Sapienza Università di Roma

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Linguaggi e Compilatori

Autore:
Simone Lidonnici

13 maggio 2025

Indice

E

Esercizi

E.1 Esercizi su automi

E.1.1 Trasformare un'espressione regolare in NFA

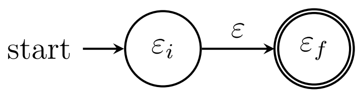
Data un'espressione regolare R con la lista di precedenze delle operazioni e la loro associatività, per prima cosa bisogna creare il suo Syntax Tree. Questo si può fare tramite una semplice operazione ricorsiva partendo dalla radice dell'albero:

1. Trovare in R l'operazione eseguita per ultima, cioè quella con meno priorità e che appare nel punto opposto alla sua associatività. Ad esempio se l'operazione con meno priorità è l'unione ($|$ o $\cup$) ed ha associatività a sinistra l'operazione da scegliere sarà l'unione più a destra.

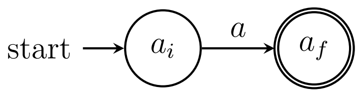
2. L'operazione scelta divide R in due sotto-espressioni R_1 e R_2, cioè i due sottoalberi su cui rieseguire il punto 1.

Possiamo poi trasformarlo in NFA eseguendo un visita in profondità del Syntax Tree e in base al nodo che visitiamo creaiamo un NFA parziale per il suo sottoalbero. Nella spiegazione dell'algoritmo i nodi s_i, s_f indicano il primo e l'ultimo nodo dell'NFA che riconosce s, negli esercizi solitamente si numerano sequanzialmente partendo da 0 in base all'ordine di visita. Eseguendo la visita in profondità l'NFA da creare cambia in base al simbolo contenuto nel nodo visitato:

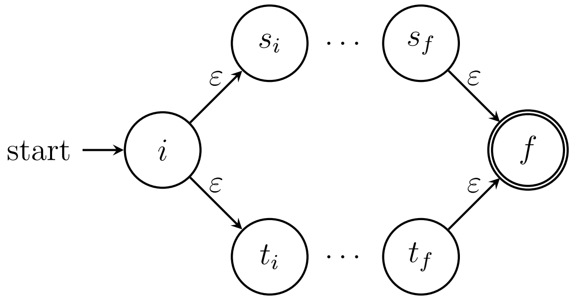
- Se stiamo visitando una foglia contente ε, costruiremo il seguente NFA:



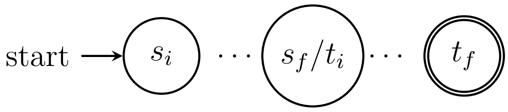
- Se stiamo visitando una foglia contente un simbolo $a \in \Sigma$, costruiremo il seguente NFA:



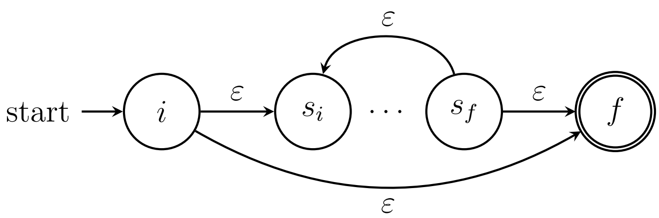
- Se stiamo visitando un nodo contente un'unione $s \cup t$, costruiremo il seguente NFA, aggiungendo gli stati i e f:



- Se stiamo visitando un nodo contente una concatenazione st, costruiremo il seguente NFA, unendo lo stato finale di s_f con lo stato iniziale di t_i:



- Se stiamo visitando un nodo contente una star di Kleene s^*, costruiremo il seguente NFA, aggiungendo gli stati i e f:



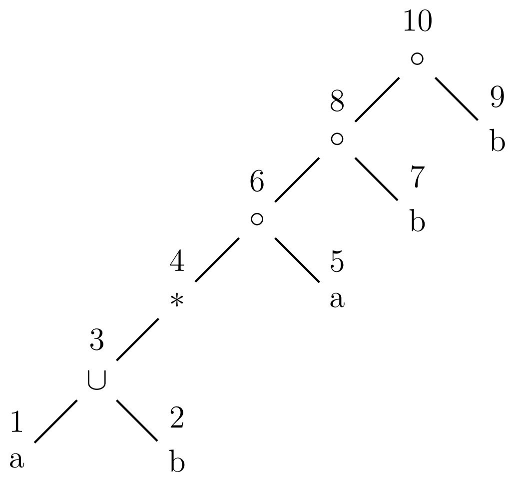
L'NFA risultante avrà un solo stato accettante e ogni stato (ad eccezione di quello accettante) avrà le transizioni uscenti in uno di questi due modi:

1. Una sola transizione con etichetta $a \in \Sigma$
2. Una o due transizioni con etichetta ε

Esempio:
Data l'espressione regolare $(a \cup b)^*abb$ con le seguenti precedenze e associatività (l'operazione più in alto ha precedenza):

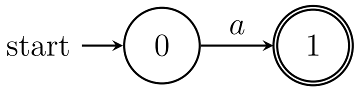
Operazione	Associatività
$*$	
$\cup$	Sinistra
$\circ$	Sinistra

Il Syntax Tree sarà ($\circ$ indica la concatenzazione):

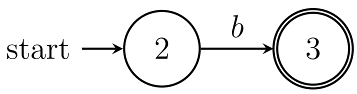


In questo esempio i nodi dell'albero sono numerati in base all'ordine in cui vanno visitati.
Eseguiamo i passi dell'algoritmo, partendo dalla foglia in basso a sinistra:

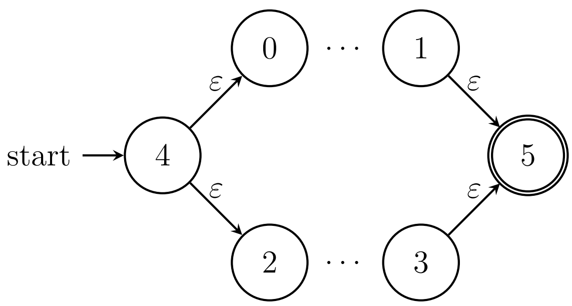
1. La foglia contiene a, quindi l'NFA corrispondente sarà:



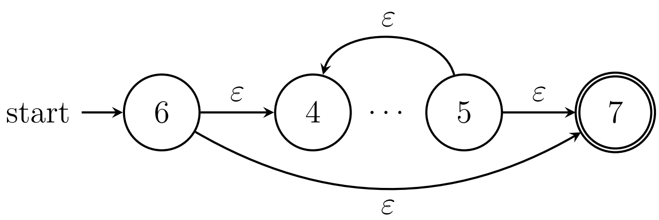
2. La foglia contiene b, quindi l'NFA corrispondente sarà:



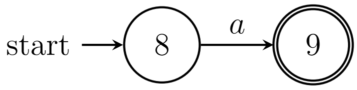
3. La foglia contiene un'unione, quindi l'NFA corrispondente a $a \cup b$ sarà:



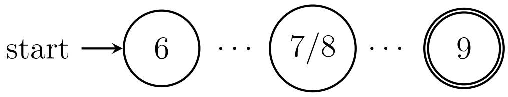
4. La foglia contiene una star di Kleene, quindi l'NFA corrispondente a $(a \cup b)^*$ sarà:



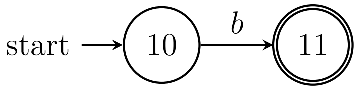
5. La foglia contiene a, quindi l'NFA corrispondente sarà:



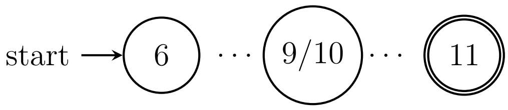
6. La foglia contiene una concatenazione, quindi l'NFA corrispondente a $(a \cup b)^*a$ sarà:



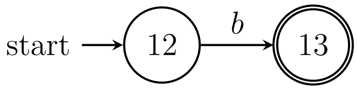
7. La foglia contiene b, quindi l'NFA corrispondente sarà:



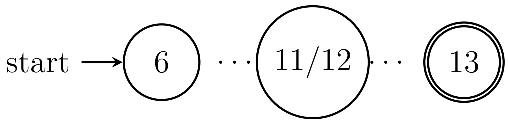
8. La foglia contiene una concatenazione, quindi l'NFA corrispondente a $(a \cup b)^*ab$ sarà:



