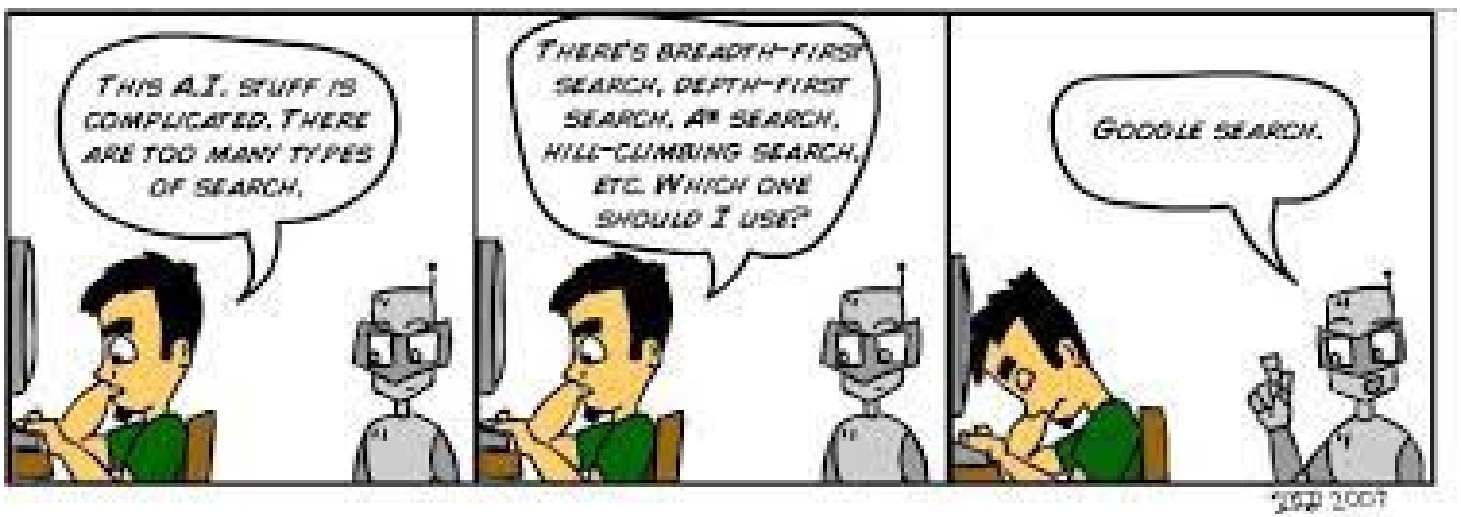
Algoritmi e strutture dati

Grafi: visita in profondità e problemi collegati

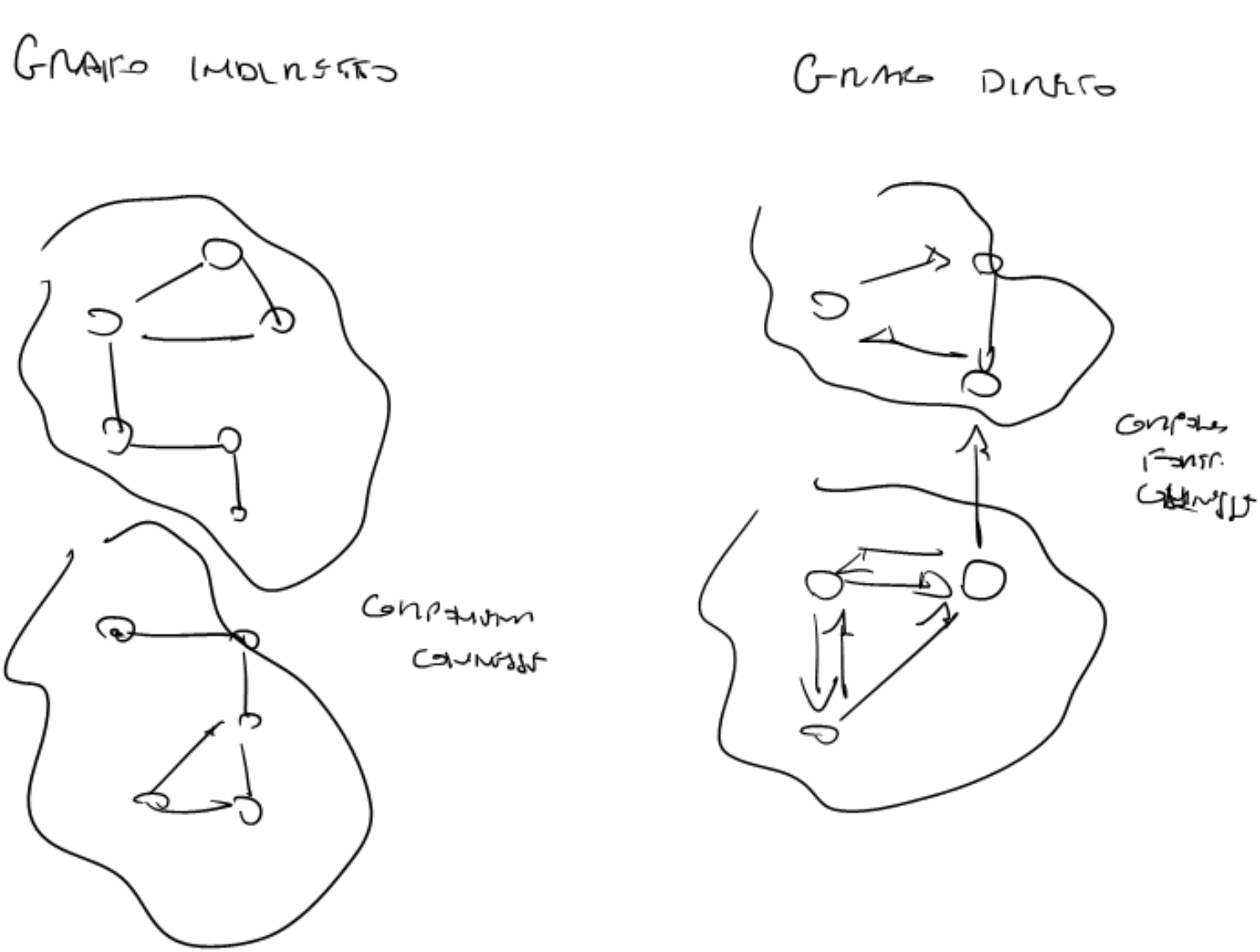


Menú di questa lezione

In questa lezione introduciamo la visita di grafi in profondità, specialmente per grafi diretti, e ne vediamo alcune applicazioni.

