore $\mathcal{O}(n)$ a $\mathcal{O}(\log n)$.

Definizione 2.11. *Sia $A = (Q, \Sigma, \delta, q_I, F)$ un automa a stati finiti deterministico. Si definisce* vettore dei successori *di $q \in Q$ il vettore $V(q) = [\delta(q, \sigma) \mid \sigma \in \Sigma]$.*

Nel dettaglio, l'algoritmo mantiene un insieme I degli stati che devono essere processati e un insieme P degli stati che sono ancora utili, entrambi inizializzati uguali a Q', una tabella hash H inizialmente vuota che mappa vettori dei successori a stati, una partizione R di Q' che ne rappresenta in partenza la partizione in singoletti, una funzione $\delta_\sim = \delta'$ e uno stato $q_{I_\sim} = q'_I$. A questo punto, fin tanto che $I \neq \emptyset$, iterativamente viene rimosso uno stato q_A da I e ne viene calcolato $V(q_A)$, se a $V(q_A)$ non è associato alcun valore in H allora si assegna a $V(q_A)$ lo stato q_A in H e si prosegue con l'iterazione successiva, altrimenti, se è già associato uno stato q_B a $V(q_A)$ in H, si determina lo stato tra q_A e q_B la cui cardinalità della parte in R della quale è elemento è minore. Supponendo che tale stato sia q_A, questo viene viene rimosso da P trattandosi di uno stato inutile in quanto collassato in q_B nei prossimi passaggi, vengono aggiunti a I tutti gli stati $q \in P$ per i quali $\exists \sigma \in \Sigma$ tale che $\delta_\sim(q, \sigma) = q_A$, si effettua il $merge(\delta_\sim, q_{I_\sim}, q_A, q_B)$ aggiornando con i valori restituiti la funzione di transizione $\delta_\sim$ e lo stato $q_{I_\sim}$ e si modifica la partizione R unendo le parti in R a cui appartengono q_A e q_B. In ultima istanza, sempre in tale caso, si aggiorna il valore associato a $V(q_A)$ in H con q_B.

Teorema 2.7. *L'algoritmo appena descritto costruisce correttamente $Q'/\sim$.*

Teorema 2.8. *Gli algoritmi descritti per l'identificazione degli stati kernel e per la costruzione di $Q'/\sim$ possono essere implementati in modo tale che la loro computazione richieda rispettivamente tempo $\mathcal{O}(m)$ e $\mathcal{O}(m \log n)$.*

Le dimostrazioni dei Teoremi 2.7 e 2.8 risultano essere particolarmente lunghe e complesse, nonchè il fulcro del lavoro di Holzer e Maletti, pertanto sono volontariamente omesse nel presente elaborato ed il lettore è rimandato a [HM10].

Banalmente, l'ultimo passaggio dell'algoritmo consiste nel costruire l'automa iperminimo A'' a partire dai risultati ottenuti dai passaggi precedenti. Iterativamente, per ogni classe di equivalenza $Q_i \in R$, se l'intersezione tra Q_i e $K(A')$ è non vuota, si sceglie arbitrariamente uno stato q al suo interno, altrimenti si sceglie un qualsiasi stato q in Q_i, e $\forall q' \in Q_i \setminus K(A')$ si collassa q' in q e si aggiornano i valori di δ' e q'_I tramite $merge(\delta', q'_I, q, q')$ ottenendo l'automa iperminimo A''.

Ipotizzando di eseguire l'algoritmo sull'automa A' in Figura 2, l'insieme C degli stati centrali risulta essere $C = \{F, G, E\}$ dal quale viene correttamente identificato l'insieme $K(A')$ degli stati kernel in A' attraverso la visita in profondità. Ulteriormente, l'insieme R delle classi di quasi-equivalenza, costruito sulla tabella H il cui contenuto al termine dell'algoritmo è riportato in Tabella 3, risulta correttamente essere $Q'/\sim$ come previsto dal Teorema 2.7.

Chiave	Valore
$[B, C]$	A
$[D, F]$	C
$[G, G]$	D
$[E, G]$	F
$[E, D]$	B
$[F, D]$	E
$[B, D]$	E
$[D, D]$	D
$[D, B]$	C

Tabella 3: Tabella hash H al termine dell'esecuzione dell'algoritmo sull'automa in Figura 2.

Capitolo 3

Implementazione

3.1 Introduzione

In questo capitolo vengono presentate delle possibili implementazioni degli algoritmi di iperminimizzazione descritti nel Capitolo 2. Ciascun algoritmo è suddiviso in funzioni che implementano i singoli passaggi presentati nella descrizione teorica, in modo da rendere più chiara la struttura dello pseudocodice e facilitarne la comprensione.

3.2 Algoritmo di Badr, Geffert e Shipman

Si presenta in primo luogo l'implementazione della funzione principale dell'algoritmo di Badr, Geffert e Shipman il cui pseudocodice è riportato nell'Algoritmo 1.

Successivamente, vengono mostrate le implementazioni delle funzioni ausiliarie utilizzate dall'algoritmo principale.

Algoritmo 1: Algoritmo di Badr, Geffert e Shipman.

Parametri: Un automa a stati finiti deterministico $A = (Q, \Sigma, \delta, q_I, F)$
Output: Uno dei possibili automi iperminimi quasi-equivalenti ad A

```
1 A' ← minimize(A)
2 K ← kernelStates(A')                    /* Algoritmo 2 */
3 R ← almostEquivalenceClasses(A')        /* Algoritmo 3 */
4 mergeStates(A', K, R)                   /* Algoritmo 4 */
5 return A'
```

Nel dettaglio:

- l'Algoritmo 2 implementa la funzione *kernelStates* che, dato un automa minimo in ingresso, restituisce l'insieme degli stati kernel nell'automa;
- l'Algoritmo 3 implementa la funzione *almostEquivalenceClasses* che, dato un automa minimo in ingresso, restituisce un vettore che associa ad ogni stato dell'automa l'indice della classe di quasi-equivalenza a cui appartiene;
- l'Algoritmo 4 implementa la funzione *mergeStates* che, dato un automa minimo in ingresso, costruisce uno dei possibili automi iperminimi quasi-equivalenti ad esso.

Algoritmo 2: Algoritmo per l'ottenimento dell'insieme degli stati kernel.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$
Output: L'insieme degli stati kernel $K(A)$

```
1  E ← matrice di zeri di dimensione ||Q|| × ||Q||
2  for i ← 1 to ||Q|| do
3  |  V_i ← breadthFirstSearch(A, q_i)
4  |  for j ← 1 to ||Q|| do
5  |  |  if q_j ∈ V_i then E[i][j] ← 1
6  |  end
7  end
8  K ← ∅
9  for i ← 1 to ||Q|| do
10 |  for j ← 1 to ||Q|| do
11 |  |  if E[1][j] = 1 and E[j][j] = 1 and E[j][i] = 1 then
12 |  |  |  K ← K ∪ {q_i}
13 |  |  |  break
14 |  |  end
15 |  end
16 end
17 return K
```

Si ritiene importante puntualizzare come, la funzione *breadthFirstSearch* utilizzata nell'Algoritmo 2 per ottenere l'insieme V_i degli stati raggiungibili da uno stato q_i in A, non sia la versione classica della Breadth-First Search, ma una versione modificata che non aggiunge nella prima iterazione lo stato q_i di partenza all'insieme degli stati visitati. Ne segue che q_i si troverà nell'insieme V_i se e solo se raggiungibile da uno stato q_j, con $q_j \neq q_i$, tale che q_j sia raggiungibile da q_i o se q_i ha un ciclo su se stesso.