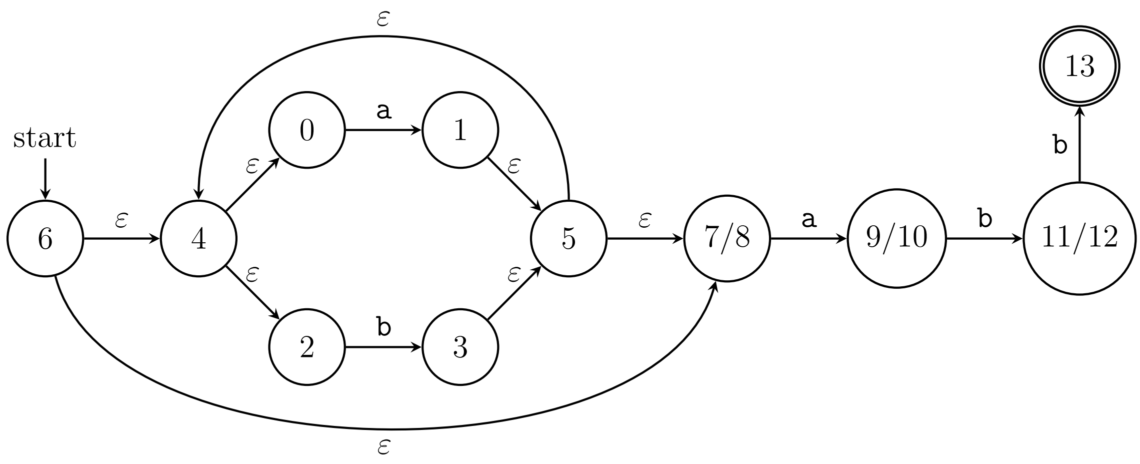
9. La foglia contiene b, quindi l'NFA corrispondente sarà:



10. La foglia contiene una concatenazione, quindi l'NFA corrispondente a $(a \cup b)^*abb$ sarà:



L'automa finale disegnato completamente sarà:



E.1.2 Trasformare un NFA in DFA

Dato un NFA $N = (Q_N, \Sigma, \delta_N, q_{0_N}, F_N)$ definiamo la ε-closure (o estensione) di uno stato $q \in Q_N$ come:

$$E(q) = \varepsilon\text{-closure}(q) = \left\{ s \in Q_N \,\middle|\, \begin{array}{c} s \text{ può essere raggiunto da } q \\ \text{tramite solamente } \varepsilon\text{-archi} \end{array} \right\}$$

La ε-closure di un insieme di stati $R \subseteq Q_N$ è definita come:

$$E(R) = \varepsilon\text{-closure}(R) = \bigcup_{q \in R} \varepsilon\text{-closure}(q)$$

La ε-closure di un insieme di stati R contiene sempre almeno R.
Per creare il DFA $D = (Q_D, \Sigma, \delta_D, q_{0_D}, F_D)$ equivalente all'NFA N, si esegue questo algoritmo:

Algoritmo: Subset Construction

```
def Subset_Construction(N):
    Q_D = {ε-closure(q_0N)}
    for R ∈ Q_D :  // anche quelli aggiunti durante il ciclo
        for a ∈ Σ :
            S = ε-closure(δ_N(R, a))
            if S ∉ Q_D :
                Aggiungo S a Q_D // Solitamente si numerano con A,B,...,Z
            Creo la transizione δ_D(R, a) = S
    return D
```

Quando uno stato $R \in Q_D$ è un insieme di stati in Q_N, la funzione δ_N viene calcolata come:

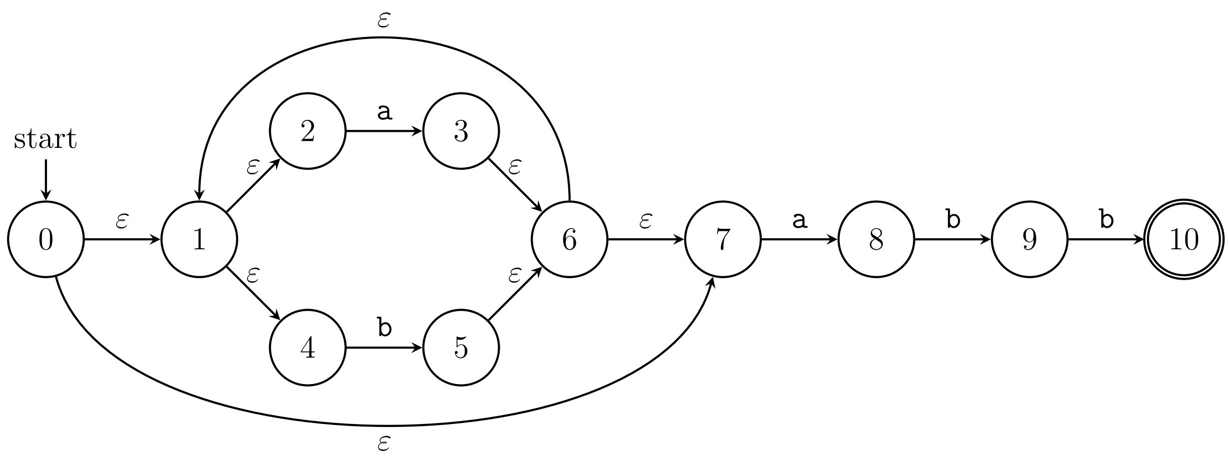
$$\delta_N(R, a) = \bigcup_{r \in R} \delta_N(r, a)$$

Gli stati accettanti in D sono tutti gli stati che contengono almeno uno stato accettante di N.
Il DFA risultante può essere disegnato oppure rappresentato come tabella con le intestazioni:

Stati NFA	Stato DFA	a	b
{0,1,2,3}	A	B	C
{1,2}	B	C	A
{3,4}	C	A	C

Con l'alfabeto $\Sigma = \{a, b\}$ la tabella avrebbe le intestazioni come sopra e la casella nella colonna a e riga A rappresenta la transizione $\delta_D(A, a)$. Nel caso l'alfabeto avesse altri simboli bisognerebbe aggiungere una colonna per ogni simbolo dell'alfabeto.

Esempio:
Dato l'NFA per il linguaggio $L = \{(a \cup b)^* abb\}$:



Eseguiamo i passaggi dell'algoritmo:

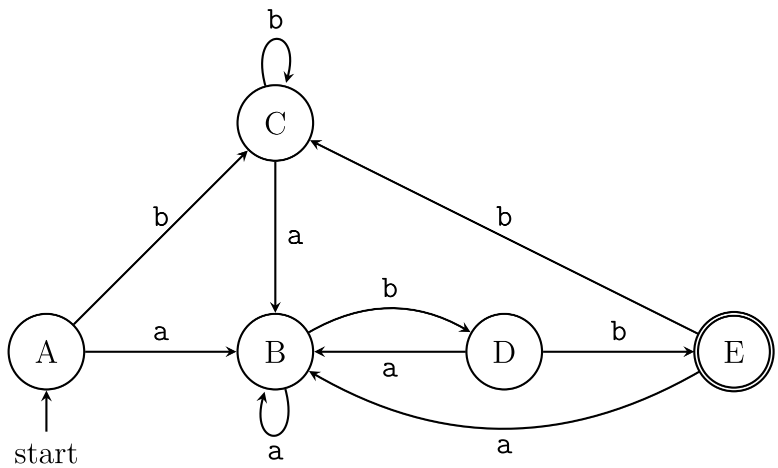
- Lo stato iniziale di D sarà ε-closure$(0) = \{0,1,2,4,7\} = A$
- Sullo stato A eseguiamo:
  1. ε-closure$(\delta_N(A,a)) = \varepsilon$-closure$(\{3,8\}) = \{1,2,3,4,6,7,8\} = B$
     Avremo una transizione $\delta_D(A,a) = B$
  2. ε-closure$(\delta_N(A,b)) = \varepsilon$-closure$(\{5\}) = \{1,2,4,5,6,7\} = C$
     Avremo una transizione $\delta_D(A,b) = C$
- Sullo stato B eseguiamo:
  1. ε-closure$(\delta_N(B,a)) = \varepsilon$-closure$(\{3,8\}) = B$
     Avremo una transizione $\delta_D(B,a) = B$
  2. ε-closure$(\delta_N(B,b)) = \varepsilon$-closure$(\{5,9\}) = \{1,2,4,5,6,7,9\} = D$
     Avremo una transizione $\delta_D(B,b) = D$
- Sullo stato C eseguiamo:
  1. ε-closure$(\delta_N(C,a)) = \varepsilon$-closure$(\{3,8\}) = B$
     Avremo una transizione $\delta_D(C,a) = B$
  2. ε-closure$(\delta_N(C,b)) = \varepsilon$-closure$(\{5\}) = C$
     Avremo una transizione $\delta_D(C,b) = C$
- Sullo stato D eseguiamo:
  1. ε-closure$(\delta_N(D,a)) = \varepsilon$-closure$(\{3,8\}) = B$
     Avremo una transizione $\delta_D(D,a) = B$
  2. ε-closure$(\delta_N(D,b)) = \varepsilon$-closure$(\{5,10\}) = \{1,2,4,5,6,7,10\} = E$
     Avremo una transizione $\delta_D(D,b) = E$

