BIOINFORMATICA
II

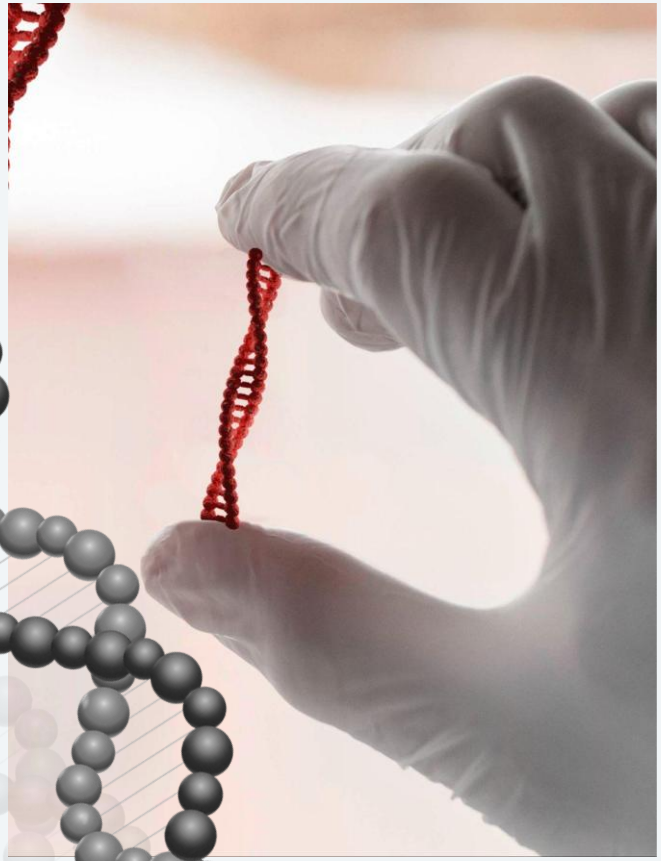
ALGORITMI DI ASSEMBLAGGIO

Argomenti

01 ASSEMBLAGGIO

02 ALGORITMI DI ASSEMBLAGIO

03 SOFTWARE DI ASSEMBLAGGIO



01

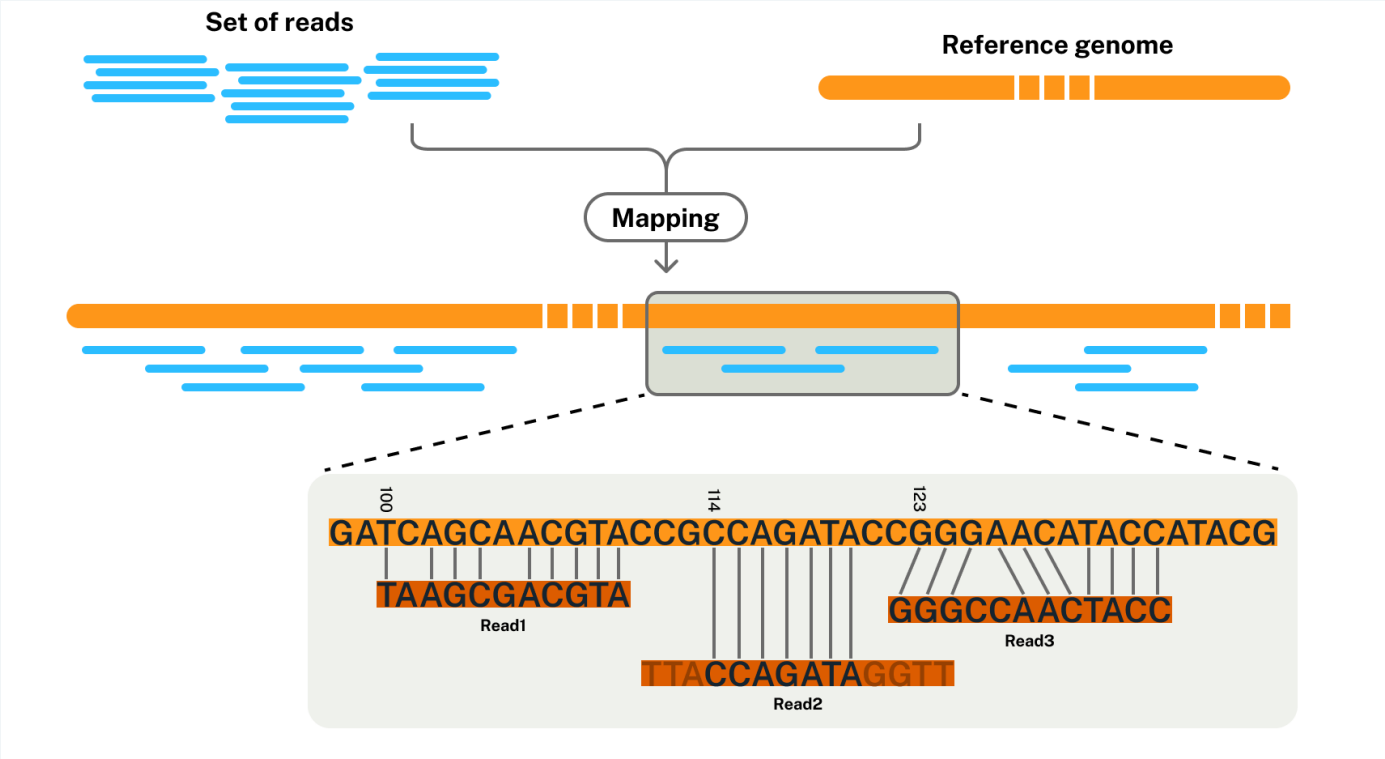
ASSEMBLAGGIO

ASSEMBLAGGIO

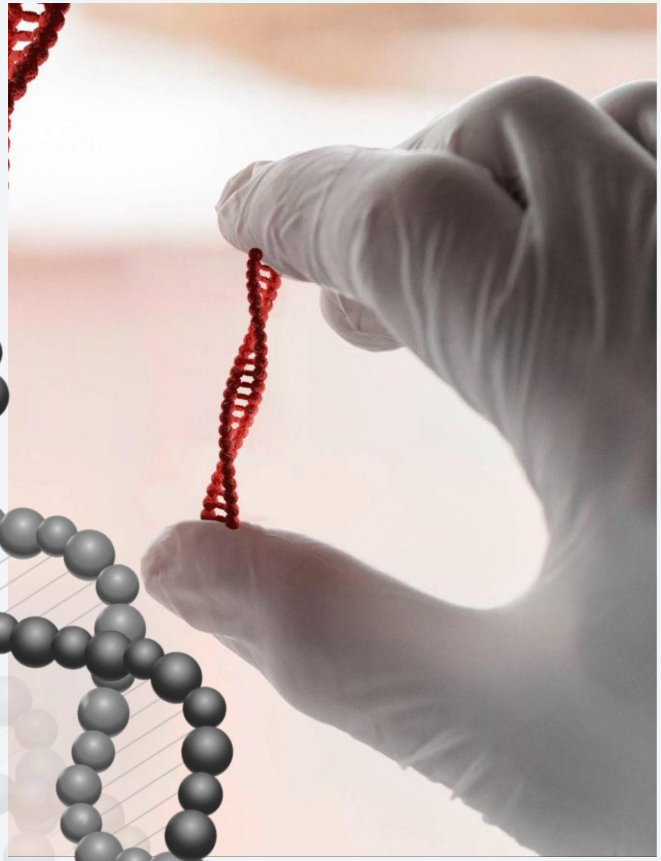
L'assemblaggio del genoma è il processo computazionale utilizzato per ricostruire l'intera sequenza del DNA di un organismo a partire da brevi frammenti di sequenza (reads) prodotti da tecnologie di sequenziamento (come Illumina o PacBio)

Questo processo è fondamentale per ottenere una rappresentazione completa e accurata del genoma, utile in molti campi della biologia, come la genomica comparativa, la genetica medica e l'agricoltura

ASSEMBLAGGIO



https://www.geneious.com/guides/assembling-dna-sequences



02

ALGORITMI DI ASSEMBLAGGIO

ALGORITMI DI ASSEMBLAGGIO

Gli algoritmi di assemblaggio si caratterizzano dalla presenza o assenza di un genoma di riferimento:

Reference-guided assembly: utilizza un genoma di riferimento già noto per guidare l'assemblaggio. Più rapido e preciso, ma non adatto a organismi molto distanti dal riferimento

De novo assembly: ricostruisce il genoma senza una sequenza di riferimento. Utilizzato per organismi nuovi, non modello o poco studiati

ASSEMBLAGGIO: Motivi

Ricostruzione di nuovi genomi: utile per studiare organismi di cui non si conosce ancora la sequenza genomica.
Genomica comparativa: confrontare genomi di diverse specie per comprendere l'evoluzione, identificare geni conservati o determinare specificità genomiche.
Scoperta di varianti genetiche: analizzare mutazioni, inversioni, duplicazioni o altri cambiamenti strutturali del DNA.
Metagenomica: identificare e ricostruire genomi di organismi da campioni ambientali, per esempio per studiare microbiomi.
Medicina personalizzata: studiare genomi umani per identificare mutazioni legate a malattie genetiche.

ALGORITMI DI ASSEMBLAGGIO

Gli algoritmi principali per l'assemblaggio del genoma si basano su diverse tecniche di gestione e analisi delle sequenze.
I più comuni:

ALGORITMI DI ASSEMBLAGGIO

Grafo di de Bruijn
Principio: Divide le sequenze (reads) in sottostringhe di lunghezza k (k-mers) e costruisce un grafo orientato, dove i nodi rappresentano i k-mers e gli archi rappresentano sovrapposizioni tra i k-mers. La ricerca di un cammino permette la ricostruzione del genoma/trascrittoma

Vantaggi: Efficiente per sequenze brevi

Svantaggi: Può soffrire di errori nella sequenza e di grafi complessi per genomi con molte parti che si ripetono

Esempi di software:

SPAdes: per il de novo assembly di genomi microbici.

Velvet: adatto per dati Illumina.

ABySS: progettato per sequenze lunghe e distribuzioni parallele.

ALGORITMI DI ASSEMBLAGGIO

Overlap-Layout-Consensus (OLC)

Principio: Confronta ogni sequenza con tutte le altre per identificare sovrapposizioni, costruisce un grafo di layout e genera una sequenza consenso, cioè una rappresentazione unica e sintetica di una regione del genoma ricostruita dalle letture sovrapposte (sequenze di input) durante l'assemblaggio del genoma

Vantaggi: Funziona bene con reads lunghi, come quelli di PacBio o Oxford Nanopore.

Svantaggi: Computazionalmente intensivo, non adatto a sequenze brevi con profondità elevata

Esempi di software: **Celera Assembler**: uno dei primi assemblatori basati su OLC. **Canu**: ottimizzato per reads lunghe e rumorose. **Flye**: progettato per assemblaggi di alta qualità con dati Nanopore.

ALGORITMI DI ASSEMBLAGGIO

String Graphs
Principio: Simile a OLC, ma elimina ridondanze

Esempi di software:
SGA (String Graph Assembler): utilizza reads corte e consente un'efficiente gestione delle sovrapposizioni.

ALGORITMI DI ASSEMBLAGGIO

Algoritmi ibridi

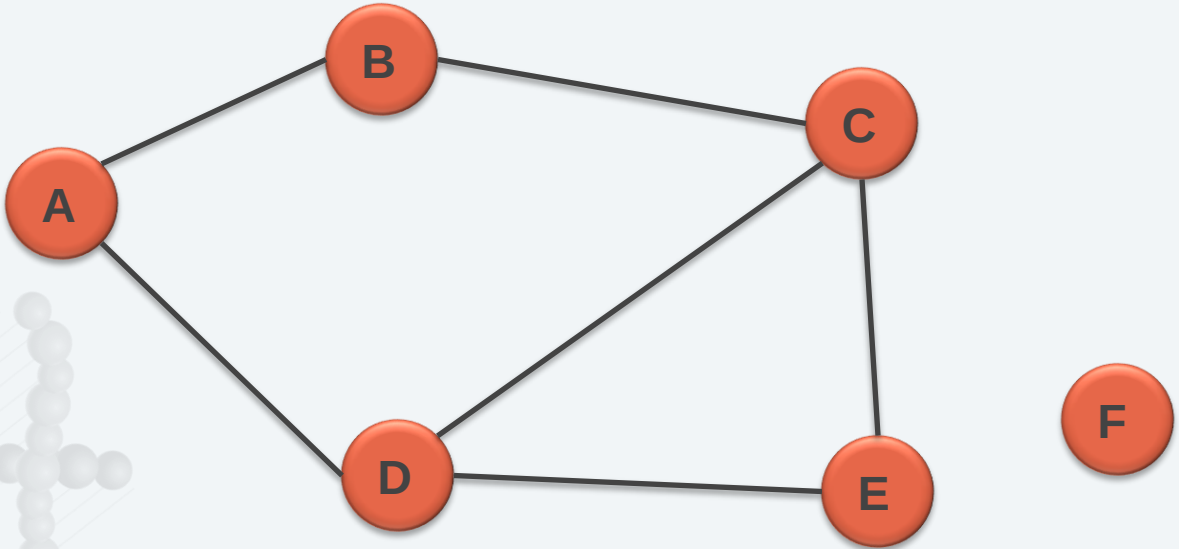
Principio: Combinano reads corte (ad alta accuratezza) e reads lunghe (ad alta copertura) per ottenere assemblaggi più completi e accurati.

Esempi di software:
MaSuRCA: ottimizza l'assemblaggio combinando dati Illumina e PacBio/Nanopore.
HybridSPAdes: integra reads lunghe e corte per migliorare la qualità.

PRINCIPI DI TEORIA DEI GRAFI

GRAFO: Definizione

Un grafo G è una coppia di elementi (*V*, *E*) dove:
V è un insieme detto **insieme dei vertici o nodi (i nodi hanno una etichetta utile a distinguerli, es: A,B,…,F)**
E è un insieme detto **insieme degli archi**

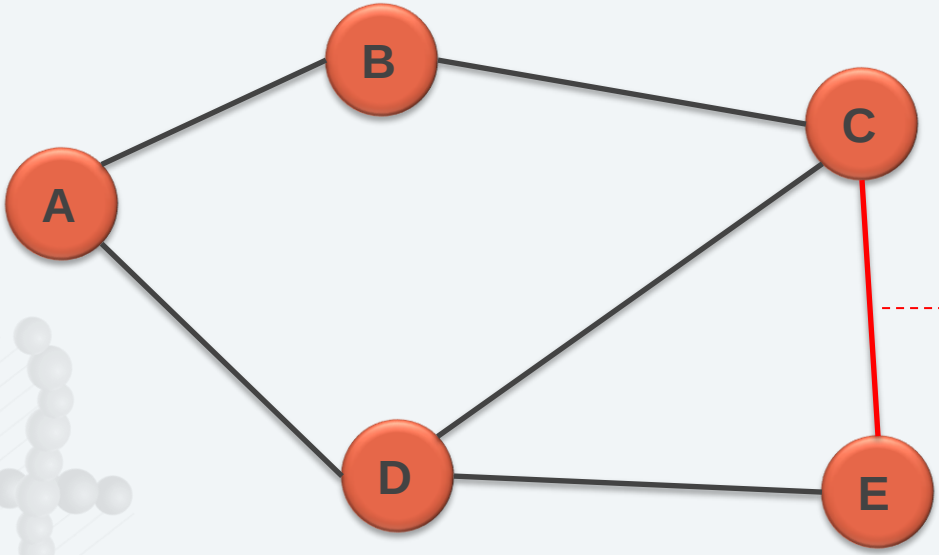


$V = \{A, B, C, D, E, F\}$

$E = \{(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)\}$

GRAFO: Definizione

Un arco di un grafo G è una coppia di nodi (u,v) dove u e v appartengono a *V*

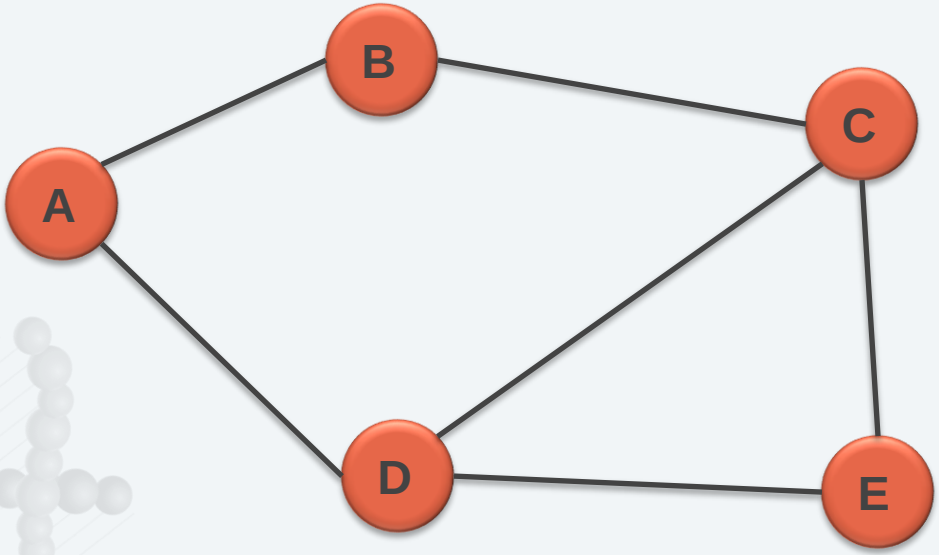


$V = \{A, B, C, D, E, F\}$

$E = \{(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)\}$

GRAFO NON ORIENTATO: Definizione

Un grafo non orientato G è una coppia (*V*, *E*) dove:
V è un insieme detto **insieme dei vertici o nodi**
E è un insieme non ordinato di coppie di nodi detto ***insieme degli archi***



F

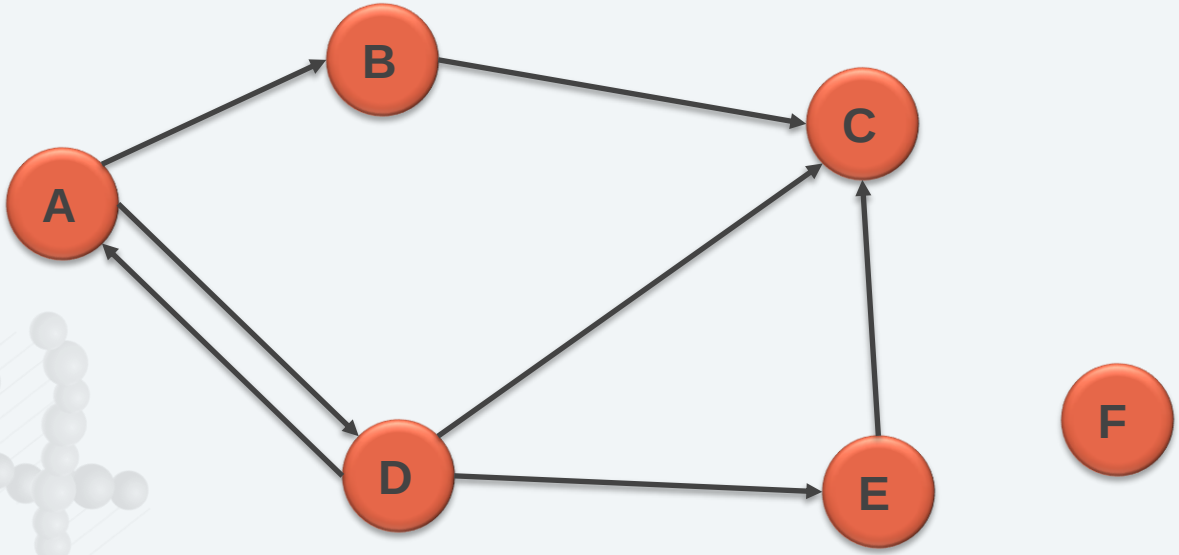
$V = \{A, B, C, D, E, F\}$

$E = \{(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)\}$

OSS: = (A,D) e (D,A) individuano lo stesso arco

GRAFO ORIENTATO: Definizione

Un grafo orientato G è una coppia (*V*, *E*) dove:
V è un insieme detto **insieme dei vertici o nodi**
E è una relazione binaria tra nodi detta **insieme degli archi orientati**