Algoritmo 3: Algoritmo per il calcolo delle classi di quasi-equivalenza.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$
Output: Un vettore che associa ad ogni stato $q_i \in Q$ l'indice j della classe di quasi-equivalenza Q_j a cui appartiene

```
1  R ← vettore di dimensione ||Q||
2  for i ← 1 to ||Q|| do
3  |  R[i] ← i
4  end
5  do
6  |  ae_found ← false
7  |  foreach q_i ∈ Q do
8  |  |  foreach q_j ∈ Q tale che j > i do
9  |  |  |  if R[i] ≠ R[j] then
10 |  |  |  |  ae ← true
11 |  |  |  |  foreach σ ∈ Σ do
12 |  |  |  |  |  if R[δ(q_i, σ)] ≠ R[δ(q_j, σ)] then
13 |  |  |  |  |  |  ae ← false
14 |  |  |  |  |  |  break
15 |  |  |  |  |  end
16 |  |  |  |  end
17 |  |  |  |  if ae then
18 |  |  |  |  |  foreach q ∈ Q do
19 |  |  |  |  |  |  if R[q] = R[j] then R[q] ← R[i]
20 |  |  |  |  |  end
21 |  |  |  |  |  ae_found ← true
22 |  |  |  |  end
23 |  |  |  end
24 |  |  end
25 |  end
26 while ae_found
27 return R
```

Algoritmo 4: Algoritmo per la costruzione dell'automa iperminimo.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$, l'insieme degli stati kernel $K(A)$ e il vettore delle classi di quasi-equivalenza R

Output: Al termine dell'esecuzione dell'algoritmo, A sarà iperminimo

```
1  for i ← 1 to ||Q'|| do
2  |  K_{Q_i}, P_{Q_i} ← ∅, ∅
3  |  for j ← 1 to ||Q'|| do
4  |  |  if R[j] = i then
5  |  |  |  if q_j ∈ K then
6  |  |  |  |  K_{Q_i} ← K_{Q_i} ∪ {q_j}
7  |  |  |  end
8  |  |  |  else
9  |  |  |  |  P_{Q_i} ← P_{Q_i} ∪ {q_j}
10 |  |  |  end
11 |  |  end
12 |  end
13 |  if P_{Q_i} ∪ K_{Q_i} ≠ ∅ then
14 |  |  q ← un elemento qualunque di P_{Q_i} ∪ K_{Q_i}
15 |  |  if P_{Q_i} ≠ ∅ and K_{Q_i} ≠ ∅ and q ∉ K_{Q_i} then
16 |  |  |  q ← un elemento qualunque di K_{Q_i}
17 |  |  end
18 |  |  foreach q' ∈ P_{Q_i} \ {q} do
19 |  |  |  collassa lo stato q' in q
20 |  |  end
21 |  end
22 end
```

3.3 Algoritmo di Badr

Seguendo la medesima linea dell'algoritmo precedente, si presenta l'implementazione dell'algoritmo di Badr iniziando dalla funzione principale e procedendo con le funzioni ausiliarie.

Algoritmo 5: Algoritmo di Badr.

Parametri: Un automa a stati finiti deterministico $A = (Q, \Sigma, \delta, q_I, F)$
Output: Uno dei possibili automi iperminimi quasi-equivalenti ad A

```
1 A' ← minimize(A)
2 U ← almostEquivalenceClasses(A')          /* Algoritmo 7 */
3 P, K ← preambleAndKernelStates(A')        /* Algoritmo 6 */
4 mergeStates(A', U, P, K)                  /* Algoritmo 8 */
5 return A'
```

In questo caso, le funzioni ausiliarie sono implementate come segue:

- l'Algoritmo 7 implementa la funzione *almostEquivalenceClasses* che, dato un automa minimo in ingresso, restituisce la partizione dell'insieme degli stati in classi di quasi-equivalenza;
- l'Algoritmo 6 implementa la funzione *preambleAndKernelStates* che, dato un automa minimo in ingresso, restituisce l'insieme degli stati preambolo e l'insieme degli stati kernel nell'automa;
- l'Algoritmo 8 implementa la funzione *mergeStates* che, dato in ingresso un automa minimo A, la partizione dei suoi stati in classi di quasi-equivalenza ed i suoi stati kernel e preambolo, costruisce uno dei possibili automi iperminimi quasi-equivalenti all'automa in ingresso.

Algoritmo 6: Algoritmo per l'ottenimento degli stati kernel e preambolo.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$
Output: Gli insiemi $P(A)$ e $K(A)$

```
1 K ← ∅
2 foreach q ∈ Q do
3 |   V ← breadthFirstSearch(A, q)
4 |   if q ∈ V then K ← K ∪ V
5 end
6 return (Q \ K, K)
```

Algoritmo 7: Algoritmo per la computazione delle classi di quasi-equivalenza.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$
Output: L'insieme quoziente $Q/\sim$

```
1  A^⊗ ← A ⊗ A
2  S ← {(q, q) : q ∈ Q}
3  R ← finiteLanguageStates(A^⊗, S)                    /* Algoritmo 9 */
4  U ← struttura QuickFind vuota
5  foreach q ∈ Q do U.makeset(q)
6  foreach (q_A, q_B) ∈ R do
7  |  Q_i ← U.find(q_A)
8  |  Q_j ← U.find(q_B)
9  |  if Q_i ≠ Q_j then U.union(Q_i, Q_j)
10 end
11 return U
```

Poiché l'implementazione dell'Algoritmo 7 per la costruzione delle classi di quasi-equivalenza risulta essere particolarmente lunga, si utilizza un'ulteriore funzione ausiliaria *finiteLanguageStates* la cui implementazione è definita nell'Algoritmo 9.

Algoritmo 8: Algoritmo per la costruzione dell'automa iperminimo.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$, l'insieme quoziente $Q/\sim$ e gli insiemi $P(A)$ e $K(A)$
Output: Al termine dell'esecuzione dell'algoritmo, A sarà iperminimo

```
1 foreach Q_i ∈ Q/∼ do
2 |  P_{Q_i} ← Q_i ∩ P(A)
3 |  K_{Q_i} ← Q_i ∩ K(A)
4 |  if K_{Q_i} ≠ ∅ then q ← un elemento qualunque di K_{Q_i}
5 |  else q ← un elemento qualunque di P_{Q_i}
6 |  foreach q' ∈ P_{Q_i} \ {q} do
7 |  |  collassa lo stato q' in q
8 |  end
9 end
```

Inoltre, si puntualizza nuovamente come, essendo l'idea dell'identificazione degli stati kernel basata sulle medesime osservazioni fatte nell'algoritmo di Badr, Geffert e Shipman e dunque l'implementazione molto simile, la funzione *breadthFirstSearch* utilizzata nell'Algoritmo 6 contiene la stessa modifica descritta in precedenza.

Algoritmo 9: Algoritmo per il calcolo degli stati che inducono un linguaggio finito.