- Sullo stato E eseguiamo:

  1. ε-closure$(\delta_N(E, a)) = \varepsilon$-closure$(\{3, 8\}) = B$
     Avremo una transizione $\delta_D(E, a) = B$
  2. ε-closure$(\delta_N(E, b)) = \varepsilon$-closure$(\{5\}) = C$
     Avremo una transizione $\delta_D(E, b) = C$

- Abbiamo finito gli stati da analizzare quindi l'algoritmo è terminato e lo stato accettante di D sarà E perchè è l'unico che contiene lo stato 10 di N.

Il DFA risultante rappresentato sotto forma di automa sarà quindi:



Sotto forma di tabella invece sarà:

Stati NFA	Stato DFA	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,3,5,6,7,10}	E	B	C

E.1.3 Minimizzare un DFA

Dato un DFA $D = (Q, \Sigma, \delta, q_0, F)$, possiamo minimizzare il numero di stati, creando un DFA equivalente D_{min} seguendo un algoritmo basato sulle partizioni di Q:

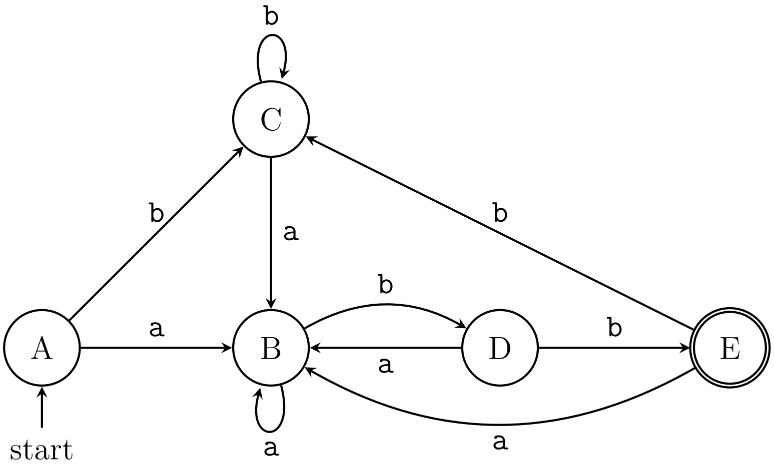
1. Partiamo con la partizione $\Pi = \{\{Q - F\}, \{F\}\}$
2. Per ogni insieme $P \in \Pi$ con $|P| \geq 2$ non ancora controllato (compresi quelli aggiunti durante il ciclo):

    2.1 Per ogni $a \in \Sigma$:

    - 2.1.1 Per ogni stato $q \in P$, controllo l'insieme P_i a cui appartiene lo stato r tale che $\delta(q, a) = r \in P_i$
    - 2.1.2 Se tutti gli stati $q \in P$ con input a non raggiungono lo stesso insieme P_i allora divido gli stati di P in base all'insieme che raggiungono
    - 2.1.3 Se tutti gli stati $q \in P$ con input a raggiungono lo stesso insieme P_i allora continuo con il prossimo insieme da controllare

3. Dopo aver controllato tutti gli insiemi ed aver trovato Π_{final}, per ogni insieme $P \in \Pi$ scelgo uno stato rappresentante.
4. Lo stato iniziale di D_{min} è lo stato rappresentante dell'insieme P per cui $s_0 \in P$
5. Gli stati finali di D_{min} sono tutti gli stati rappresentanti di insiemi P in cui esiste $q \in F$ tale che $q \in P$

Esempio:
Dato un DFA D non minimizzato:



Eseguiamo l'algoritmo:

1. $\Pi = \{\overbrace{\{A, B, C, D\}}^{S-F}, \overbrace{\{E\}}^{F}\}$
2. Per l'insieme $S - F$:

	a	b
A	$S-F$	$S-F$
B	$S-F$	$S-F$
C	$S-F$	$S-F$
D	$S-F$	F

   Divido l'insieme aggiungendo a Π i sottoinsiemi $A = \{A, B, C\}$ e $B = \{D\}$

3. $\Pi = \{\overbrace{\{A, B, C\}}^{A}, \overbrace{\{D\}}^{B}, \overbrace{\{E\}}^{F}\}$
4. Per l'insieme A:

	a	b
A	A	A
B	A	B
C	A	A

   Divido l'insieme aggiungendo a Π i sottoinsiemi $C = \{A, C\}$ e $D = \{B\}$

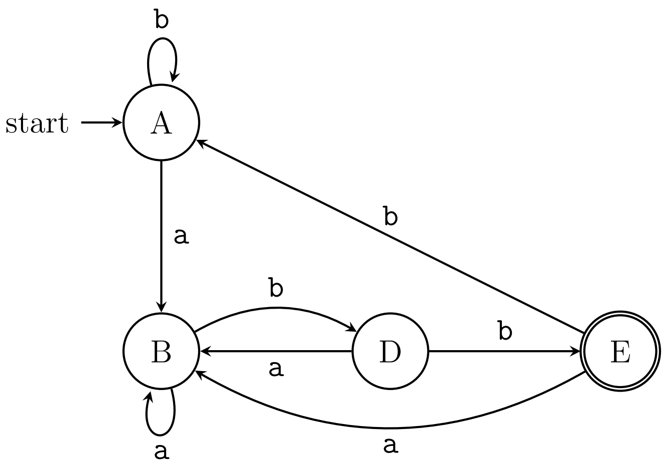
5. $\Pi = \{\overbrace{\{A, C\}}^{C}, \overbrace{\{B\}}^{D}, \overbrace{\{D\}}^{B}, \overbrace{\{E\}}^{F}\}$
6. Per l'insieme C:

	a	b
A	D	C
C	D	C

Non ci sono discordanze quindi non divido ulteriormente questo insieme.

7. Abbiamo finito gli insiemi da controllare e quindi $\Pi_{final} = \{\{A, C\}, \{B\}, \{D\}, \{E\}\}$ e nell'automa minimizzato:
   - Lo stato iniziale è il rappresentante dell'insieme $\{A, C\} = A$
   - Lo stato finale è il rappresentante dell'insieme {E}

Il DFA minimizzato D_{min} sarà:



E.2 Esercizi su grammatiche

E.2.1 Eliminare la ricorsione sinistra

Una grammatica $G = (V, \Sigma, R, S)$ ha una ricorsione sinistra se:

$$\exists A \in V | A \overset{+}{\Rightarrow} A\alpha \quad \alpha \in (V \cup \Sigma)^*$$

Il simbolo $\overset{+}{\Rightarrow}$ indica la derivazione in almeno un passo, se $A \Rightarrow A\alpha$ (in esattamente un passo) allora si dice che la grammatica ha una **ricorsione immediata a sinistra**.
Per eliminare una ricorsione immediata a sinistra nella forma:

$$\begin{aligned} &A \to A\alpha_1 | A\alpha_2 | \dots | A\alpha_n \\ &A \to \beta_1 | \dots | \beta_m \end{aligned}$$

Per cui $\forall i \ \alpha_i, \beta_i \in (V \cup \Sigma)^*$ e per cui il primo simbolo non è A.
Per eliminare la ricorsione immediata a sinistra modifico le regole facendo diventare la grammatica:

$$\begin{aligned} &A \to \beta_1 A' | \dots | \beta_m A' \\ &A' \to \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \varepsilon \end{aligned}$$

Per eliminare invece la ricorsione a sinistra generica, possiamo utilizzare un algoritmo se la grammatica segue due regole:

1. G non contiene cicli, cioè non esiste $A \overset{+}{\Rightarrow} A$
2. G non ha regole $A \to \varepsilon$ (in alcuni casi potrebbe non essere un problema)