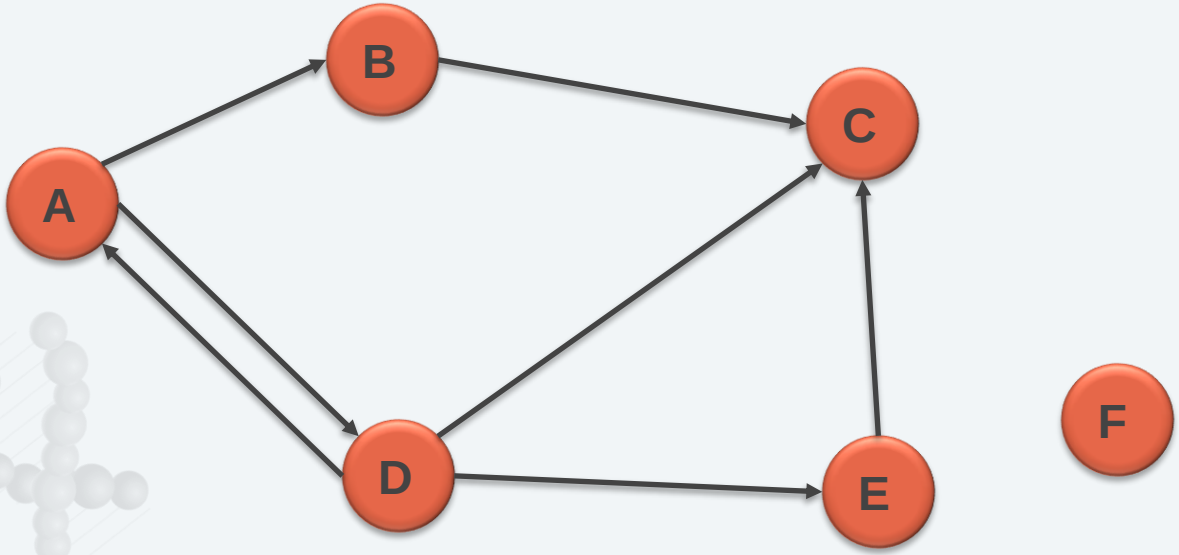
V = {A, B, C, D, E}

E ={(A,B),(A,D),(B,C),(D,C), (E,C), (D,E),(D,A)}

OSS: = (A,D) e (D,A) individuano due archi diversi

GRAFO ORIENTATO: Incidenza

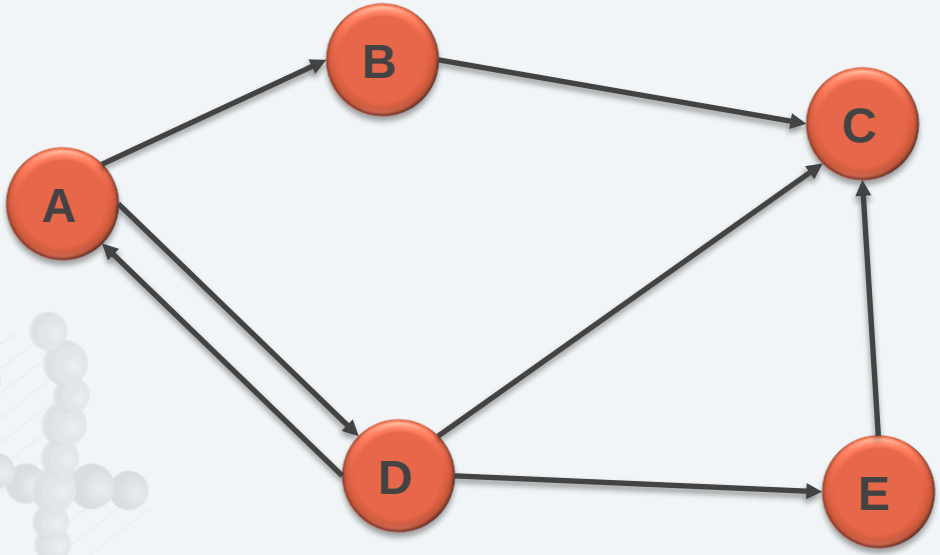
In un grafo orientato, un **arco (w,v)** si dice **incidente da w in v se l'arco parte dal nodo w e giunge nel nodo v**



(A,B) è incidente da A in B
(A,D) è incidente da A in D
(B,C) è incidente da B in C
(D,C) è incidente da D in C
(E,C) è incidente da E in C
(D,E) è incidente da D in E
(D,A) è incidente da D in A

GRAFO ORIENTATO: Adiacenza

Un nodo **w si dice adiacente a v** se e solo se l'arco (v,w) appartiene ad *E*

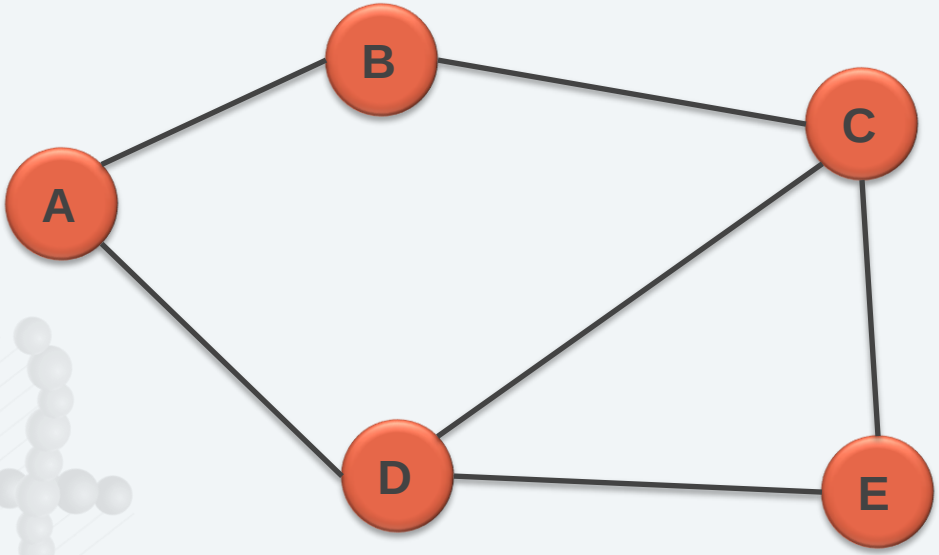


F

B è adiacente ad A
C è adiacente a B e a D e a E
A è adiacente a D
D è adiacente a A
E è adiacente a D
B NON è adiacente a D né a E né F
F NON è adiacente ad alcun nodo

GRAFO NON ORIENTATO: Adiacenza

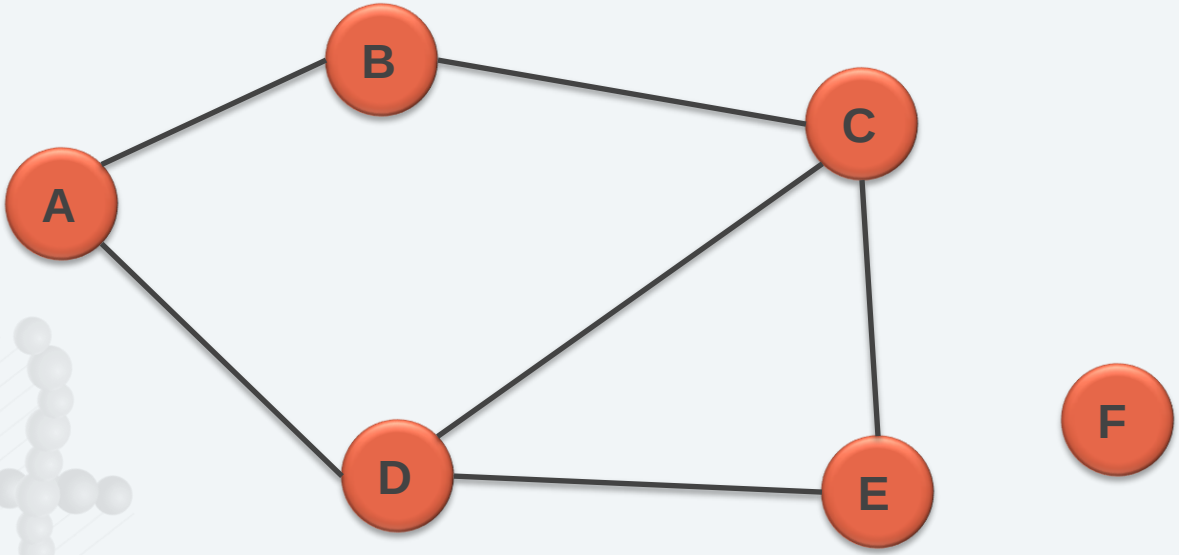
In un grafo non orientato la relazione di adiacenza tra nodi è simmetrica



A è adiacente a D e B e viceversa
B è adiacente a A e C e viceversa
C è adiacente a B, D e E e viceversa
D è adiacente a C e A e E e viceversa
E è adiacente a C e D e viceversa
F NON è adiacente ad alcun nodo

GRAFO NON ORIENTATO: Grado

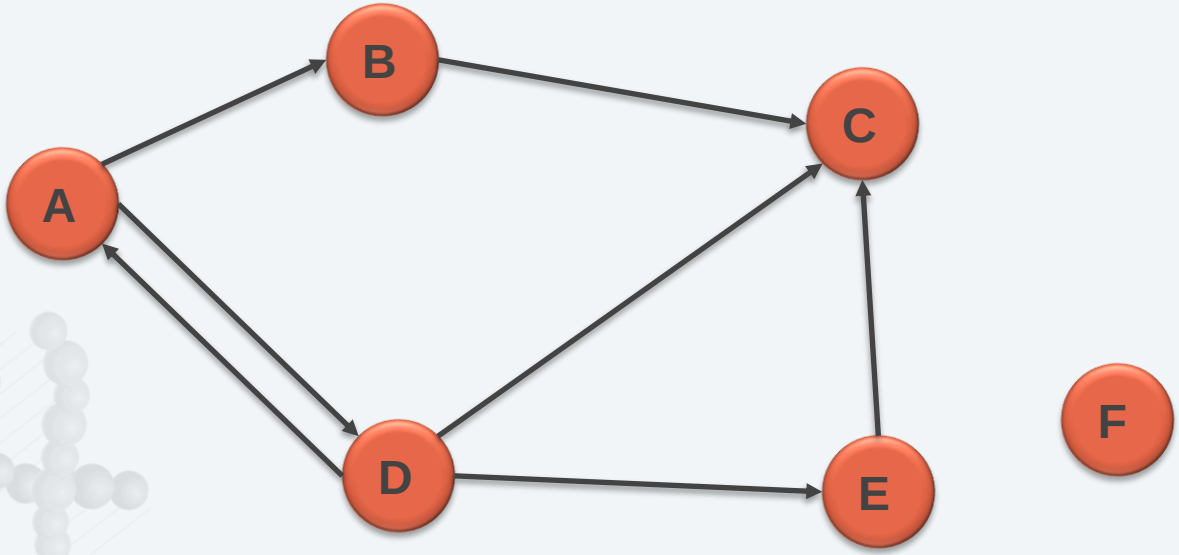
In un grafo non orientato il grado di un nodo è il numero di archi che da esso dipartono (o entrano)



A, B ed E hanno grado 2
C e D hanno grado 3
F ha grado 0

GRAFO ORIENTATO: Grado Uscente

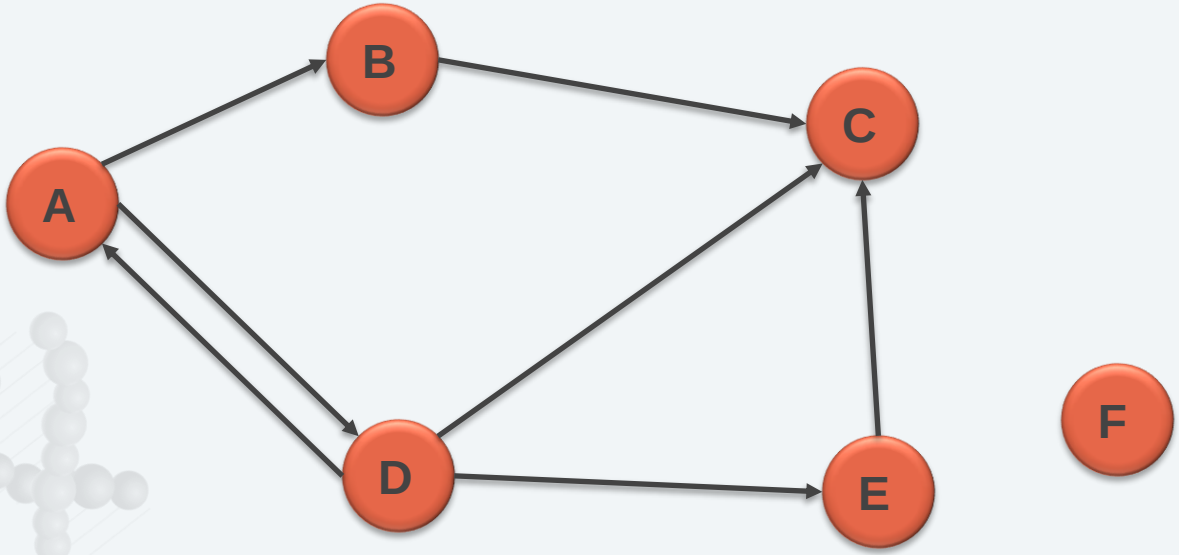
In un grafo orientato il grado uscente di un nodo è il numero di archi incidenti da esso (numero di archi uscenti per il nodo considerato)



A ha grado uscente 2
B ha grado uscente 1
C ha grado uscente 0
D ha grado uscente 3
D ha grado uscente 1
F ha grado uscente 0

GRAFO ORIENTATO: Grado Entrante

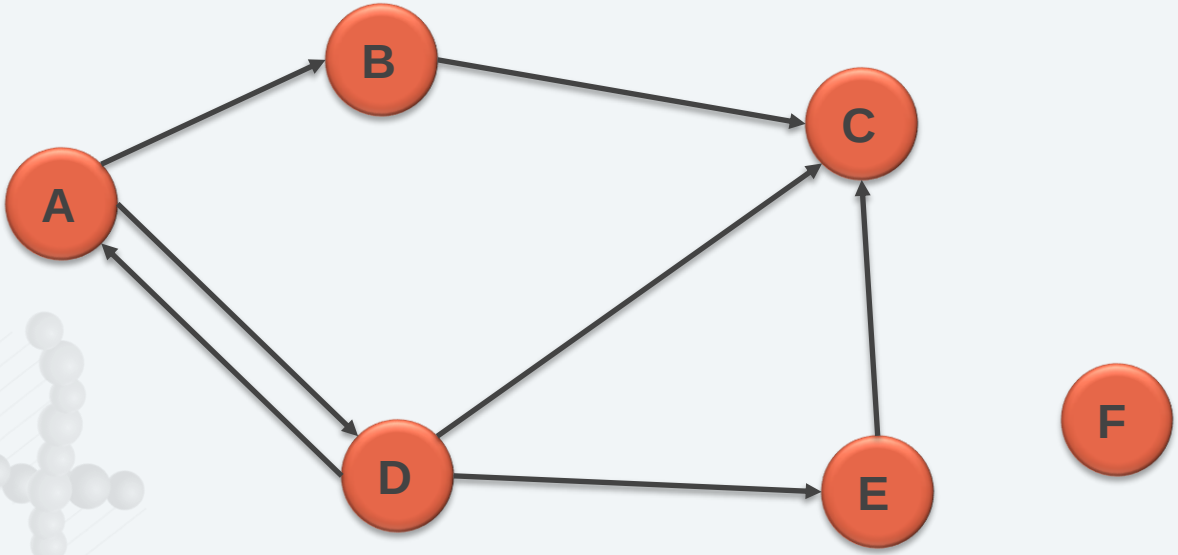
In un grafo orientato il grado entrante di un nodo è il numero di archi incidenti in esso (numero di archi entranti per il nodo considerato)



A ha grado entrante 1
B ha grado entrante 1
C ha grado entrante 3
D ha grado entrante 1
E ha grado entrante 1
F ha grado uscente 0

GRAFO ORIENTATO: Grado

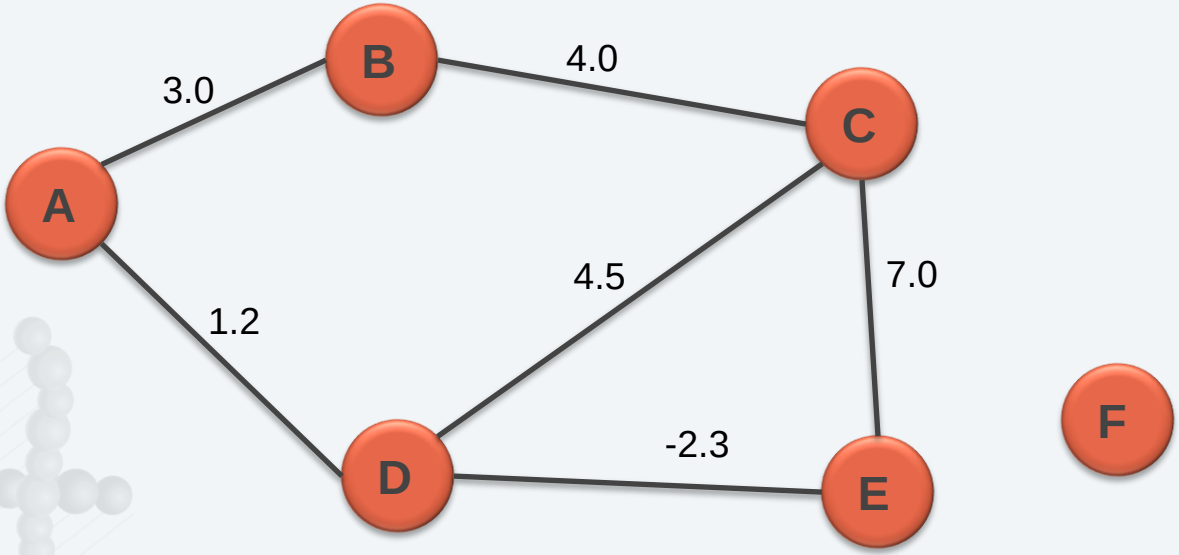
In un grafo orientato il grado di un nodo è la somma del suo grado entrante e del suo grado uscente



A e C hanno grado 3
B e E hanno grado 2
D ha grado 4
F ha grado 0

GRAFO PESATO: Definizione

In alcuni casi ogni arco ha un peso (o costo) associato.
Il costo può essere determinato tramite una funzione ***c: E→R***, dove R è l'insieme dei numeri reali
Quando tra due nodi non esiste un arco, si dice che il peso è infinito



c(A,B)=3.0
c(B,C)=4.0
c(A,D)=1.2
c(D,C)=4.5
c(D,E)=–2.3
c(C,E)=7.0
c(C,F)=infinito
c(A,E)=infinito
...

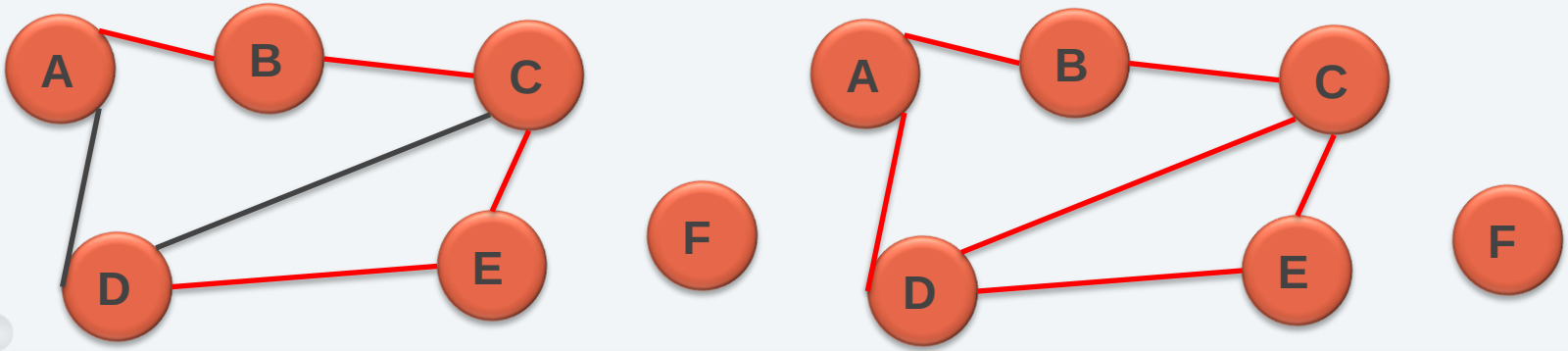
PERCORSO DI UN GRAFO: Definizione

Sia G = (V, E) un grafo

Un percorso nel grafo è una sequenza di nodi $<w_1, w_2, ..., w_n>$ tale che $(w_i, w_{i+1}) \in E$ per $1 \leq i \leq n-1$ (cioè ogni nodo è adiacente al successivo nella sequenza).