Grafi: visita in profonditá

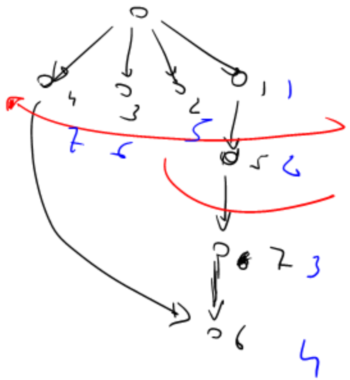
Consideriamo, di nuovo, un grafo G. Come nel caso della visita in ampiezza, è indifferente se G è o no pesato (i pesi non vengono considerati), o se G è o no diretto. Ma se $G = (V, E)$ è, in effetti, diretto, allora visitarlo in maniera diversa da quella vista precedentemente può fornire informazioni utili. Ci proponiamo, in particolare, di risolvere i seguenti tre problemi (definiti solo su grafi diretti): stabilire se G è **ciclico**: stabilire se contiene almeno un ciclo (in realtà questo problema è definito anche su grafi indiretti); costruire un **ordinamento topologico** di G: elencare tutti i suoi vertici in un ordine qualsiasi tale che ogni vertice v è elencato solo se tutti i vertici **dai quali** v si puó raggiungere sono stati elencati prima; e conoscere ed enumerare tutte le **componenti fortemente connesse** di G: elencare tutti i sottoinsiemi massimali di V tali che, ogni vertice in ogni sottoinsieme raggiunge ogni altro vertice in quel sottoinsieme. Le soluzioni a questi problemi hanno in comune la stessa procedura: la visita in profondità, probabilmente introdotta da Charles Pierre Tremaux, nel 19 secolo.