L'algoritmo per eliminare la ricorsione sinistra è (considerando $n = |V|$):

Algoritmo: Eliminazione ricorsione a sinistra

```
def Elimina_Ricorsione(G):
    Ordina V
    for i in [1,n] :
        for j in [1,i-1] :
            Sostituisco le regole A_i → A_j α con A_i → βα per cui A_j → β
        Elimina ricorsione immediata a sinistra se esiste
    return G
```

Esempio:
Data la grammatica G con regole:

$$S \to Aa|b$$
$$A \to Ac|Sd|\varepsilon$$

Applichiamo l'algoritmo:

1. Ordiniamo $V = \{S, A\}$
2. Per S non esistono variabili precedenti e non c'è ricorsione immediata
3. Per A:
   3.1 Sostituiamo la regola $A \to Sd$ utilizzando le regole $S \to Aa|b$ aggiungendo le regole $A \to Aad|bd$, facendo diventare la grammatica:

   $$S \to Aa|b$$
   $$A \to Ac|Aad|bd|\varepsilon$$

   3.2 Eliminiamo la ricorsione immediata a sinistra togliendo le regole di A aggiungendo una variabile A' e le regole:

   $$A \to bdA'|A'$$
   $$A' \to cA'|adA'|\varepsilon$$

4. La gramatica diventa quindi:

$$S \to Aa|b$$
$$A \to bdA'|A'$$
$$A' \to cA'|adA'|\varepsilon$$

E.2.2 Fattorizzare una grammatica

Una grammatica $G = (V, \Sigma, R, S)$ è fattorizzabile a sinistra se ha delle regole nella forma:

$$A \to \alpha\beta_1|\alpha\beta_2$$

Per fattorizzarla possiamo usare l'algoritmo:

1. $\forall A \in V$:

   1.1 $P(A) = \{(A \to \alpha) \in R\}$

   1.2 Scelgo $P' \subseteq P(A)$ tale che (si possono scegliere anche più di un insieme contemporaneamente sulla stessa variabile A per eliminare tutti i prefissi):
   - Esiste un prefisso π comune a tutte le regole in P'
   - $\pi \neq \varepsilon$
   - $|P'| \geq 2$
   - Non esiste P'' tale che $P' \subset P''$ e P'' ha le stesse proprietà (1-3) sopra (il prefisso può essere sia il più lungo che quello che compare più volte, conviene considerare sempre il più lungo)

   1.3 Se P' non esiste passiamo alla prossima variabile

   1.4 Se P' esiste creiamo una nuova variabile $A' \notin V$ e sostituiamo le regole in P' aggiungendo le regole:

   $$\begin{aligned} &A \to \pi A' \\ &A' \to \beta_1|\dots|\beta_n \end{aligned}$$

   Per cui $\beta_1, \dots, \beta_n$ appartengono alle regole $A \to \pi\beta_i$.

2. Se nell'ultimo ciclo la grammatica è cambiata, rieseguiamo il ciclo sulle variabili (comprese quelle aggiunte nel ciclo precedente), sennò abbiamo la grammatica finale fattorizzata.

Esempio:
Data una grammatica non fattorizzata G:

$$A \to AaCb|AaCa|ABc|BCbB|BAC|BCbC|aa|a$$
$$B \to CbC|CAa|CbA|BAc|BAb|\varepsilon$$
$$C \to CBa|AaC|CBC|BaB|BaA|c$$

Per fattorizzarla eseguiamo l'algoritmo (per le variabili non scritte si sottointende che non esista un insieme $P' \subset P(A)$):

1. Per A:

   1.1 Prendiamo $P(A)$:

   $$A \to AaCb|AaCa|ABc|BCbB|BAC|BCbC|aa|a$$

   1.2 Scegliamo $\pi = AaC$ e modifichiamo le regole aggiungendo A':

   $$A \to AaCA'|ABc|BCbB|BAC|BCbC|aa|a$$
   $$A' \to b|a$$

   1.3 Scegliamo $\pi = BCb$ e modifichiamo le regole aggiungendo A'':

   $$A \to AaCA'|ABc|BCbA''|BAC|aa|a$$
   $$A'' \to B|C$$

   1.4 Scegliamo $\pi = a$ e modifichiamo le regole aggiungendo A''':

   $$A \to AaCA'|ABc|BCbA''|BAC|aA'''$$
   $$A''' \to a|\varepsilon$$

   1.5 Scegliamo $\pi = B$ e modifichiamo le regole aggiungendo A'''':

   $$A \to AaCA'|ABc|BA''''|aA'''$$
   $$A'''' \to CbA''|AC$$

   1.6 Scegliamo $\pi = A$ e modifichiamo le regole aggiungendo A''''':

   $$A \to AA'''''|BA''''|aA'''$$
   $$A''''' \to aCA'|Bc$$

2. Per B:

   2.1 Prendiamo $P(B)$:
   $$B \to CbC|CAa|CbA|BAc|BAb|\varepsilon$$

   2.2 Scegliamo $\pi = Cb$ e modifichiamo le regole aggiungendo B':

   $$B \to CbB'|CAa|BAc|BAb|\varepsilon$$
   $$B' \to C|A$$

2.3 Scegliamo $\pi = BA$ e modifichiamo le regole aggiungendo B'':

$$\begin{aligned} &B \to CbB'|CAa|BAB''|\varepsilon \\ &B'' \to c|b \end{aligned}$$

2.4 Scegliamo $\pi = C$ e modifichiamo le regole aggiungendo B''':

$$\begin{aligned} &B \to CB'''|BAB''|\varepsilon \\ &B''' \to bB'|Aa \end{aligned}$$

3. Per C:

3.1 Prendiamo $P(C)$:

$$C \to CBa|AaC|CBC|BaB|BaA|c$$

3.2 Scegliamo $\pi = CB$ e modifichiamo le regole aggiungendo C':

$$\begin{aligned} &C \to CBC'|AaC|BaB|BaA|c \\ &C' \to a|C \end{aligned}$$

3.3 Scegliamo $\pi = Ba$ e modifichiamo le regole aggiungendo C'':

$$\begin{aligned} &C \to CBC'|AaC|BaC''|c \\ &C'' \to B|A \end{aligned}$$

4. Nessun altra variabile ha delle regole con prefissi comuni, quindi l'algoritmo termina.

La grammatica fattorizzata ottenuta è:

$$\begin{aligned} &A \to AA'''''|BA''''|aA''' \\ &A' \to b|a \\ &A'' \to B|C \\ &A''' \to a|\varepsilon \\ &A'''' \to CbA''|AC \\ &A''''' \to aCA'|Bc \\ &B \to CB'''|BAB''|\varepsilon \\ &B' \to C|A \\ &B'' \to c|b \\ &B''' \to bB'|Aa \\ &C \to CBC'|AaC|BaC''|c \\ &C' \to a|C \\ &C'' \to B|A \end{aligned}$$

E.2.3 Calcolare FIRST e FOLLOW di variabili e stringhe

Data una grammatica G e una stringa $\alpha \in (V \cup \Sigma)^*$, si ha che:

$$\text{FIRST}(\alpha) = \{u \in \Sigma | \exists \beta \in (V \cup \Sigma)^* \quad \alpha \overset{*}{\Rightarrow} u\beta\}$$

Per poter calcolare l'insieme FIRST di una stringa, bisogna prima calcolare gli insiemi FIRST delle variabili e terminali che la compongono. Per un generico $X \in (V \cup \Sigma)$ si può calcolare FIRST(X) con l'algoritmo:

Algoritmo: Calcolo FIRST di una variabile o terminale

```
def Calc_FIRST(X):
    if X ∈ Σ :
        FIRST(X)={X}
    if X ∈ V :
        for (X → Y_1 ... Y_k) ∈ R :
            for a ∈ Σ :
                for i in [1,k] :
                    if a ∈ FIRST(Y_i) and ε ∈ FIRST(Y_1) ∩ ··· ∩ FIRST(Y_{i−1}) :
                        FIRST(X)+={a}
            if ε ∈ FIRST(Y_1) ∩ ··· ∩ FIRST(Y_k) :
                FIRST(X)+={ε}
    return FIRST(X)
```

Nel caso di una stringa $\alpha = X_1 \dots X_n$ si può calcolare FIRST(α) con l'algoritmo:

Algoritmo: Calcolo FIRST di una stringa

```
def Calc_FIRST(α):
    FIRST(α)=FIRST(X_1)−{ε}
    for i in [2,n] :
        if ε ∈ FIRST(X_{i−1}) :
            FIRST(α)+=FIRST(X_i)−{ε}
    if ε ∈ FIRST(X_n) :
        FIRST(α)+={ε}
    return FIRST(α)
```

Per una variabile $A \in V$, si ha che:

$$\text{FOLLOW}(A) = \{u \in \Sigma | \exists \alpha, \beta \in (V \cup \Sigma)^* \quad S \overset{*}{\Rightarrow} \alpha A u \beta\}$$

Si può calcolare l'insieme FOLLOW di una variabile con l'algoritmo:

Algoritmo: Calcolo FOLLOW di una variabile

```
def Calc_FOLLOW(A):
    FOLLOW(S)={$}
    while FOLLOW(A) viene modificato :
        for (A → αBβ) ∈ R :  // per una qualsiasi A
            FOLLOW(B)+=FIRST(β)−{ε}
            if ε ∈ FIRST(β) :
                FOLLOW(B)+=FOLLOW(A)
        for (A → αB) ∈ R :  // per una qualsiasi A
            FOLLOW(B)+=FOLLOW(A)
    return FOLLOW(A)
```

Esempio:
Data la grammatica G:

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE'|\varepsilon \\
&T \to FT' \\
&T' \to *FT'|\varepsilon \\
&F \to (E)|\text{id}
\end{aligned}
$$

Possiamo calcolare l'insieme FIRST delle variabili eseguendo l'algoritmo:

- $\text{FIRST}(F) = \{(, \text{id}\}$
- $\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \text{id})\}$
- $\text{FIRST}(E') = \{+, \varepsilon\}$
- $\text{FIRST}(T') = \{*, \varepsilon\}$

Ora eseguiamo l'algoritmo per calcolare l'insieme FOLLOW delle variabili (per tutte le operazioni non scritte si da per scontato che non modifichino nessun insieme):

1. $\text{FOLLOW}(E) = \{\$\}$
2. Ciclo 1:

    2.1 Per la regola $(E \to TE')$:

    2.1.1 $\text{FOLLOW}(T) += \text{FIRST}(E') - \{\varepsilon\}$:

    $$\text{FOLLOW}(T) = \{+\}$$

    2.1.2 $\varepsilon \in \text{FIRST}(E')$ quindi $\text{FOLLOW}(T) += \text{FOLLOW}(E)$:

    $$\text{FOLLOW}(T) = \{+, \$\}$$

    2.1.3 $\text{FOLLOW}(E') += \text{FOLLOW}(E)$:

    $$\text{FOLLOW}(E') = \{\$\}$$

    2.2 Per la regola $(E' \to +TE')$:

    2.2.1 Non effettua nessun cambiamento negli insiemi FOLLOW

    2.3 Per la regola $(T \to FT')$:

    2.3.1 $\text{FOLLOW}(F) += \text{FIRST}(T') - \{\varepsilon\}$:

    $$\text{FOLLOW}(F) = \{*\}$$

    2.3.2 $\varepsilon \in \text{FIRST}(T')$ quindi $\text{FOLLOW}(F) += \text{FOLLOW}(T)$:

    $$\text{FOLLOW}(F) = \{*, +, \$\}$$

    2.3.3 $\text{FOLLOW}(T') += \text{FOLLOW}(T)$:

    $$\text{FOLLOW}(T') = \{+, \$\}$$

2.4 Per la regola ($T' \to *FT'$):

2.4.1 Non effettua nessun cambiamento negli insiemi FOLLOW

2.5 Per la regola ($F \to (E)$):

2.5.1 FOLLOW(E)+ = FIRST(")") − {ε}:

$$\text{FOLLOW}(E) = \{\$,)\}$$

3. Ciclo 2:

3.1 Per la regola ($E \to TE'$):

3.1.1 $\varepsilon \in$ FIRST(E') quindi FOLLOW(T)+ = FOLLOW(E):

$$\text{FOLLOW}(T) = \{+, \$,)\}$$

3.1.2 FOLLOW(E')+ = FOLLOW(E):

$$\text{FOLLOW}(E') = \{\$,)\}$$

3.2 Per la regola ($E' \to +TE'$):

3.2.1 Non effettua nessun cambiamento negli insiemi FOLLOW

3.3 Per la regola ($T \to FT'$):

3.3.1 $\varepsilon \in$ FIRST(T') quindi FOLLOW(F)+ = FOLLOW(T):

$$\text{FOLLOW}(F) = \{*, +, \$,)\}$$

3.3.2 FOLLOW(T')+ = FOLLOW(T):

$$\text{FOLLOW}(T') = \{+, \$,)\}$$

3.4 Per la regola ($T' \to *FT'$):

3.4.1 Non effettua nessun cambiamento negli insiemi FOLLOW

4. Ciclo 3:

4.1 Nessuna regola effettua cambiamenti negli insiemi FOLLOW

5. L'ultimo ciclo non ha fatto cambiamenti agli insiemi FOLLOW quindi l'algoritmo termina

Gli insiemi FOLLOW delle variabili saranno quindi:

- FOLLOW(F) = $\{*, +, \$,)\}$
- FOLLOW(E) = FOLLOW(E') = $\{\$,)\}$
- FOLLOW(T) = FOLLOW(T') = $\{+, \$,)\}$

E.2.4 Creare un'automa LR(0) per una grammatica

Data una grammatica G aumentata, cioè per cui esiste solo una regola iniziale $S' \to S$, definiamo un **item LR(0)** come una coppia:

$$(A \to \beta, p)$$

in cui $(A \to \beta) \in R$ e p è una posizione in β. Viene rappresentato graficamente come $A \to \beta_1 \cdot \beta_2$ in cui il punto indica la posizione p.
Dato un insieme di item I, definiamo CLOSURE(I):

Algoritmo: Calcolo CLOSURE di un insieme di item

```
def Calc_CLOSURE(I):
    while CLOSURE(I) viene modificato :
        for (A → α · Bβ) ∈ CLOSURE(I) :  // per una qualsiasi A
            for (B → γ) ∈ R :
                if (B → ·γ) ∉ CLOSURE(I) :
                    CLOSURE(I)+={B → ·γ}
    return CLOSURE(I)
```

Dato un insieme di item I e $X \in V \cup \Sigma$, definiamo GOTO(I, X):

$$\text{GOTO}(I, X) = \text{CLOSURE}(\{A \to \alpha X \cdot \beta | (A \to \alpha \cdot X\beta) \in I\})$$

Date le due definizioni sopra per creare l'automa LR(0) della grammatica G si usa il seguente algoritmo:

Algoritmo: Calcolo stati dell'automa LR(0)

```
def Calc_LR0(G):
    C = {CLOSURE({S' → ·S})}
    while C viene modificato :
        for I in C :
            for X ∈ V ∪ Σ :
                C+=GOTO(I, X)
    return C
```

Ogni insieme I contenuto in C rappresenta uno stato dell'automa.

Esempio:
Data la grammatica G:

$$\begin{aligned} &S' \to S \\ &S \to AB|aA \\ &A \to bB|cAb|a \\ &B \to Bb|BA|b \end{aligned}$$

Eseguiamo l'algoritmo:

1. $I_0 = \text{CLOSURE}(\{S' \to \cdot S\})$:

$$\begin{aligned} &S' \to \cdot S \\ &S \to \cdot AB| \cdot aA \\ &A \to \cdot bB| \cdot cAb| \cdot a \end{aligned}$$

2. Per I_0:

   2.1 $\text{GOTO}(I_0, S) = I_1$:

$$S' \to S \cdot$$

   2.2 $\text{GOTO}(I_0, A) = I_2$:

$$\begin{aligned} &S \to A \cdot B \\ &B \to \cdot Bb| \cdot BA| \cdot b \end{aligned}$$

   2.3 $\text{GOTO}(I_0, a) = I_3$:

$$\begin{aligned} &S \to a \cdot A \\ &A \to a \cdot | \cdot bB| \cdot cAb| \cdot a \end{aligned}$$

   2.4 $\text{GOTO}(I_0, b) = I_4$:

$$\begin{aligned} &A \to b \cdot B \\ &B \to \cdot Bb| \cdot BA| \cdot b \end{aligned}$$