Cammino 1 (sinistra): <(A,B),(B,C),(C,E),(E,D)>

Cammino 2 (destra) <(A,B),(B,C),(C,E),(E,D),(D,C),(C,E)(E,D)(D,A)>

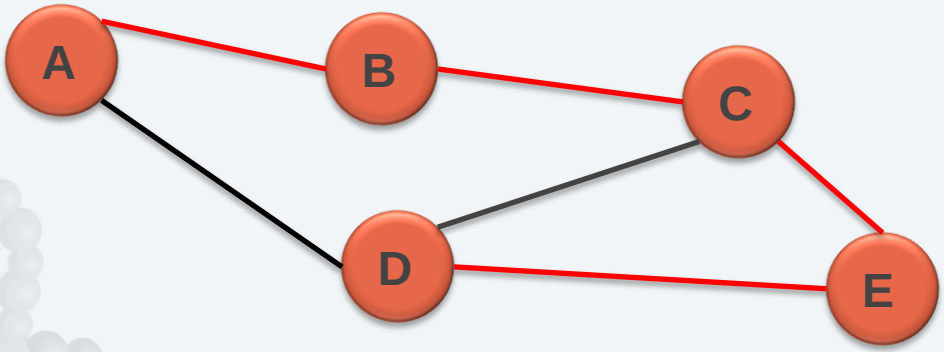


Il ***percorso*** $<w_1, w_2, ..., w_n>$ si dice che ***contiene*** i vertici $w_1, w_2, \ldots, w_n$ e gli archi (w_1,w_2) (w_2,w_3) $\ldots(w_{n-1},w_n)$

PERCORSO DI UN GRAFO: Semplice

Sia G = (V, E) un grafo

Un percorso nel grafo è detto semplice se i nodi non si ripetono



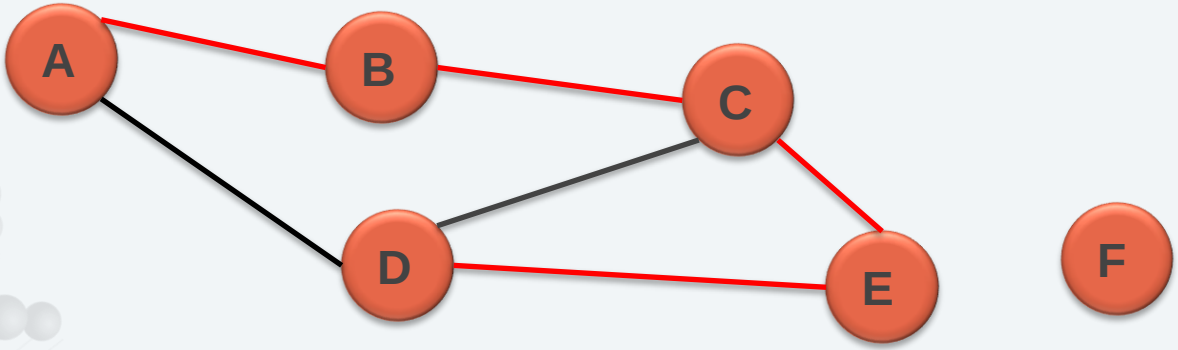
{A→B→C→E→D} è semplice
{A→B→C→E→D→A} è semplice

{A→B→C→E→D→C} non è semplice, perché C è ripetuto

PERCORSO DI UN GRAFO: Lunghezza

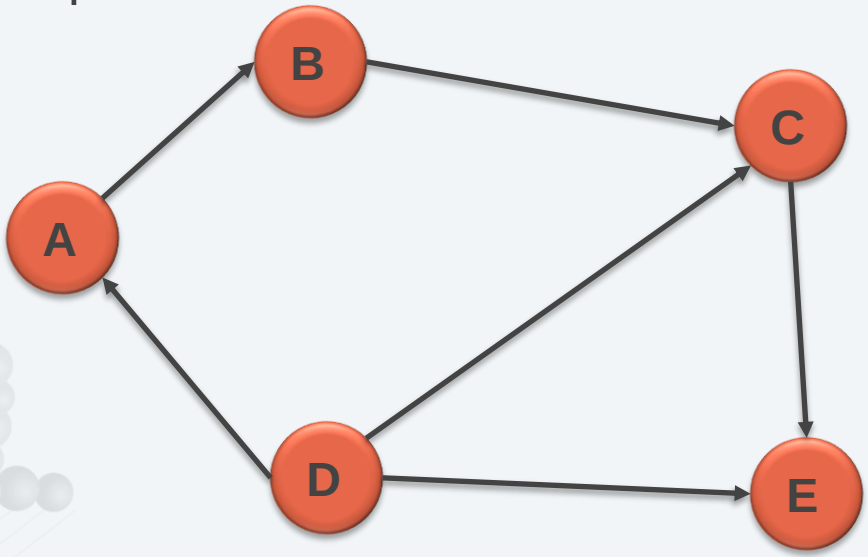
Sia <w1, w2, . . . , wn> un percorso.
La lunghezza del percorso è il numero di archi che connettono i nodi nell'ordine della sequenza (uno in meno del numero di nodi nella sequenza)

La lunghezza del percorso <A, B, C, E,D> è 4



PERCORSO DI UN GRAFO ORIENTATO

In un grafo orientato **ogni arco (wi, wi+1) nel percorso è ordinato** (ha archi con orientamento a seguire).
In altre parole gli archi del percorso sono sempre orientati lungo il percorso

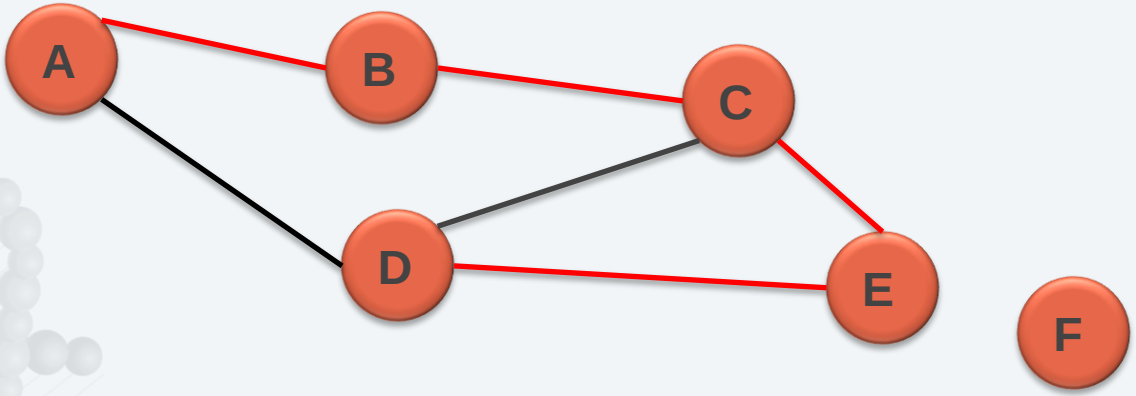


{A→B→C→E} è un percorso nel grafo orientato

{A→B→C→D} non è un percorso, poiché (C, D) non è un arco con orientamento a seguire

PERCORSO: Raggiungibilità grafo non orientato

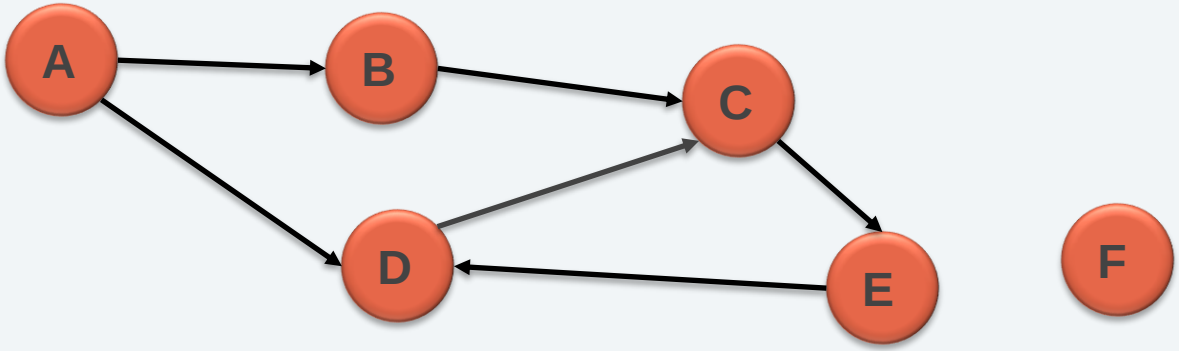
In un grafo non orientato se esiste un percorso p tra i nodi v e w, si dice che w è raggiungibile da v tramite p



A è raggiungibile da D e viceversa
{A→B→C→E→D} e {D→E→C→B→A}

PERCORSO: Raggiungibilità grafo orientato

In un grafo orientato esiste un percorso p tra i nodi v e w, si dice che w è raggiungibile da v tramite p



C è raggiungibile da E e viceversa
{E→D→C} e {C→E}

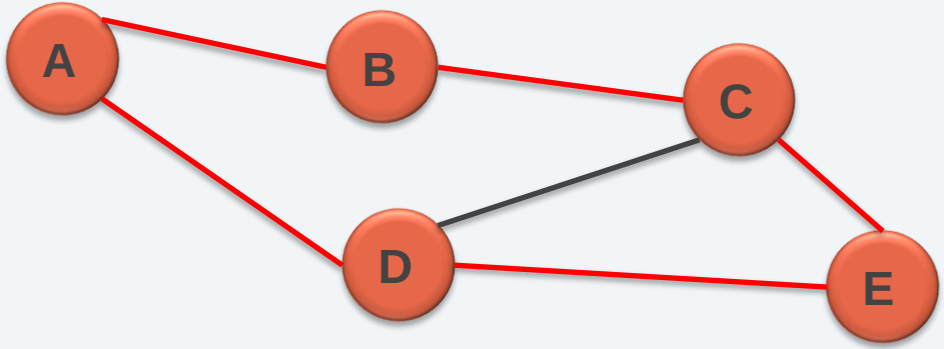
D è raggiungibile da A ma NON è vero il contrario
{A→B→C→E→D}

CIRCUITO: Definizione

Un circuito è un percorso che:

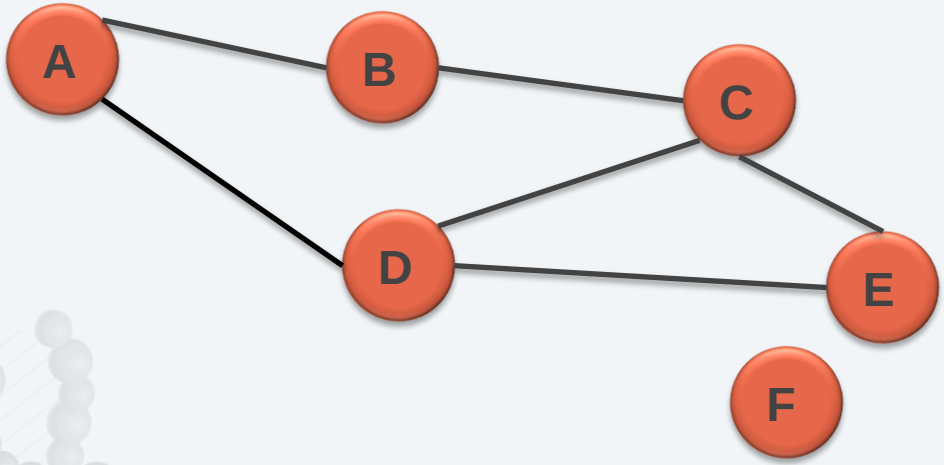
- inizia e termina nello stesso nodo **(in questo caso si dice che il grafo ha un ciclo)**
- se è un circuito semplice, non attraversa alcuno arco o nodo più di una volta, tranne il nodo iniziale/finale (**in questo caso si parla di ciclo semplice**)

In altre parole, un circuito chiude su sé stesso. Per esempio, un circuito può essere rappresentato come $v1$, $v2$,…,vk,v,1

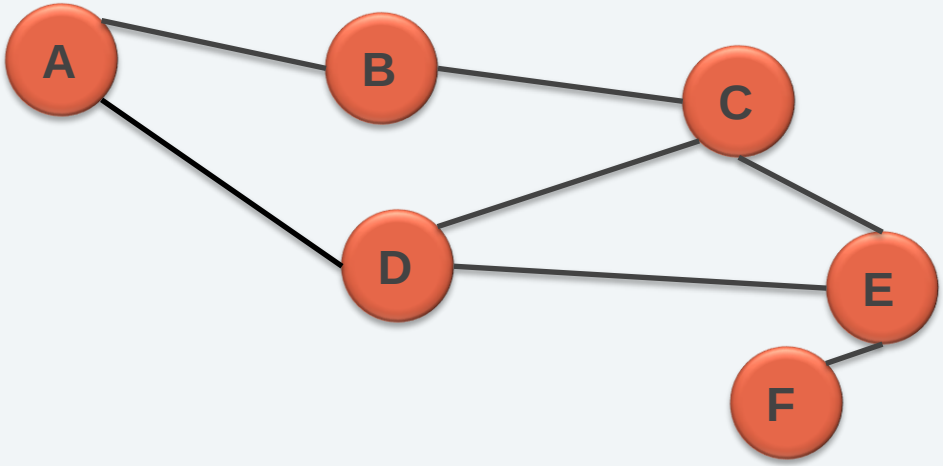


GRAFO CONNESSO

Se G è un grafo diciamo che G è connesso se esiste un percorso da ogni nodo ad ogni altro nodo



Grafo non orientato NON connesso



Grafo non orientato connesso

RAPPRESENTAZIONE DI GRAFI

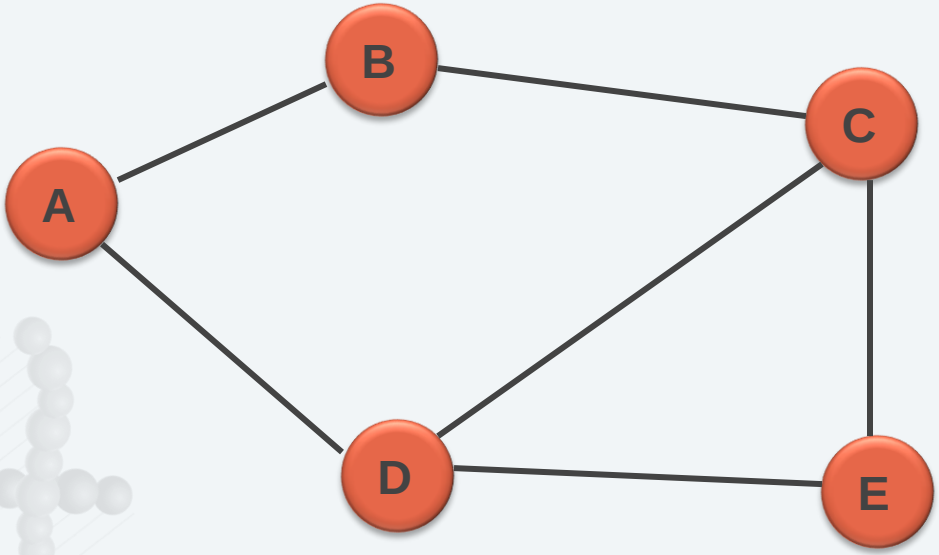
Ci sono due modi standard per **rappresentazione un grafo** in informatica:

Rappresentazione con matrice di adiacenza

Rappresentazione con liste di adiacenza

MATRICE DI ADIACENZA: GRAFO NON ORIENTATO

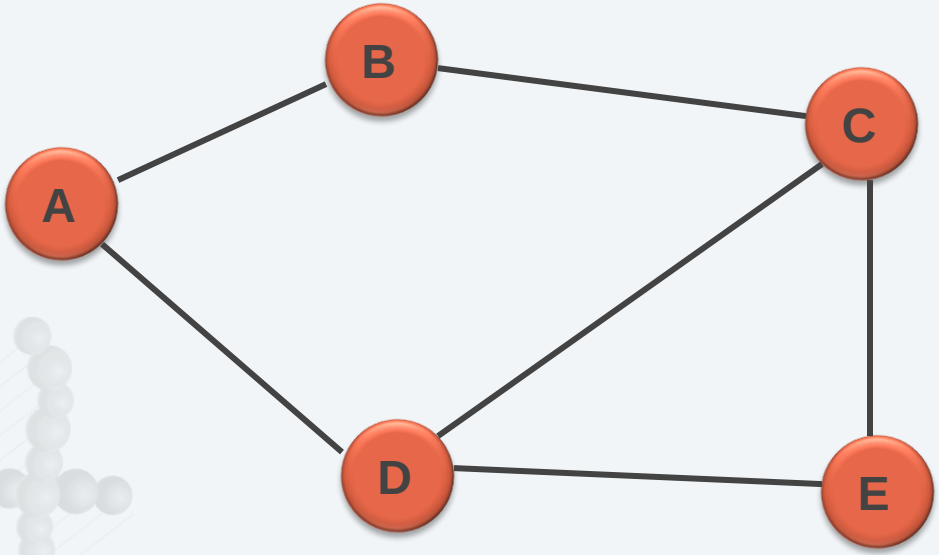
$$M(u,v)\begin{cases}1 \text{ se } (u,v)\in E\\ 0 \text{ altrimenti}\end{cases}$$

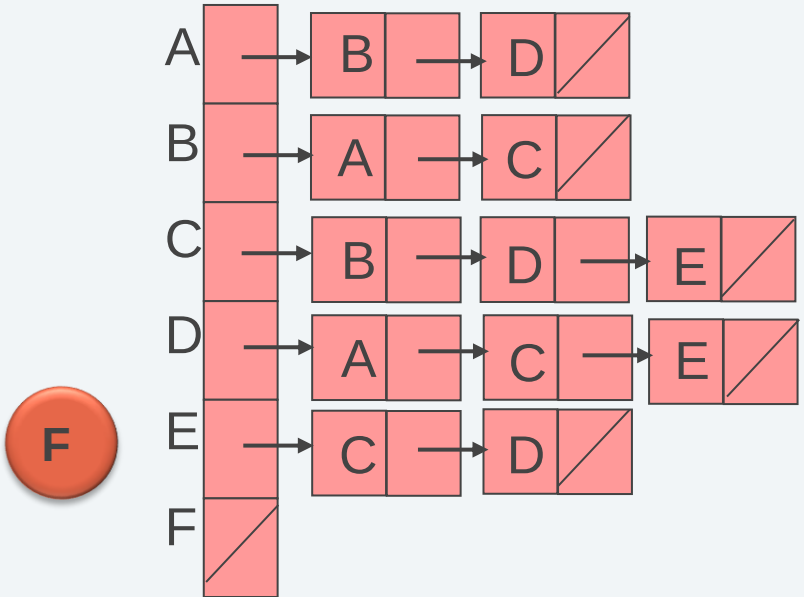


	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0

LISTA DI ADIACENZA: GRAFO NON ORIENTATO

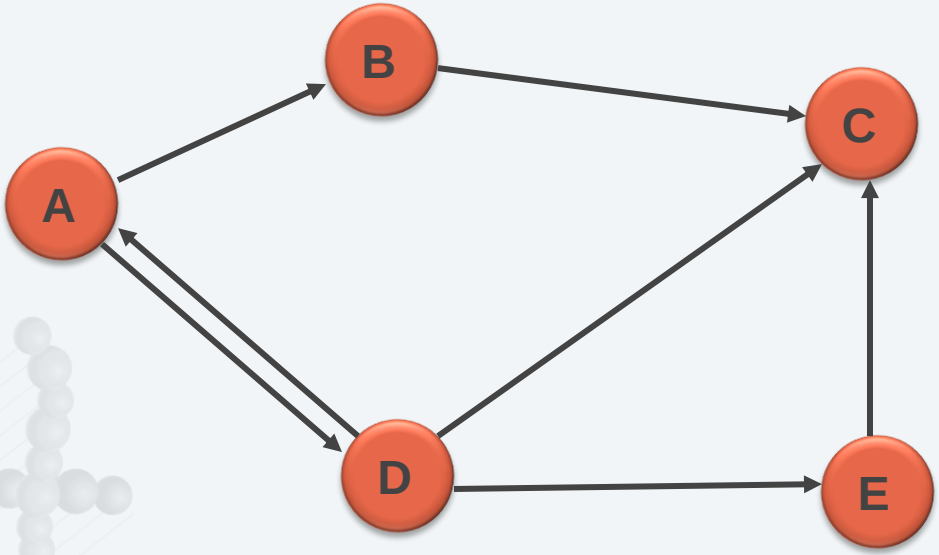
L(v) = lista di w, tale che (v, w)∈E, per v∈V





MATRICE DI ADIACENZA: GRAFO ORIENTATO

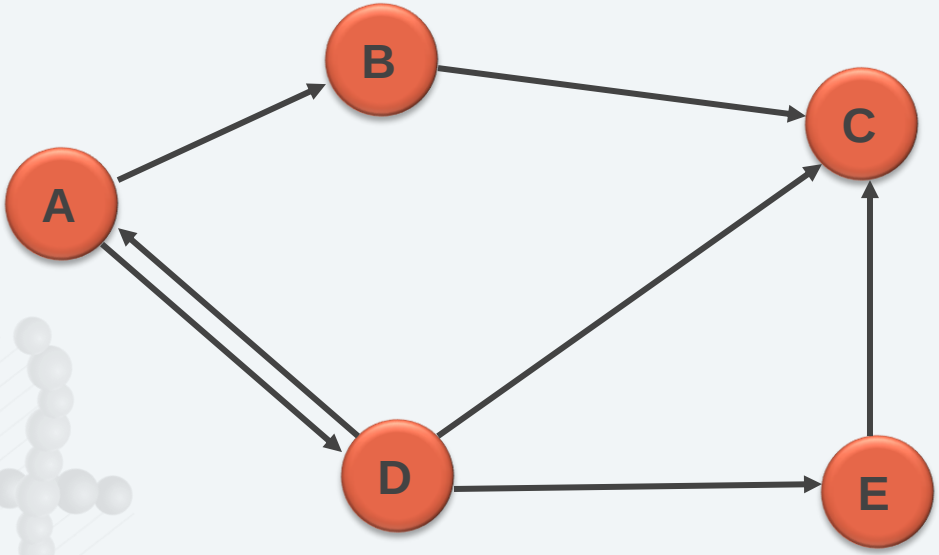
$$M(u,v)\begin{cases}1 \text{ se } (u,v)\in E\\ 0 \text{ altrimenti}\end{cases}$$

