PROGRAMMA ARNALDO

Grafi e Pathfinding

A.A 2022/23

In today’s episode…

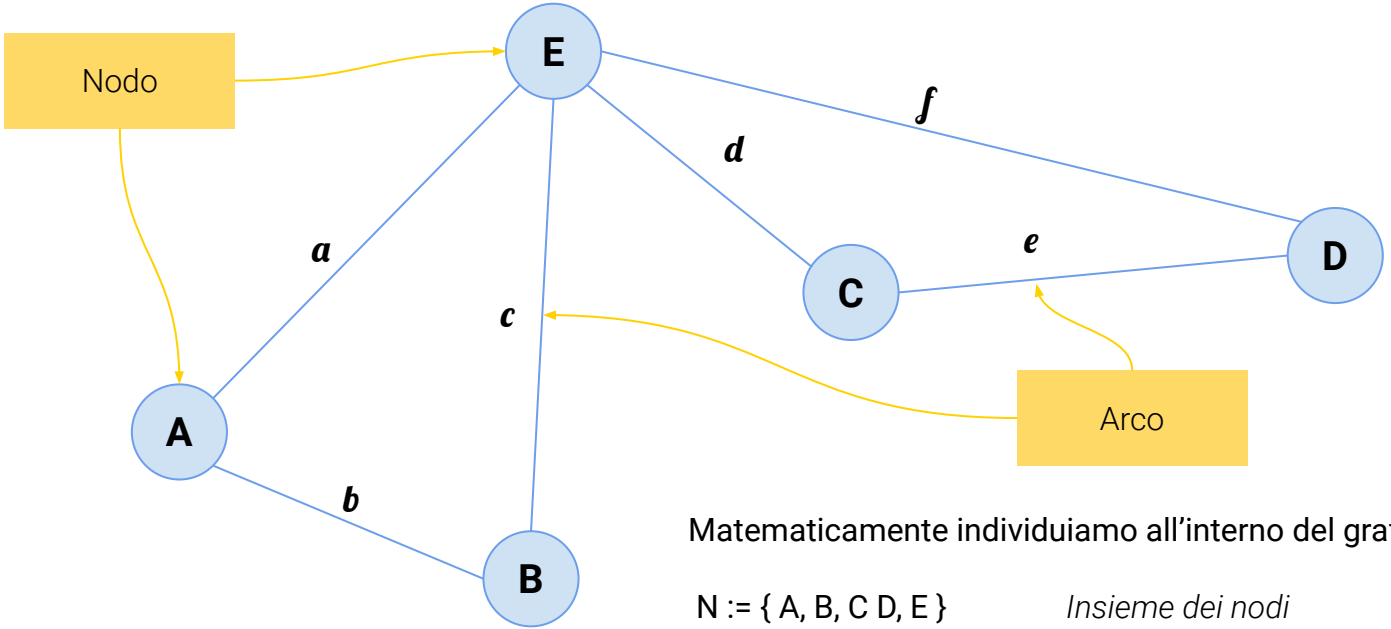
- Concetti preliminari
  - Minimum spanning tree
- Alberi
  - DFS,
  - BFS
- Pathfinding
  - ~~Djiksta~~
    ~~Distra~~
    ~~Digstria~~
    ~~Dykestra~~
    Dijkstra
  - A*
- Esercizio 6 ????

Concetti base sui grafi

"In a forest of a hundred thousand trees, no two leaves are alike. And no two journeys along the same path are alike."

- Paulo Coelho

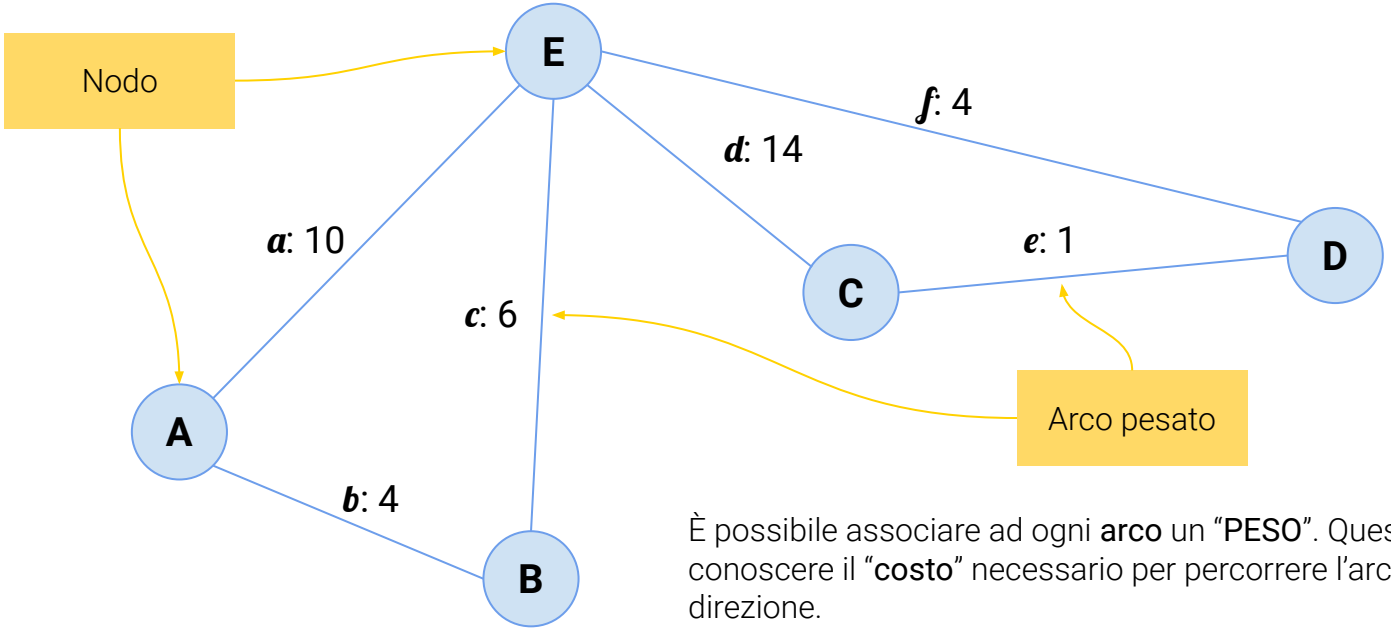
Vi presento Grafo



Matematicamente individuiamo all'interno del grafo due insiemi:

N := { A, B, C D, E } *Insieme dei nodi*
V := { a, b, c, d, e, f } *Insieme degli archi*

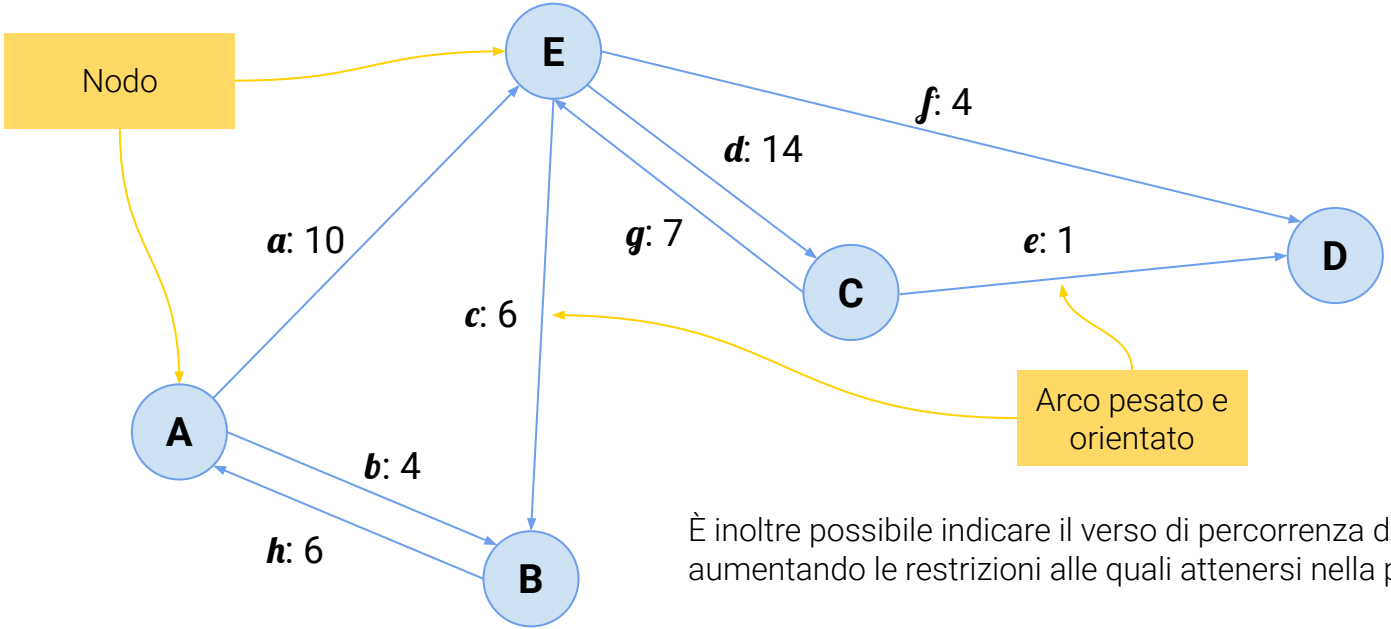
Vi presento Grafo, un po' cresciuto



È possibile associare ad ogni **arco** un "**PESO**". Questo permette di conoscere il "**costo**" necessario per percorrere l'arco in qualsiasi direzione.

Questa struttura prende il nome di "**Grafo Pesato**"

Vi presento Grafo, sotto steroidi



È inoltre possibile indicare il verso di percorrenza degli archi, aumentando le restrizioni alle quali attenersi nella percorrenza del grafo.

Possono così esserci collegamenti asimmetrici bidirezionali.

Pesi e cammini

Il peso associato ad un arco di un grafo potrebbe rappresentare:

- se il grafo rappresenta una mappa geografica:
  - la distanza in km
  - il tempo di percorrenza in minuti
  - il costo in termini di pedaggi autostradali
- se il grafo rappresenta una rete di telecomunicazioni:
  - la latenza in millisecondi tra due nodi
  - la velocità del canale in termini di banda
  - lo stato di congestione del collegamento tra due nodi (variabile nel tempo)

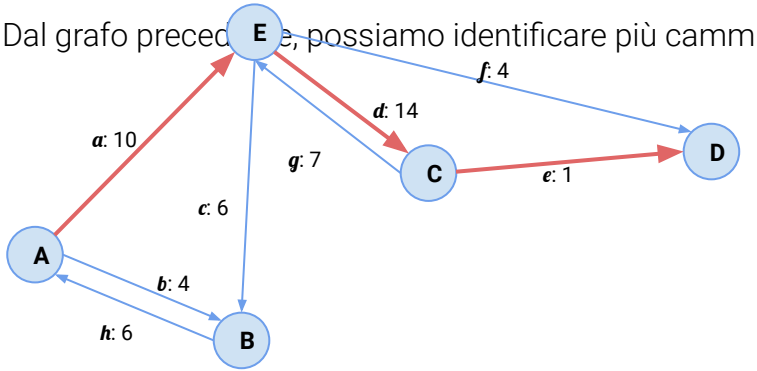
Il peso potrebbe anche essere calcolato dalla combinazione di due o più di questi parametri.