Parametri: Un automa a stati finiti deterministico $A = (Q, \Sigma, \delta, q_I, F)$ e l'insieme S degli stati che inducono un linguaggio vuoto in A

Output: L'insieme degli stati in A che inducono un linguaggio finito

```
1  S^c ← Q \ S
2  foreach q ∈ Q do
3  |  T_q^→ ← ∅
4  |  T_q^← ← ∅
5  end
6  foreach q ∈ S^c do
7  |  foreach σ ∈ Σ do
8  |  |  q' ← δ(q, σ)
9  |  |  T_q^→ ← T_q^→ ∪ {(q', σ)}
10 |  |  T_q'^← ← T_q'^← ∪ {(q, σ)}
11 |  end
12 end
13 R ← ∅
14 while S ≠ ∅ do
15 |  q ← un elemento qualunque di S
16 |  S ← S \ {q}
17 |  R ← R ∪ {q}
18 |  foreach (q', σ) ∈ T_q^← do
19 |  |  T_q'^→ ← T_q'^→ \ {(q, σ)}
20 |  |  if T_q'^→ = ∅ then
21 |  |  |  S ← S ∪ {q'}
22 |  |  end
23 |  end
24 end
25 return R
```

3.4 Algoritmo di Holzer e Maletti

Nuovamente, si procede con la presentazione dell'implementazione dell'algoritmo di Holzer e Maletti partendo dalla funzione principale e proseguendo con le funzioni ausiliarie.

Algoritmo 10: Algoritmo di Holzer e Maletti.

Parametri: Un automa a stati finiti deterministico $A = (Q, \Sigma, \delta, q_I, F)$
Output: Uno dei possibili automi iperminimi quasi-equivalenti ad A

```
1 A' ← minimize(A)
2 K ← kernelStates(A')                          /* Algoritmo 11 */
3 R ← almostEquivalenceClasses(A')              /* Algoritmo 15 */
4 mergeStates(A', R, K)                         /* Algoritmo 13 */
5 return A'
```

Gli algoritmi che implementano le funzioni ausiliarie utilizzate dalla funzione principale sono organizzati come segue:

- l'Algoritmo 11 implementa la funzione *kernelStates* che, dato in ingresso un automa minimo, ne identifica e restituisce l'insieme degli stati kernel;
- l'Algoritmo 15 implementa la funzione *almostEquivalenceClasses* che, dato in ingresso un automa minimo, ne computa e restituisce la partizione degli stati in classi di quasi-equivalenza;
- l'Algoritmo 13 implementa la funzione *mergeStates* che, dato in ingresso un automa minimo, la partizione dei suoi stati in classi di quasi-equivalenza e l'insieme degli stati kernel, costruisce uno dei possibili automi iperminimi quasi-equivalenti all'automa in ingresso.

Algoritmo 11: Algoritmo per l'identificazione degli stati kernel.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$
Output: L'insieme degli stati kernel $K(A)$

```
1 index, low ← tabelle hash vuote che associano q ∈ Q a n ∈ ℕ
2 i ← 0
3 S ← stack vuoto
4 K ← ∅
5 tarjan(A, q_I, index, low, i, S, K)           /* Algoritmo 14 */
6 return K
```

Si puntualizza come, nelle implementazioni dei diversi algoritmi che costituiscono le funzioni ausiliarie, la funzione *merge* corrisponda alla specifica strategia di collasso degli stati ideata da Holzer e Maletti di cui una sua possibile implementazione è data nell'Algoritmo 12.

Algoritmo 12: Algoritmo per il collasso degli stati.

Parametri: Una funzione di transizione δ, lo stato iniziale q_I e due stati q_A, q_B
Output: La funzione di transizione δ' e lo stato iniziale q'_I in seguito al collasso dello stato q_A in q_B

```
1  δ' ← ∅
2  q'_I ← q_I
3  if q_I = q_A then q'_I ← q_B
4  foreach (q, σ, q') ∈ δ do
5  |  if q ≠ q_A then
6  |  |  if q' = q_A then δ' ← δ' ∪ {(q, σ, q_B)}
7  |  |  else δ' ← δ' ∪ {(q, σ, q')}
8  |  end
9  end
10 return (δ', q'_I)
```

Algoritmo 13: Algoritmo per la costruzione dell'automa iperminimo.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$, l'insieme quoziente $Q/\sim$ e l'insieme $K(A)$
Output: Al termine dell'esecuzione dell'algoritmo, A sarà iperminimo

```
1  foreach Q_i ∈ Q/∼ do
2  |  q ← un elemento qualunque di Q_i
3  |  if K(A) ∩ Q_i ≠ ∅ and q ∉ K(A) then
4  |  |  q ← un elemento qualunque di K(A) ∩ Q_i
5  |  end
6  |  foreach q' ∈ Q_i \ K(A) do δ, q_I ← merge(δ, q_I, q, q')
7  end
```

Ulteriormente, negli algoritmi che si avvalgono dell'utilizzo di tabelle hash, si assume che tali strutture implementino alcune funzioni necessarie al loro utilizzo i cui dettagli implementativi non sono riportati. In particolare, la funzione *containsKey* restituisce il valore booleano *true* se la chiave passata come argomento è presente nella tabella hash, $false$ altrimenti. La funzione *deleteKey* rimuove la chiave passata come argomento, e conseguentemente il valore ad essa associato, dalla tabella hash. Infine, la funzione *values* restituisce la lista dei valori associati alle chiavi presenti in tabella.

Algoritmo 14: Algoritmo di Tarjan per l'identificazione degli stati kernel.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$, uno stato $q \in Q$, tabelle hash *index* e *low* che associano $q \in Q$ a $n \in \mathbb{N}$, un intero i, uno stack S, l'insieme K nel quale vengono inseriti gli stati kernel

Output: Al termine dell'esecuzione dell'algoritmo, l'insieme K conterrà $K(A)$

```
1  index[q], low[q] ← i, i
2  i ← i + 1
3  S.push(q)
4  foreach σ ∈ Σ do
5  |  if not index.containsKey(δ(q, σ)) then
6  |  |  tarjan(A, δ(q, σ), index, low, i, S, C)      /* chiamata ricorsiva */
7  |  |  low[q] ← min(low[q], low[δ(q, σ)])
8  |  end
9  |  else if δ(q, σ) ∈ S then
10 |  |  low[q] ← min(low[q], index[δ(q, σ)])
11 |  end
12 |  if δ(q, σ) = q then
13 |  |  K ← K ∪ {q}
14 |  end
15 end
16 if low[q] = index[q] then
17 |  scc ← ∅
18 |  while S.top() ≠ q do
19 |  |  scc ← scc ∪ {S.pop()}
20 |  end
21 |  scc ← scc ∪ {S.pop()}
22 |  if ||scc|| > 1 then
23 |  |  q_scc ← un elemento qualunque di scc
24 |  |  V ← depthFirstSearch(A, q_scc)
25 |  |  K ← K ∪ V
26 |  end
27 end
```

L'Algoritmo 14 corrisponde alla versione semplificata dell'algoritmo di Tarjan [DFI08] per la ricerca delle componenti fortemente connesse in un grafo introdotta nella descrizione dell'algoritmo di iperminimizzazione di Holzer e Maletti. In particolare, l'algoritmo è stato modificato per adattarlo all'identificazione degli stati centrali in un automa a stati finiti deterministico minimizzato.

Algoritmo 15: Algoritmo per il calcolo delle classi di quasi-equivalenza.