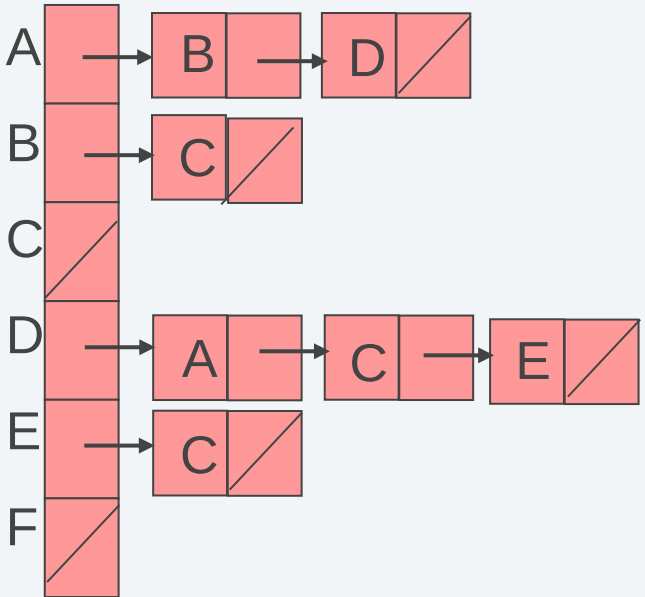


	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	0	0	0
D	1	0	1	0	1	0
E	0	0	1	0	0	0
F	0	0	0	0	0	0

LISTA DI ADIACENZA: GRAFO ORIENTATO

L(v) = lista di w, tale che (v, w)∈E, per v∈V





UN ESEMPIO DELL'USO DEI GRAFI: LA RICERCA DI UN CAMMINO MINIMO - *ALGORITMO DI DIJKSTRA*

ALGORITMO DI DIJKSTRA

L'**algoritmo di Dijkstra** è un algoritmo utilizzato per cercare un **cammino minimo in un grafo** non orientato (o orientato), ciclico e con pesi non negativi sugli archi

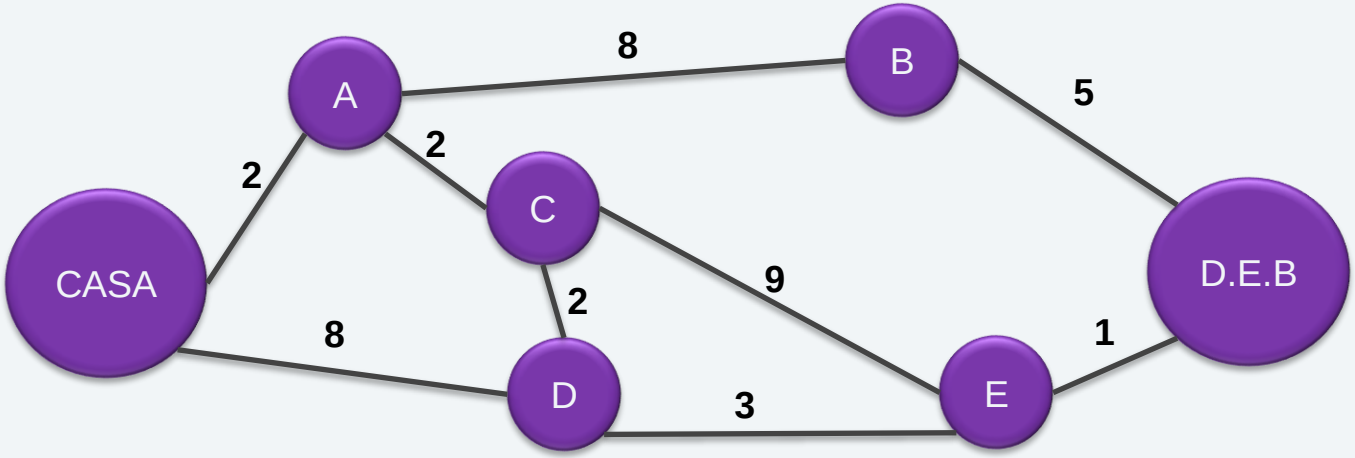
Fu inventato nel 1956 dall'informatico olandese Edsger Dijkstra che lo pubblicò successivamente nel 1959.

Tale algoritmo trova applicazione in molteplici contesti quale l'ottimizzazione nella realizzazione di reti (idriche, telecomunicazioni, stradali, circuitali, ecc.) o l'organizzazione e la valutazione di percorsi in tempo reale nel campo della robotica

ALGORITMO DI DIJKSTRA: Esempio

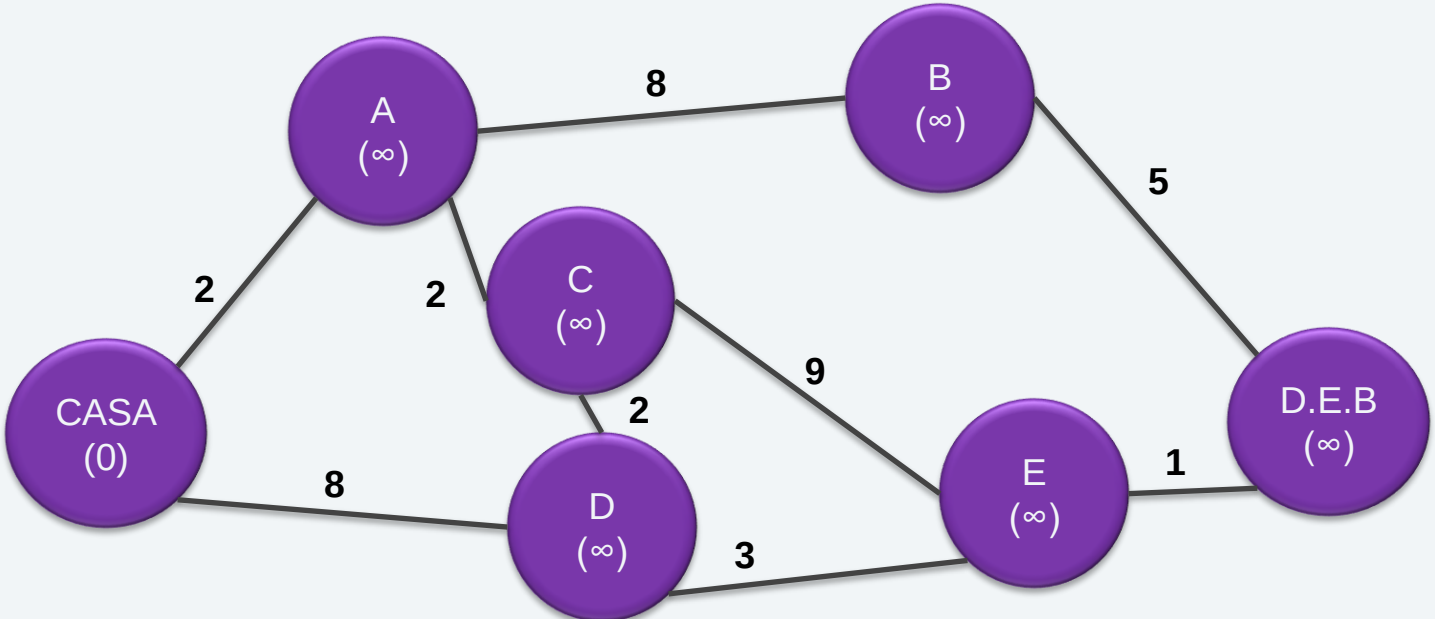
Si consideri il problema in cui si vuole calcolare il percorso minimo tra casa e l'Università.

Si schematizzino tutti i possibili percorsi e il relativo tempo di percorrenza (supponendo di voler calcolare il percorso più breve in fatto di tempo di percorrenza).
I nodi A, B, C, D, E indicano le cittadine per cui è possibile passare, gli archi le strade.



ALGORITMO DI DIJKSTRA: Esempio

Inizializzazione - Ogni nodo ha, all'inizio **potenziale +∞**
Il nodo di partenza (in questo caso “casa”) ha **potenziale 0** (ovvero dista zero da sé stesso);

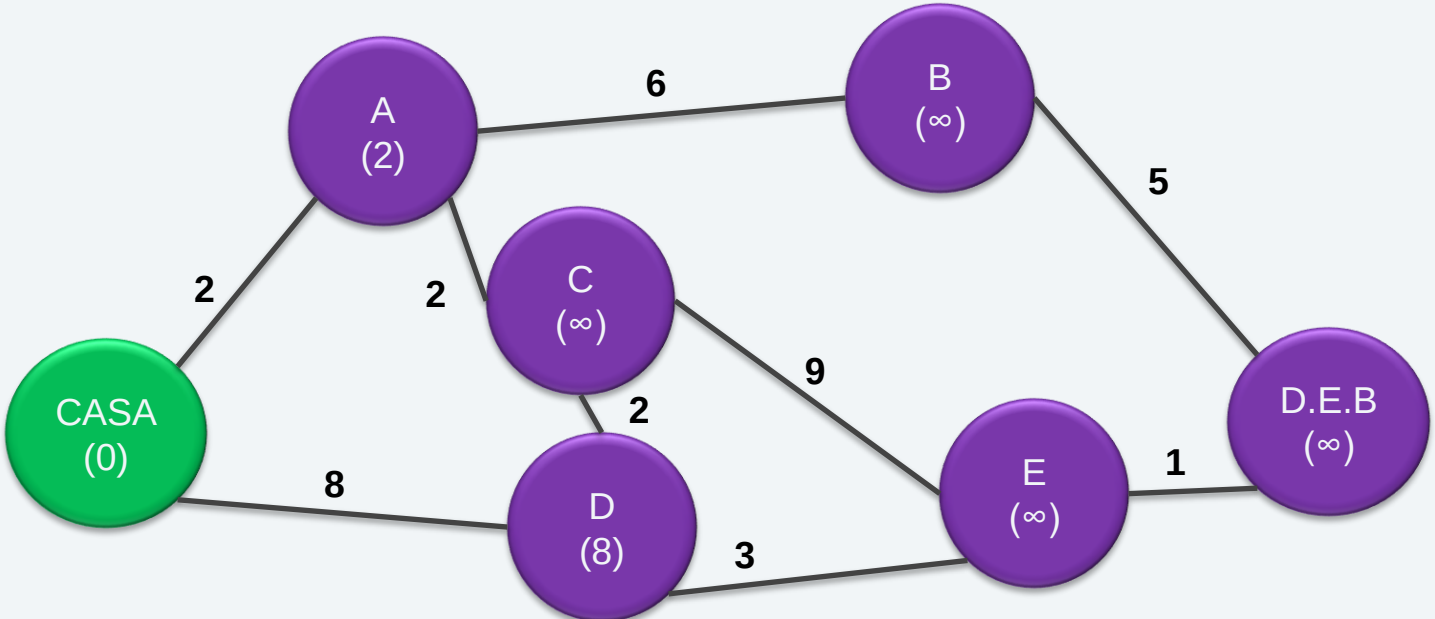


ALGORITMO DI DIJKSTRA: Esempio

Passo Iterativo - Si sceglie il nodo con potenziale minore e lo si rende definitivo (colorando il potenziale) e si aggiornano i nodi adiacenti (il potenziale di un nodo adiacente è ottenuto sommando il potenziale del nodo di partenza più il costo del collegamento)
Non si aggiornano i potenziali dei nodi resi definitivi (i potenziali definitivi indicano la distanza di quel nodo da quello di partenza)

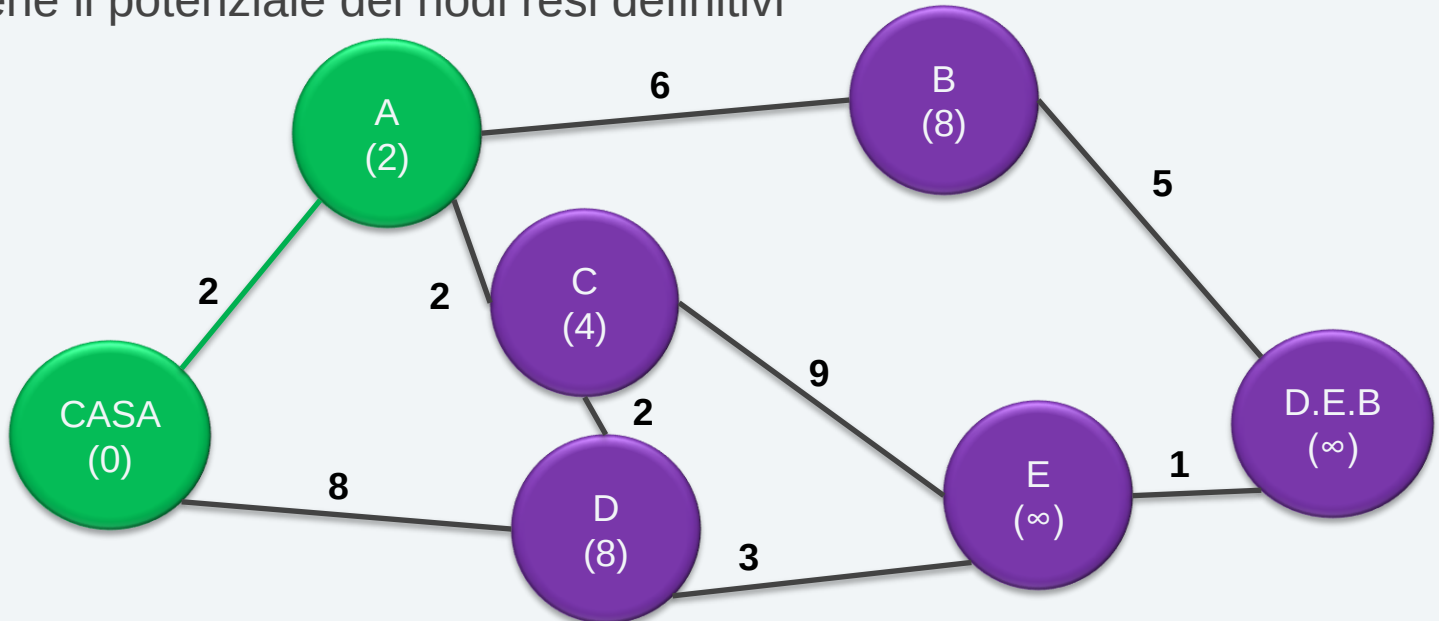
ALGORITMO DI DIJKSTRA: Esempio

Passo 1 – Si sceglie il nodo con potenziale minore (CASA) che diventa a potenziale definitivo (verde) e si aggiorna il potenziale degli adiacenti sommando il potenziale di CASA (0) con il percorso A=0+2 e D=0+8



ALGORITMO DI DIJKSTRA: Esempio

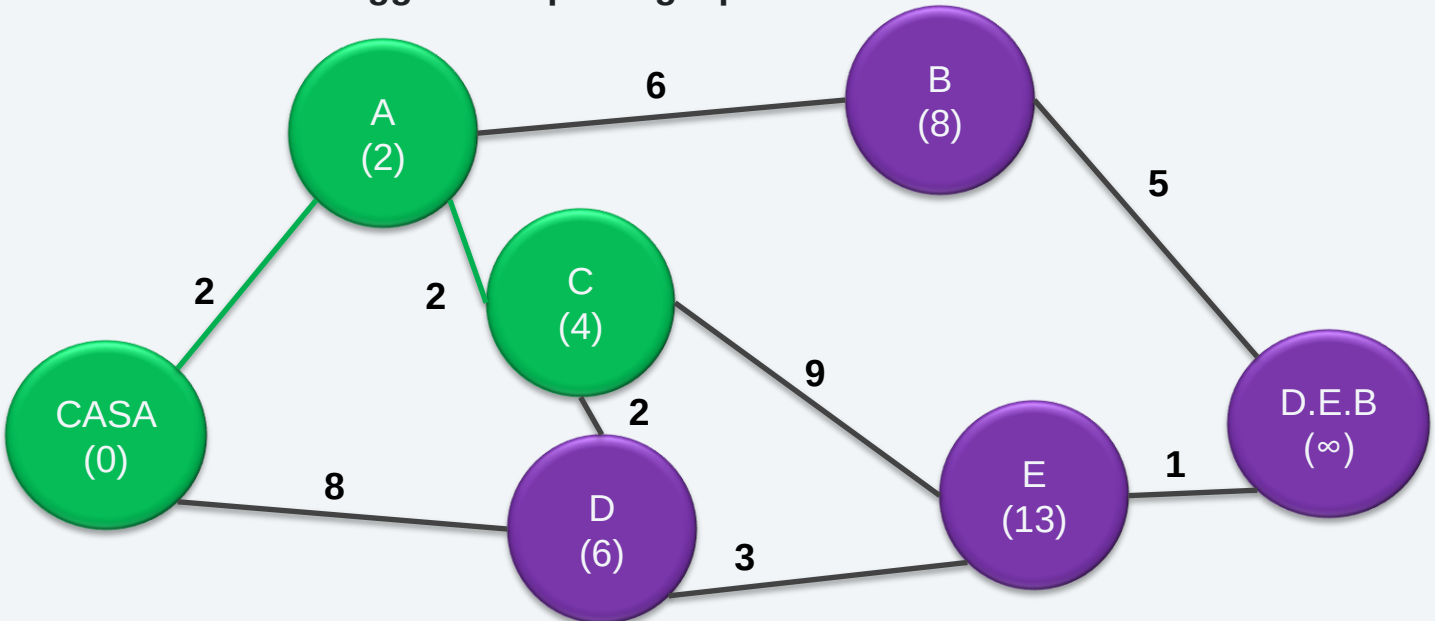
Passo 2 - Bisogna ora considerare il nodo non definitivo (ovvero quelli viola) con potenziale minore (il nodo A). Lo si rende definitivo e si aggiornano i potenziali dei nodi adiacenti (cioè B e C). Si indica con una freccia da dove proviene il potenziale dei nodi resi definitivi



ALGORITMO DI DIJKSTRA: Esempio

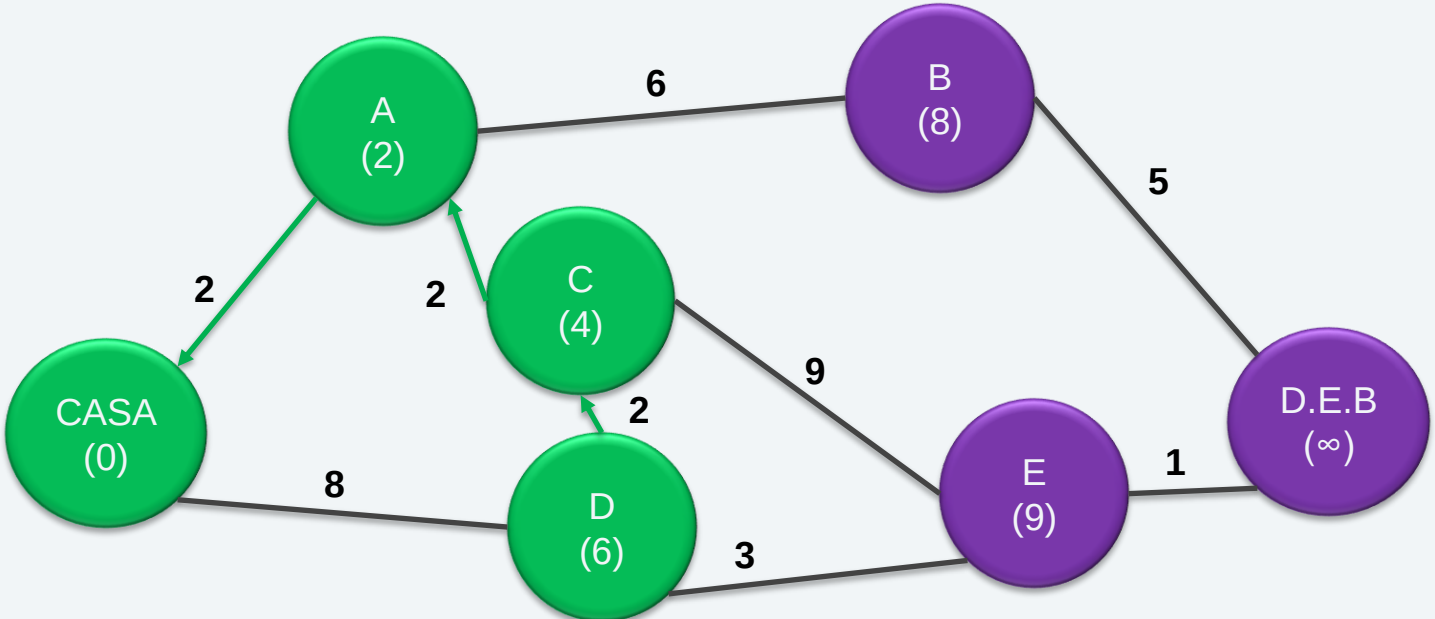
Passo 3 - Il nodo con potenziale minore ora è C. lo si rende definitivo e si aggiornano quelli adiacenti (D ed E).