   2.5 $\text{GOTO}(I_0, c) = I_5$:

$$A \to c \cdot Ab| \cdot bB| \cdot cAb| \cdot a$$

3. Per I_2:

   3.1 $\text{GOTO}(I_2, B) = I_6$:

$$\begin{aligned} &S \to AB \cdot \\ &A \to \cdot bB| \cdot cAb| \cdot a \\ &B \to B \cdot b|B \cdot A \end{aligned}$$

3.2 GOTO$(I_2, b) = I_7$:

$$B \to b\cdot$$

4. Per I_3:

4.1 GOTO$(I_3, A) = I_8$:

$$S \to aA\cdot$$

4.2 GOTO$(I_3, a) = I_9$:

$$A \to a\cdot$$

4.3 GOTO$(I_3, b) = I_4$

4.4 GOTO$(I_3, c) = I_5$

5. Per I_4:

5.1 GOTO$(I_4, B) = I_{10}$:

$$\begin{aligned} &A \to bB \cdot | \cdot bB| \cdot cAb| \cdot a \\ &B \to B \cdot b|B \cdot A \end{aligned}$$

5.2 GOTO$(I_4, b) = I_7$

6. Per I_5:

6.1 GOTO$(I_5, A) = I_{11}$:

$$A \to cA \cdot b$$

6.2 GOTO$(I_5, a) = I_9$

6.3 GOTO$(I_5, b) = I_4$

6.4 GOTO$(I_5, c) = I_5$

7. Per I_6:

7.1 GOTO$(I_6, A) = I_{12}$:

$$B \to BA\cdot$$

7.2 GOTO$(I_6, a) = I_9$

7.3 GOTO$(I_6, b) = I_{13}$:

$$\begin{aligned} &A \to b \cdot B \\ &B \to Bb \cdot | \cdot Bb| \cdot BA| \cdot b \end{aligned}$$

7.4 GOTO$(I_6, c) = I_5$

8. Per I_{10}:
   8.1 GOTO$(I_{10}, A) = I_{12}$
   8.2 GOTO$(I_{10}, a) = I_9$
   8.3 GOTO$(I_{10}, b) = I_{13}$
   8.4 GOTO$(I_{10}, c) = I_5$
9. Per I_{11}:
   9.1 GOTO$(I_{11}, b) = I_{14}$:
   $$A \to cAb\cdot$$
10. Per I_{13}:
    10.1 GOTO$(I_{13}, B) = I_{10}$
    10.2 GOTO$(I_{13}, b) = I_7$

E.3 Esercizi su blocchi di codice

E.3.1 Verificare se un flow graph è riducibile

Dato un flow graph e due nodi u, v diciamo che u **domina** v se:

- Per ogni cammino dal primo nodo e a v bisogna per forza passare per u

Definiamo quindi gli insiemi:

$$\text{DOM}(v) = \{u \in V | u \text{ domina } v\}$$
$$\text{IDOM}(v) = \{u \in \text{DOM}(v) - \{v\} | \min(\text{dist}(u, v)\ \forall u)\}$$

Ogni insieme DOM(u) contiene anche u stesso e e (primo nodo del grafo) è contenuto in tutti gli insiemi DOM.
Per calcolare gli insiemi DOM per ogni nodo di un grafo si può usare questo algoritmo in cui e è il primo nodo del grafo, n il numero di nodi e pred(v) l'insieme di tutti i vicini entranti di v:

Algoritmo: Calcolo DOM di un flow graph

```
def Calc_DOM(G):
    DOM(e)={e}
    for v ∈ V − {e} :
        DOM(v)={1, ..., n}
    while un qualsiasi DOM viene modificato :
        for v ∈ V − {e} :
            DOM(v)=( ⋂_{p∈pred(v)} DOM(p) ) ∪ {n}
```

Definiamo inoltre la **dominance frontier** di un nodo v, cioè l'insieme di tutti i nodi u per cui:

1. $v = u$ oppure $v \notin \text{DOM}(u)$
2. esiste $x \in \text{pred}(u)$ per cui v domina x

Per calcolare la DOMFRONT per ogni nodo si può usare questo algoritmo:

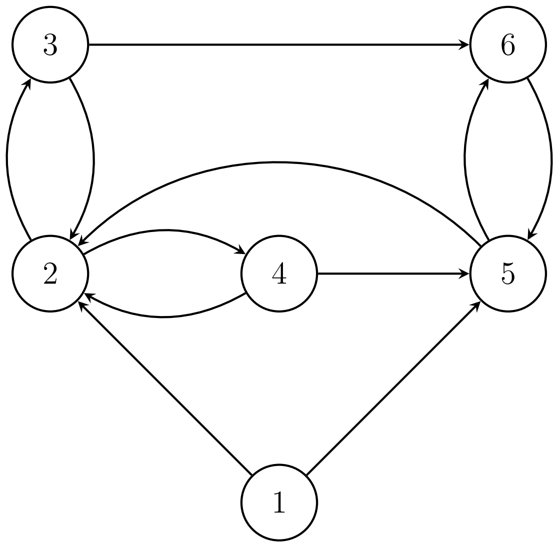
Algoritmo: Calcolo DOMFRONT di un flow graph

```
def Calc_DOMFRONT(G):
    for v ∈ V :
        if |pred(v)| ≥ 2 :
            for p ∈ pred(v) :
                while p!=IDOM(v) :
                    DOMFRONT(p)+={v}
                    p = IDOM(p)
```

Una volta calcolati i tre insiemi precedenti possiamo verificare se il flow graph F sia riducibile tramite una serie di passaggi:

1. Eseguiamo una DFS su F
2. Eliminiamo da F tutti i **back edge**, ovvero gli archi all'indietro (u, v) in cui $v \in \mathrm{DOM}(u)$
3. Se in F esistono ancora archi all'indietro allora F non è riducibile, altrimenti lo è

Esempio:
Dato il seguente flow graph F:



Eseguiamo l'algoritmo per calcolare gli insiemi DOM:

1. $\mathrm{DOM}(1) = \{1\}$
2. Ciclo 1:

   2.1 $\mathrm{DOM}(2) = (\mathrm{DOM}(1) \cap \mathrm{DOM}(3) \cap \mathrm{DOM}(4) \cap \mathrm{DOM}(5)) \cup \{2\} = \{1, 2\}$
   2.2 $\mathrm{DOM}(3) = \mathrm{DOM}(2) \cup \{3\} = \{1, 2, 3\}$
   2.3 $\mathrm{DOM}(4) = \mathrm{DOM}(2) \cup \{4\} = \{1, 2, 4\}$
   2.4 $\mathrm{DOM}(5) = (\mathrm{DOM}(1) \cap \mathrm{DOM}(4) \cap \mathrm{DOM}(6)) \cup \{5\} = \{1, 5\}$
   2.5 $\mathrm{DOM}(6) = (\mathrm{DOM}(3) \cap \mathrm{DOM}(5)) \cup \{6\} = \{1, 6\}$

Gli IDOM di conseguenza saranno:

- $\mathrm{IDOM}(1) = 1$
- $\mathrm{IDOM}(2) = 1$
- $\mathrm{IDOM}(3) = 2$
- $\mathrm{IDOM}(4) = 2$
- $\mathrm{IDOM}(5) = 1$
- $\mathrm{IDOM}(6) = 1$

Eseguiamo l'algoritmo per calcolare gli insiemi DOMFRONT:

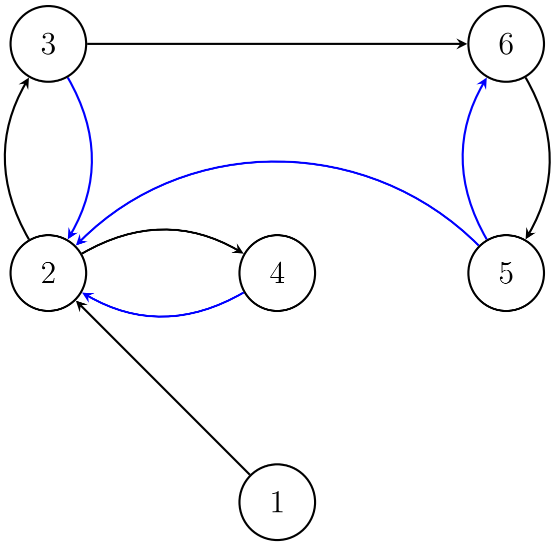
1. 1 non ha vicini entranti
2. Per 2:
    2.1 Preso 3:
        2.1.1 DOMFRONT(3) = $\{2\}$
        2.1.2 DOMFRONT(2) = $\{2\}$
    2.2 Preso 4:
        2.2.1 DOMFRONT(4) = $\{2\}$
    2.3 Preso 5:
        2.3.1 DOMFRONT(5) = $\{2\}$
3. 3 ha solo 2 come vicino entrante
4. 4 ha solo 2 come vicino entrante
5. Per 5:
    5.1 Preso 4:
        5.1.1 DOMFRONT(4) = $\{2, 5\}$
        5.1.2 DOMFRONT(2) = $\{2, 5\}$
    5.2 Preso 6:
        5.2.1 DOMFRONT(6) = $\{5\}$
6. Per 6:
    6.1 Preso 3:
        6.1.1 DOMFRONT(3) = $\{2, 6\}$
        6.1.2 DOMFRONT(2) = $\{2, 5, 6\}$
    6.2 Preso 5:
        6.2.1 DOMFRONT(5) = $\{2, 6\}$