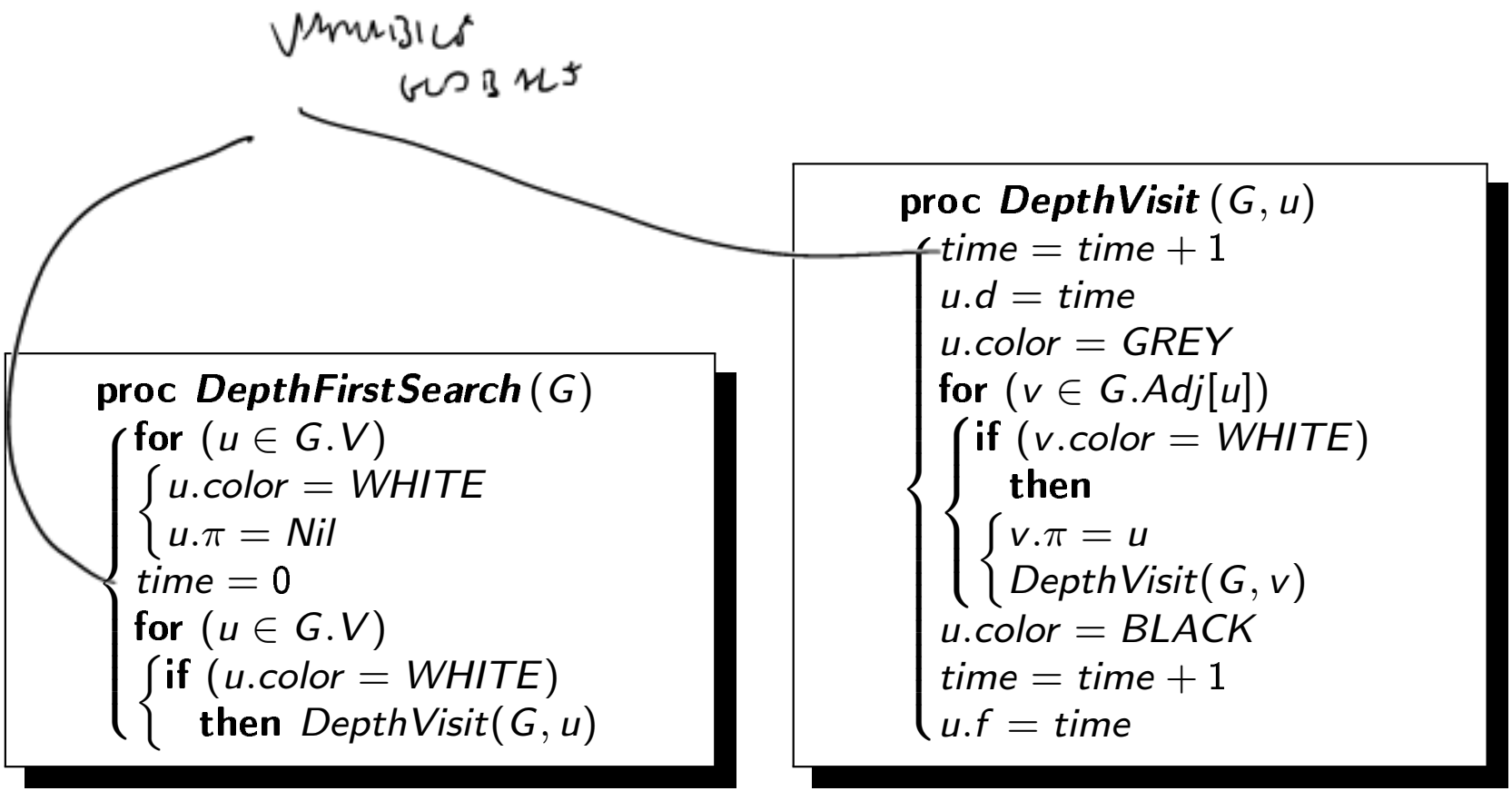
Grafi: visita in profonditá

Il proposito della visita in profonditá è quello di scoprire tutti i vertici raggiungibili da ogni potenziale sorgente s. La differenza tra le due visite è che i vertici nella visita in profonditá vengono scoperti il **prima possibile** a partire da quelli giá scoperti: la visita in profonditá è inerentemente ricorsiva. Anche *DepthFirstSearch* usa un sistema di colorazione (bianco, grigio e nero) per ricordare i vertici ancora da scoprire e quelli giá scoperti. Anche *DepthFirstSearch* assume che G sia rappresentato con liste di adiacenza, e riempe un campo $v.\pi$ per generare un **albero di visita in profonditá** come risultato della visita; la differenza qui è che invece di un solo albero, si produce una **foresta** di alberi, uno per ogni sorgente (ogni sotto-grafo connesso di G). *DepthFirstSearch* riempie anche dei campi $v.d$ e $v.f$: il primo rappresenta il **momento** della scoperta, e il secondo il *momento* nel quale il vertice viene abbandonato (tutto il suo sotto-grafo è stato scoperto). I campi $v.d$ e $v.f$ sono interi ~~tra 1 e $2 \cdot |V|$~~. Vengono chiamati genericamente **tempi** (di scoperta e di abbandono). Ogni nuovo evento (una scoperta o un abbandono) provoca che il tempo si incrementa. Chiaramente, per ogni vertice u, abbiamo $u.d < u.f$.

DIFFERENZA TRA AMPIEZZA E PROFONDITÀ

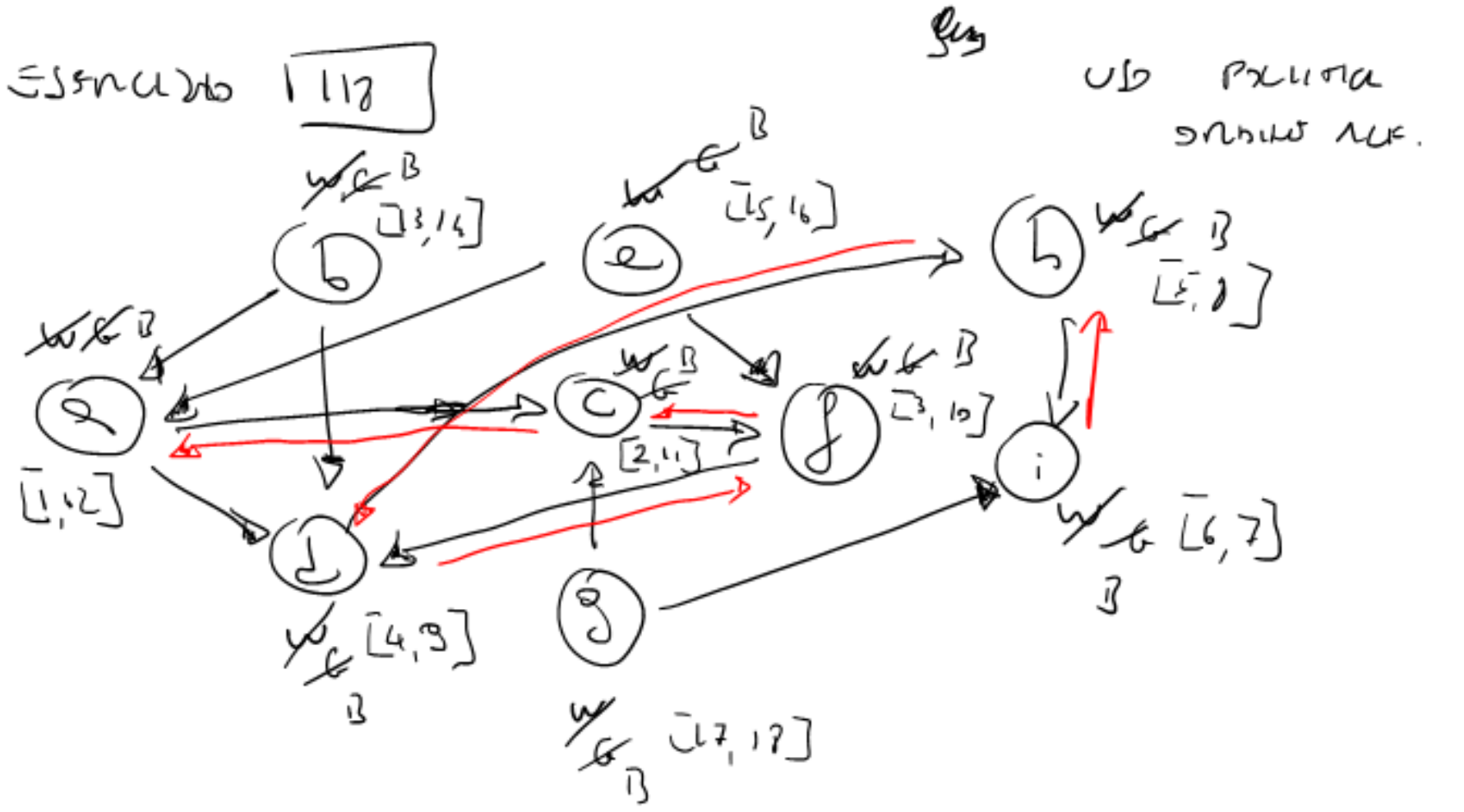


Grafi: visita in profonditá



```
proc DepthFirstSearch(G)
  for (u ∈ G.V)
    { u.color = WHITE
      u.π = Nil
  time = 0
  for (u ∈ G.V)
    { if (u.color = WHITE)
        then DepthVisit(G, u)
```

```
proc DepthVisit(G, u)
  { time = time + 1
    u.d = time
    u.color = GREY
    for (v ∈ G.Adj[u])
      { if (v.color = WHITE)
          then
            { v.π = u
              DepthVisit(G, v)
    u.color = BLACK
    time = time + 1
    u.f = time
```


a
b
c
d
e
f
g
h
i
[1,12]
[3,14]
[5,16]
[2,11]
[3,10]
[4,9]
[6,7]
[7,17]