Parametri: Un automa a stati finiti deterministico minimo $A = (Q, \Sigma, \delta, q_I, F)$
Output: L'insieme quoziente $Q/\sim$

```
1  I = Q
2  P = Q
3  δ_∼, q_I∼ = δ, q_I
4  H = tabella hash vuota che associa vettori di successori a stati
5  R = tabella hash vuota che associa ad uno stato la parte di cui è elemento
6  foreach q ∈ Q do
7  |  R[q] = {q}
8  end
9  while I ≠ ∅ do
10 |  q_A = un elemento qualunque di I
11 |  I = I \ {q_A}
12 |  if q_A ∈ P and H.containsKey(V(q_A)) then
13 |  |  q_B = H[V(q_A)]
14 |  |  if ||R[q_B]|| ≥ ||R[q_A]|| then
15 |  |  |  q_A, q_B = q_B, q_A
16 |  |  end
17 |  |  P = P \ {q_B}
18 |  |  I = I ∪ {q ∈ P | ∃σ ∈ Σ : δ_∼(q, σ) = q_B}
19 |  |  δ_∼, q_I∼ = merge(δ_∼, q_I∼, q_A, q_B)
20 |  |  R[q_A] = R[q_A] ∪ R[q_B]
21 |  |  R.deleteKey(q_B)
22 |  end
23 |  H[V(q_A)] = q_A
24 end
25 return R.values()
```

Capitolo 4

Risultati sperimentali

4.1 Ambiente e scelte implementative

La prima fase del progetto è consistita nell'individuazione dell'ambiente di sviluppo e degli strumenti ad esso annessi più adatti all'implementazione degli algoritmi di iperminimizzazione e all'analisi delle relative prestazioni. Un attento studio delle possibilità offerte dai diversi linguaggi di programmazione general purpose e delle librerie messe a disposizione dall'ampia community open-source ha portato alla scelta di *Python 3.12.3* [Pyt24] come linguaggio principale per lo sviluppo del progetto.

Nel dettaglio, si è fatto ampio uso delle strutture dati offerte dalla libreria standard di Python e di alcune librerie di terze parti per la manipolazione di automi a stati finiti deterministici e per la manipolazione dei dati ottenuti dai test effettuati. In particolare, è stata utilzzata la libreria *Automata* [Eva24] per gestire i DFA all'interno del codice. Questa è stata scelta poiché implementa già al suo interno l'algoritmo di minimizzazione di Hopcroft e permette la rappresentazione degli automi sotto forma di grafi usando il popolare strumento di visualizzazione *Graphviz* [Lab24].

4.2 Generazione del campione

Si rende necessario comporre un campione di automi a stati finiti deterministici su cui effettuare i test per valutare le prestazioni degli algoritmi di iperminimizzazione implementati. Poiché non sono stati trovati dataset di DFA in letteratura che soddisfino i requisiti richiesti, si è optato per la generazione pseudo casuale del campione.

Seguendo quanto detto, si è dovuto trovare una strategia per generare automi a stati finiti deterministici che potessero essere iperminimizzati. Infatti, la generazione totalmente pseudo casuale porta nella stragrande maggioranza dei casi alla generazione di automi già iperminimi. L'idea alla base della generazione del campione è stata quella di costruire automi a stati finiti deterministici con un alfabeto binario fissato $\Sigma = \{a, b\}$ ed il cui insieme di stati preambolo sia generato in maniera regolare mentre l'insieme di stati kernel sia generato in modo pseudo casuale. Successivamente, si sono svolti dei test variando alcuni parametri per ottenere i settaggi ottimali per massimizzare la percentuale di automi iperminimizzabili su quelli generati.

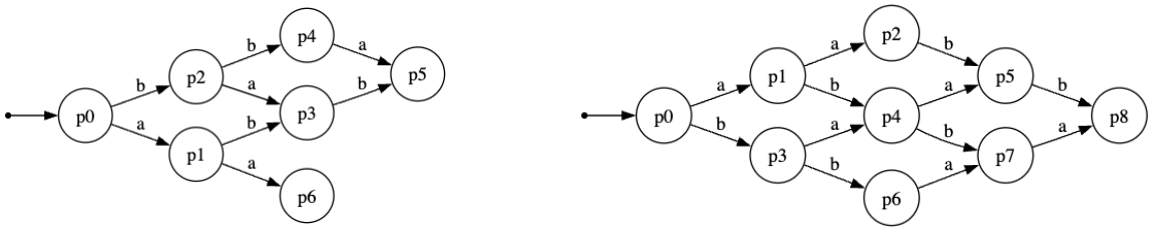
Nel dettaglio, il preambolo viene generato in modo tale che stati e transizioni formino una griglia. Supponendo di voler generare un preambolo composto da n stati, con $n \in \mathbb{N}$, in primo luogo si compone una matrice di $\lfloor\sqrt{n}\rfloor$ righe e $\lfloor\sqrt{n}\rfloor$ colonne, cioè di $m = (\lfloor\sqrt{n}\rfloor)^2$ elementi, in cui ogni elemento corrisponde ad uno stato del preambolo. Sia $q_{i,j}$ lo stato di riga i e colonna j, lo stato iniziale dell'automa sarà $q_{1,1}$, ovvero lo stato che si trova nella prima colonna della prima riga della matrice. La funzione di transizione è definita $\forall q_{i,j}$ con $i < \sqrt{m}$ e $j < \sqrt{m}$ e $\forall \sigma \in \Sigma$, ricordando $\Sigma = \{a, b\}$, come segue:

$$\delta(q_{i,j}, \sigma) = \begin{cases} q_{i+1,j} & \text{se } \sigma = a \\ q_{1,j+1} & \text{se } \sigma = b \end{cases}$$

Se n non è un quadrato perfetto (i.e. $\sqrt{n} \notin \mathbb{N}$), gli $n - m$ stati preambolo rimanenti vengono suddivisi in due gruppi R e B, rispettivamente di dimensione $r = \lfloor\frac{n-m}{2}\rfloor$ e $b = \lceil\frac{n-m}{2}\rceil$. Partendo dalla matrice costruita in precedenza, siano q_{r_i} l'i-esimo elemento del gruppo R e q_{b_j} il j-esimo elemento del gruppo B, si completa la costruzione della griglia degli stati preambolo come segue: $\forall i \in \{1, 2, ..., r\}$, $\delta(q_{i,\sqrt{m}}, a) = q_{r_i}$ e $\forall j \in \{1, 2, ..., b\}$, $\delta(q_{\sqrt{m},j}, b) = q_{b_i}$. In altri termini, viene aggiunta una colonna sulla destra contentente gli stati in R ed una riga sotto contentente gli stati in B.

Si osservi come la funzione di transizione risultante sia parziale in quanto non è definita per tutti i simboli in ingresso per tutti gli stati del preambolo. Nello specifico, non è definita per gli stati in ultima posizione di ogni riga che ricevono in ingresso il simbolo a e per gli stati in ultima riga che ricevono in ingresso il simbolo b. Queste transizioni sono sfruttate e sono successivamente definite in modo da portare gli stati preambolo agli stati kernel.

L'insieme di stati kernel può essere invece generato in due modi differenti, che ne modificano la composizione, il che costituisce uno dei parametri che sono stati variati per ottenere i settaggi ottimali per massimizzare la percentuale di automi iperminimizzabili su quelli generati. In particolare, il kernel può essere composto da una singola



(a) Preambolo di dimensione $n = 7$

(b) Preambolo di dimensione $n = 9$

Figura 5: Due esempi di preamboli generati seguendo la tecnica descritta:
(a) con un numero di stati non quadrato perfetto,
(b) con un numero quadrato perfetto di stati.

componente fortemente connessa oppure da un numero pseudo-casuale, ovviamente minore del numero di stati kernel, di componenti fortemente connesse, connesse tra loro. La strategia per la generazione della singola componente fortemente connessa, supponendo di voler generare una componente composta da n stati, con $n \in \mathbb{N}$, consiste nel definire la funzione di transizione per il generico stato q_i, corrispondente all'i-esimo stato della componente, come segue: $\delta(q_i, a) = q_{(i+1) \bmod n}$ e $\delta(q_i, b)$ invece uguale ad uno stato scelto pseudo-casualmente tra quelli della componente. Il numero di stati all'interno del kernel facenti parte dell'insieme degli stati finali è anch'esso un parametro variabile, la scelta di quali stati siano finali è invece pseudo casuale.