Va notato come il nodo D ha potenziale 6 in quanto 6 è minore di 8 e quindi lo si aggiorna. Se si fosse ottenuto un valore maggiore di quello già presente si sarebbe dovuto lasciare invariato



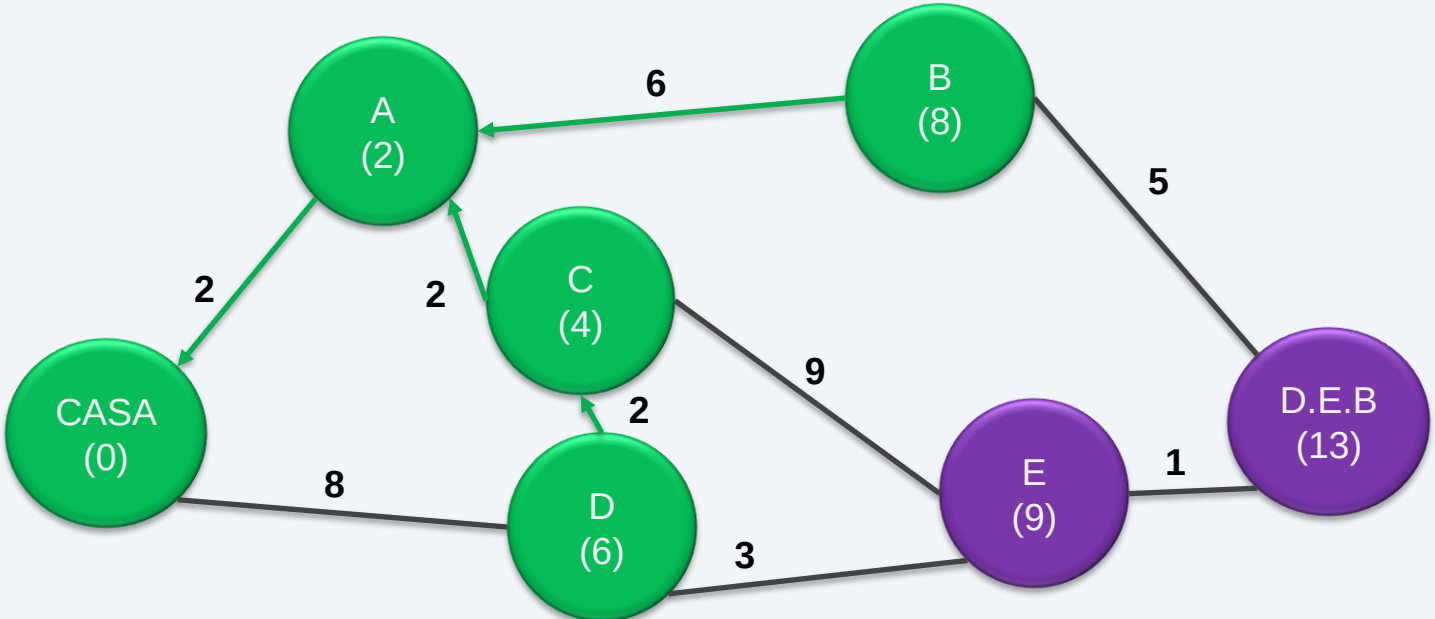
ALGORITMO DI DIJKSTRA: Esempio

Passo 4 – Si sceglie il nodo minimo che è D. Mentre il nodo E si aggiorna al valore 9



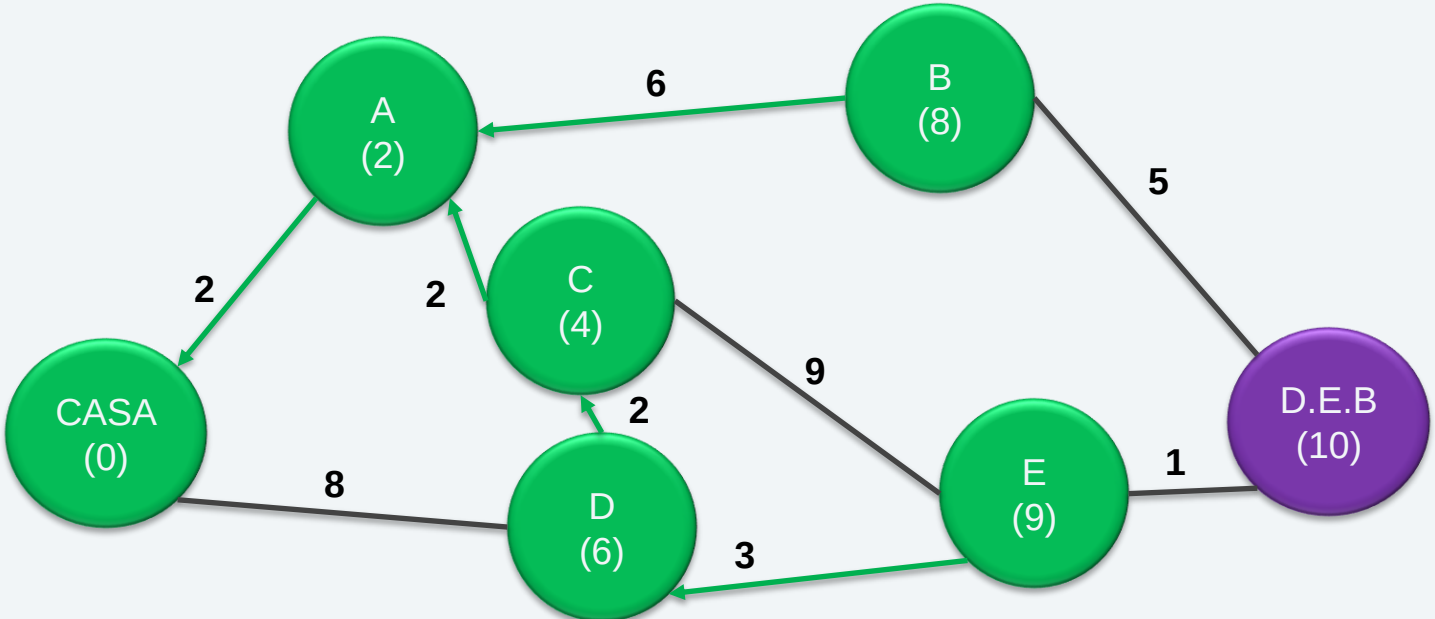
ALGORITMO DI DIJKSTRA: Esempio

Passo 5 – Si scegli il nodo minimo che è B. Mentre il nodo DEB si aggiorna al valore 13



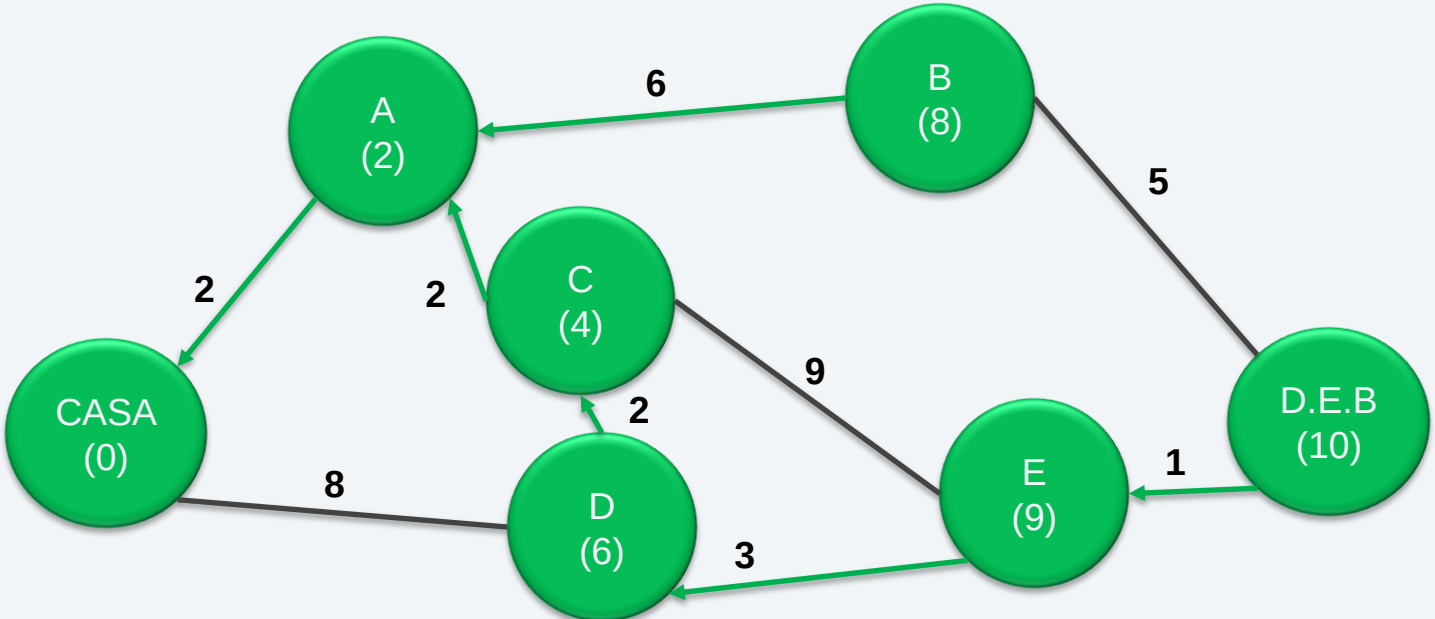
ALGORITMO DI DIJKSTRA: Esempio

Passo 6 – Si scegli il nodo minimo che è E. Mentre il nodo DEB si aggiorna al valore 10



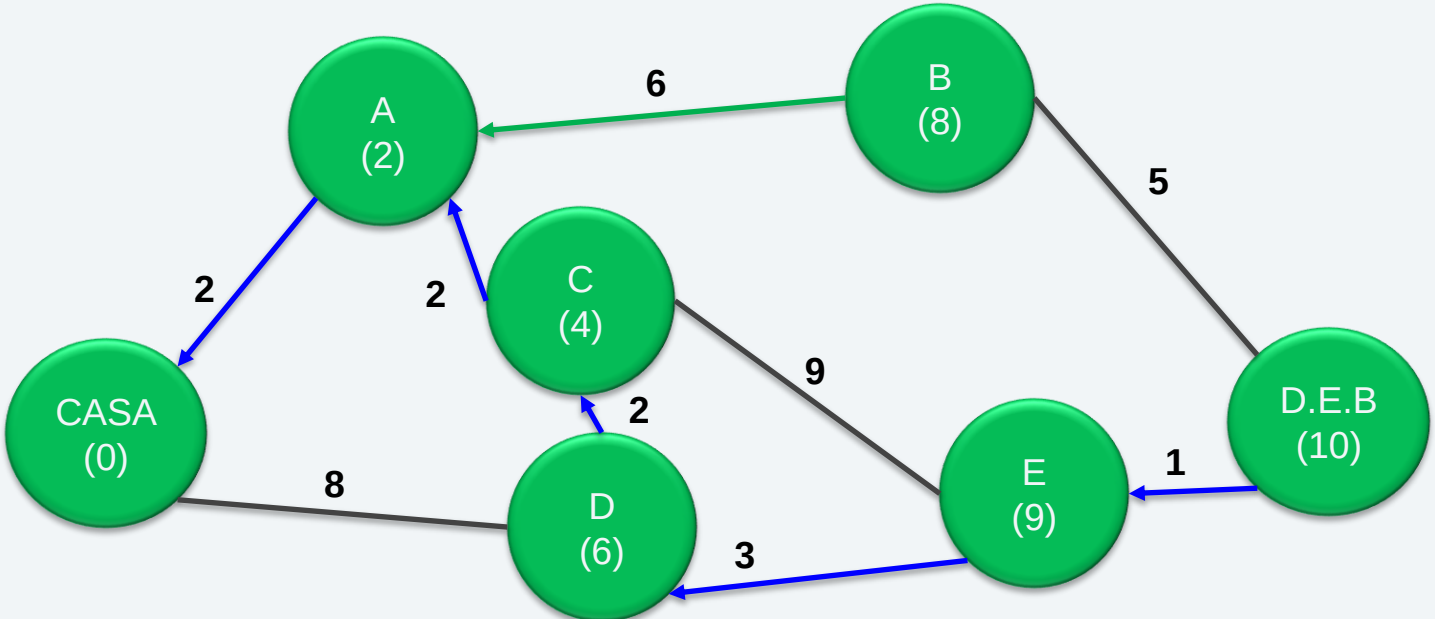
ALGORITMO DI DIJKSTRA: Esempio

Passo 7 – Si scegli il nodo minimo che è DEB e si termina perché arrivati a destinazione



ALGORITMO DI DIJKSTRA: Esempio

Terminazione – Seguendo a partire dal nodo di destinazione (DEB) all'indietro le frecce si ottiene il percorso minimo che (come indicato dal potenziale) ha peso “10”.



CIRCUITO EULERIANO

CIRCUITO HAMILTONIANO

I GRAFI NEL MONDO SCIENTIFICO

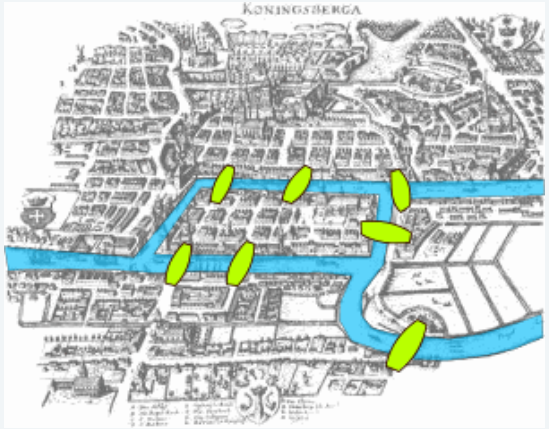
I grafi sono spesso associati alla matematica ricreativa ed ai cosiddetti ”rompicapo”, problemi di natura logico-matematica la cui formulazione ricorre a situazioni reali e può essere compresa anche senza una specifica preparazione teorica.

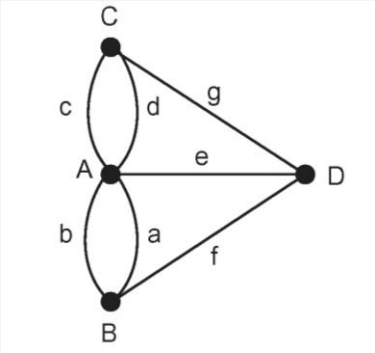
Nel 1736 furono studiati come ramo della matematica: atti dell’Accademia delle Scienze di Pietroburgo di una memoria di Leonardo Eulero, nella quale egli affrontava il Problema dei ponti di Konigsberg

Con l’avvento dell’Informatica sono stati adottati per l’analisi di problemi in diversi campi applicativi: teoria degli automi, geometria dei poliedri, algebre di Lie, schematizzazione di programmi, circuiti, reti di computer, mappe di siti, sistemi fluviali, reti stradali, trasporti, nella linguistica strutturale, nella storia (alberi genealogici, filologia dei testi)

CIRCUITO EULERIANO

La città di Konigsberg (oggi Kaliningrad) sorge alla foce del fiume Pregel, che in quel punto forma due isole. Nel settecento le varie parti della città erano collegate da un sistema di sette ponti, ed era costume delle famiglie borghesi del tempo recarsi a passeggiare, nelle domeniche di bel tempo, lungo le rive del fiume e le sue isole





Il problema era il seguente: **è possibile fare una passeggiata che partendo ed arrivando nello stesso luogo porti ad attraversare una ed una sola volta tutti e sette i ponti di Konigsberg?**

CIRCUITO EULERIANO

Un percorso completo si dice euleriano se si visita ogni arco una sola volta (formalmente: un circuito si dice euleriano se l'insieme dei nodi che lo compongono è tutto l'insieme E)

Teorema di Eulero. *Un qualsiasi grafo è percorribile se e solo se ha tutti i nodi di grado pari, o due di essi sono di grado dispari. Per percorrere un grafo "possibile" con due nodi di grado dispari, è necessario partire da uno di essi, e si termina sull'altro nodo di grado dispari*

Pertanto è impossibile percorrere Königsberg come richiesto dalla tesi perché tutti i nodi sono di grado dispari

CIRCUITO EULERIANO: Hierholzer (alg. euristico)

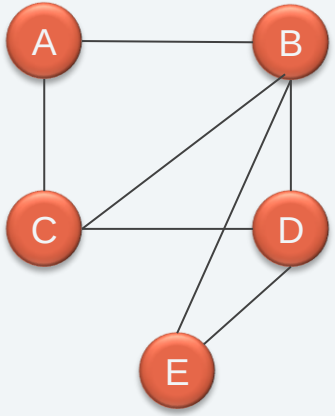
Scegliere un nodo iniziale con grado dispari

Scegliere un nodo con grado pari e unirlo

Procedere lungo gli archi non ancora visitati, marcandoli man mano. Se si arriva in un nodo che già non ha più archi non visitati, termina il percorso parziale

Se ci sono ancora archi non visitati, scegliere un nodo nel percorso attuale con archi non ancora attraversati e costruire un nuovo percorso a partire da lì. Unire il nuovo percorso a quello già trovato

Ripetere fino a quando non si è attraversato tutti gli archi



Grado dei nodi:
A: 2
B: 4
C: 3
D: 3
E: 2

Due nodi hanno grado dispari (C e D), quindi esiste un cammino euleriano

CIRCUITO EULERIANO: Hierholzer (alg. euristico)

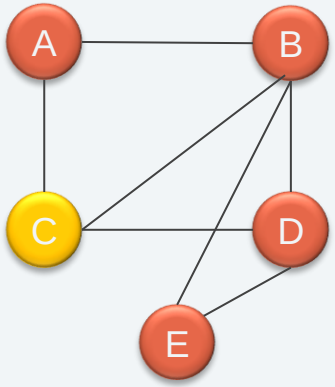
Scegliere un nodo iniziale con grado dispari

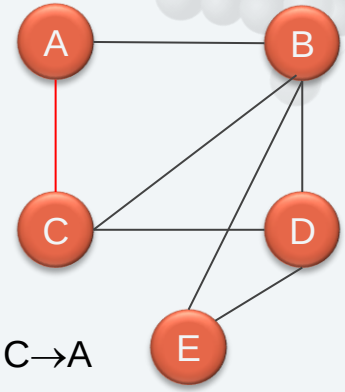
Scegliere un nodo con grado pari e unirlo

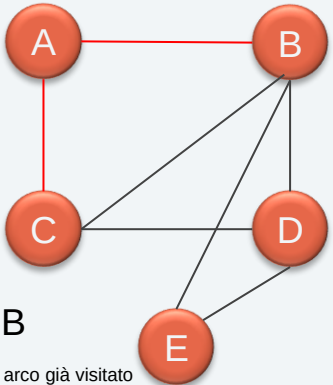
Procedere lungo gli archi non ancora visitati, marcandoli man mano. Se si arriva in un nodo che già non ha più archi non visitati, termina il percorso parziale

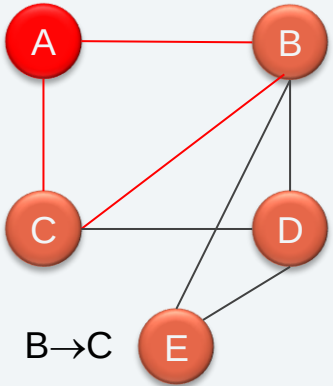
Se ci sono ancora archi non visitati, scegliere un nodo nel percorso attuale con archi non ancora attraversati e costruire un nuovo percorso a partire da lì. Unire il nuovo percorso a quello già trovato

Ripetere fino a quando non si è attraversato tutti gli archi









Non posso scegliere C perché arco già visitato

CIRCUITO EULERIANO: Hierholzer (alg. euristico)

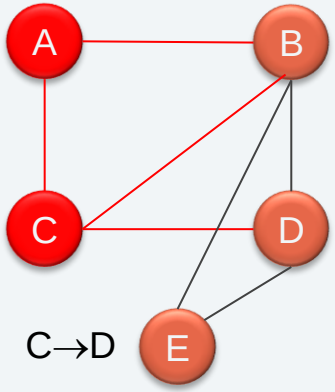
Scegliere un nodo iniziale con grado dispari

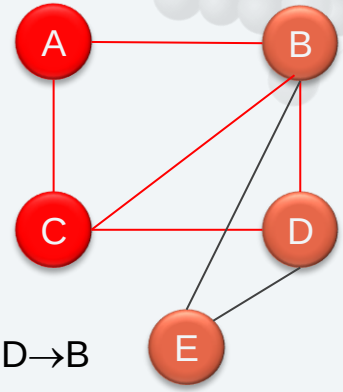
Scegliere un nodo con grado pari e unirlo

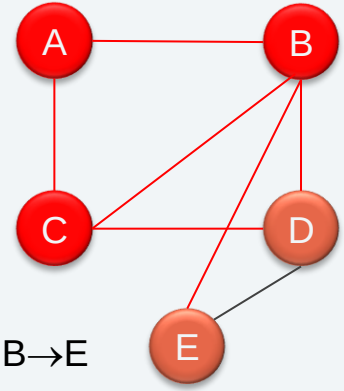
Procedere lungo gli archi non ancora visitati, marcandoli man mano. Se si arriva in un nodo che non ha più archi non visitati, termina il percorso parziale

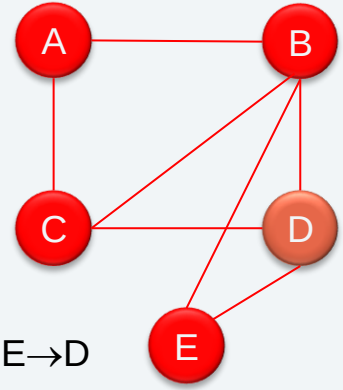
Se ci sono ancora archi non visitati, scegliere un nodo nel percorso attuale con archi non ancora attraversati e costruire un nuovo percorso a partire da lì. Unire il nuovo percorso a quello già trovato

Ripetere fino a quando non si è attraversato tutti gli archi









CIRCUITO EULERIANO: Hierholzer (alg. euristico)