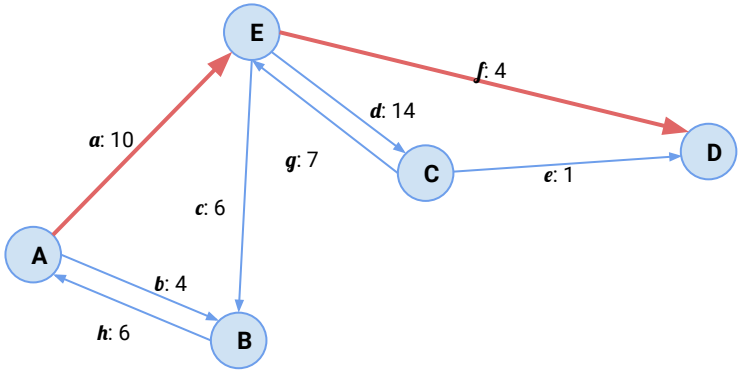
Pesi e cammini

Come detto in precedenza il peso associato a un arco è una misura di costo per per percorrerlo.
L'utilità del peso la si ha quando si introducono i **cammini.**

Definiamo come "**Cammino dal nodo A al nodo D**" la lista di archi percorsi nel percorso tra A e D all'interno del grafo.

Dal grafo precedente, possiamo identificare più cammini:



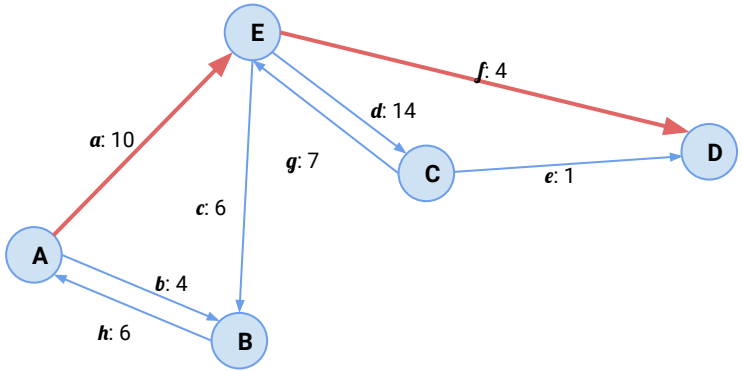


Il cammino [a, d, e] avrà peso 25, mentre il cammino [a, f] avrà peso 14

Cammino minimo

Il fatto che vi sia la possibilità di avere più cammini tra due nodi richiede che esista un algoritmo - efficiente - in grado di determinare quale sia il cammino con peso minore (o maggiore).

Dall'esempio precedente, data la presenza di due soli cammini possibili, possiamo intuire facilmente che il cammino minimo è il secondo.

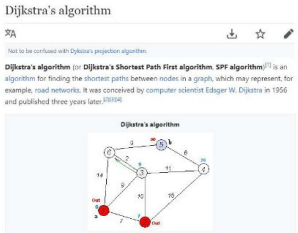


Bae: Come over

Dijkstra: Can't, I'm busy

Bae: I'm home alone...

Dijkstra:



Un gentile signore ha trovato un modo per fare questo compito: vedremo in seguito in dettaglio.

Casi particolari di grafi

Un grafo può essere **ciclico o aciclico** se esiste (o non esiste) un cammino da un generico nodo a se stesso passando per un numero dispari di nodi maggiore di uno.

Un grafo si dice **connesso** se da ogni nodo è possibile raggiungere tutti gli altri nodi con almeno un cammino.

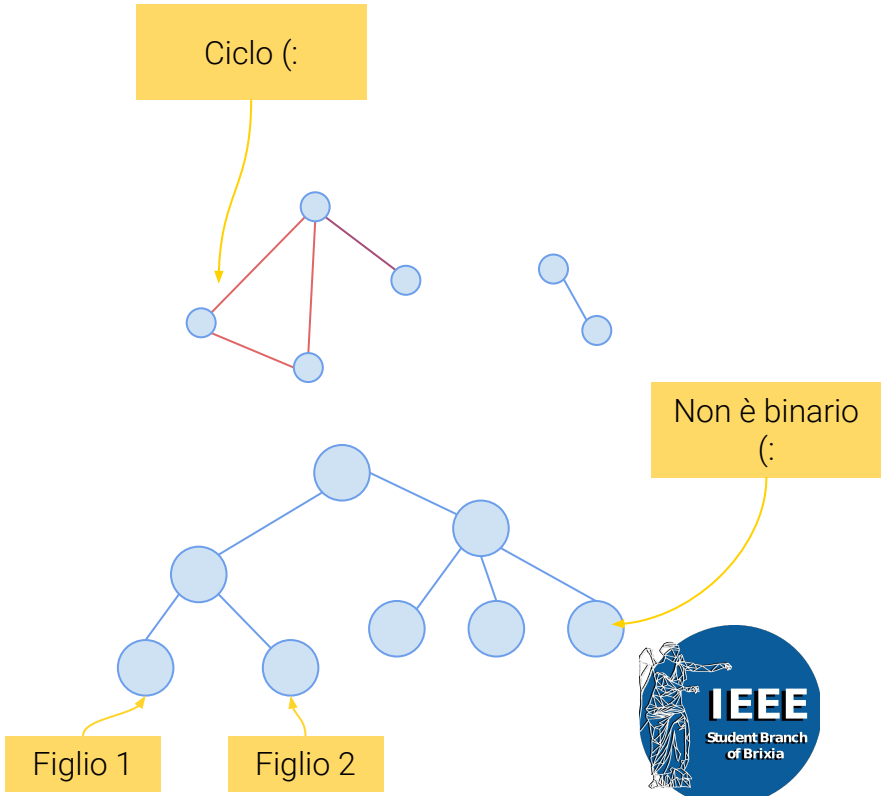
Nell'esempio accanto vi sono due nodi che non potranno mai essere raggiunti se non quando il cammino partirà da uno dei due.

In teoria dei grafi, un albero è un **grafo non orientato** nel quale due vertici qualsiasi sono connessi da **uno e un solo cammino**.

In altre parole, è un grafo non orientato, connesso e privo di cicli (**aciclico**).

Un albero binario è un albero dove ogni nodo ha al massimo due figli

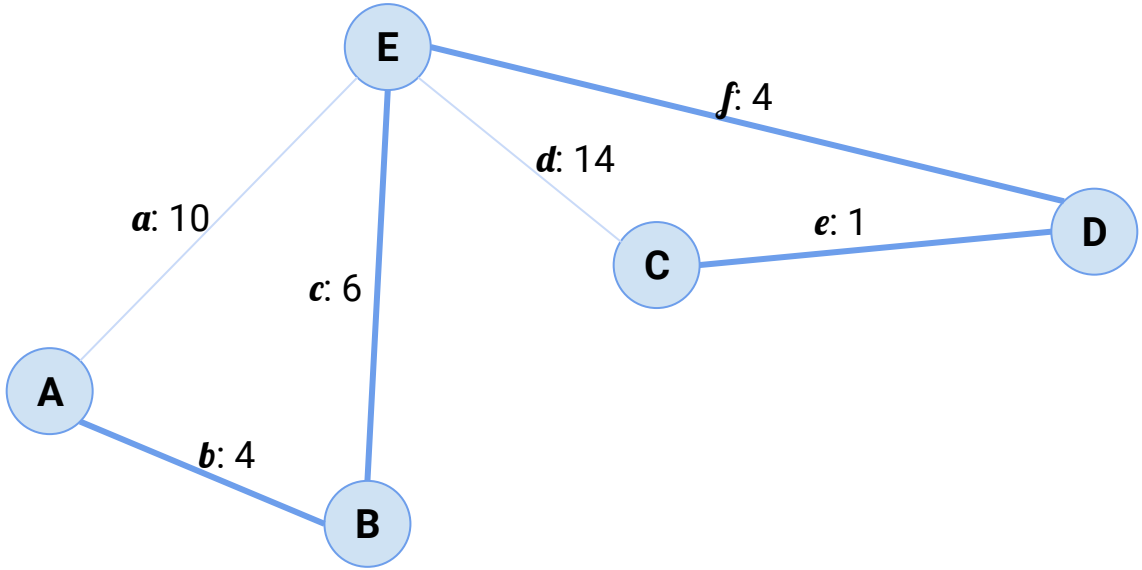
È da notare che aggiungendo un qualunque arco a un albero, il grafo diventerà ciclico, mentre rimuovendo un arco, il grafo diventerà sconnesso.



Minimum Spanning Tree

L'albero di copertura del grafo è un semplicissimo albero dove i nodi sono quelli del grafo. È una rappresentazione aciclica e minimamente connessa del grafo iniziale.

Per ultima cosa, definiamo **Minimum Spanning Tree** l'albero di copertura del grafo dove la somma dei pesi degli archi è minima e il grafo è totalmente connesso.

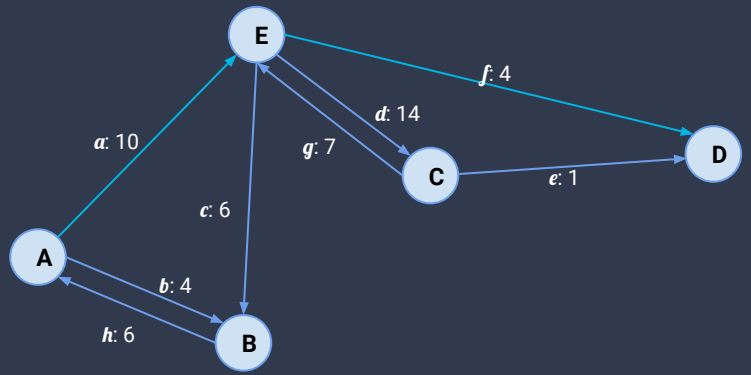


OK, MA IL CODICE?

La rappresentazione di grafi come struttura dati all'interno di un programma può avvenire in modi diversi in base alle necessità dell'algoritmo.

Metodologie note sono **Matrice di adiacenza** o la **rappresentazione in memoria.**

Ok ma il codice?
Un approccio non troppo OOP



	A	B	C	D	E
A	0	4	∞	∞	10
B	6	0	∞	∞	∞
C	∞	∞	0	1	7
D	∞	∞	∞	0	∞
E	∞	6	14	4	0

Matrice di adiacenza

La matrice di adiacenza permette, mediante una semplice struttura dati nativa, di rappresentare tutte le connessioni presenti tra i nodi con il relativo peso.