Infine, la connessione tra gli stati preambolo e gli stati kernel avviene in maniera pseudo-casuale: per ogni stato preambolo q_A descritto precedentemente di cui non è stata definita la transizione per un simbolo $\sigma \in \Sigma$, si sceglie uno stato kernel q_B e si definisce la transizione $\delta(q_A, \sigma) = q_B$.

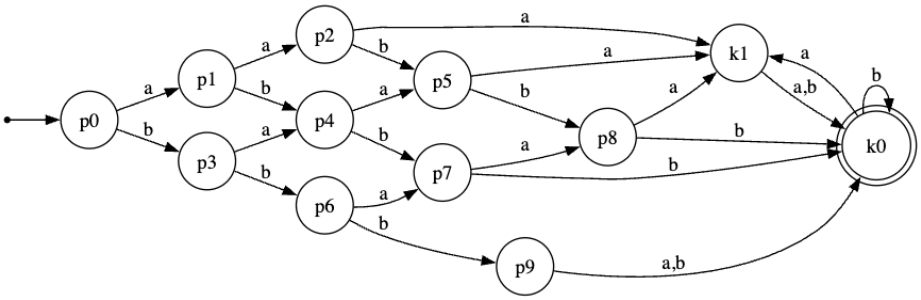


Figura 6: Un esempio di automa generato dalla procedura descritta.

Un esempio di un automa a stati finiti deterministico A, generato seguendo la procedura descritta, con $||P(A)|| = 10$, $||K(A)|| = 2$ e kernel composto da una singola componente fortemente connessa, è mostrato in Figura 6.

Come già accennato precedentemente, sono stati effettuati dei test variando alcuni parametri della procedura di generazione, i cui risultati ottenuti sono riportati di seguito, al fine di ottenere i settaggi ottimali per massimizzare la percentuale di automi iperminimizzabili su quelli generati. Maggiormente nello specifico, sono stati variati: il rapporto tra gli stati preambolo e gli stati kernel, la percentuale di stati kernel finali, la composizione del kernel e il numero di stati dell'automa.

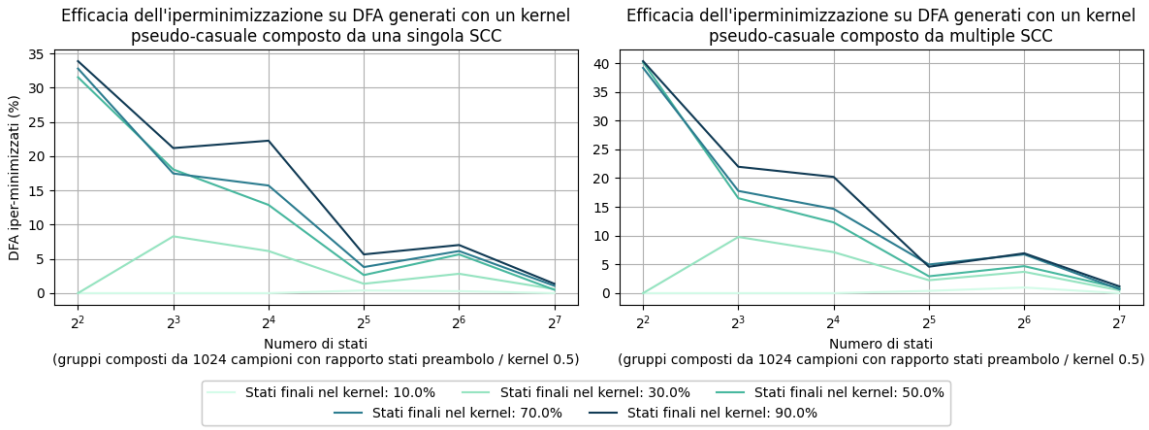


Figura 7: Efficacia dell'iperminimizzazione su automi con differente numero di stati kernel finali e numero di stati.

La Figura 7 mostra come varia l'efficacia dell'iperminimizzazione su automi con differente numero di stati kernel finali all'aumentare del numero di stati dell'automa. Se ne deduce che, nonostante all'aumentare del numero di stati diminuisca la percentuale di automi iperminimizzabili, automi con un kernel composto da più di una componente fortemente connessa e con un alto numero di stati finali nel kernel sono più proni all'essere iperminimizzati.

In Figura 8 è possibile osservare come varia l'efficacia dell'iperminimizzazione su automi con differente numero di stati kernel finali all'aumentare del rapporto tra stati preambolo e kernel. Risulta chiaro che automi con un kernel composto da più di una componente fortemente connessa e con kernel di dimensioni ridotte contenente un alto numero di stati finali sono più probabilmente iperminimizzabili.

La Figura 9 mostra come varia l'efficacia dell'iperminimizzazione su automi con differente rapporto tra stati preambolo e kernel all'aumentare del numero di stati dell'automa. Si osservi come, ancora una volta, nonostante all'aumentare del numero di stati diminuisca la percentuale di automi iperminimizzabili, automi con un kernel composto da più di una componente fortemente connessa e con un alto numero di stati preambolo e basso numero di stati kernel sono più proni all'essere iperminimizzati.

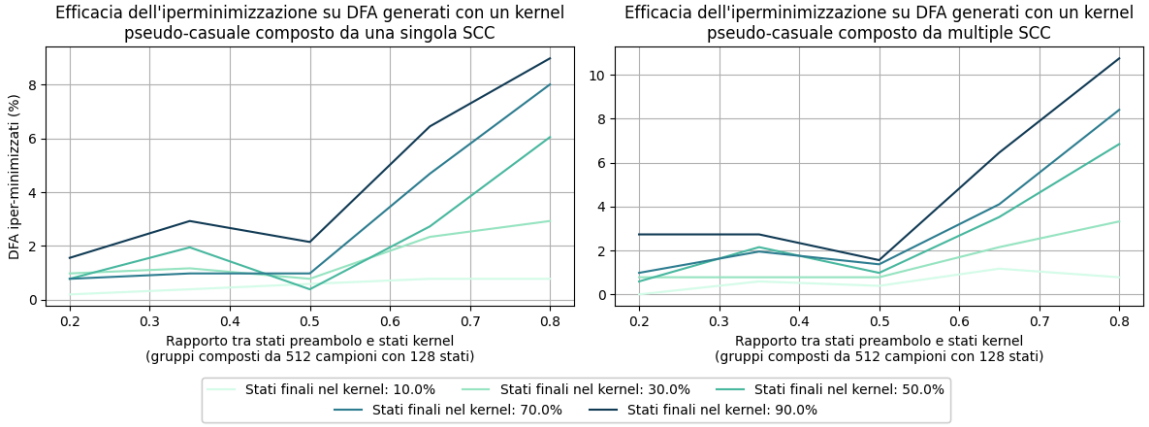


Figura 8: Efficacia dell'iperminimizzazione su automi con differente numero di stati kernel finali e rapporto tra stati preambolo e kernel.