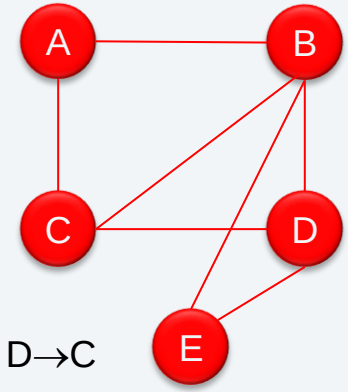
Scegliere un nodo iniziale con grado dispari

Scegliere un nodo con grado pari e unirlo

Procedere lungo gli archi non ancora visitati, marcandoli man mano. Se si arriva in un nodo che non ha più archi non visitati, termina il percorso parziale

Se ci sono ancora archi non visitati, scegliere un nodo nel percorso attuale con archi non ancora attraversati e costruire un nuovo percorso a partire da lì. Unire il nuovo percorso a quello già trovato

Ripetere fino a quando non si è attraversato tutti gli archi



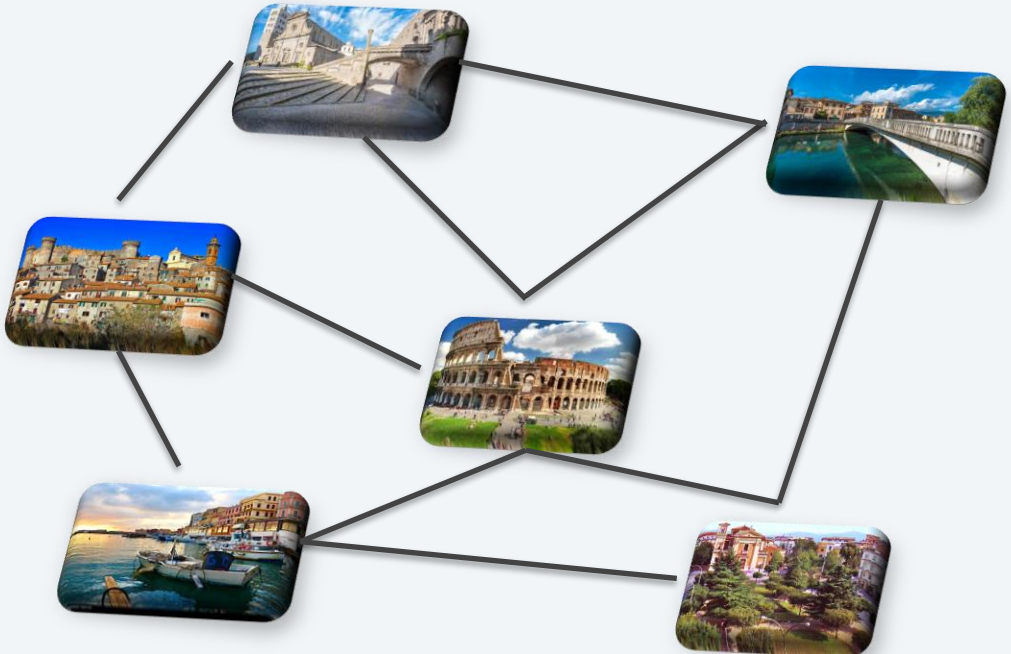
D→C

C→A→B→C→D→B→E→D

La complessità temporale dell'algoritmo di Hierholzer è O(M), dove M è il numero di archi del grafo

CIRCUITO HAMILTONIANO

Dato un insieme di città trovare il tragitto che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta



CIRCUITO HAMILTONIANO

Un **percorso completo si dice hamiltoniano** se si visita ogni nodo una sola volta

Teorema. *Sia G un grafo con n nodi, tale che per ogni coppia di nodi v,w non adiacenti si ha grado(v) + grado(w) ≥ n. Allora G è hamiltoniano.*

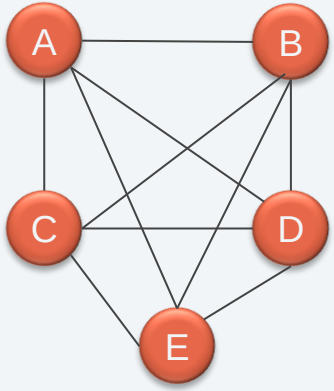
Corollario. Sia G un grafo di n nodi e tale che grado(v)≥ n/2 per ogni nodo v. Allora G è hamiltoniano

CIRCUITO HAMILTONIANO

Si sceglie un nodo iniziale casualmente

Si seleziona il nodo adiacente che:
Non è già stato visitato.
{non è stato visitato ed ha il costo più basso se è un grafo pesato}

Si ripete fino a quando non sono visitati tutti i nodi o non è più possibile proseguire



Grado dei nodi:
A: 4
B: 4
C: 4
D: 4
E: 4

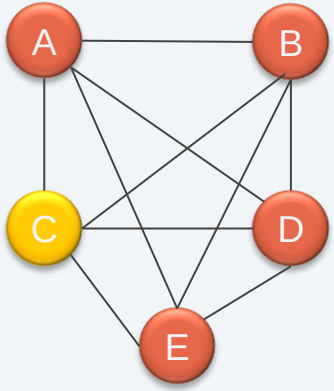
Tutti i nodi hanno grado maggiore o uguale di 5/2 quindi esiste un circuito hamiltoniano

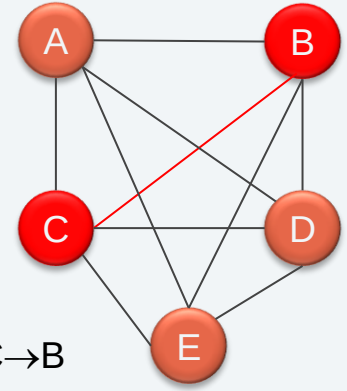
CIRCUITO HAMILTONIANO

Si sceglie un nodo iniziale casualmente

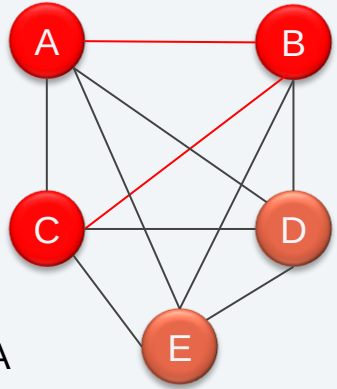
si seleziona il nodo adiacente che:
Non è già stato visitato.
{non è stato visitato ed ha il costo più basso se è un grafo pesato}

Si ripete fino a quando non vengono visitati tutti i nodi o non è più possibile proseguire

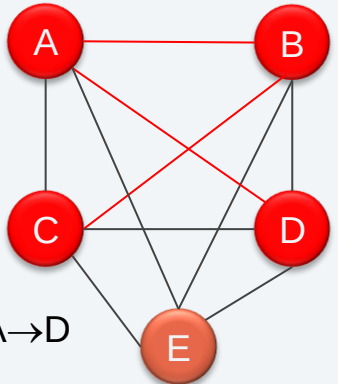




C→B



B→A



A→D

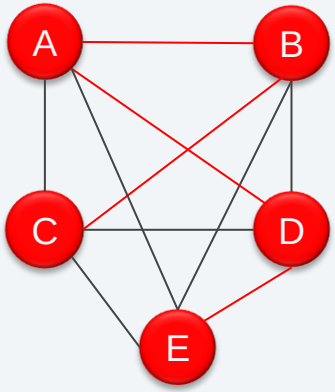
CIRCUITO HAMILTONIANO

Si sceglie un nodo iniziale casualmente

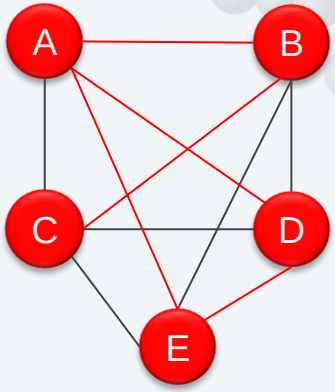
si seleziona il nodo adiacente che:
Non è già stato visitato.

{non è stato visitato ed ha il costo più basso se è un grafo pesato}

Si ripete fino a quando non vengono visitati tutti i nodi o non è più possibile proseguire



D→E



E→A

C→B → A → D → E → A

GRAFO DI DE BRUIJN

GRAFO DI DE BRUIJN: Definizione

Un grafo di de Bruijn, dB(m,n), è un grafo orientato composto a partire da un alfabeto di cardinalità m e un numero intero n
Il grafo possiede m^n nodi che contengono tutte le sequenze di lunghezza n (denominate sequenze di de Bruijn)

Un grafo di de Bruijn è un tipo di di grafo utilizzato nella teoria dei sistemi e in bioinformatica

GRAFO DI DE BRUIJN: Definizione

Sia S={s1,…,sm} l'alfabeto di simboli

Sia S^n=V={(s1,…,s1),(s1,…,s2),…,(sm,…,sm)} il dizionario delle sequenze di de Bruijn di lunghezza n

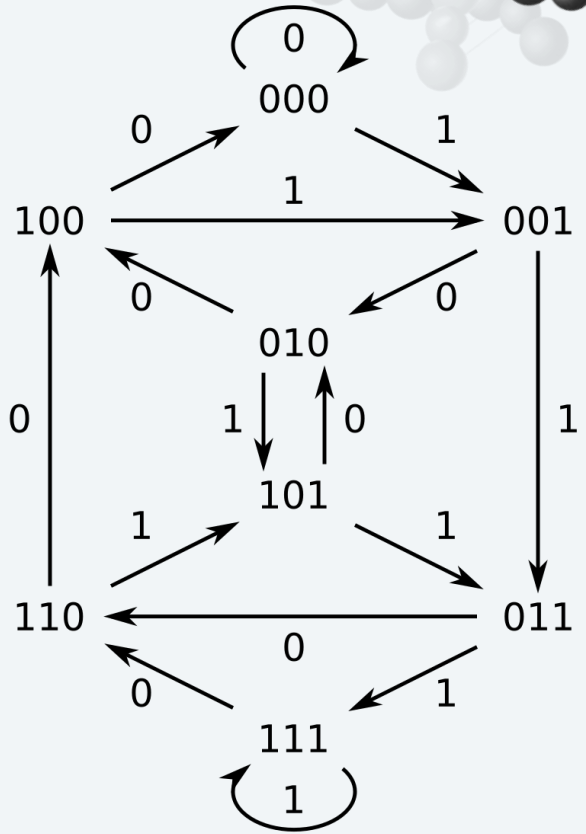
L'insieme degli archi del grafo di de Bruijn è definito da E={((v1,v2,…,vm,si),(v2,…,vm,si),…) dove i=1,…,m}

GRAFO DI DE BRUIJN: Esempio

Sia S={0,1} l'alfabeto di simboli

Sia S^3=V={(000),(001),(010),(011),(100),(101),(110),(111)} il dizionario delle sequenze di de Bruijn di lunghezza n (nonché insieme dei nodi del grafo)

L'insieme degli archi del grafo di de Bruijn è definito da E={(110,100):0,(100,000):0;(000,000):0,…}



GRAFO DI DE BRUIJN – Step 1

Input: Una o più letture di sequenza genomica (reads) ottenute da tecnologie di sequenziamento

Esempio: TAATGCCATGGGA

GRAFO DI DE BRUIJN – Step 2

Si sceglie un valore k, che rappresenta la lunghezza del **k-mer**.
Questo valore deve essere sufficientemente grande da evitare ambiguità ma piccolo abbastanza da catturare sovrapposizioni
(per far questo si può usare la programmazione dinamica o delle strutture dai chiamate Suffix Tree)

Esempio: AATGCCATGGGA
K=3

GRAFO DI DE BRUIJN – Perché KMER

Una read Illumina da 100pb può essere vista come un k-mer di lunghezza 100 ma:

- non è sempre possibile avere tutti i k-mer di lunghezza 100
- ci possono essere errori di sequenziamento
- il coverage aiuta ma non risolve
- possibile violazione dell'assunzione Euleriana

Se si spezzano le reads in k-mers più corti di 100pb

- I k-mers ottenuti tendono a rappresentare meglio quelli di cui è composto un genoma
- Identificazione del k ottimale (generalmente k da 15 a 60)

GRAFO DI DE BRUIJN – Step 3

Ogni sequenza viene suddivisa in sottostringhe consecutive di lunghezza k

AATGCCATGGGA

AAT

ATG

TGC

GCC

CCA

CAT

ATG

TGG

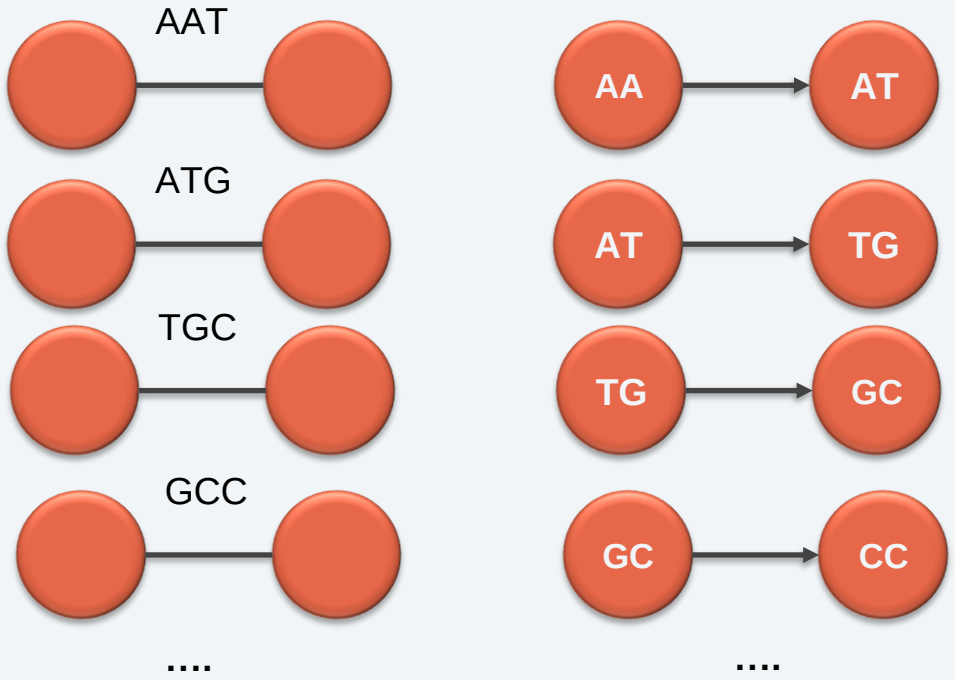
GGG

GGA

GRAFO DI DE BRUIJN – Step 4

Ogni k-mer diventa un arco del grafo

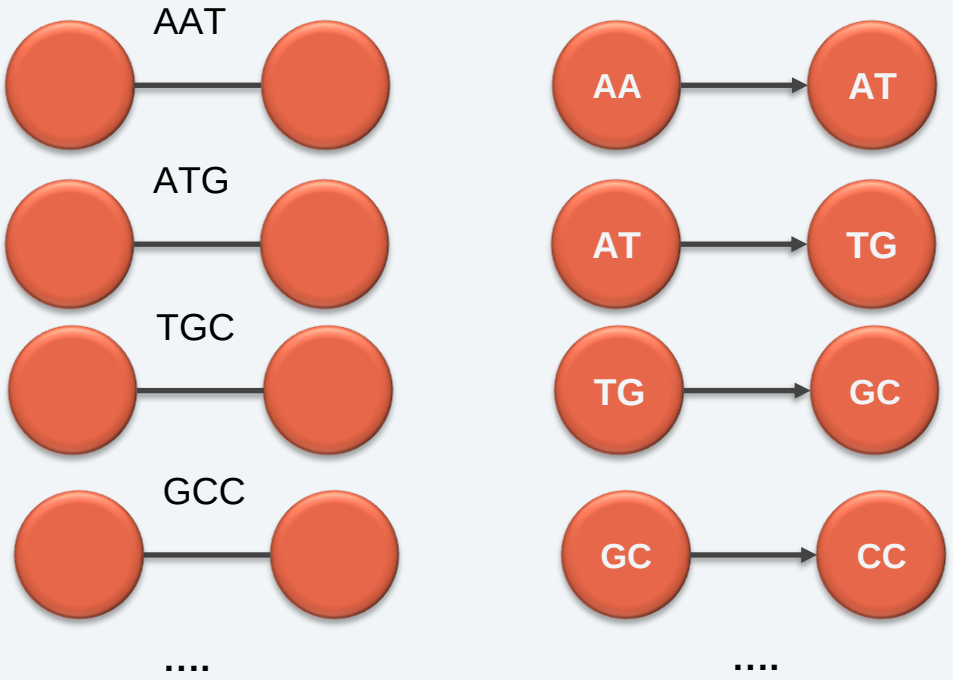
AAT
ATG
TGC
GCC
CCA
CAT
ATG
TGG
GGG
GGA



GRAFO DI DE BRUIJN – Step 5

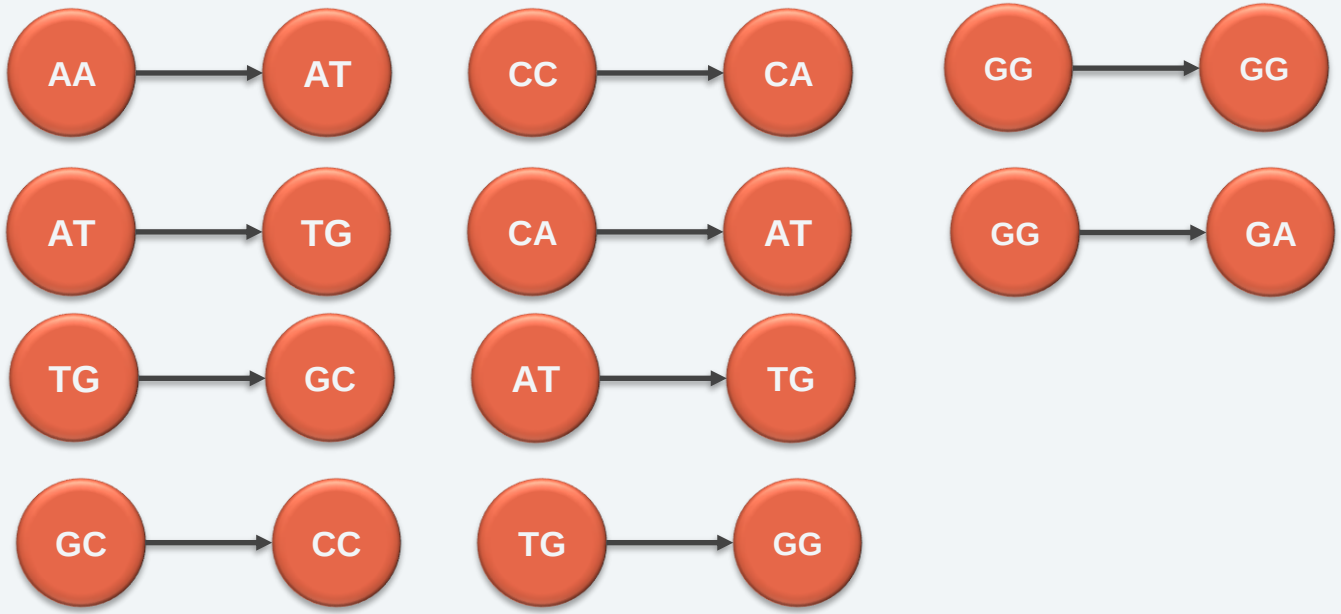
Ogni k-1-mer (sottostringhe di k−1 basi) diventa un nodo del grafo