	A	B	C	D	E
A	0	4	∞	∞	10
B	6	0	∞	∞	∞
C	∞	∞	0	1	7
D	∞	∞	∞	0	∞
E	∞	6	14	4	0

L'indice di colonna (`j`) indica il nodo di arrivo dell'arco

In questo caso abbiamo che i nodi A e B sono connessi da due archi, Uno da B verso A con costo 6 e uno da A verso B con costo 4

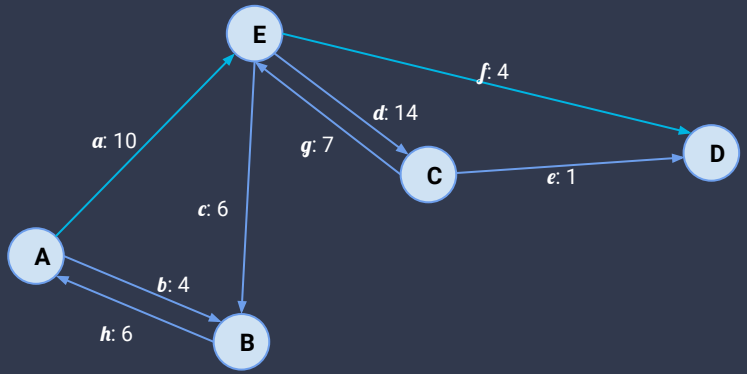
Il valore dell'elemento (`i`, `j`) rappresenta il costo dell'arco dal nodo `i` al nodo `j`

Un valore particolare indica la mancanza di collegamento. Se i pesi sono definiti positivi o nulli, il valore `-1` può essere usato.

L'indice di riga (`i`) indica il nodo di partenza dell'arco

IEEE Student Branch of Brixia

Ok ma il codice?
Un approccio stra OOP



LinkedList/Map style

In questo caso è possibile creare una classe `Nodo`, con definiti i metodi `equals` e `hashCode`.
Questa classe avrà, tra i suoi attributi una mappa `<Nodo, Integer>` che conterrà i nodi al quale è connessa l'istanza (`this`) con il relativo peso.

Se ogni nodo è un'istanza di `Nodo`, l'`HashMap` sarà in grado di determinare l'uguaglianza e di reperire il corretto valore per l'arco.

Un trucco che potrebbe fare comodo è l'implementazione di un metodo equals tra due Nodi e un metodo equals tra una stringa e un nodo: se ogni nodo ha un nome, come stringa, e il metodo hashcode ritornasse il valore dell'hashCode della stringa, sarà possibile eguagliare un nodo al suo nome e la ricerca nella mappa...

Alberi

“Nel mezzo del cammin di nostra vita mi ritrovai per una selva oscura, ché la diritta via era smarrita.”

- Dante Alighieri

Attraversamento

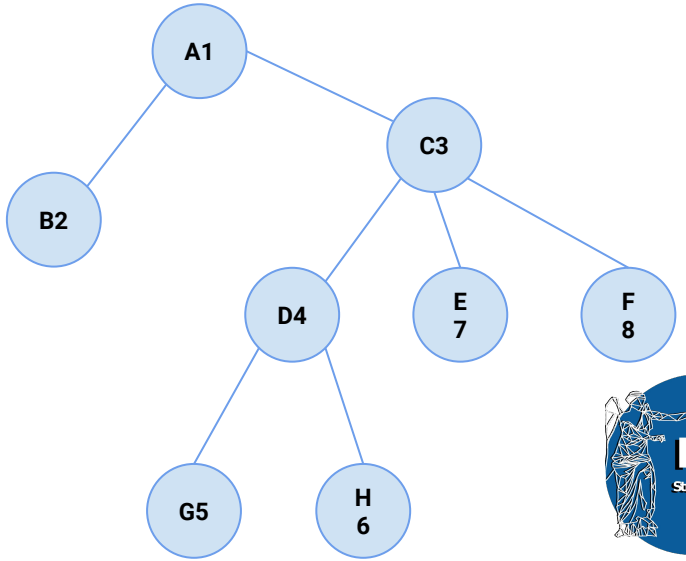
L’attraversamento di un albero è molto utilizzato al fine di indicizzare e/o ricercare all’interno dell’albero.
Questi sono algoritmi che possono essere implementati ricorsivamente e che permettono di avere una percezione completa (una volta attraversato tutto l’albero) dei dati contenuti.

DFS - Depth first search

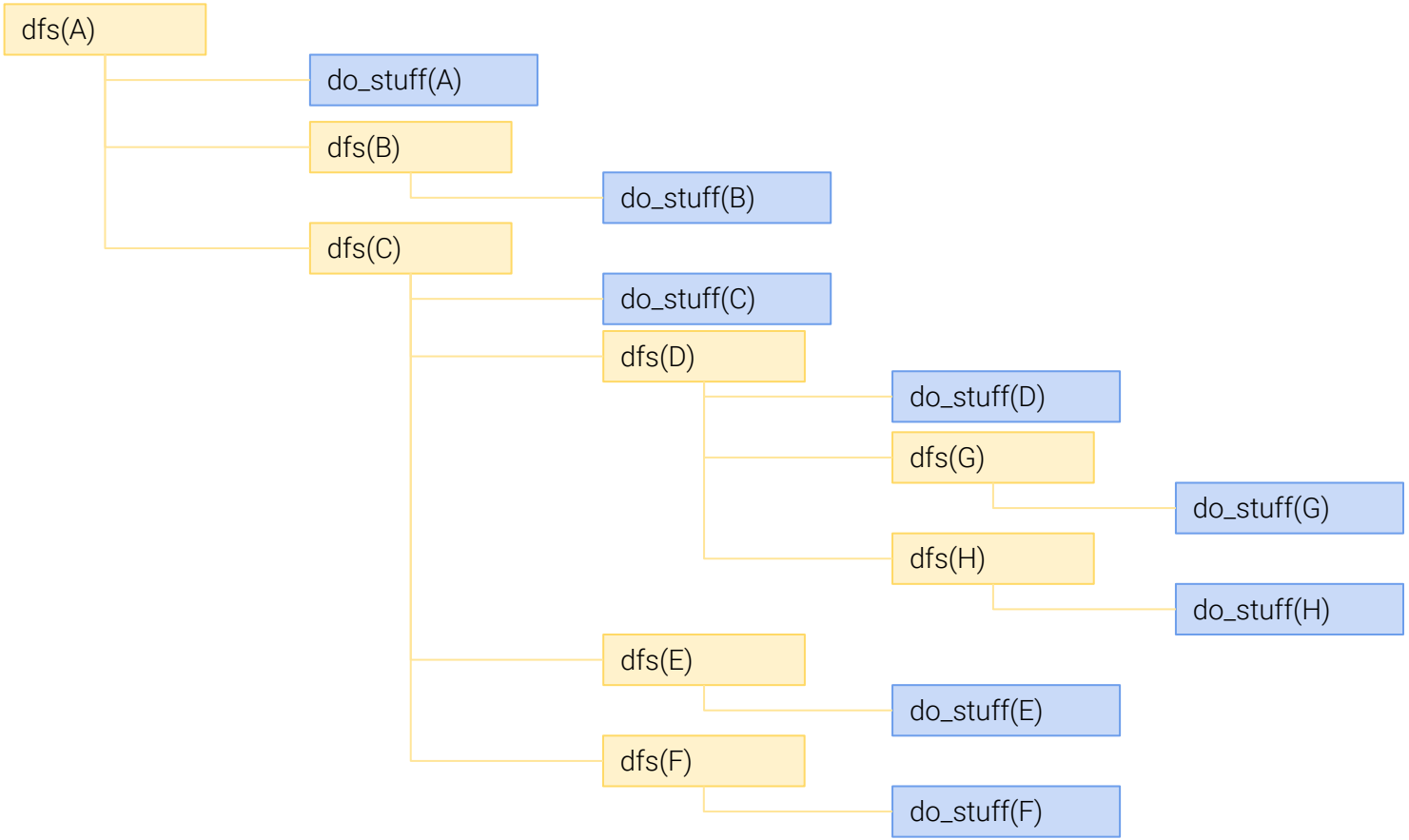
```
func dfs(root R):
    S = empty_stack
    S.push(R)
    while S is not empty:
        V = S.pop()
        do_stuff(V)
        foreach child C of V:
            S.push(C)
```

```
func dfs(root R):
    do_stuff(R)
    for each child C of R:
        dfs(C)
```

È una tecnica algoritmica che permette di effettuare attraversamenti o ricerche su alberi e, con qualche piccola modifica, su generici grafi. Il concetto di fondo è molto semplice: a partire da un nodo radice, si scorrono tutti i vertici dell'albero procedendo in profondità.



DFS - ricorsiva



Tempo

DFS - Iterativa

i	Passo	V	STACK
	Push A		A

Tempo

DFS - Iterativa

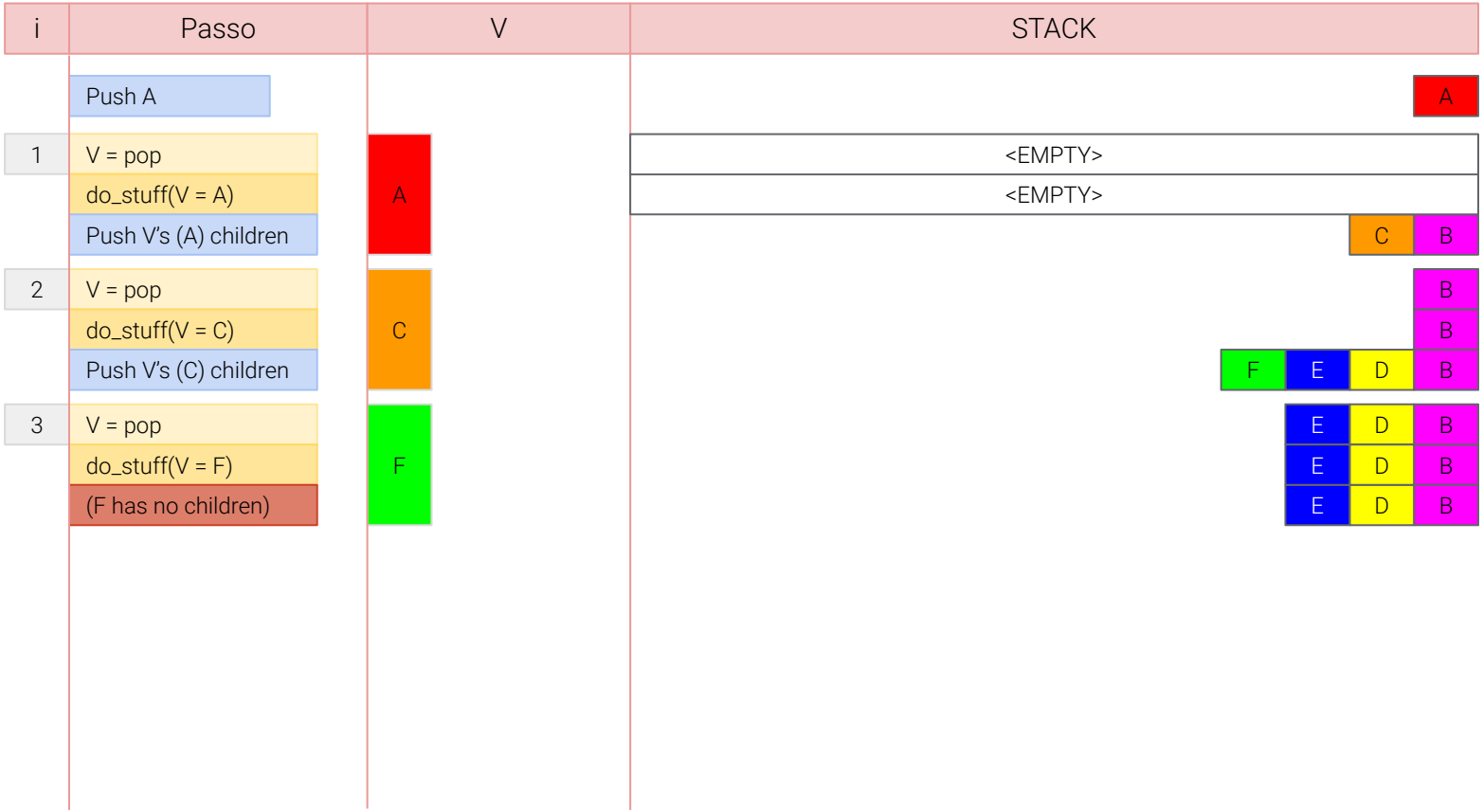
i	Passo	V	STACK
	Push A		A
1	V = pop do_stuff(V = A) Push V's (A) children	A	<EMPTY> <EMPTY> C B