Grafi: visita in profonditá

Ogni volta che *DepthVisit* viene chiamata, si inizia un nuovo albero della foresta degli alberi di visita in profonditá. Se G è tale che tutti i vertici sono raggiungibili dal primo vertice visitato, allora ci sará un solo albero; altrimenti ce ne saranno di più. Ogni vertice u sul quale si chiama *DepthVisit* è inizialmente bianco; il suo tempo di scoperta (incrementato di 1) viene marcato come tempo di inizio ($u.d$), e u viene colorato grigio. Ogni vertice nella lista di adiacenza di u viene esplorato ricorsivamente; alla fine di questa esplorazione u viene marcato come nero, ed il suo tempo finale (di nuovo, incrementato di 1) $u.f$ viene registrato.

Complessità di *DepthFirstSearch*

Come per il caso di *BreadthFirstSearch*, l'analisi della **complessitá** è semplice, attraverso la tecnica dell'analisi aggregata. *BreadthFirstSearch* esegue un ciclo che costa $|V|$ per la colorazione iniziale, e poi, per ogni vertice chiama *DepthVisit*. Ma il massimo numero di chiamate a *DepthVisit*, anche contando le chiamate ricorsive, è $|E|$. A differenza della visita in ampiezza, però, la visita in profondità garantisce che tutte le parti del grafo siano visitate (grazie al ciclo in *DepthFirstSearch*). Quindi la complessità è $\Theta(|V| + |E|)$, che, nel caso peggiore di grafi densi, diventa $\Theta(|V|^2)$ (ma, come nel caso della visita in ampiezza, accettiamo $\Theta(|V| + |E|)$). Come detto in precedenza, per parlare di correttezza dobbiamo stabilire qual è l'obbiettivo dell'algoritmo. In questo caso, definiamo tre problemi che hanno in comune la visita in profondità e li risolviamo, stabilendo la correttezza dei relativi algoritmi.

Grafi diretti: cicli

In un grafo diretto, chiamiamo **ciclo** un percorso $v_1, v_2, \ldots, v_k$ di vertici tali che per ogni i esiste l'arco (v_i, v_{i+1}), e che $v_1 = v_k$. Quando un grafo diretto è privo di cicli, lo chiamiamo **DAG** (Directed Acyclic Graph). Come abbiamo visto precedentemente, uno degli utilizzi di *DepthFirstSearch* è quello di stabilire se un dato grafo G diretto è o no acilico. Risolviamo dunque il seguente problema: dato un grafo diretto G stabilire se presenta, o no, un ciclo. L'algoritmo *CycleDet* che usiamo prevede di eseguire *DepthFirstSearch* modificata in maniera da interrompersi se si visita un nodo grigio: al momento di visitare un nodo grigio, siamo certi di aver trovato un ciclo.

Esempio di ciclo euleriano / non ciclo euleriano

$v_1 \dots v_k = v_1$

$\forall i$

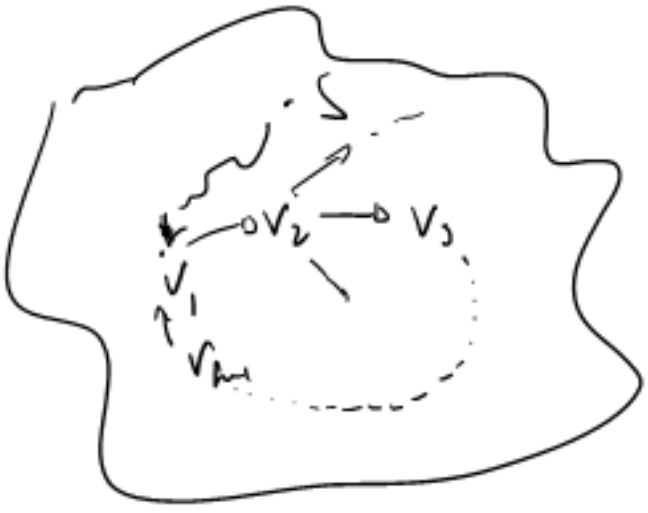
$(v_i, v_{i+1}) \in E$

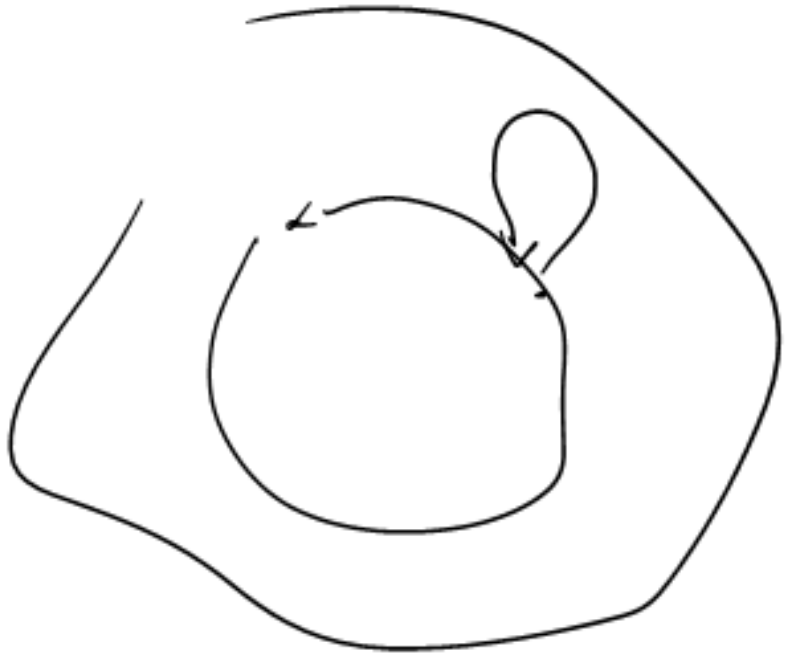
Grafi: cicli

```
proc CycleDet (G)
  cycle = False
  for (u ∈ G.V) u.color = WHITE
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitCycle(G, u)
  return cycle
```

```
proc DepthVisitCycle (G, u)
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitCycle(G, v)
    if (v.color = GREY)
      then cycle = True
  u.color = BLACK
```