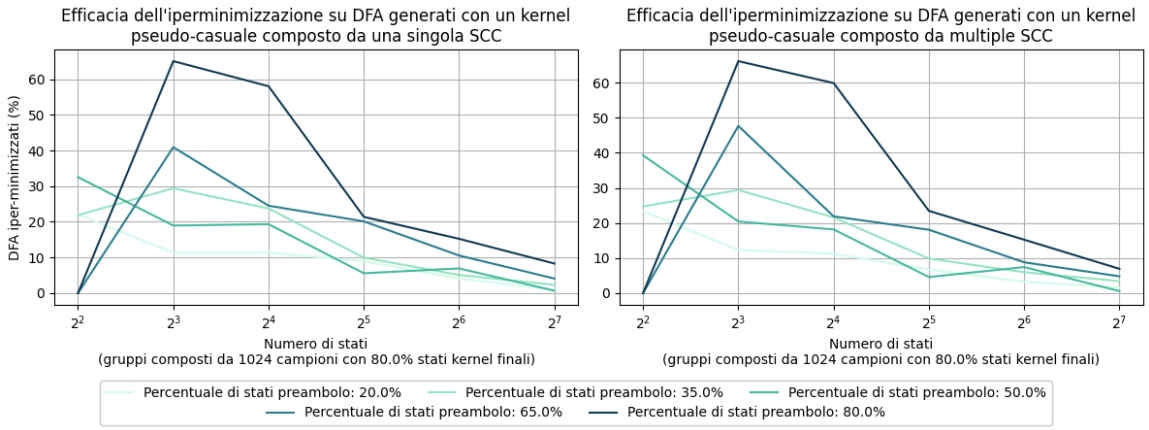


Figura 9: Efficacia dell'iperminimizzazione su automi con differente numero di stati e rapporto tra stati preambolo e kernel.

In conclusione, risulta chiaro che, per massimizzare la percentuale di automi generati iperminimizzabili, è necessario generare automi con un kernel composto da più di una componente fortemente connessa, con un alto numero di stati finali nel kernel e con un rapporto tra stati preambolo e kernel che favorisca nettamente gli stati preambolo.

4.3 Prestazioni

È stata effettuata una serie di test sperimentali per valutare le prestazioni degli algoritmi di iperminimizzazione implementati. Gli esperimenti sono stati condotti su

una macchina con processore Apple M1, 16GB di RAM e sistema operativo macOS Sonoma 14.5 e su di un campione di automi generati come descritto precedentemente con un numero di stati compreso tra 2^2 e 2^{10}, in particolar modo 256 automi per ogni potenza di 2 inclusa nell'intervallo (estremi inclusi), in modo tale da osservare come il tempo di esecuzione di ciascun algoritmo varia in relazione alla dimensione, in termini di numero di stati, dell'automa in ingresso.

Sulla base dei risultati ottenuti nella Sezione 4.2, sono stati generati automi con un rapporto tra stati preambolo e kernel che favorisca nettamente gli stati preambolo, pari a 0.8, e con un kernel formato da più di una componente fortemente connessa nel quale l'80% degli stati da cui è composto appartenga all'insieme degli stati finali.

Gli esiti ottenuti dagli esperimenti sono riportati in Figura 10, 11 e 12.

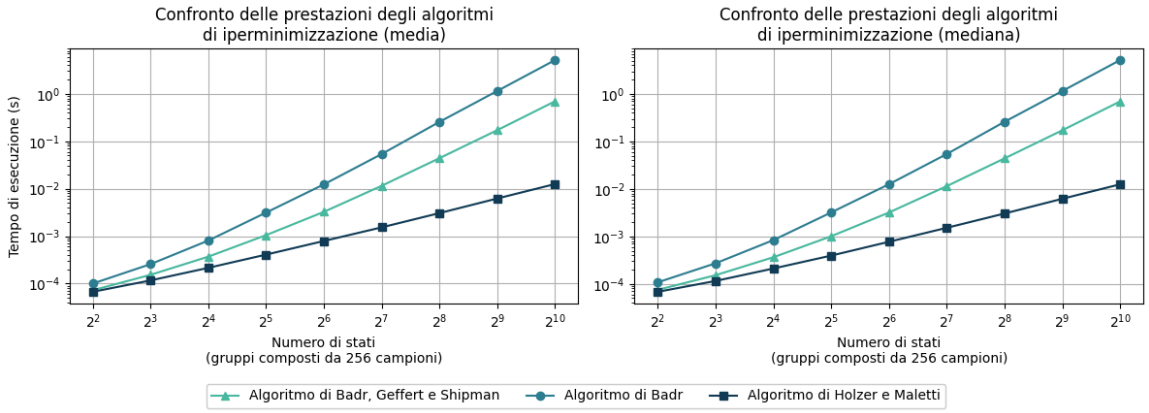


Figura 10: Performance degli algoritmi di iperminimizzazione sull'intero campione.

I risultati sperimentali confermano le aspettative per quanto riguarda le prestazioni dell'algoritmo di Holzer e Maletti, che risulta essere il più efficiente tra quelli implementati, mentre smentiscono parzialmente le aspettative per quanto riguarda gli altri due algoritmi. Infatti, l'algoritmo di Badr risulta essere più lento dell'algoritmo di Badr, Geffert e Shipman. Questo risultato è dovuto al fatto che l'algoritmo di Badr, nella funzione che si occupa della costruzione dello XOR cross product dell'automa in ingresso minimizzato con se stesso ed in quella che effettua calcolo degli stati che inducono un linguaggio finito su quest'ultimo, effettua $\Theta(n^2)$ operazioni, dove n è il numero di stati dell'automa in ingresso, il che significa che eseguirà sempre esattamente un numero di operazioni nell'ordine di n^2. Al contrario, l'algoritmo di Badr, Geffert e Shipman, nel calcolo delle classi di quasi-equivalenza, effettua un numero di operazioni $\mathcal{O}(n^3 \cdot m)$, dove m è il numero di simboli dell'alfabeto dell'automa in ingresso, questo implica che l'algoritmo effettua al più un numero di operazioni nell'ordine di $n^3 \cdot m$, ma come dimostrano i risultati sperimentali, tale caso accade

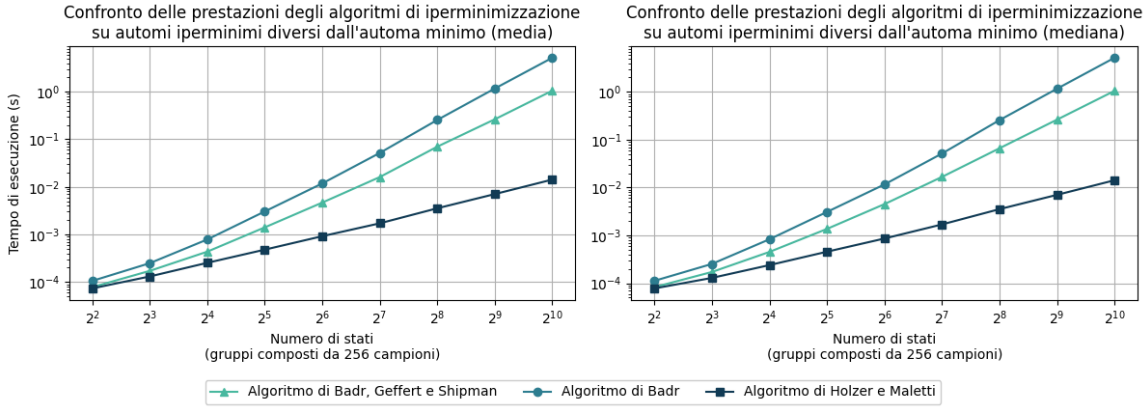


Figura 11: Performance degli algoritmi di iperminimizzazione sul campione considerando solo gli automi iperminimizzazati diversi dell'automa minimo.