Tempo

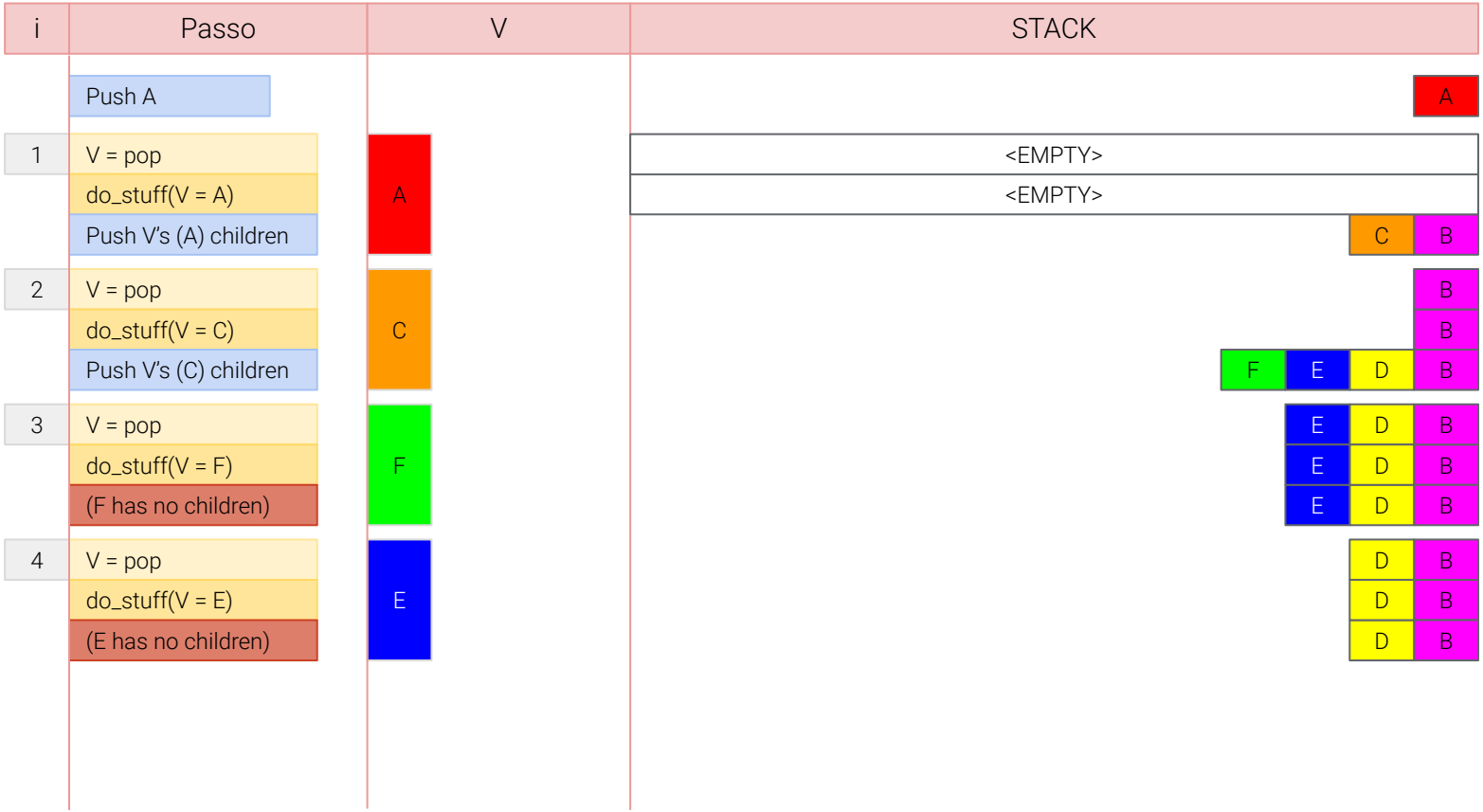
DFS - Iterativa

i	Passo	V	STACK
	Push A		A
1	V = pop do_stuff(V = A) Push V's (A) children	A	<EMPTY> <EMPTY> C B
2	V = pop do_stuff(V = C) Push V's (C) children	C	B B F E D B

DFS - Iterativa



DFS - Iterativa



DFS - Iterativa

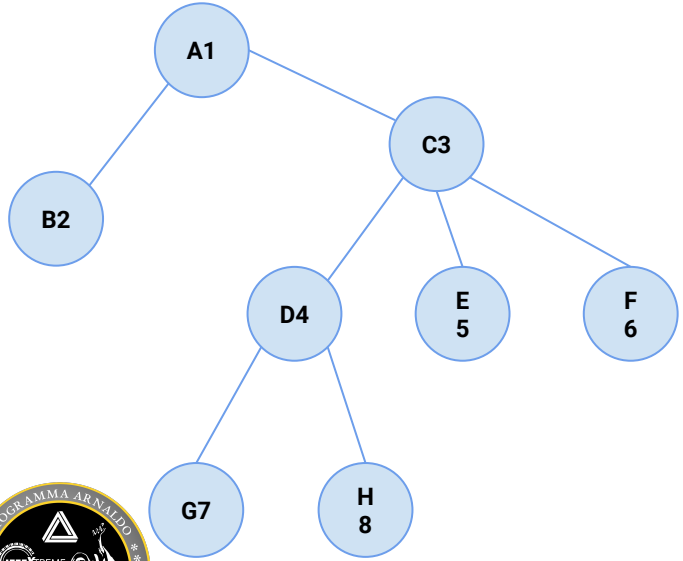
i	Passo	V	STACK
	Push A		A
1	V = pop do_stuff(V = A) Push V's (A) children	A	<EMPTY> <EMPTY> C B
2	V = pop do_stuff(V = C) Push V's (C) children	C	B B F E D B
3	V = pop do_stuff(V = F) (F has no children)	F	E D B E D B E D B
4	V = pop do_stuff(V = E) (E has no children)	E	D B D B D B
6	V = pop do_stuff(V = D) Push V's (D) children	D	B B H G B

Tempo

Iterazioni 7 - 9 omesse perchè foglie

BFS - breadth first search

È una tecnica algoritmica che permette di effettuare attraversamenti o ricerche su alberi e, con qualche piccola modifica, su generici grafi. Il concetto di fondo è molto semplice: a partire da un nodo radice, si scorrono tutti i vertici dell'albero procedendo in profondità.




```
func bfs(root R):
    S = empty_queue
    S.add(R)
    while S is not empty:
        V = S.remove()
        do_stuff(V)
        for each child C of V:
        S.add(C)
```

BFS - Iterativa

i	Passo	V	Queue
	Add A		A

Tempo

BFS - Iterativa

i	Passo	V	Queue
	Add A		A
1	V = remove do_stuff(V = A) add A's children	A	<EMPTY> <EMPTY> B C

Tempo

BFS - Iterativa

i	Passo	V	Queue
	Add A		A
1	V = remove	A	<EMPTY>
	do_stuff(V = A)		<EMPTY>
	add A's children		B C
2	V = remove	B	C
	do_stuff(V = B)		C
	(B has no children)		C