G ha un ciclo se DV(G) ha un solo grigio





Se V_1 è il primo vertice visitato del ciclo poi $V_2 \dots V_{k-1}$ non sono stati visitati

Correttezza e complessità di *CycleDet*

Correttezza: l'algoritmo è corretto se e solo se restituisce True quando G è ciclico, e False altrimenti. Per dimostrare che l'algoritmo è corretto, consideriamo, prima, un grafo ciclico G. Siccome è ciclico, presenta un percorso $v_1, v_2, \ldots, v_k$ di vertici tali che per ogni i esiste l'arco (v_i, v_{i+1}), e che $v_1 = v_k$. Siccome *DepthFirstSearch* visita tutti i vertici, anche i vertici tra v_1 e v_k sono, prima o poi, visitati; sia v_i il primo vertice del ciclo su cui *DepthFirstSearch* viene chiamata. Per ipotesi, allora tutti i vertici $v_{i+1}, \ldots, v_k, v_1, \ldots, v_{i-1}$ sono bianchi in quell'istante. Pertanto, esiste certamente la sequenza di chiamate ricorsive $v_{i+1}, \ldots, v_k, v_1, \ldots, v_{i-1}$; all'ultimo passo, si chiamerà *DepthFirstSearch* su v_i, che è grigio, e pertanto il ciclo viene trovato. Il ragionamento da fare nel caso di grafi acilici è simile. Chiaramente la **complessità** del nostro algoritmo è la stessa della visita in profondità, e la **terminazione** è ovvia.

Grafi: ordinamento topologico di grafi diretti

Uno degli usi più interessanti dei grafi diretti consiste nel rappresentare un insieme di vincoli, per esempio temporali, rispetto a un insieme finito di compiti (o **task**). In molte applicazioni, per esempio in ingegneria, ci sono situazioni dove i task da eseguire sono vincolati tra loro in maniera non totale; ad esempio, la costruzione dalla base di un edificio deve essere preceduta dallo scavo delle fondamenta e dal calcolo delle caratteristiche delle fondamenta stesse, ma questi due task non sono necessariamente ordinati tra loro. Se tutti i task sono inseriti in un grafo diretto G dove un arco (u, v) rappresenta che u deve essere posto prima di v, una delle possibili soluzioni al problema di stabilire un ordine lineare (possibile) tra i task è dato dall' **ordinamento topologico**. Il problema dell'ordinamento topologico non ha senso se il grafo diretto è ciclico.

Ejemplo de ordenamiento

topológico

Grafi: ordinamento topologico di grafi diretti

Quindi il problema dell'ordinamento topologico prende in input un grafo connesso G diretto senza cicli e restituisce una lista collegata $v_1, \ldots, v_{|V|}$ di vertici topologicamente ordinati; per ogni coppia v_i, v_j di vertici, v_i appare prima nella lista di v_j se e solo se v_i precede topologicamente v_j. Nell'algoritmo *TopologicalSort*, prima, chiamiamo *DepthFirstSearch* su G per computare $v.f$ per ogni $v \in G.V$, e, poi, per ogni nodo finito v lo inseriamo in testa a una lista collegata; finalmente, restituiamo la lista.

Grafi: ordinamento topologico di grafi diretti

```
proc TopologicalSort(G)
  for (u ∈ G.V) u.color = WHITE
  L = ∅
  time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitTS(G, u)
  return L
```

```
proc DepthVisitTS(G, u)
  time = time + 1
  u.d = time
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitTS(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time
  ListInsert(L, u)
```

Correttezza e complessità di *TopologicalSort*

Per la **correttezza** di *TopologicalSort*: dopo l'esecuzione, la lista di uscita è topologicamente ordinata. Dimostriamo questa affermazione mostrando che per ogni arco (u, v), succede che $v.f < u.f$. Infatti, consideriamo un arco (u, v) qualsiasi esplorato da *DepthFirstSearch*: quando scopriamo v, questo non può essere grigio (perchè, altrimenti, G sarebbe ciclico). Pertanto, è bianco o nero. Se è bianco, allora v diventa discendente di u, e abbiamo già visto che in questo caso $v.f < u.f$. Se invece è nero, allora è già finito il suo ruolo e $v.f$ ha già un valore, mentre $u.f$ non ancora: pertanto, quando l'avrà, succederà che $v.f < u.f$. La **complessità** é la stessa della visita in profondità, e la **terminazione** è ovvia.