AAT
ATG
TGC
GCC
CCA
CAT
ATG
TGG
GGG
GGA



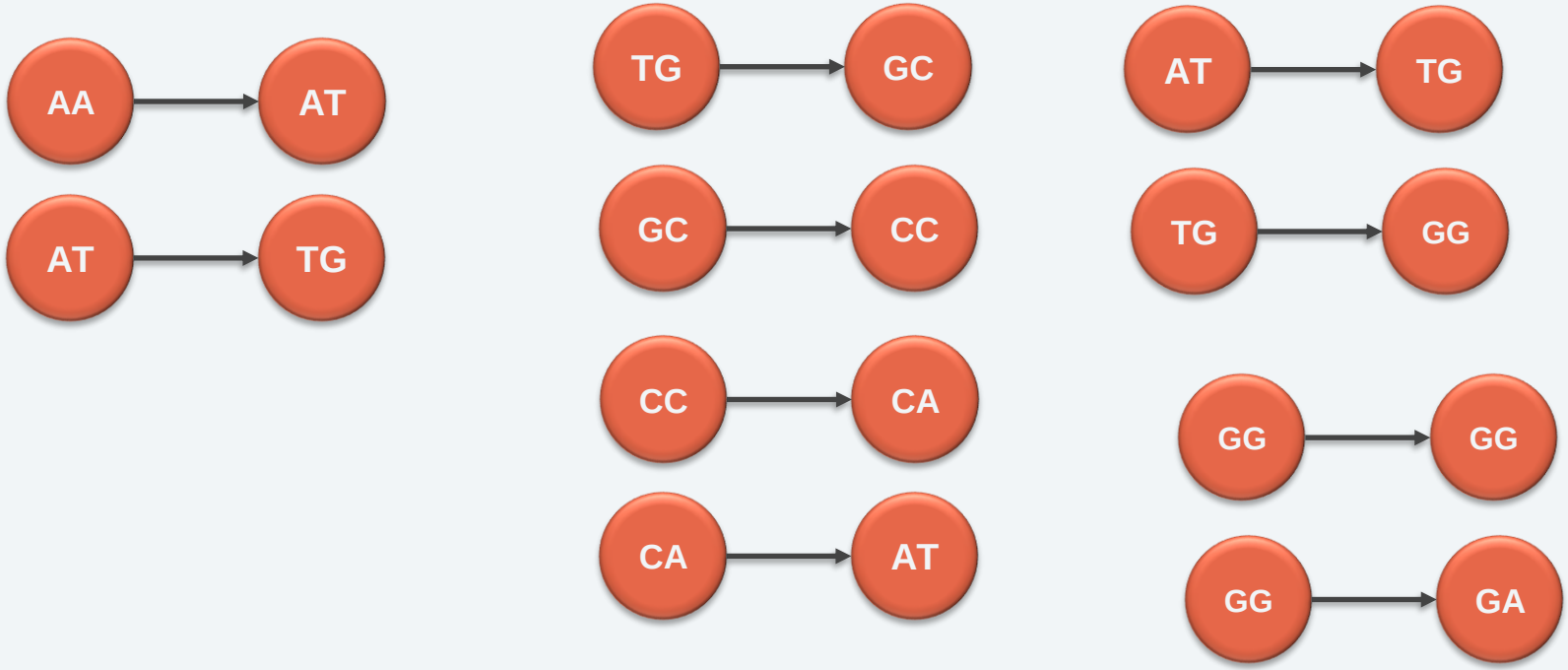
GRAFO DI DE BRUIJN – Step 5

Ogni k-1-mer (sottostringhe di k−1 basi) diventa un nodo del grafo



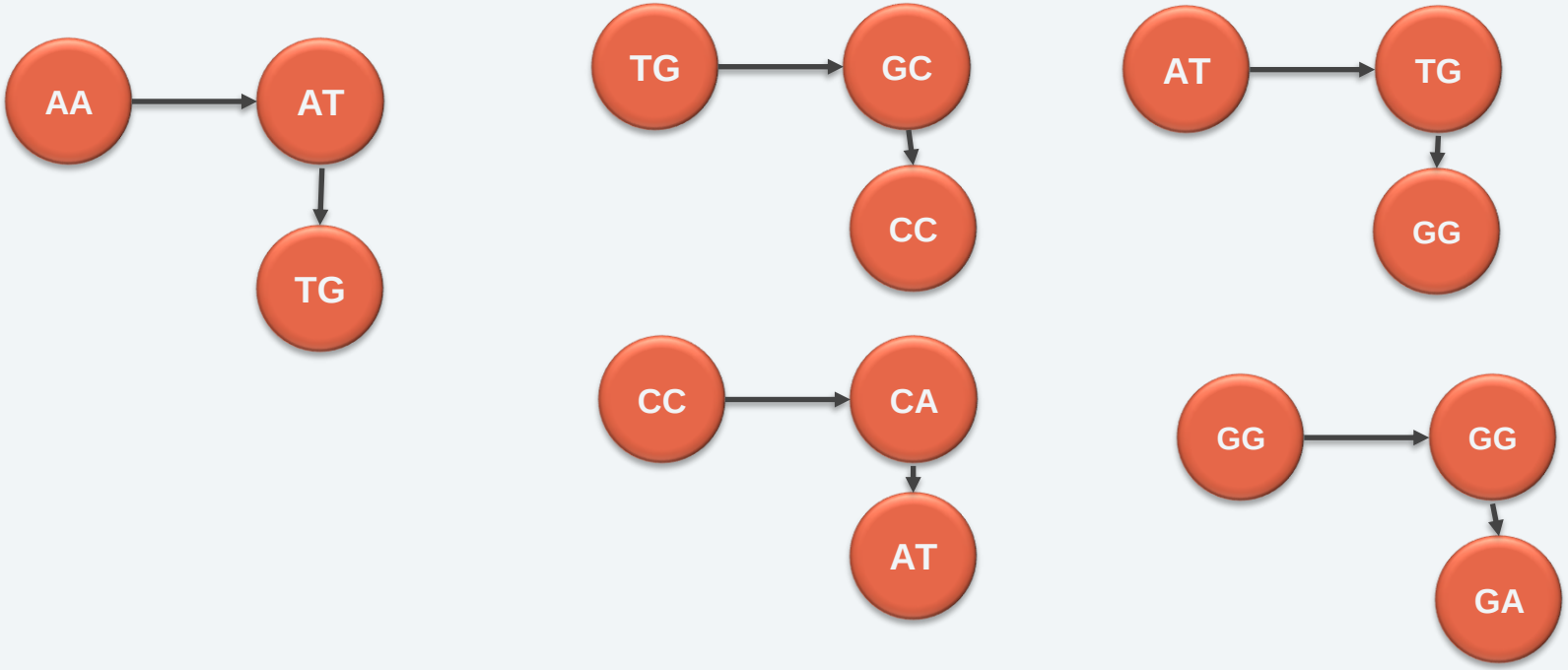
GRAFO DI DE BRUIJN – Step 6

Si avvicinano i k-mer più simili



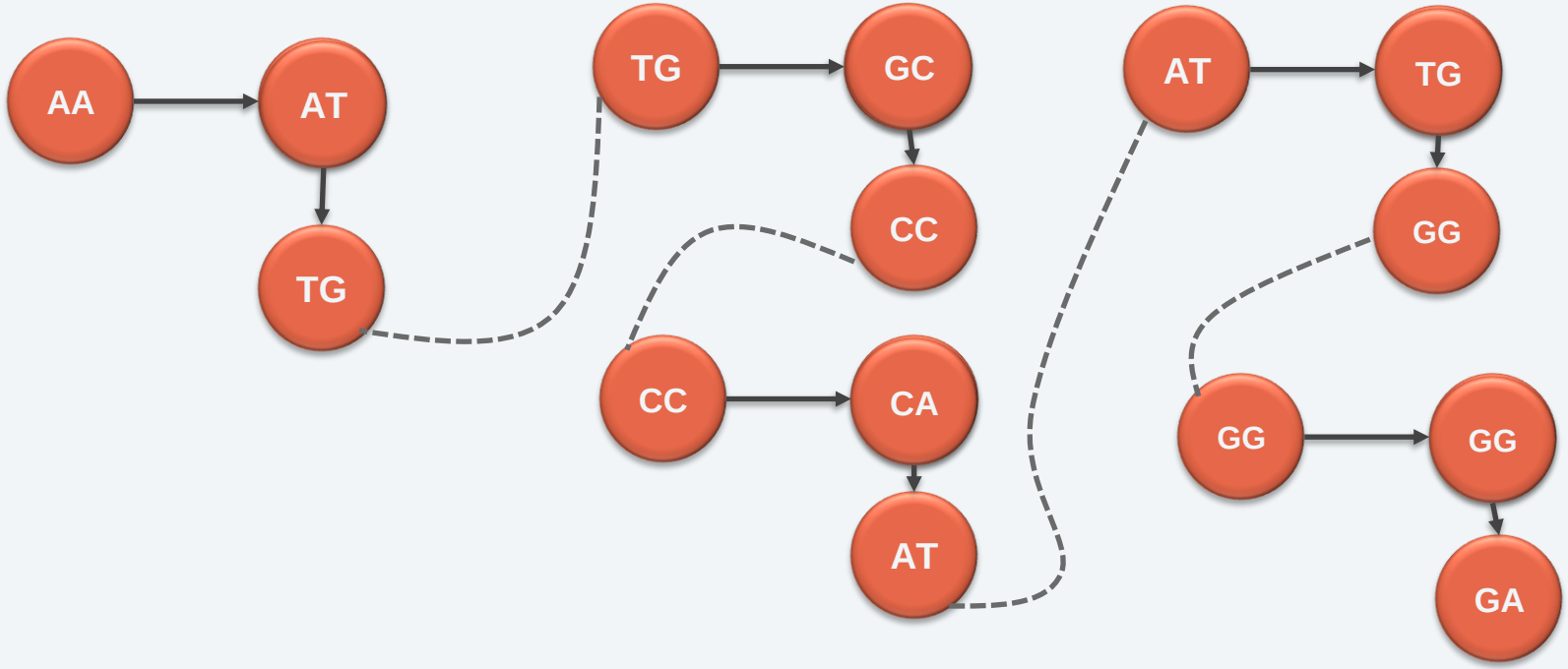
GRAFO DI DE BRUIJN – Step 7

Si uniscono (*glue operation*) **i nodi con la stessa etichetta**



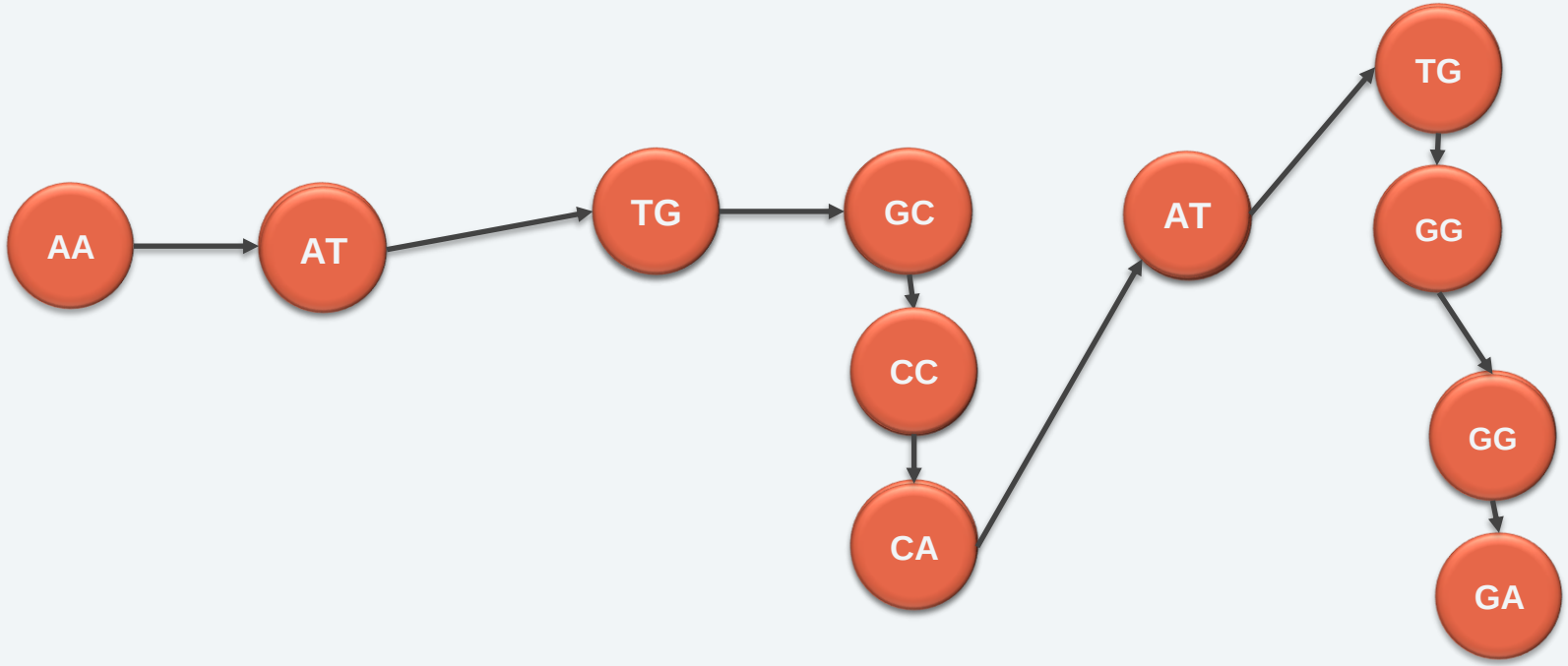
GRAFO DI DE BRUIJN – Step 7

Si uniscono (*glue operation*) **i nodi con la stessa etichetta**



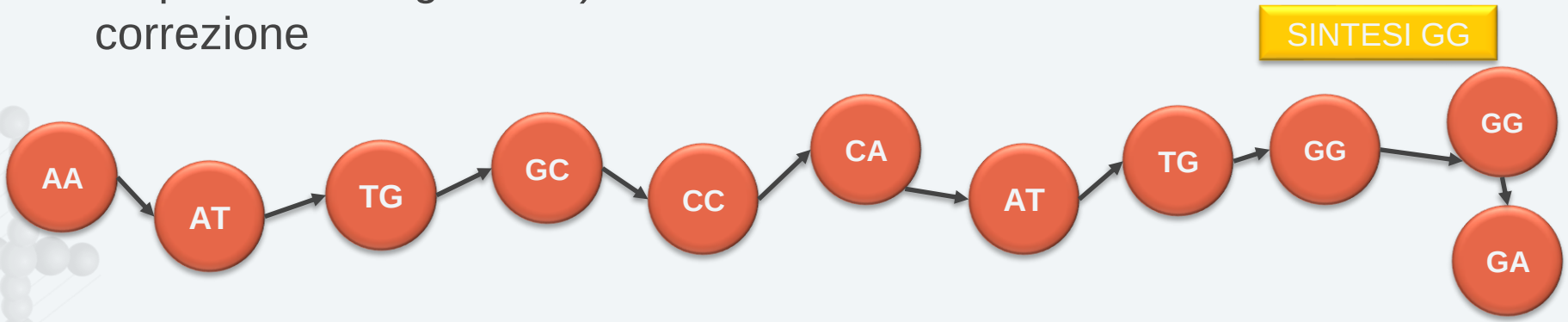
GRAFO DI DE BRUIJN – Step 8

Il grafo risultante è un grafo orientato, in cui i nodi rappresentano i k−1-mer e gli archi rappresentano i k-mer

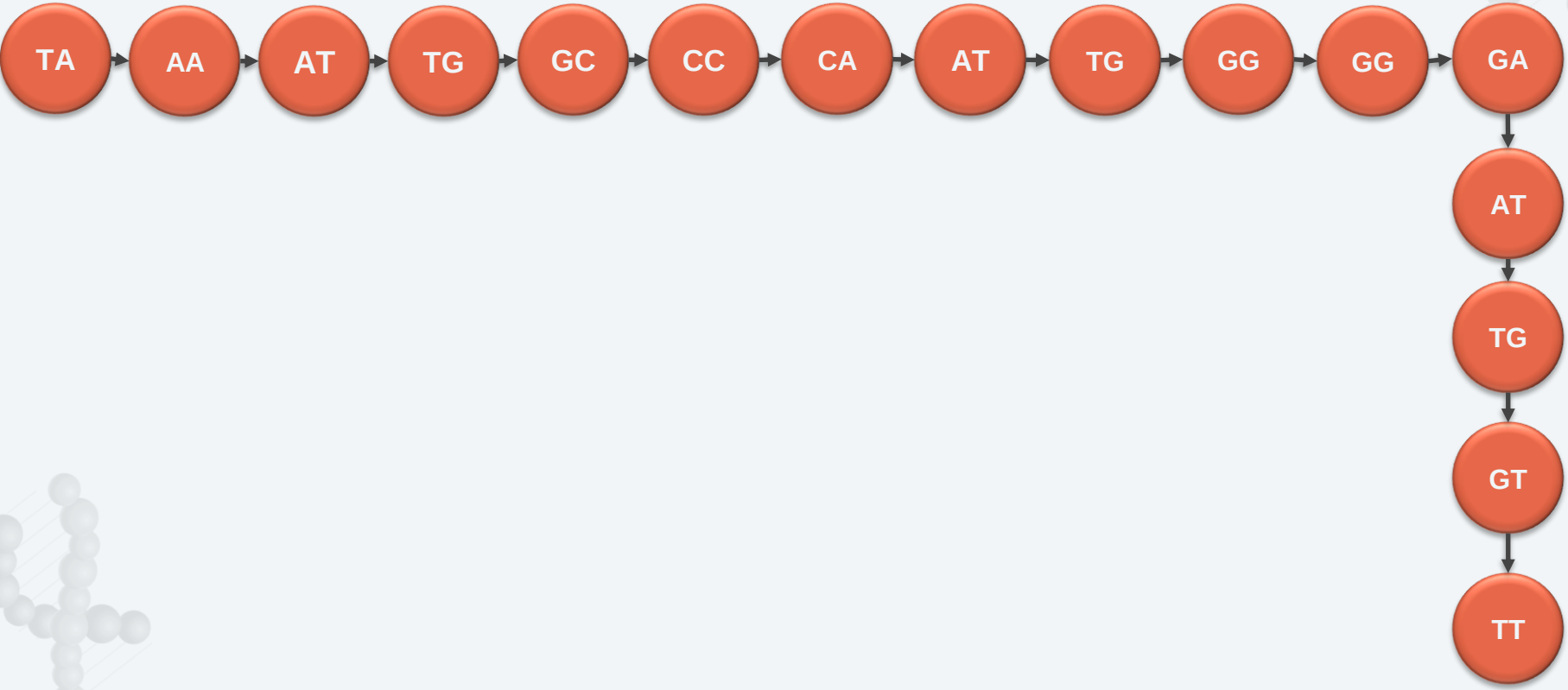


GRAFO DI DE BRUIJN – Step 9

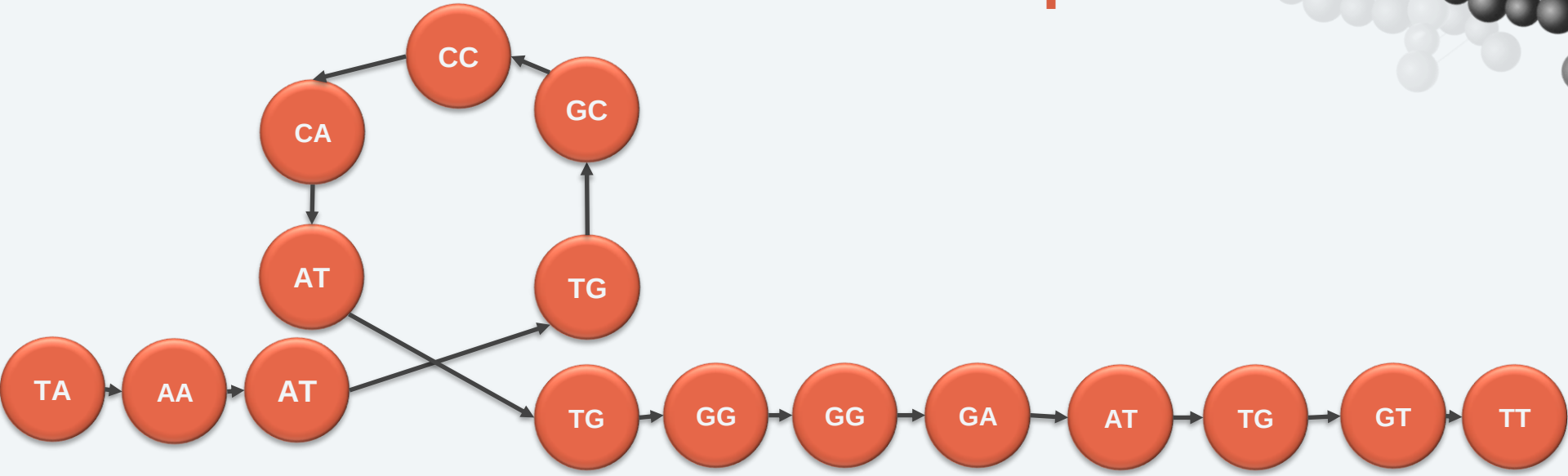
Una volta costruito, il grafo di de Bruijn è utilizzato per ricostruire la sequenza genomica originale:
Si identifica un circuito che attraversa tutti gli archi almeno una volta (cammino euleriano), ricostruendo la sequenza dai nodi visitati.
Eventuali biforcazioni nel grafo (causate da errori di sequenziamento o ripetizioni nel genoma) sono risolte utilizzando metodi di correzione