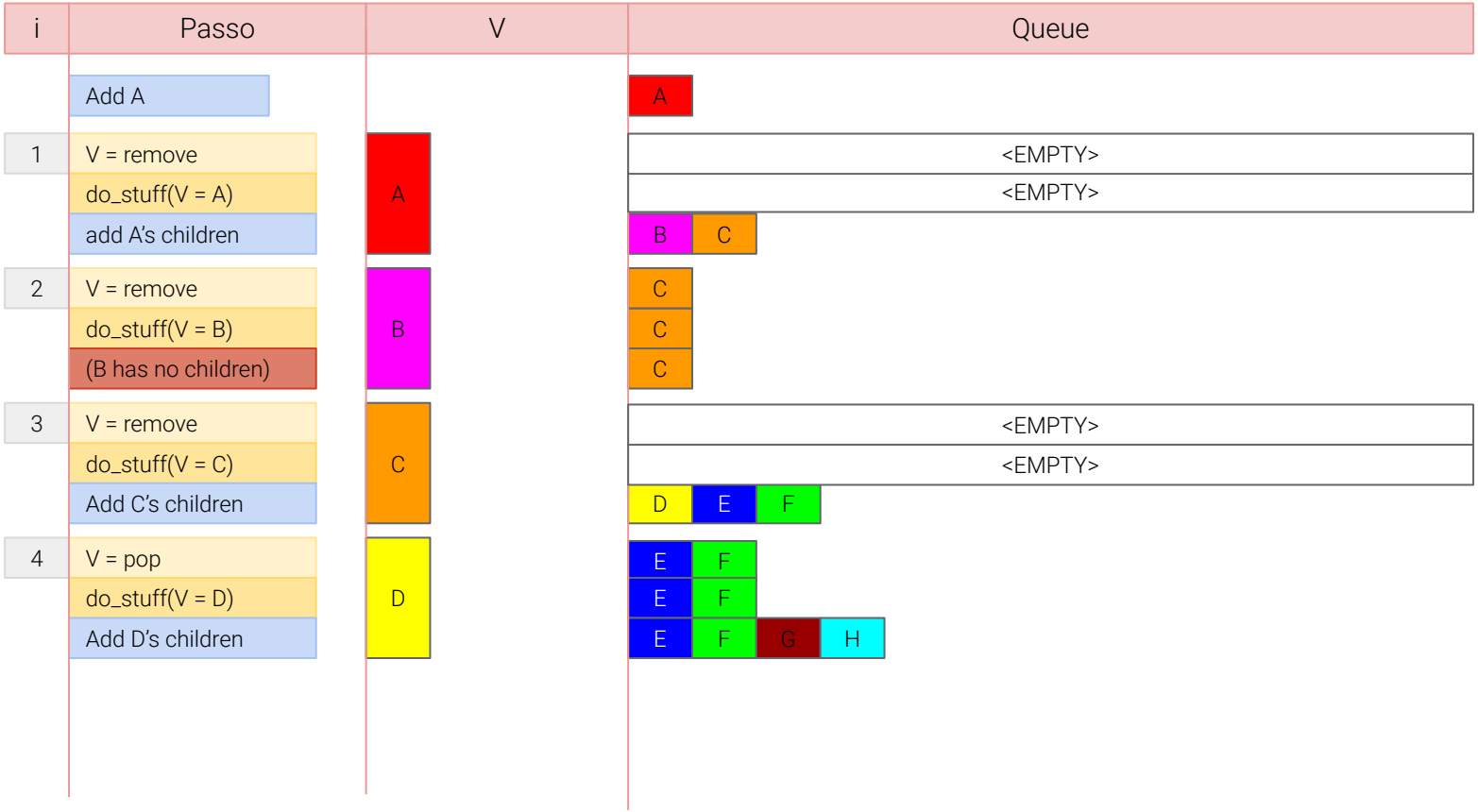
Tempo

BFS - Iterativa

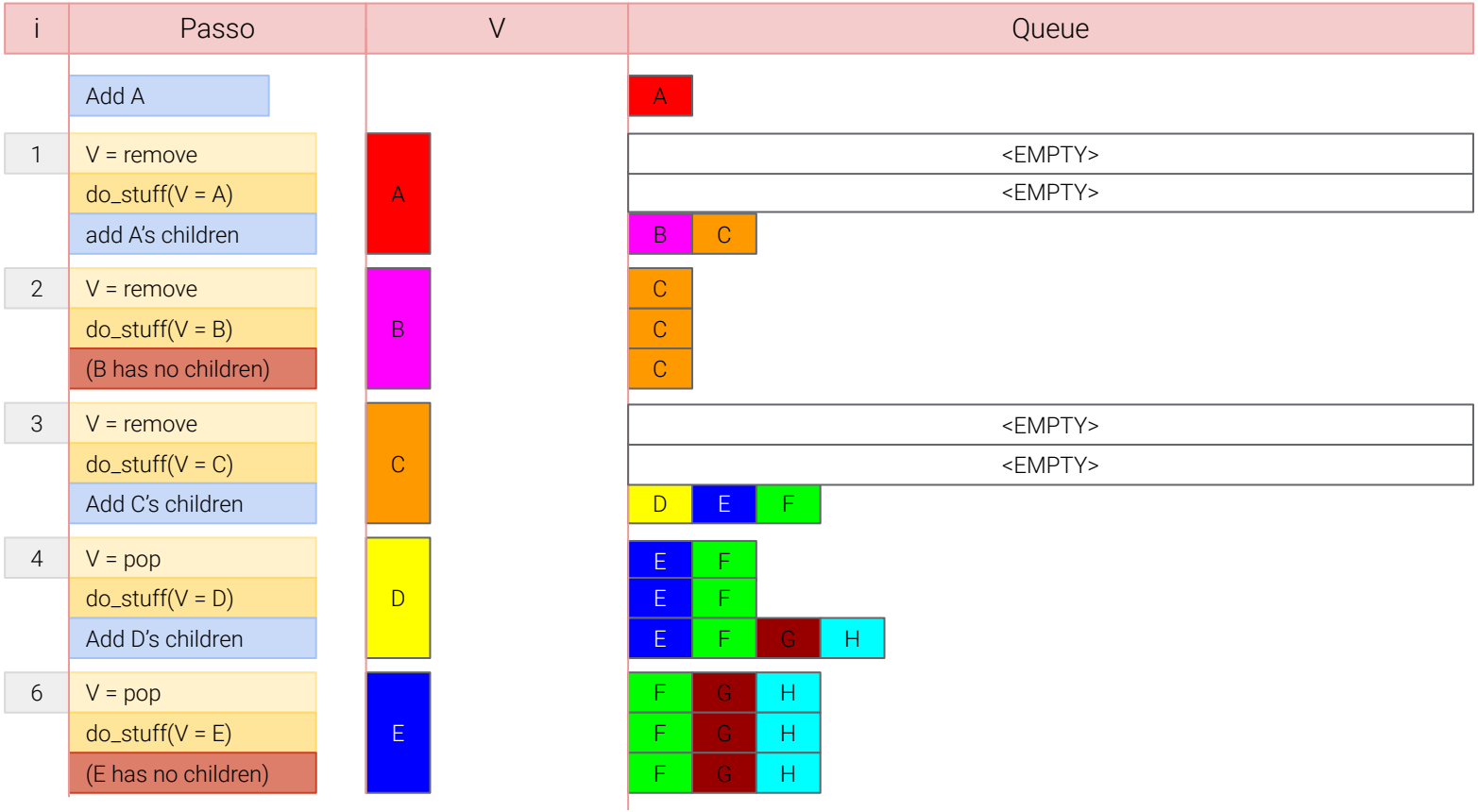
i	Passo	V	Queue
	Add A		A
1	V = remove	A	<EMPTY>
	do_stuff(V = A)		<EMPTY>
	add A's children		B C
2	V = remove	B	C
	do_stuff(V = B)		C
	(B has no children)		C
3	V = remove	C	<EMPTY>
	do_stuff(V = C)		<EMPTY>
	Add C's children		D E F

Tempo

BFS - Iterativa



BFS - Iterativa

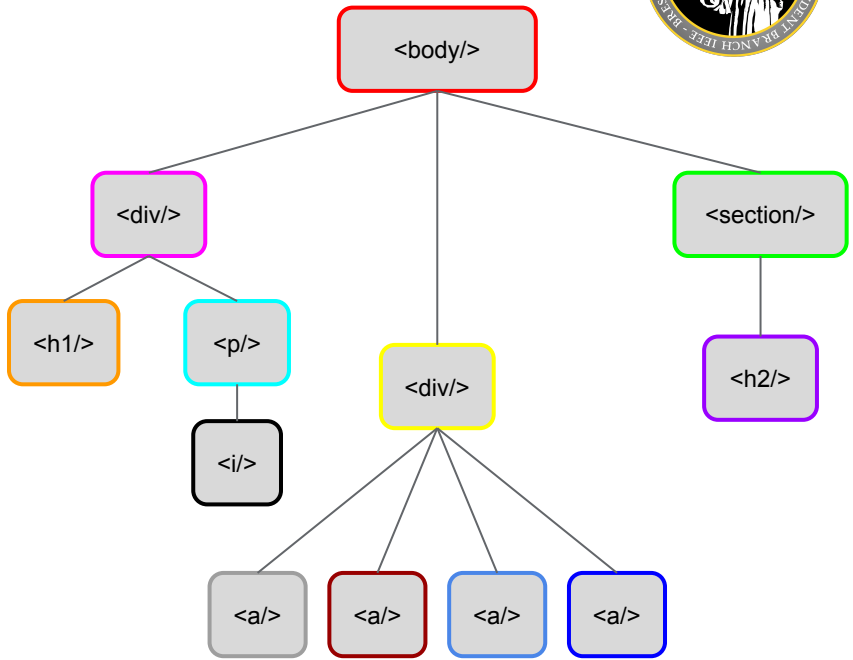


Tempo

Iterazioni 7 - 9 omesse perchè foglie

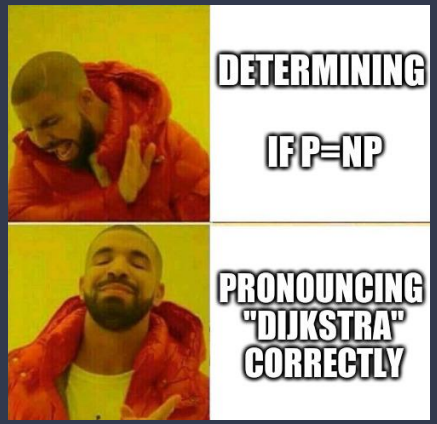
L'XML non è altro che un albero!

```
<body>
        <div class="header">
                <h1>PROGRAMMA ARNALDO</h1>
                <p><i>student driven learning</i></p>
        </div>
        <div class="navbar">
            <a href="---">Discord</a>
            <a href="---">Telegram</a>
            <a href="lessons.html">PDF</a>
            <a href="https://ieeesb.unibs.it">Student
        Branch</a>
        </div>
        <section class="container" style="max-width:600px">
            <h2>IL PROGRAMMA</h2>
        </section>
</body>
```



Pathfinding

Dijkstra's algorithm

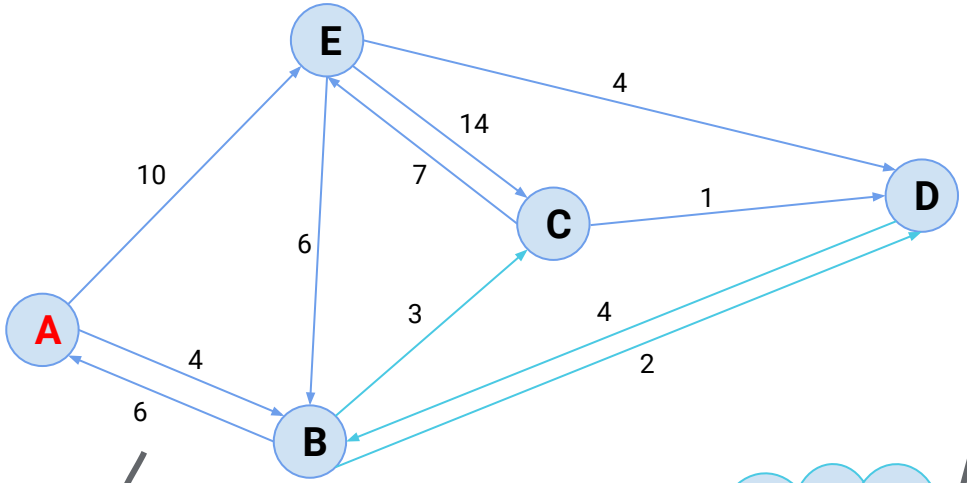


Ideato dall'informatico olandese **Edsger Dijkstra** nel 1956, è un algoritmo utilizzato nei più svariati ambiti per ottenere, dato un grafo, l'albero dei cammini minimi da un determinato **nodo di partenza verso tutti gli altri**.

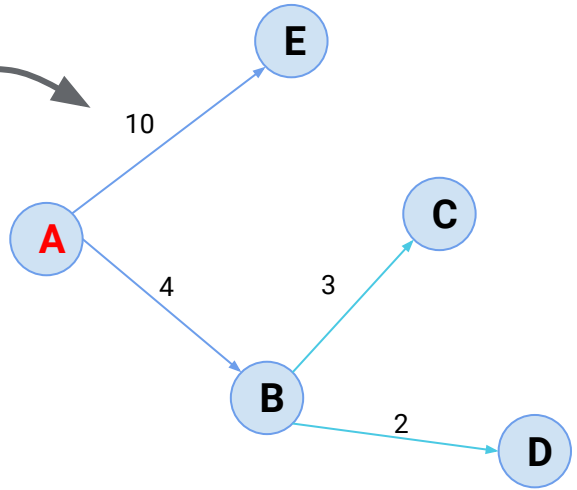
Tale algoritmo si basa sul semplice ma fondamentale **principio di ottimalità**:

Dati due nodi in un grafo, il cammino ottimo che unisce tali nodi, se esiste, è composto dall'unione di sotto-cammini ottimi.