Essencia 123 :

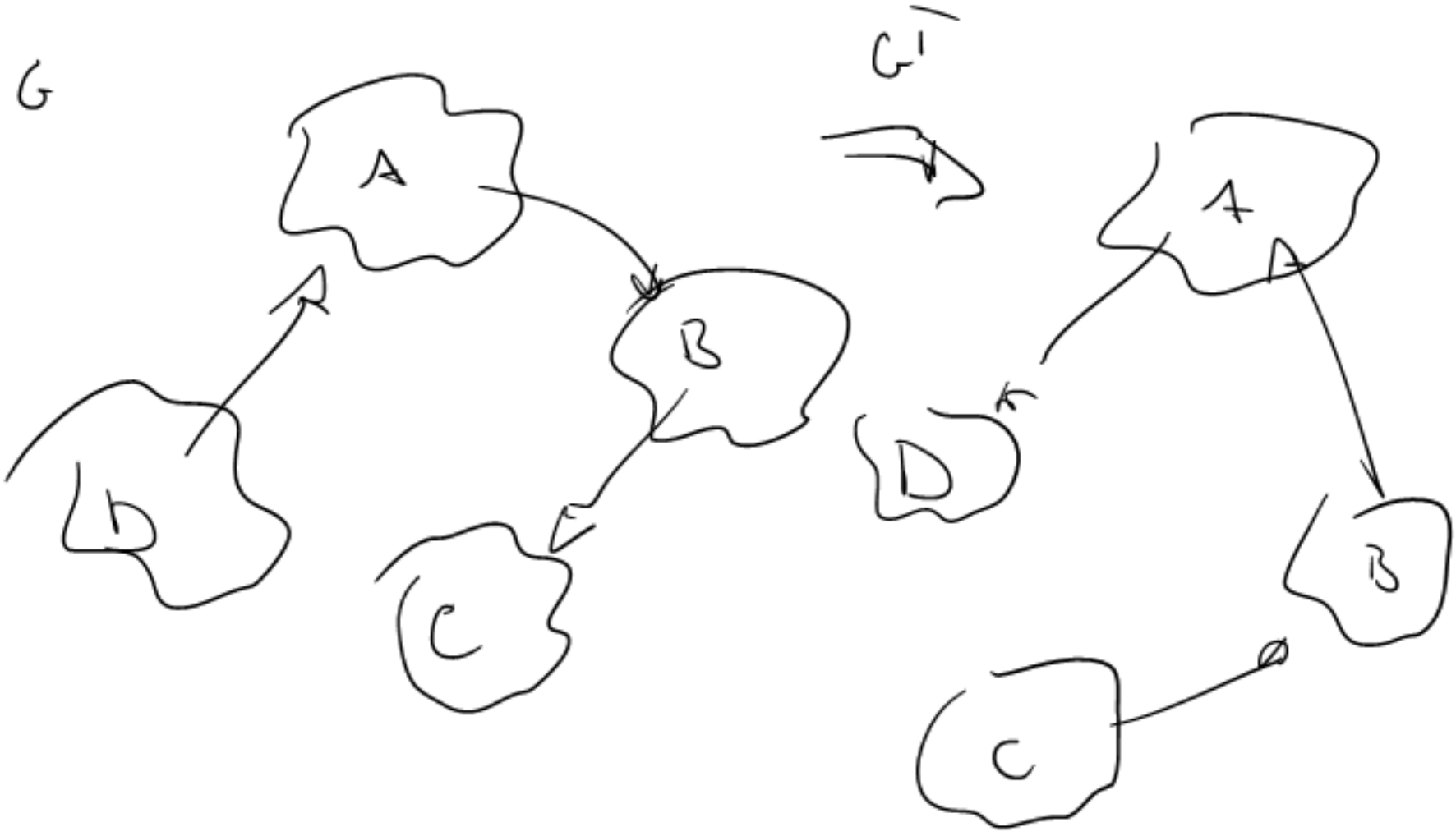
Grafi: componenti fortemente connesse

Trovare le componenti fortemente connesse di un grafo diretto G è la terza applicazione classica di *DepthFirstSearch* che vediamo. Dato un grafo diretto G, una **componente fortemente connessa** (SCC) è un sottoinsieme massimale $V' \subseteq V$ tale che, per ogni $u, v \in V'$, succede che $u \rightsquigarrow v$ e che $v \rightsquigarrow u$. Ci proponiamo di elencare tutte e sole le SCC di G. Applicativamente, supponiamo che G rappresenti le conoscenze (che si suppongono non simmetriche, ma transitive) in un gruppo di persone: una SCC di G rappresenta un sotto-gruppo di persone dove tutti sono possono arrivare a tutti attraverso conoscenze dirette. Osserviamo che il concetto corrispondente nel caso di grafi indiretti è quello di **componente connessa**; due termini per indicare la stessa idea in due contesti leggermente diversi.