Gli insiemi DOMFRONT saranno quindi:

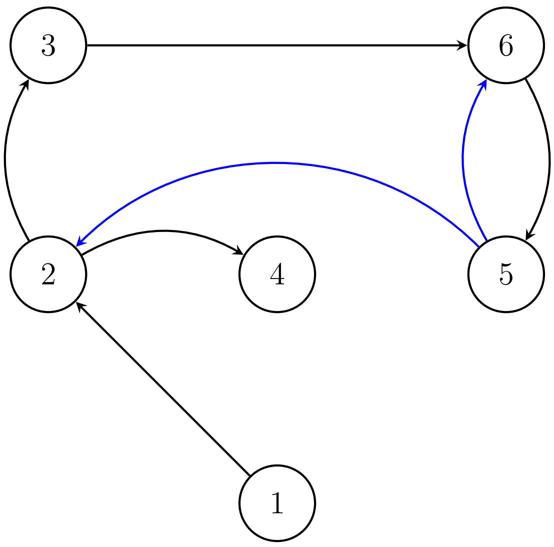
- DOMFRONT(1) = $\{\}$
- DOMFRONT(2) = $\{2, 5, 6\}$
- DOMFRONT(3) = $\{2, 6\}$
- DOMFRONT(4) = $\{2, 5\}$
- DOMFRONT(5) = $\{2, 6\}$
- DOMFRONT(6) = $\{5\}$

Ora dobbiamo verificare se il flow graph F è riducibile:

1. Eseguiamo una DFS su F e otteniamo la seguente arborescenza (in blu sono segnati gli archi all'indientro in F):



2. Eliminiamo gli archi all'indietro (u, v) in cui $v \in \mathrm{DOM}(u)$, ottenendo il grafo:



3. Essendo rimasti degli archi all'indietro nel grafo, allora F non è riducibile.

E.3.2 Creare e colorare un interference graph

Dato un programma P, definiamo un **interference graph** un grafo non orientato in cui:

- Ogni vertice v corrispende ad una variabile in P
- c'è un arco (u, v) se una determinata definizione di u è viva subito dopo una definizione di v, cioè la definizione di u viene utilizzata successivamente alla definizione di v senza essere ridefinita prima

Per creare l'interference graph G si utilizza la seguente procedura:

1. Presa ogni definizione x di una variabile u
2. Scorriamo tutte le definizione successive fino alla prossima definizione di u (se non c'è fino alla fine del programma)
3. Se la definizione attuale di u viene ancora utilizzata dopo la definizione di questa altra variabile v, aggiungiamo un arco (u, v) a G

Per calcolare se sia possibile usare k registri per eseguire questo codice dobbiamo controllare se il grafo G è k-colorabile, per farlo utilizziamo un algoritmo di colorazione parziale (essendo il problema NP-Completo):

1. Definiamo il grafo iniziale come G_0
2. Finchè G_i non è vuoto:
   - 2.1 Se esiste un nodo v con meno di k vicini lo eliminiamo da G_i, creando il grafo $G_{i+1} = G_i - \{v\}$
   - 2.2 Se tutti i nodi hanno più o uguale a k vicini, scegliamo un nodo a caso v e lo contrassegnamo come `ToSpill` e lo eliminiamo da G_i, creando $G_{i+1} = G_i - \{v\}$
3. Consideriamo i grafi G_i e i vertici v_i partendo dagli ultimi:
   - 3.1 Se v_i è contrassegnato come `ToSpill` lo ignoriamo
   - 3.2 Sennò assegnamo a v_i un colore non assegnato a nessuno dei suoi vicini in G_i

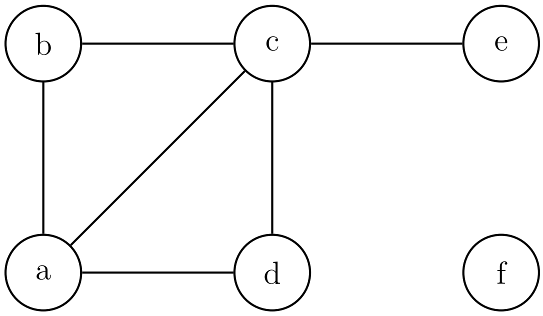
Esempio:
Dato il programma P:

```
a=12
b=2
c=a*b
d=b+9
e=a+8
f=c+3
```

Eseguiamo l'algoritmo per creare l'interference graph:

1. Presa la definizione `a=12`:
   - 1.1 Controlliamo la definizione `b=2`, dopo viene ancora utilizzata la definizione di a, quindi aggiungiamo un arco (a, b)
   - 1.2 Controlliamo la definizione `c=a*b`, dopo viene ancora utilizzata la definizione di a, quindi aggiungiamo un arco (a, c)
   - 1.3 Controlliamo la definizione `d=b+9`, dopo viene ancora utilizzata la definizione di a, quindi aggiungiamo un arco (a, d)
   - 1.4 Controlliamo la definizione `e=a+8`, dopo non viene più utilizzata la definizione di a
2. Presa la definiamo `b=2`:
   - 2.1 Controlliamo la definizione `c=a*b`, dopo viene ancora utilizzata la definizione di b, quindi aggiungiamo un arco (b, c)
   - 2.2 Controlliamo la definizione `d=b+9`, dopo non viene più utilizzata la definizione di b
3. Presa la definizione `c=a*b`:
   - 3.1 Controlliamo la definizione `d=b+9`, dopo viene ancora utilizzata la definizione di c, quindi aggiungiamo un arco (c, d)
   - 3.2 Controlliamo la definizione `e=a+8`, dopo viene ancora utilizzata la definizione di c, quindi aggiungiamo un arco (c, e)
   - 3.3 Controlliamo la definizione `f=c+3`, dopo non viene più utilizzata la definizione di c
4. Presa la definizione `d=a*b`, non viene mai usata
5. Presa la definizione `e=a+8`, non viene mai usata
6. Presa la definizione `f=c+3`, non viene mai usata

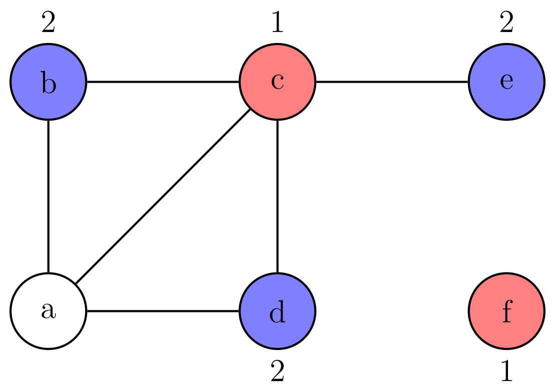
L'interference graph finale quindi sarà:



Ora eseguiamo l'algoritmo di colorazione parziale con $k = 2$ (variabile in input di esempio):

1. Finchè G non è vuoto:
   1.1 Eliminiamo $v_1 = f$, creando G_1
   1.2 Eliminiamo $v_2 = e$, creando G_2
   1.3 Contrassegnamo $v_3 = a$ come `ToSpill`, creando il grafo G_3
   1.4 Eliminiamo $v_4 = b$, creando G_4
   1.5 Eliminiamo $v_5 = d$, creando G_5
   1.6 Eliminiamo $v_6 = c$, creando G_6
2. Prendiamo in ordine inverso i v_i e G_i:
   2.1 Assegnamo $col(c) = 1$
   2.2 Assegnamo $col(d) = 2$
   2.3 Assegnamo $col(b) = 2$
   2.4 Saltiamo a
   2.5 Assegnamo $col(e) = 2$
   2.6 Assegnamo $col(f) = 1$