Dijkstra

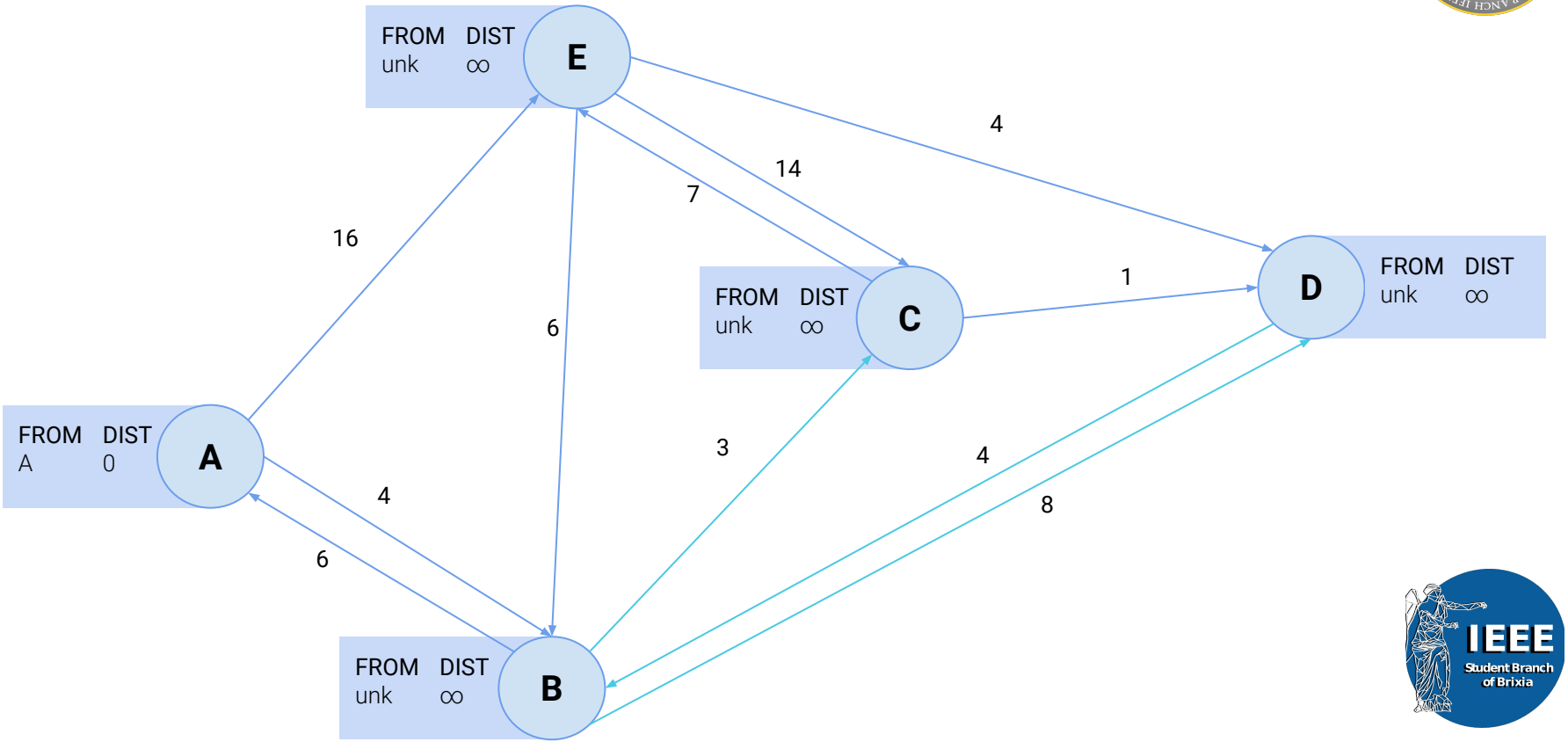


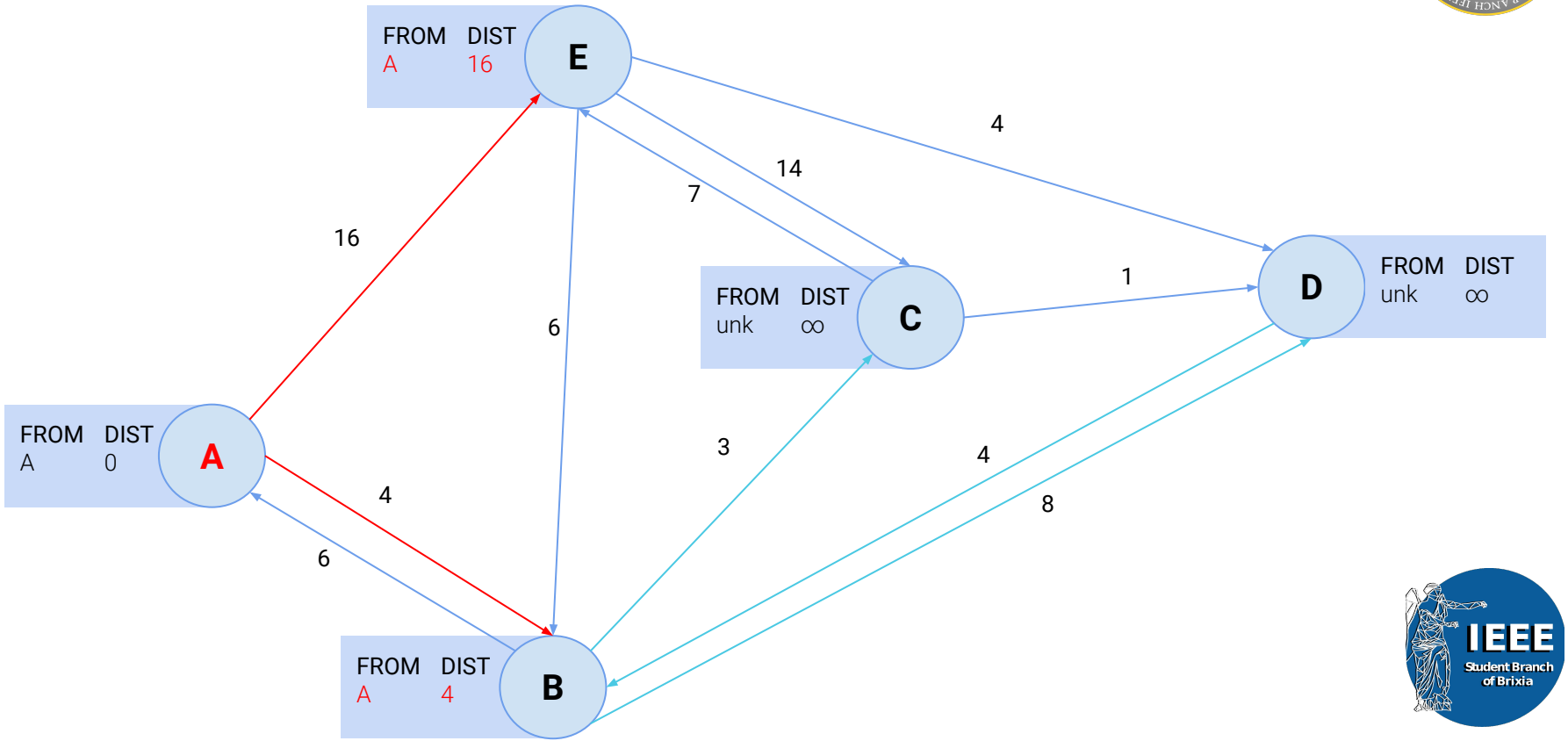
DIJKSTRA's
MAGIC

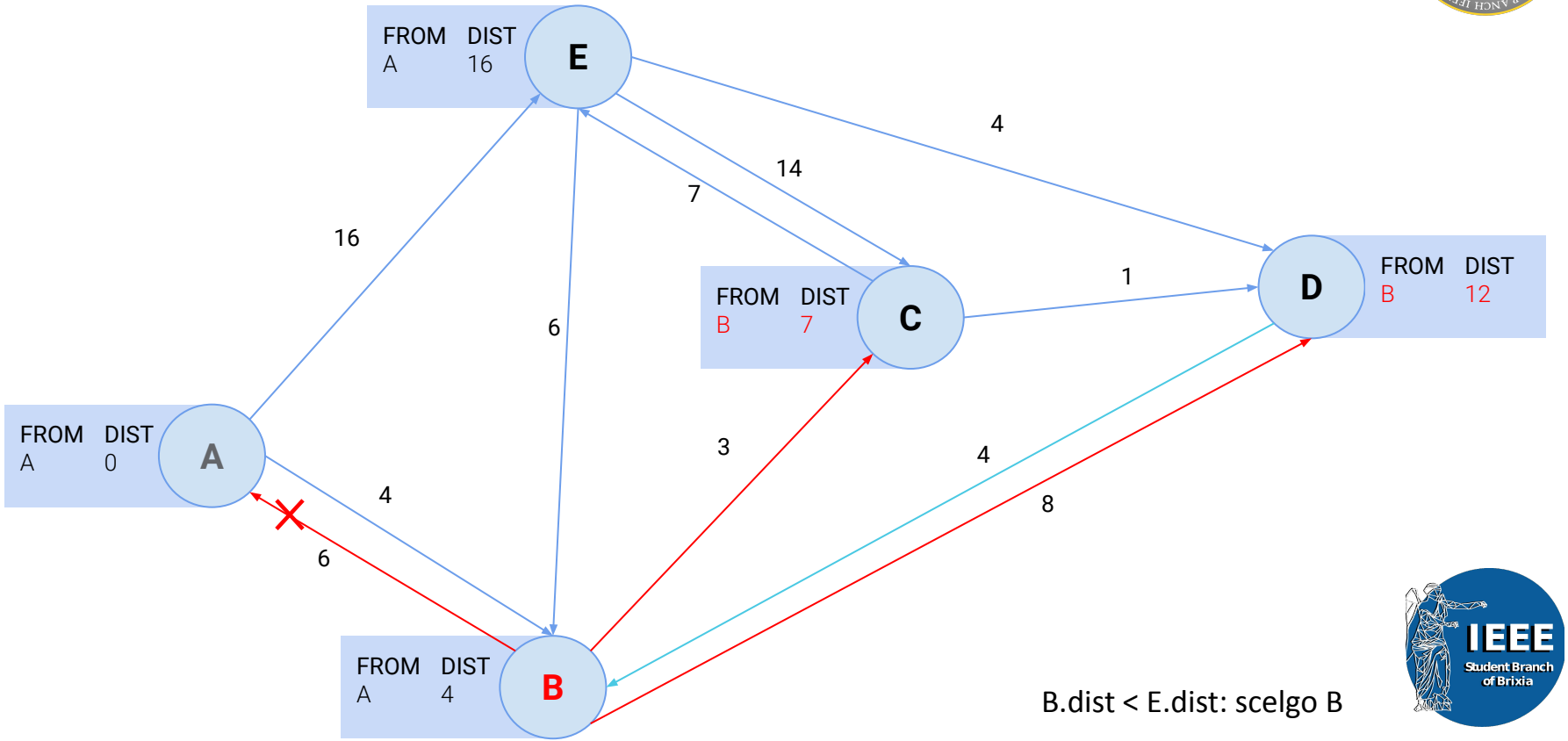


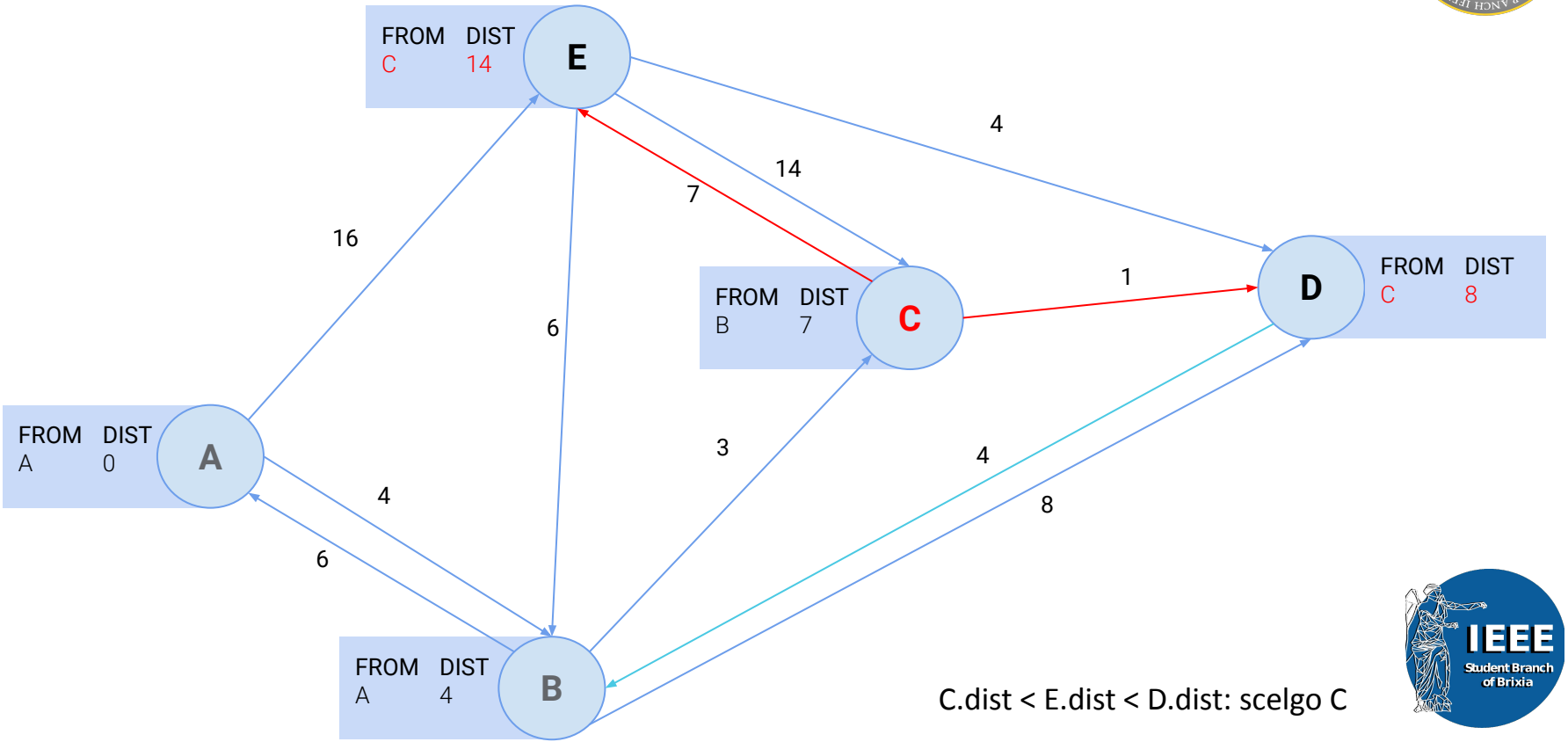
Dijkstra's magic

1. Si crea un insieme **Q** contenente tutti i nodi da visitare:
   inizialmente, tutti i nodi del grafo
2. Si crea una tabella contenente, per ogni nodo del grafo, la distanza minima calcolata dal nodo origine -*inizialmente posta a infinito*- e il nodo precedente nel cammino minimo -*inizialmente sconosciuto*-
3. Si itera su tutti i nodi in **Q**, prendendo di volta in volta il nodo **T** che, rispetto agli altri nodi di **Q**, ha distanza minore dall'origine. ***La precedenza è importantissima per il principio di ottimalità***