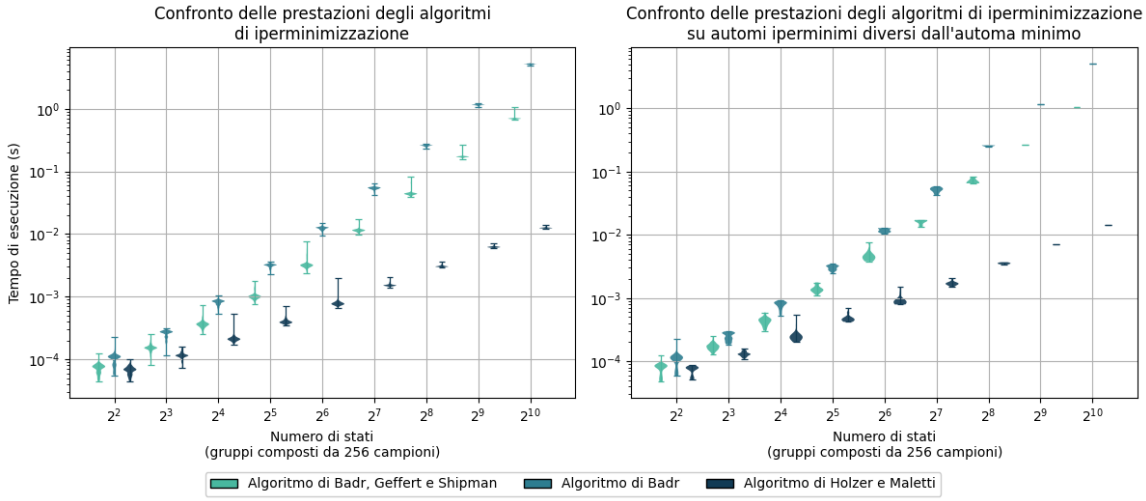


Figura 12: Distribuzioni delle performance degli algoritmi di iperminimizzazione.

molto raramente e dunque nella pratica effettua molte meno operazioni. Nel dettaglio, l'algoritmo di Badr, Geffert e Shipman risulta essere il più lento tra i tre algoritmi implementati quando la ricerca delle coppie di stati quasi-equivalenti richiede tutte le $n-1$ iterazioni al massimo richieste per trovare $Q/\sim$, il che, nuovamente, accade molto di rado. La dimostrazione di quanto detto è osservabile in Figura 13, dove è stata eseguita una versione modificata dell'algoritmo dove la ricerca delle coppie di stati quasi-equivalenti non termina nel momento in cui non ci sono più coppie che soddisfino tale condizione ma continua fino a che non ha effettuato tutte le $n-1$ iterazioni, ovvero il caso peggiore possibile.

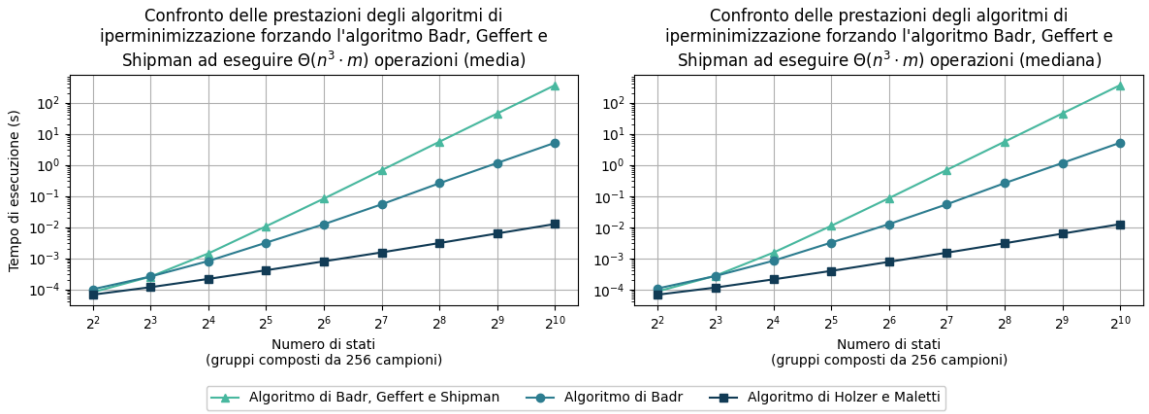


Figura 13: Performance degli algoritmi di iperminimizzazione sul campione forzando l'algoritmo di Badr, Geffert e Shipman ad eseguire $\Theta(n^3 \cdot m)$ operazioni.