Il grafo finale parzialmente colorato sarà:



E.3.3 Generare codice tramite gli Ershov Number

Dato un programma P, per calcolare gli **Ershov Number** dobbiamo prima creare un albero. Questo si può fare tramite un'operazione partendo dall'ultima riga del programma:

- Presa l'ultima definizione `a=b op c`, creaiamo il nodo `a` come radice dell'albero e `b` e `c` come figli.
- Scorriamo le definizioni dal basso verso l'alto
- Presa una definizione `a=b op c`, aggiungiamo `b` e `c` come figli di `a`

Dato l'albero assegnamo gli Ershov Number $\mathcal{N}_e$ ad ogni nodo v come segue:

- Se v è una foglia, allora $\mathcal{N}_e(v) = 1$
- Se v ha un solo figlio u allora $\mathcal{N}_e(v) = \mathcal{N}_e(u)$
- Se v ha due figli x e y, ci sono due casi possibili:
  - Se $\mathcal{N}_e(x) = \mathcal{N}_e(y)$ allora $\mathcal{N}_e(v) = \mathcal{N}_e(x) + 1$ (non cambia tra x e y essendo uguali)
  - Se $\mathcal{N}_e(x) \neq \mathcal{N}_e(y)$ allora $\mathcal{N}_e(v) = \max(\mathcal{N}_e(x), \mathcal{N}_e(y))$

Dall'albero possiamo applicare 2 algoritmi che generano del codice. Il primo è `Ershov(v,b)` in cui `v` è un nodo e `b` è il primo registro da cui partire:

1. Se v è una foglia che contiene una variabile `x`, generiamo il codice:
   LD $\mathtt{R_b}$, x
2. Se v ha due figli x e y per cui $\mathcal{N}_e(x) = \mathcal{N}_e(y) = k - 1$, allora chiamiamo ricorsivamente:
   2.1 `Ershov(y,b+1)`
   2.2 `Ershov(x,b)`
   2.3 Data l'operazione `op` di v, generiamo il codice:
   OP $\mathtt{R_{b+k-1}}$, $\mathtt{R_{b+k-2}}$, $\mathtt{R_{b+k-1}}$
3. Se v ha due figli x e y per cui $\mathcal{N}_e(x) \neq \mathcal{N}_e(y)$, chiamiamo *max* il figlio con $\mathcal{N}_e$ massimo e *min* l'altro per cui $\mathcal{N}_e(min) = m$. Chiamiamo ricorsivamente:
   3.1 `Ershov(max,b)`
   3.2 `Ershov(min,b)`
   3.3 Data l'operazione `op` di v, se *max* è il figlio destro generiamo il codice:
   OP $\mathtt{R_{b+k-1}}$, $\mathtt{R_{b+m-1}}$, $\mathtt{R_{b+k-1}}$
   3.4 Data l'operazione `op` di v, se *max* è il figlio sinistro generiamo il codice:
   OP $\mathtt{R_{b+k-1}}$, $\mathtt{R_{b+k-1}}$, $\mathtt{R_{b+m-1}}$

Se il numero di registri disponibili è minore dell'$\mathcal{N}_e$ della radice dell'albero, bisogna usare un altro algoritmo `ErshovGen(v,r,b)` in cui `r` è il numero di registri disponibili:

1. Prima dei 3 casi dell'algoritmo precedente, viene aggiunto un caso 0
2. Se $\mathcal{N}_e(v) = k > r$ e v ha due figli x e y, chiamiamo *max* il figlio con $\mathcal{N}_e$ massimo e *min* l'altro (se sono uguali è indifferente). Eseguiamo:

    2.1 `ErshovGen(max,r,1)`

    2.2 Generiamo il codice:
    `ST` $\mathtt{s_k}$ `,` $\mathtt{R_r}$

    2.3 Se $\mathcal{N}_e(min) = j \geq r$, allora chiamiamo `ErshovGen(min,r,1)`

    2.4 Se invece $\mathcal{N}_e(min) = j < r$, allora chiamiamo `ErshovGen(min,r,r-j+1)`

    2.5 Generiamo il codice:
    `LD` $\mathtt{R_{r-1}}$ `,` $\mathtt{s_k}$

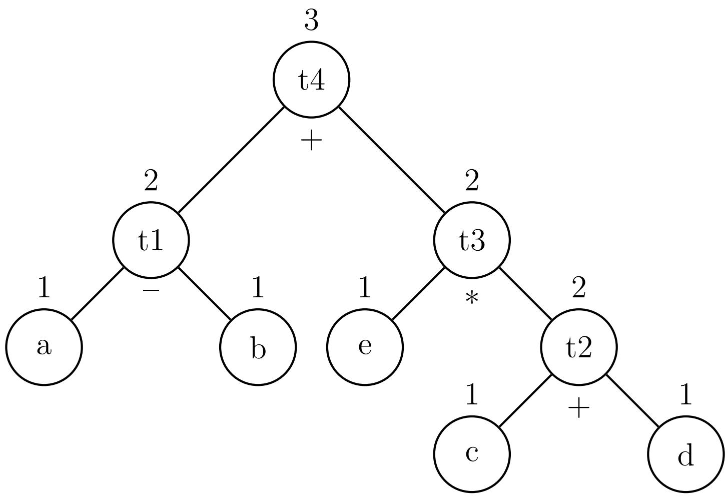
    2.6 Data l'operazione `op` di v, se *max* è il figlio destro generiamo il codice:
    `OP` $\mathtt{R_r}$ `,` $\mathtt{R_r}$ `,` $\mathtt{R_{r-1}}$

    2.7 Data l'operazione `op` di v, se *max* è il figlio sinistro generiamo il codice:
    `OP` $\mathtt{R_r}$ `,` $\mathtt{R_{r-1}}$ `,` $\mathtt{R_r}$

Esempio:
Dato il programma P:

```
t1=a-b
t2=c+d
t3=e*t2
t4=t1+t3
```

L'albero corrispondente a questo programma è:



Eseguiamo ora l'algoritmo `Ershov(t4,1)`:

1. `Ershov(t4,1)`: `t4` ha due figli e $\mathcal{N}_e(\texttt{t3}) = \mathcal{N}_e(\texttt{t1}) = 2$
2. `Ershov(t3,2)`: `t3` ha due figli e $\mathcal{N}_e(\texttt{e}) = 1 \neq \mathcal{N}_e(\texttt{t2}) = 2$
3. `Ershov(t2,2)`: `t2` ha due figli e $\mathcal{N}_e(\texttt{c}) = \mathcal{N}_e(\texttt{d}) = 1$
4. `Ershov(d,3)`: `d` è una foglia quindi viene generato il codice:
   `LD R3, d`
5. `Ershov(c,2)`: `c` è una foglia quindi viene generato il codice:
   `LD R2, c`
6. `Ershov(t2,2)`: viene generato il codice:
   `ADD R3, R2, R3`
7. `Ershov(e,2)`: `e` è una foglia quindi viene generato il codice:
   `LD R2, e`
8. `Ershov(t3,2)`: viene generato il codice:
   `MUL R3, R2, R3`
9. `Ershov(t1,1)`: `t1` ha due figli e $\mathcal{N}_e(\texttt{a}) = \mathcal{N}_e(\texttt{b}) = 1$
10. `Ershov(b,2)`: `b` è una foglia quindi viene generato il codice:
    `LD R2, b`
11. `Ershov(a,1)`: `a` è una foglia quindi viene generato il codice:
    `LD R1, a`

12. `Ershov(t1,1)`: viene generato il codice:
    `SUB R2, R1, R2`

13. `Ershov(t4,1)`: viene generato il codice:
    `ADD R3, R2, R3`

Il codice generato è:

```
LD R3,d
LD R2,c
ADD R3,R2,R3
LD R2,e
MUL R3,R2,R3
LD R2,b
LD R1,a
SUB R2,R1,R2
ADD R3,R2,R3
```

Se usassimo `ErshovGen(t4,2,1)` il codice generato sarebbe:

```
LD R2,d
LD R1,c
ADD R2,R1,R2
LD R1,e
MUL R2,R1,R2
ST s3,R2
LD R2,b
LD R1,a
SUB R2,R1,R2
LD R1,s3
ADD R2,R2,R1
```