4. Per ogni vicino (neighbour) **N** del nodo **T**:
   a. si calcola la distanza
      ```
      calc_dist = distanza(origine, T) + peso(T, N)
      ```
   b. se `calc_dist` è minore della distanza presente in tabella per il nodo **N**, allora nella riga **N** della tabella si scrivono `calc_dist` come distanza e **T** come nodo precedente
5. Terminata l'iterazione, **T** viene rimosso da **Q**.
   Quando **Q** è vuoto l'algoritmo termina.

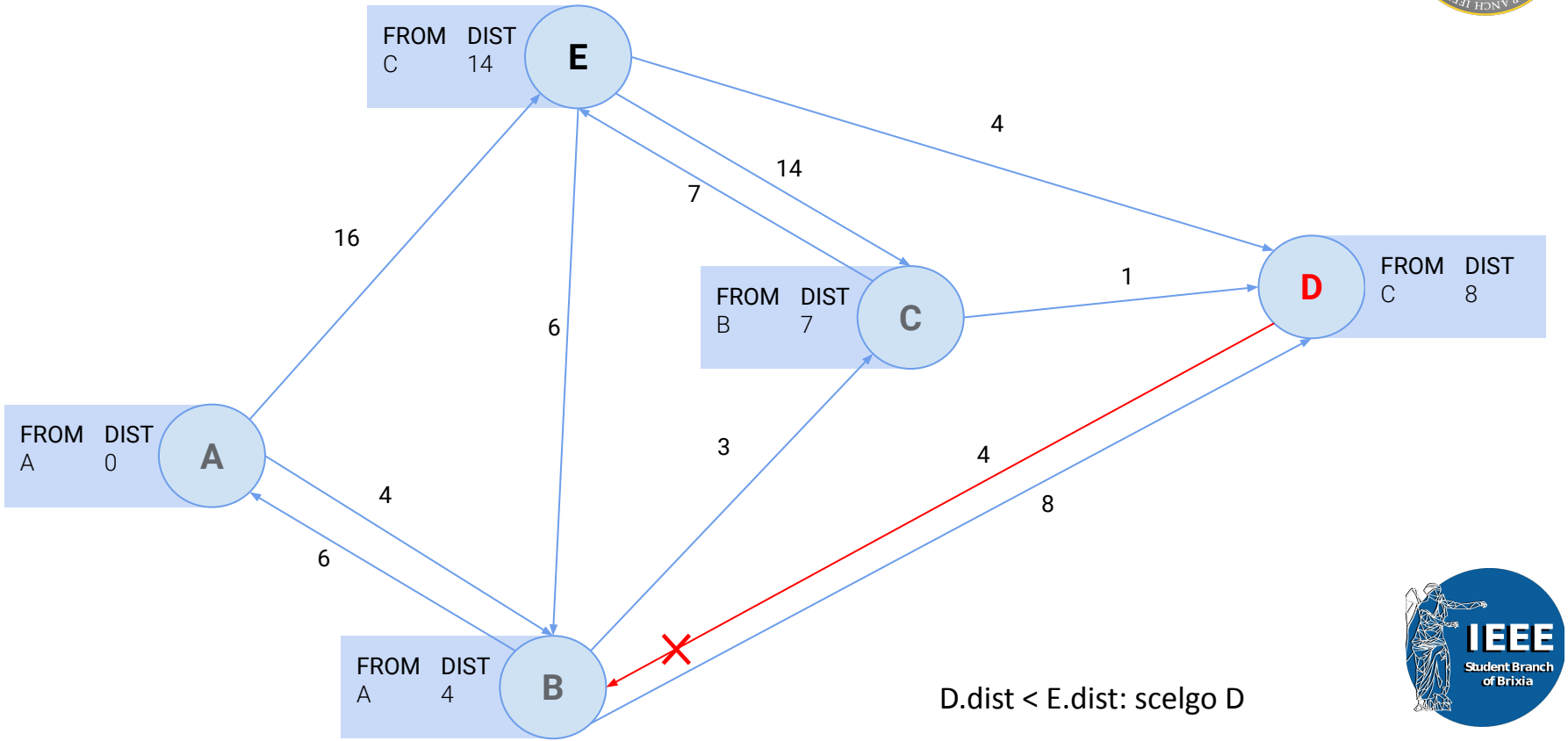

FROM DIST
unk ∞
E
4
14
7
16
FROM DIST
unk ∞
C
1
D
FROM DIST
unk ∞
6
FROM DIST
A 0
A
3
4
4
6
8
FROM DIST
unk ∞
B