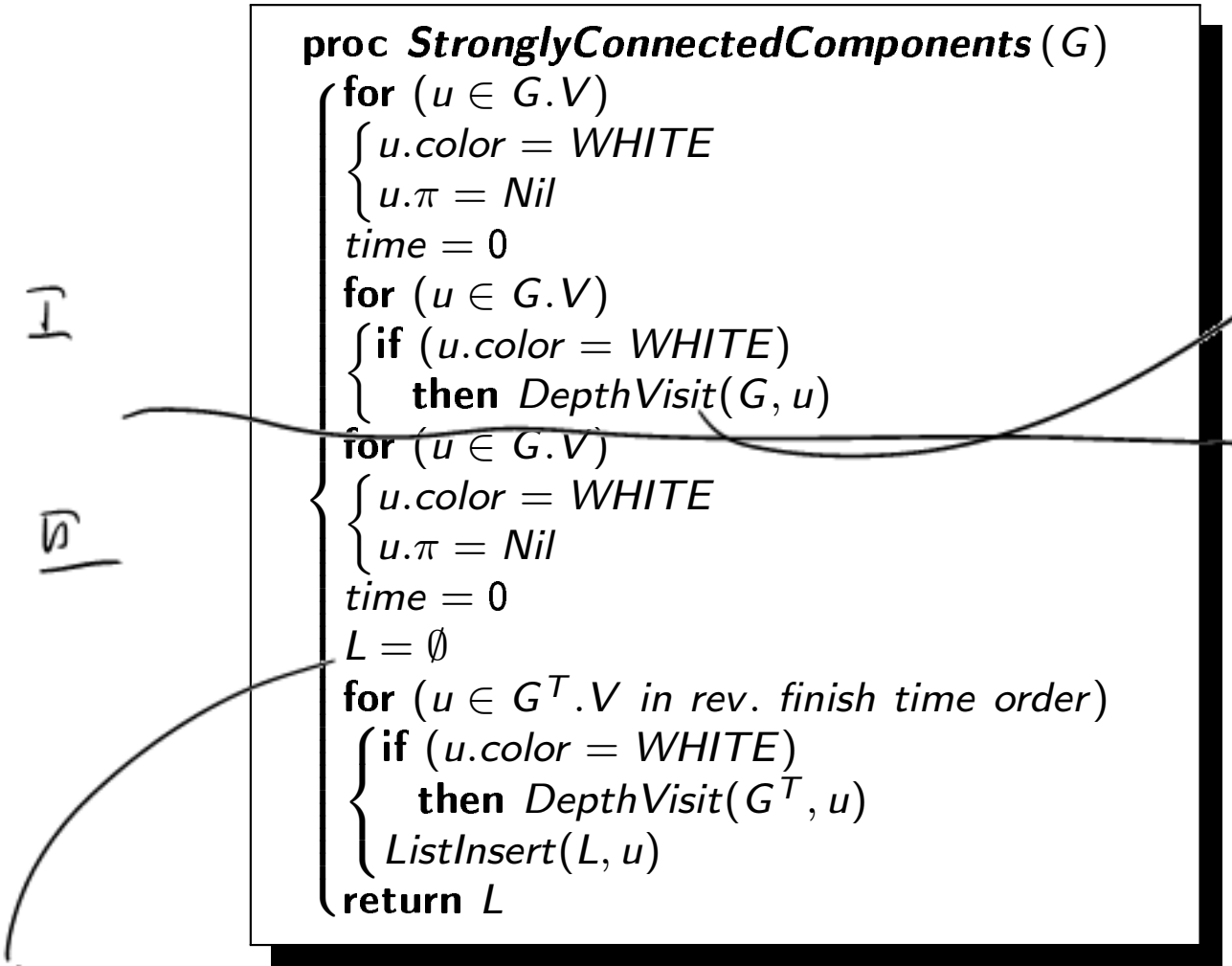
Grafi: componenti fortemente connesse

Un elemento fondamentale nello studio delle SCC é il grafo trasposto di G. Dato G diretto, il **grafo trasposto** G^T di G è ottenuto invertendo la direzione di ogni arco. Questo si puó ottenere facilmente, ed il problema ha complessitá $\Theta(|V| + |E|)$ quando G è rappresentato con liste di adiacenza. La proprietá piú interessante di G^T è che G e G^T hanno *le stesse SCC*, ed é immediato dimostrarlo. Andiamo a vedere l'algoritmo *StronglyConnectedComponents*. Dato un grafo diretto G, chiamiamo *DepthFirstSearch* su G per computare $v.f$ per ogni $v \in G.V$. Poi, computiamo G^T e chiamiamo *DepthFirstSearch* su G^T, dove nel ciclo principale consideriamo i vertici in ordine decrescente di $u.f$ (computato alla prima visita); infine, restituiamo la lista degli alberi di visita ottenuti dall'ultimo passaggio: ogni albero è una componente fortemente connessa.

Esempio di calcolo delle SCC



Grafi: componenti fortemente connesse



```
proc StronglyConnectedComponents(G)
  for (u ∈ G.V)
    { u.color = WHITE
    { u.π = Nil
  time = 0
  for (u ∈ G.V)
    { if (u.color = WHITE)
    {   then DepthVisit(G, u)
  for (u ∈ G.V)
    { u.color = WHITE
    { u.π = Nil
  time = 0
  L = ∅
  for (u ∈ G^T.V in rev. finish time order)
    { if (u.color = WHITE)
    {   then DepthVisit(G^T, u)
    { ListInsert(L, u)
  return L
```

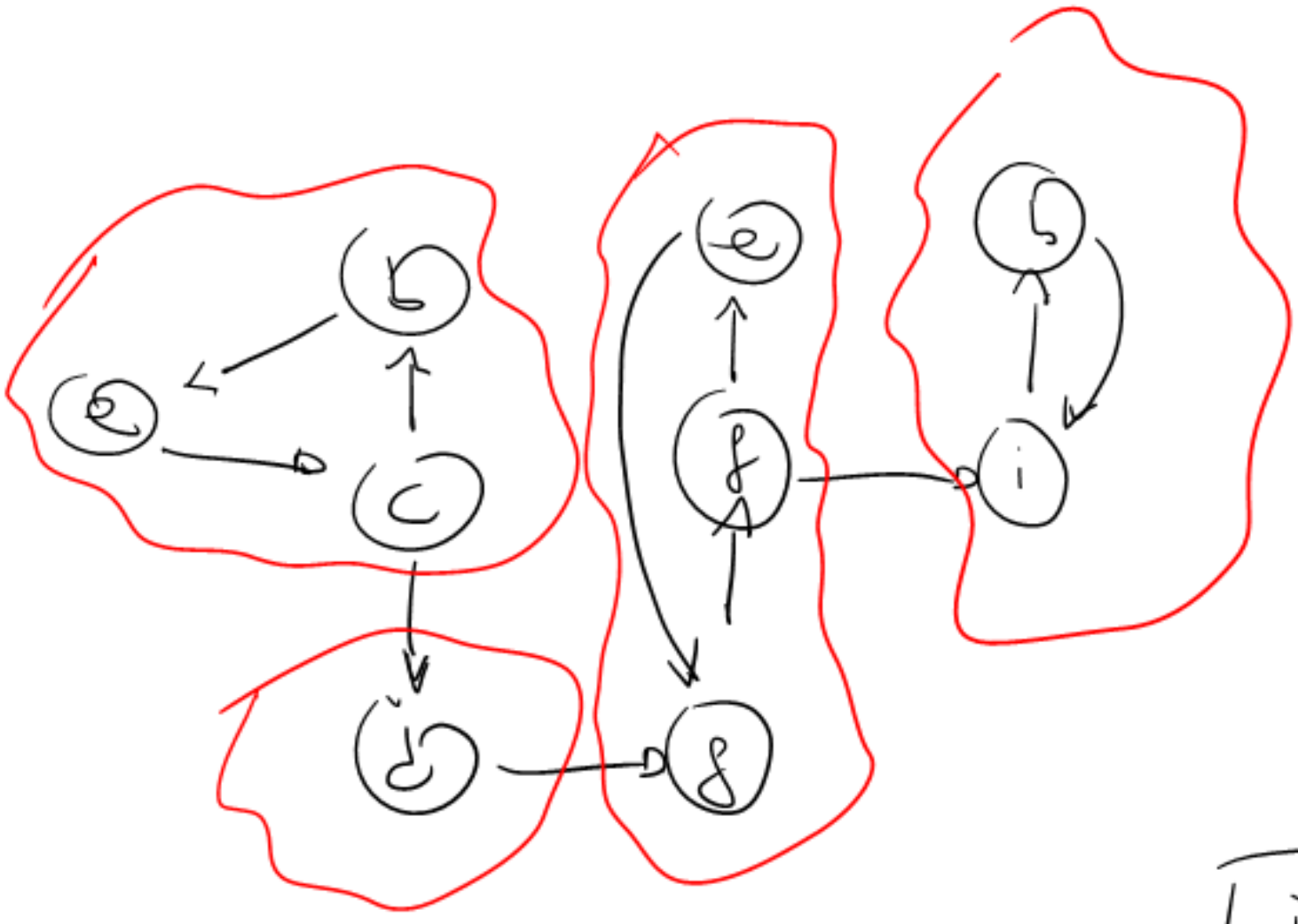
Correttezza e complessità di *StronglyConnectedComponents*