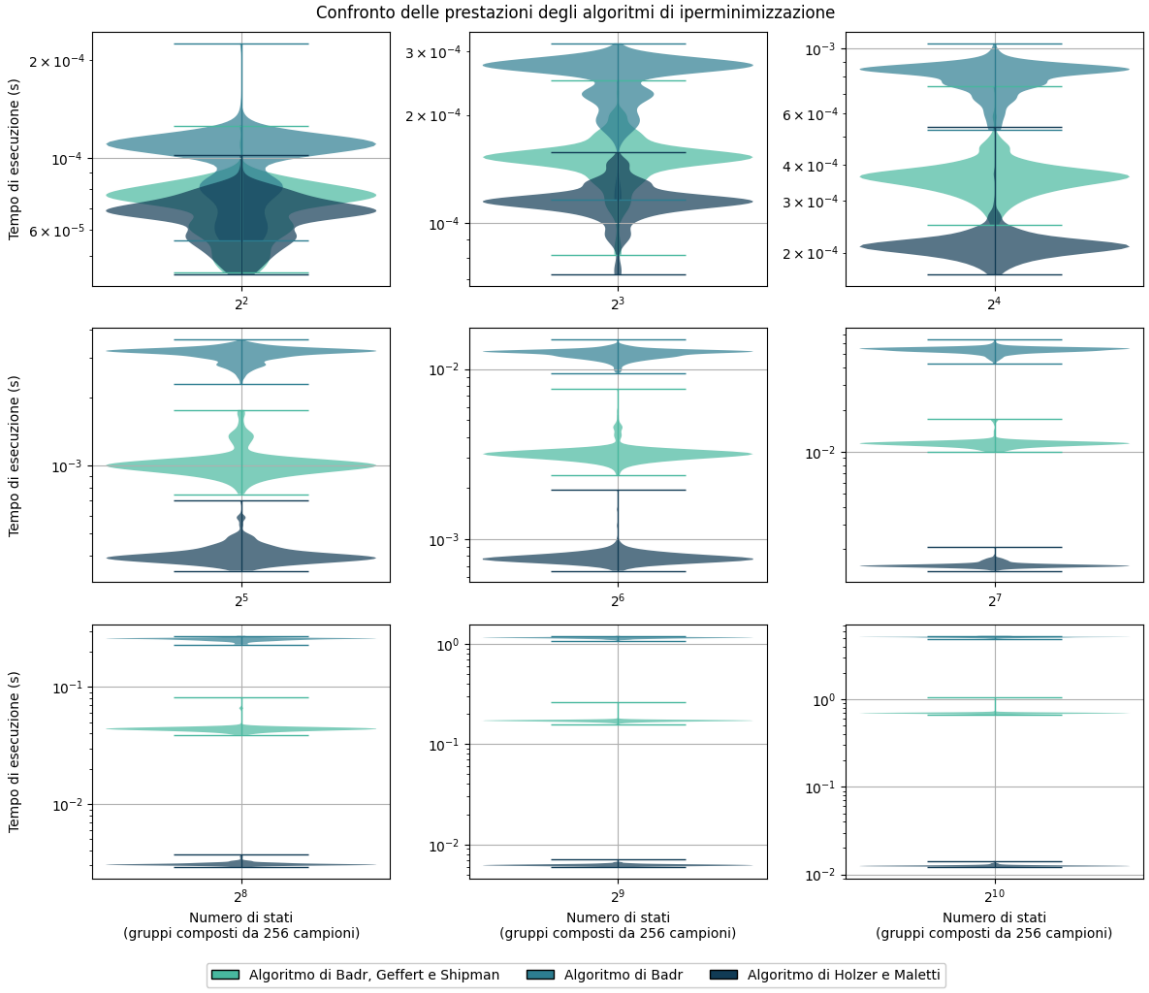


Figura 14: Focus sulle distribuzioni delle performance degli algoritmi di iperminimizzazione sull'intero campione.

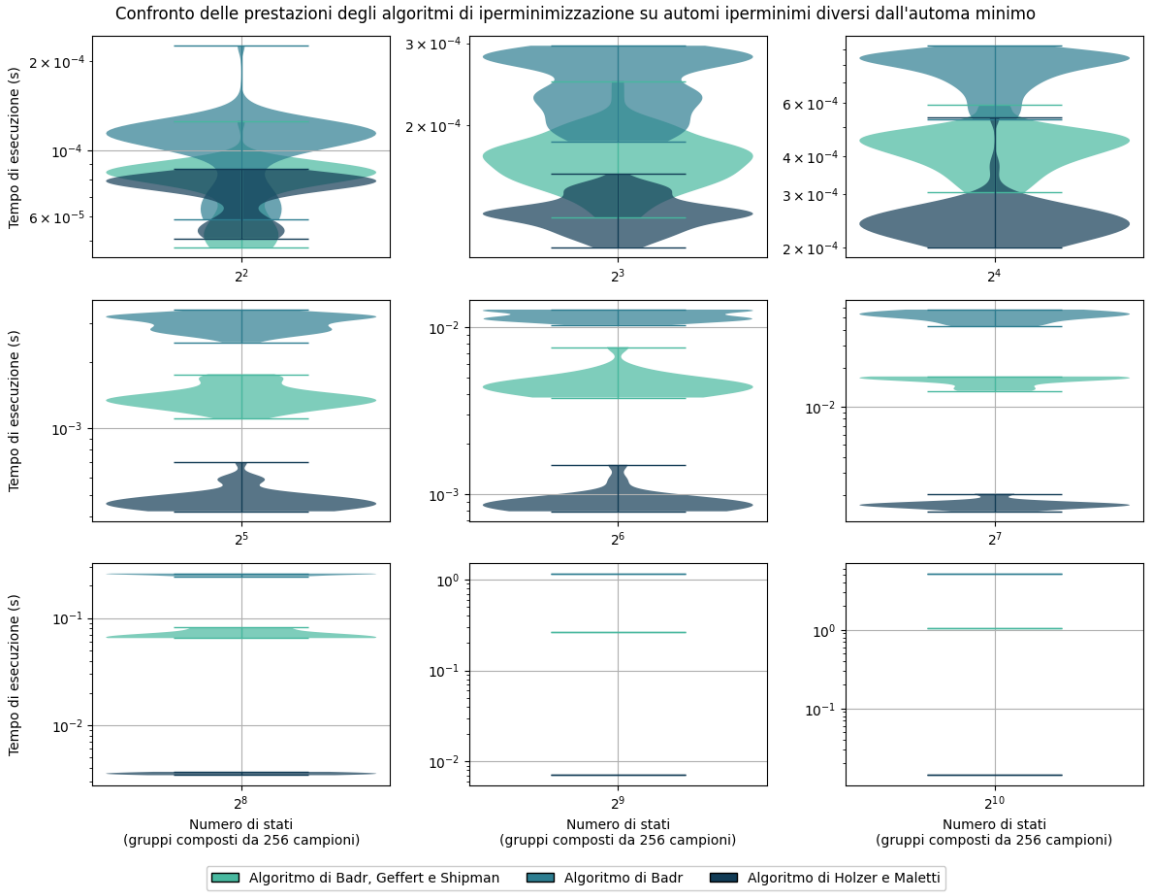


Figura 15: Focus sulle distribuzioni delle performance degli algoritmi di iperminimizzazione considerando solo gli automi il cui numero di stati in seguito all'iperminimizzazione è minore dell'automa minimo.

Bibliografia

Pubblicazioni

[Bad08] Andrew Badr. «Hyper-Minimization in $\mathcal{O}(n^2)$». In: *Implementation and Applications of Automata*. A cura di Oscar H. Ibarra e Bala Ravikumar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 223–231. ISBN: 978-3-540-70844-5.

[BGS09] Andrew Badr, Viliam Geffert e Ian Shipman. «Hyper-minimizing minimized deterministic finite state automata». In: *RAIRO - Theoretical Informatics and Applications* 43.1 (2009), pp. 69–94. DOI: 10.1051/ita:2007061.

[HM10] Markus Holzer e Andreas Maletti. «An $n \log n$ algorithm for hyper-minimizing a (minimized) deterministic automaton». In: *Theoretical Computer Science* 411.38 (2010). Implementation and Application of Automata (CIAA 2009), pp. 3404–3413. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2010.05.029. URL: https://www.sciencedirect.com/science/article/pii/S030439751000321X.

[Hop71] John Hopcroft. «An $n \log n$ algorithm for minimizing states in a finite automaton». In: *Theory of Machines and Computations*. A cura di Zvi Kohavi e Azaria Paz. Academic Press, 1971, pp. 189–196. ISBN: 978-0-12-417750-5. DOI: 10.1016/B978-0-12-417750-5.50022-1. URL: https://www.sciencedirect.com/science/article/pii/B9780124177505500221.

Testi

[DFI08] Camil Demetrescu, Irene Finocchi e Giuseppe F. Italiano. *Algoritmi e strutture dati*. 2ª ed. McGraw-Hill, 2008. ISBN: 9788838664687.

[HMU06] John E. Hopcroft, Rajeev Motwani e Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. 3ª ed. Addison Wesley, 2006. ISBN: 9780321455369.

[Ros18] Kenneth Rosen. *Discrete Mathematics and Its Applications*. 8th ed. McGraw-Hill Higher Education, 2018. ISBN: 9781259676512; 125967651X.

Software

[Eva24] Caleb Evans. *Automata*. Ver. 8.3.0. 23 Mar. 2024. URL: `https://pypi.org/project/automata-lib/`.

[Lab24] AT&T Research Labs. *Graphviz*. Ver. 11.0.0. 28 Apr. 2024. URL: `https://graphviz.org/`.

[Pyt24] Python Software Foundation. *Python Programming Language*. Ver. 3.12.3. 9 Apr. 2024. URL: `https://www.python.org`.