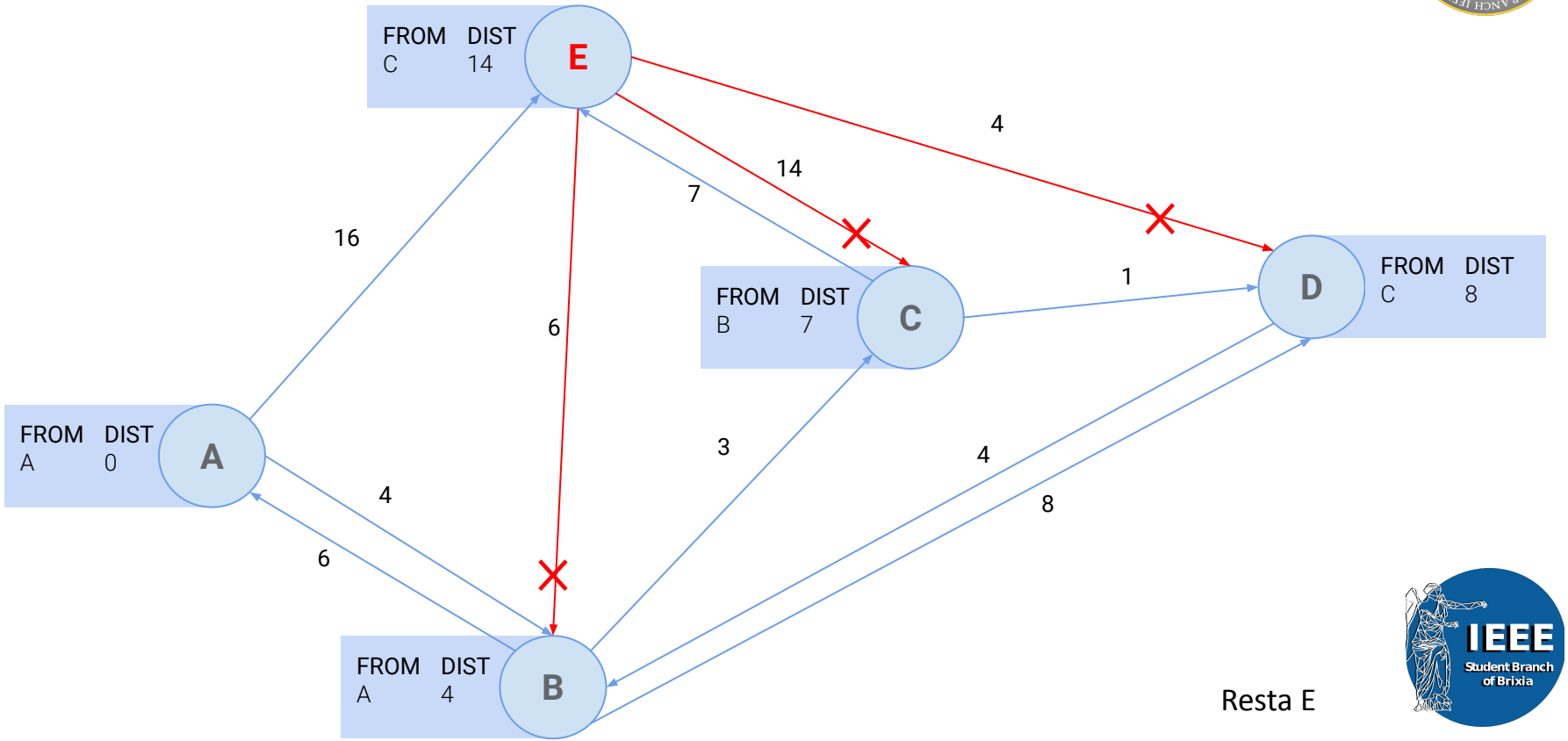

FROM DIST
A 16
E
4
14
7
16
6
FROM DIST
unk ∞
C
1
D
FROM DIST
unk ∞
FROM DIST
A 0
A
3
4
4
8
6
FROM DIST
A 4
B


FROM DIST
A 16
E
4
14
7
16
6
FROM DIST
B 7
C
1
D
FROM DIST
B 12
FROM DIST
A 0
A
4
6
3
4
8
FROM DIST
A 4
B
B.dist < E.dist: scelgo B

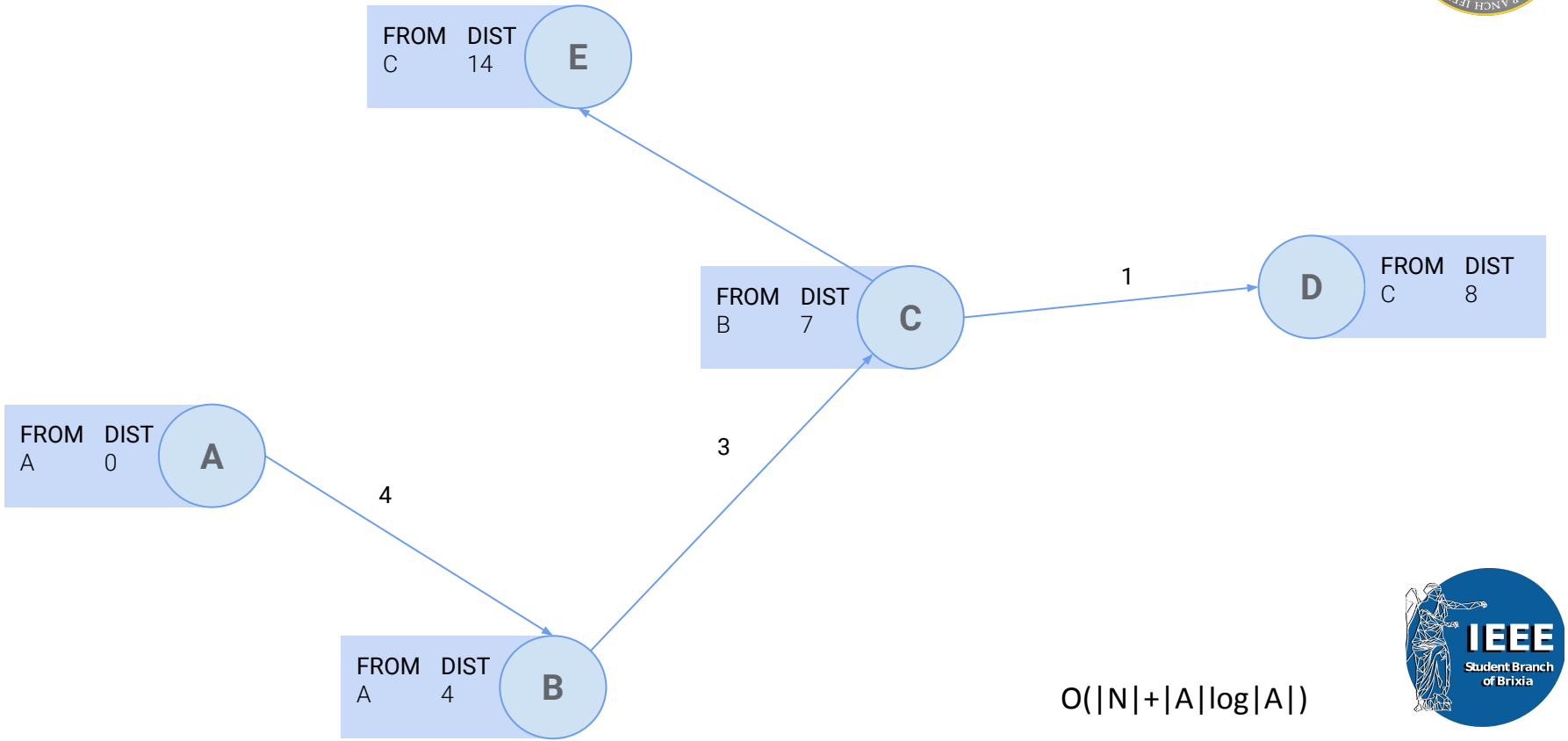


C.dist < E.dist < D.dist: scelgo C


FROM DIST
C 14
E
4
14
7
16
1
FROM DIST
B 7
C
D
FROM DIST
C 8
6
FROM DIST
A 0
A
3
4
8
4
6
FROM DIST
A 4
B
D.dist < E.dist: scelgo D


FROM DIST
C 14
E
4
14
7
16
6
FROM DIST
B 7
C
1
D
FROM DIST
C 8
FROM DIST
A 0
A
4
6
3
4
8
FROM DIST
A 4
B

Resta E


FROM DIST
C 14
E
FROM DIST
B 7
C
1
D
FROM DIST
C 8
FROM DIST
A 0
A
4
3
FROM DIST
A 4
B

$O(|N|+|A|\log|A|)$

Dijkstra - osservazioni

- L'algoritmo di Dijkstra richiede che i pesi degli archi siano sempre positivi o nulli. In alcune varianti (Bellman-Ford) questi sono concessi ma ne risente la complessità e possono sorgere problemi di loop infiniti.
- Una volta che un nodo è stato elaborato una volta ed è stato rimosso da **Q**, esso non viene più toccato.
  Di conseguenza, se si vuole trovare il cammino minimo verso un determinato nodo, **ci si può fermare subito dopo l'iterazione che lo riguarda**. Al contrario altri algoritmi, come **Bellman-Ford** ri-inseriscono i nodi modificati nell'insieme **Q** ma hanno richieste riguardo al nodo scelto per l'iterazione corrente.
- L'algoritmo di Dijkstra esegue sostanzialmente una **BFS** su grafo: lo si può notare chiaramente facendo girare l'algoritmo su un grafo con tutti i pesi posti a 1.
- La complessità dell'algoritmo, nella sua implementazione base, è **$O(n^2)$**. Esistono tuttavia versioni specializzate dell'algoritmo, basate su ipotesi più stringenti, che riescono ad ottenere prestazioni migliori;
- Altre tecniche di ottimizzazione ricorrono all'utilizzo di strutture dati "migliori"

A*

* *Non scherzo, si chiama proprio "A* " (A STAR, non "A asterisco")*

L'algoritmo di Dijkstra funziona benissimo per grafi di dimensioni modeste, ma quando il numero di nodi aumenta, calcolare l'intero albero dei cammini minimi può essere oneroso.

Abbiamo già visto che possiamo interrompere l'esecuzione dell'algoritmo una volta elaborato il nodo di destinazione; tuttavia possiamo fare di meglio.

L'algoritmo A (pronuncia: "A-star") è un algoritmo di ricerca su grafi che "si fa aiutare" da una funzione euristica: una funzione in grado di stimare qual è la direzione migliore, probabilisticamente, in cui esplorare il grafo.*

L'implementazione di tale funzione dipende da cosa rappresenta il grafo e non è sempre possibile.

Intuitivamente, se il grafo rappresenta una mappa geografica e noi vogliamo spostarci da **Brescia** a **Roma**, la funzione euristica dovrebbe favorire l'esplorazione dei nodi verso sud.

A*

Il funzionamento di **A*** è, grossolanamente, simile a quello dell'algoritmo di Dijkstra:

- Ad ogni iterazione, **A*** decide in che direzione esplorare il grafo, scegliendo il nodo n che minimizzi la funzione **f(n) = g(n) + h(n)**, in cui:
  - **n** è il nodo successivo
  - **g(n)** = (costo del cammino fino al nodo attuale) + (costo dal nodo attuale a n)
  - **h(n)** = funzione euristica che stima il costo da n alla destinazione
- La funzione euristica può essere implementata in qualunque modo, a patto che sia ammissibile: non deve mai sovrastimare il costo tra due nodi.
  - Un esempio di euristica ammissibile per una mappa geografica è la distanza in linea d'aria tra due punti.
- Nelle Risorse aggiuntive è presente il link ad un possibile pseudocodice dell'algoritmo.

A è sempre in grado di trovare il cammino ottimo tra due nodi; se la struttura del grafo non è particolarmente "intricata" può impiegare un tempo molto minore rispetto a Dijkstra.*

- Algoritmo di Kruskal, un altro algoritmo di pathfinding con un approccio diverso: https://it.wikipedia.org/wiki/Algoritmo_di_Kruskal
- Definizione di euristica: https://it.wikipedia.org/wiki/Euristica
- Algoritmo A* con spiegazione dettagliata e pseudocodice: https://en.wikipedia.org/wiki/A*_search_algorithm
- Algoritmo di Floyd-Warshall, un algoritmo per calcolare il cammino minimo per tutte le coppie di un grafo: https://it.wikipedia.org/wiki/Algoritmo_di_Floyd-Warshall
- Algoritmo di Bellman-Ford, una variante dell'algoritmo di Dijkstra che ammette anche archi di peso negativo: https://it.wikipedia.org/wiki/Algoritmo_di_Bellman-Ford

Presentazione realizzata per lo Student Branch IEEE dell'Universitá degli Studi di Brescia, in occasione del Programma Arnaldo 2023

corazzinamarco33@ieee.org
matteo.boniotti@ieee.org
stefano.agnelli@ieee.org
kibo@ieee.org

ieeesb.unibs.it

Grazie per l'attenzione