Mentre il calcolo della **complessitá** è molto semplice, e porta a concludere che la complessitá di *StronglyConnectedComponents* è la stessa della visita in profondità (perchè il calcolo del grafo trasposto ha costo $\Theta(|V| + |E|)$), la dimostrazione di **correttezza** è piú complessa. Osserviamo prima di tutto che da un grafo G possiamo ricavare il suo **grafo delle componenti connesse** G^{SCC} semplicemente considerando tutti i vertici di ogni SCC di G come un unico vertice di G^{SCC}, e impostando che esiste un arco (u, v) in G^{SCC} se e solo se esiste un arco in G da uno dei nodi simboleggiati da u a uno dei nodi simboleggiato da v. Questo grafo G^{SCC} è, evidentemente, un DAG: se cosí non fosse, allora ci sarebbe un ciclo $u \rightsquigarrow v \rightsquigarrow u$ in G^{SCC}, il porterebbe a concludere che tutti i vertici simboleggiati da u raggiungono e sono raggiungibili da tutti i vertici simboleggiati da v, cioé che u e v sono *la stessa componente connessa* in G. Allora, se G^{SCC} è un DAG, possiamo calcolare il suo ordinamento topologico.

Correttezza e complessità di *StronglyConnectedComponents*

Per un gruppo di vertici C (che potrebbe essere una SCC) chiamiamo $f(C)$ il massimo tempo $u.f$ tra tutti gli $u \in C$. Adesso osserviamo che la prima esecuzione di *DepthFirstSearch* ci dá informazione sulle SCC di G: infatti, se C, C' sono due SCC di G, ed esiste un arco tra un vertice di C ad un vertice C', $f(C) > f(C')$. In altre parole, la componente C' viene scoperta tutta prima della fine della scoperta di C. Questo significa che una esecuzione di *DepthFirstSearch* sul grafo G^T ha esattamente la proprietá contraria.

Esercizio 125



D

Correttezza e complessità di *StronglyConnectedComponents*

Ragioniamo adesso per induzione sull'indice k che indica il k-esimo albero di visita in profonditá generato dalla **seconda** visita *DepthFirstSearch*, quella effettuata su G^T. Vogliamo mostrare che tutti questi alberi sono, in effetti, SCC di G (e di G^T) (e questa è l'**invariante** che stiamo cercando).

- Quando $k = 0$ (**caso base**) il risultato è triviale.
- Per il **caso induttivo** supponiamo che i primi k alberi restituiti coincidano con k SCC di G, e dimostriamo che questo é ancora vero per il $(k+1)$-esimo albero. Supponiamo che u sia la radice del $(k+1)$-esimo albero. Questo vertice é tale che $u.f = f(C)$ (dove C è la SCC alla quale appartiene) ed è maggiore di $f(C')$ per ogni SCC C' ancora da visitare. Tutti i vertici in C sono bianchi a questo punto, e tutti diventano discendenti di u. Quindi, l'albero risultante contiene tutti gli elementi di C (ricordiamo che le componenti di G e G^T sono le stesse!).

Correttezza e complessità di *StronglyConnectedComponents*

Rimane da mostrare che nessun v che *non* appartenga a C appare come discendente di u. Infatti, se v è discendente di u, deve succedere in G^T che $u \rightsquigarrow v$, e che v era bianco al momento di considerare u. Ma questo accade solo se $u \not\rightsquigarrow v$ in G. Quindi, nella prima visita di G, v è ancora bianco quando $u.f$ viene deciso, e quindi, quando v viene visitato (e abbandonato) il suo $v.f$ sará maggiore di $u.f$. Pertanto, v è giá apparso in qualche albero di visita (radicato in qualche u') e quando viene visto durante la visita a partire da u è giá nero, e non inserito nel rispettivo albero di visita. Allora, come volevamo dimostrare, anche l'albero k corrisponde ad una SCC di G.

Conclusione

La visita in profonditá è anche piú versatile di quella in ampiezza. I tre problemi che abbiamo visto, risolti con la visita in profonditá, sono solo tre esempi di un insieme molto piú ampio di problemi che possono essere risolti in questo modo.