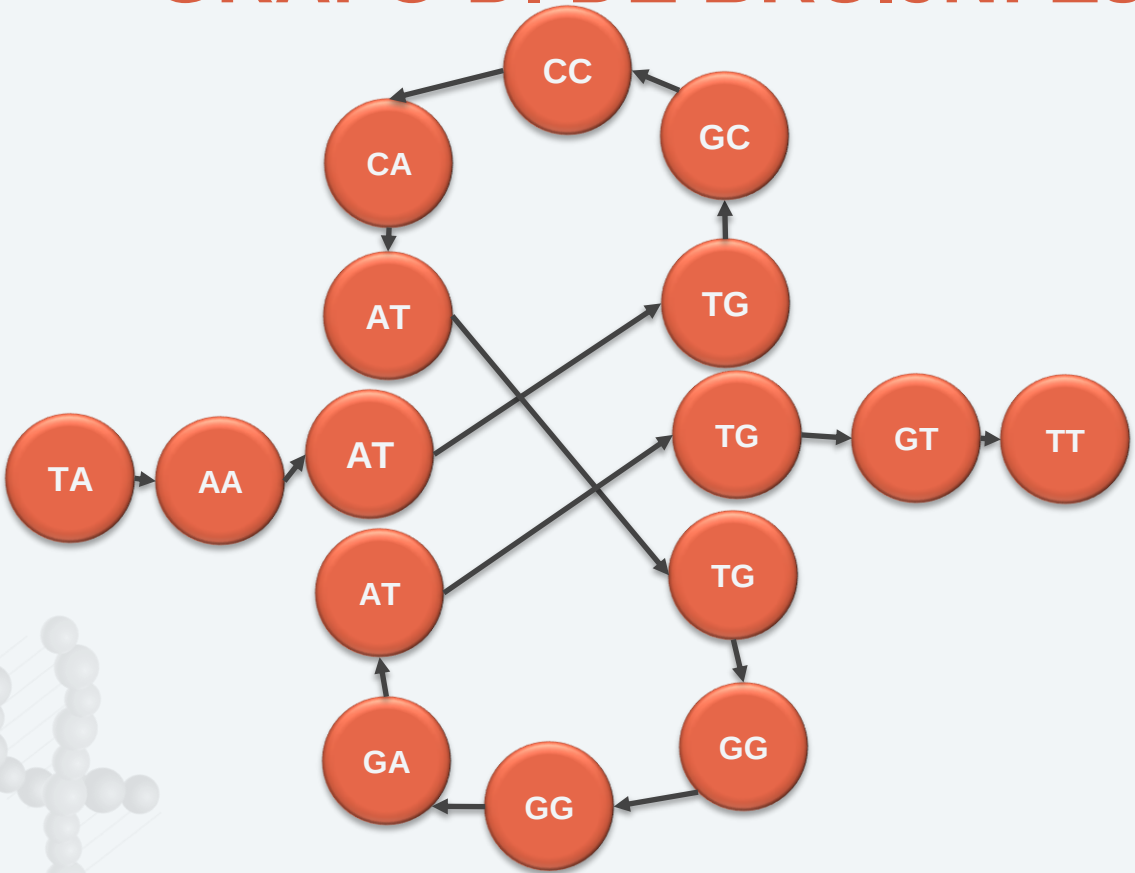
GRAFO DI DE BRUIJN: Esempio



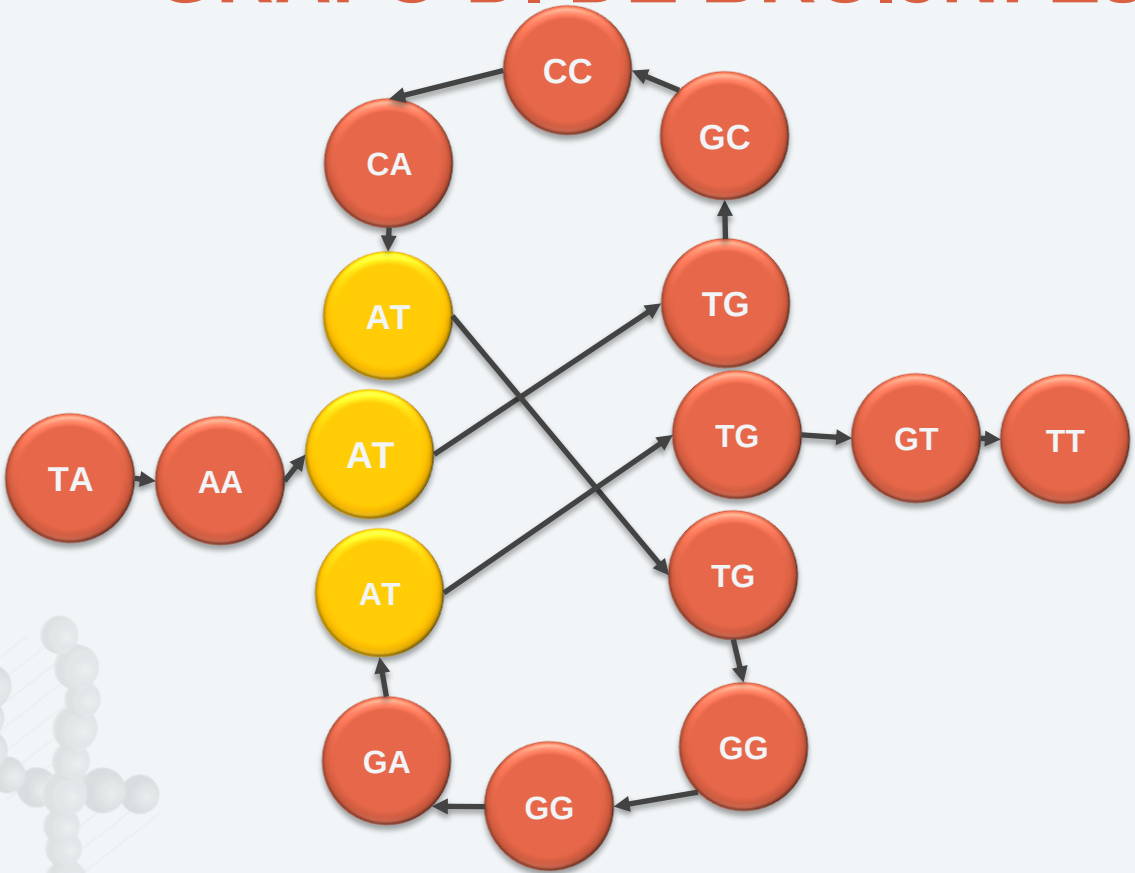
GRAFO DI DE BRUIJN: Esempio



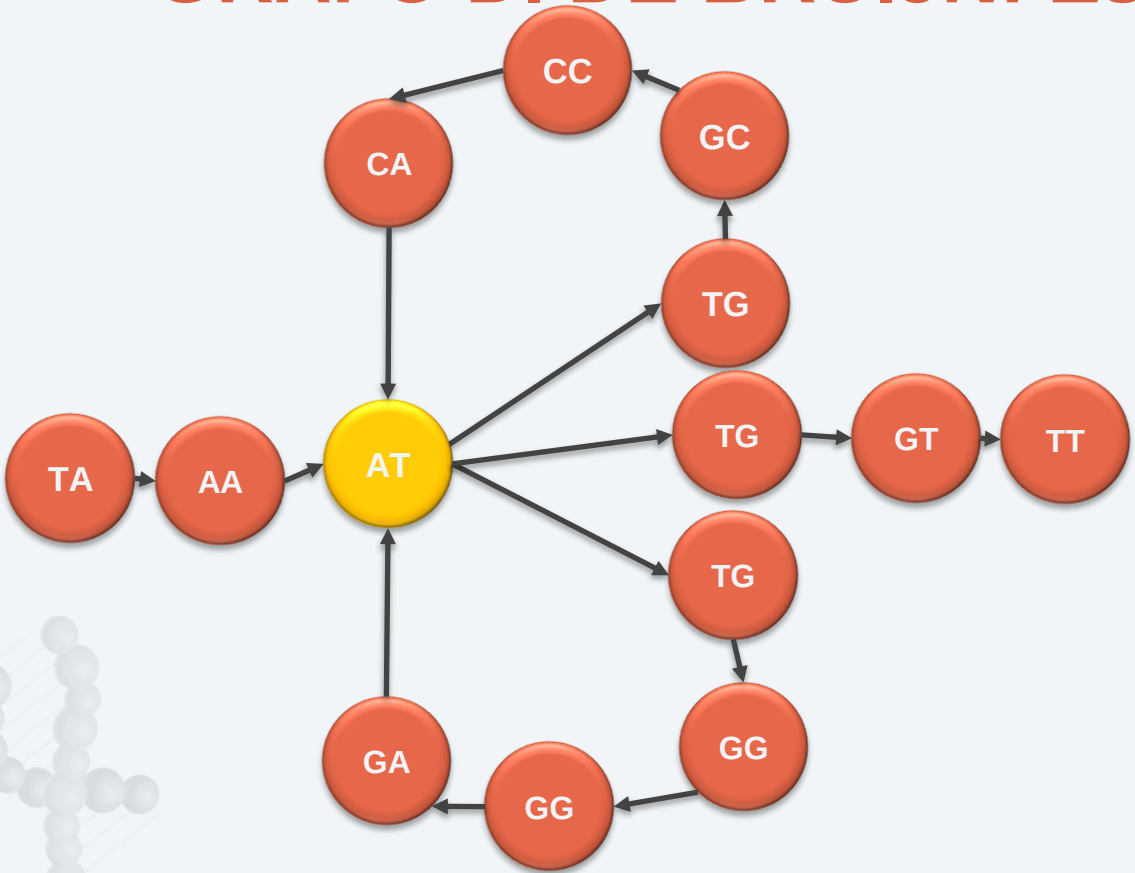
GRAFO DI DE BRUIJN: Esempio



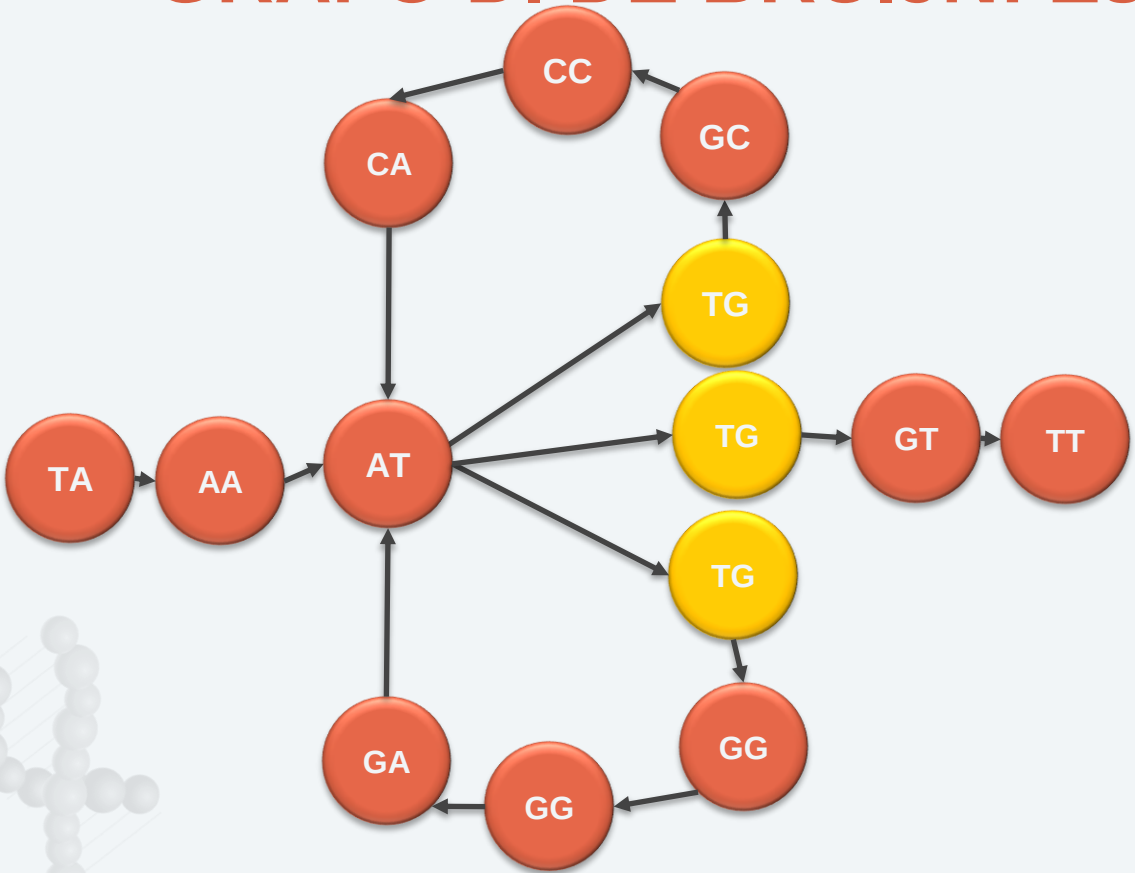
GRAFO DI DE BRUIJN: Esempio



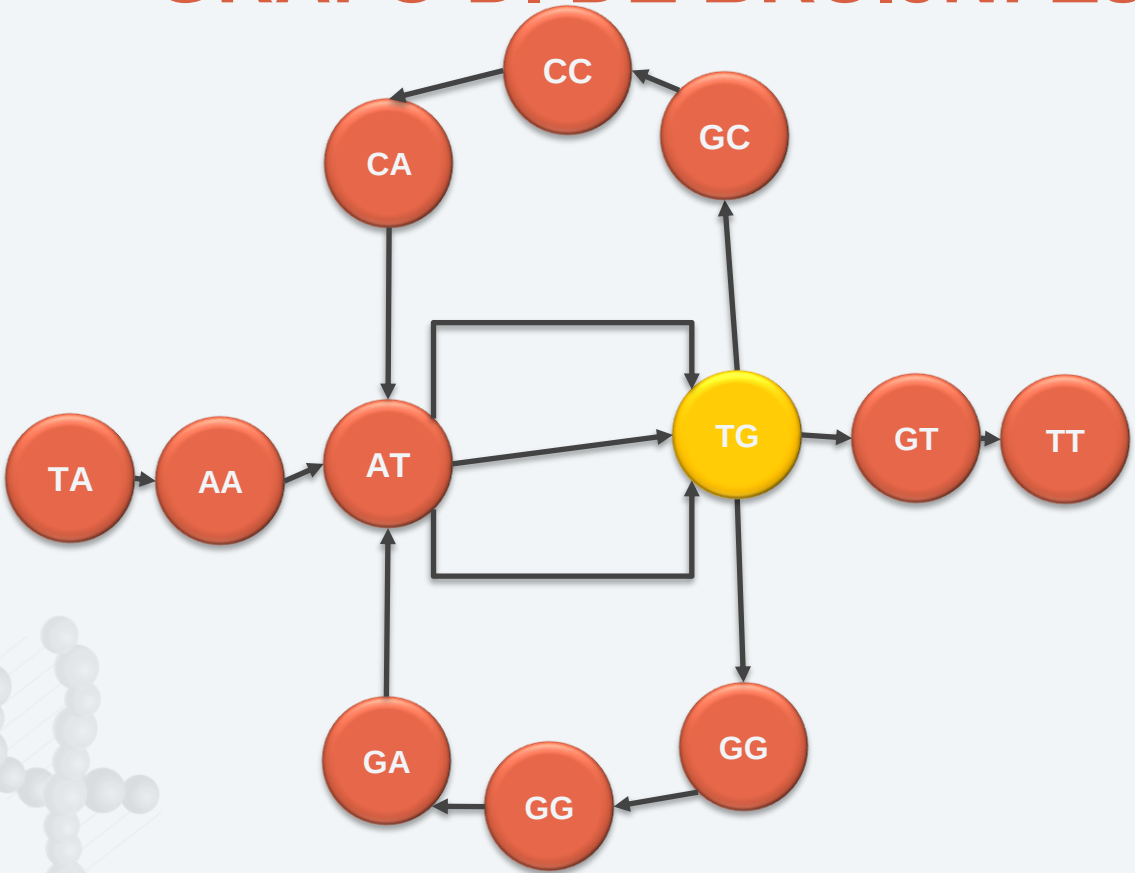
GRAFO DI DE BRUIJN: Esempio



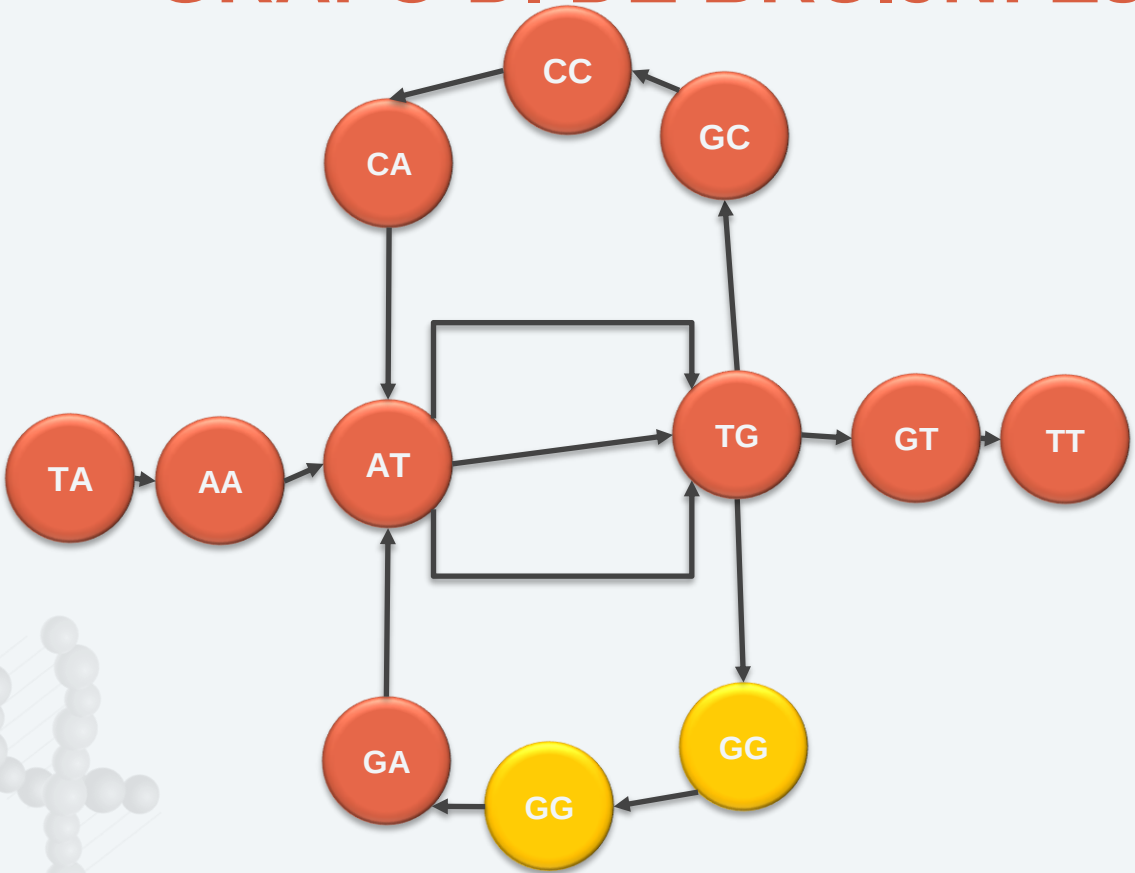
GRAFO DI DE BRUIJN: Esempio



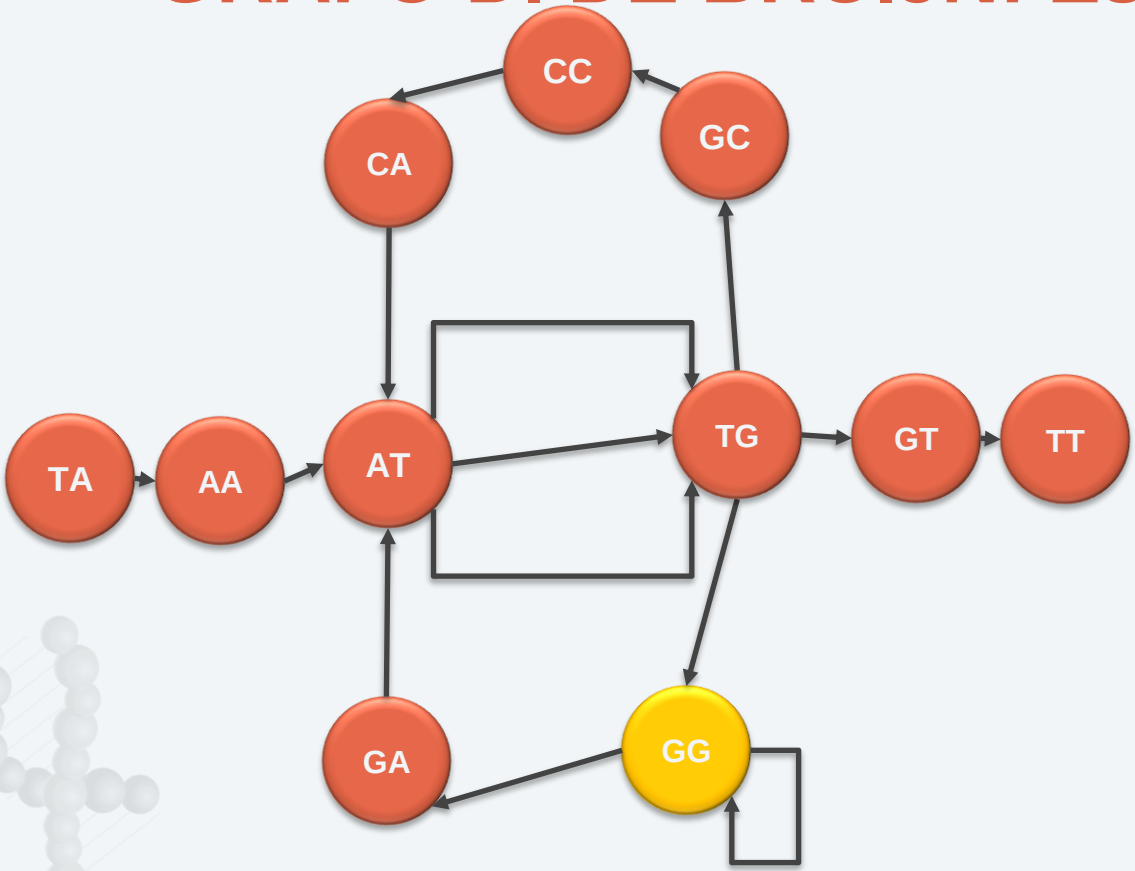
GRAFO DI DE BRUIJN: Esempio



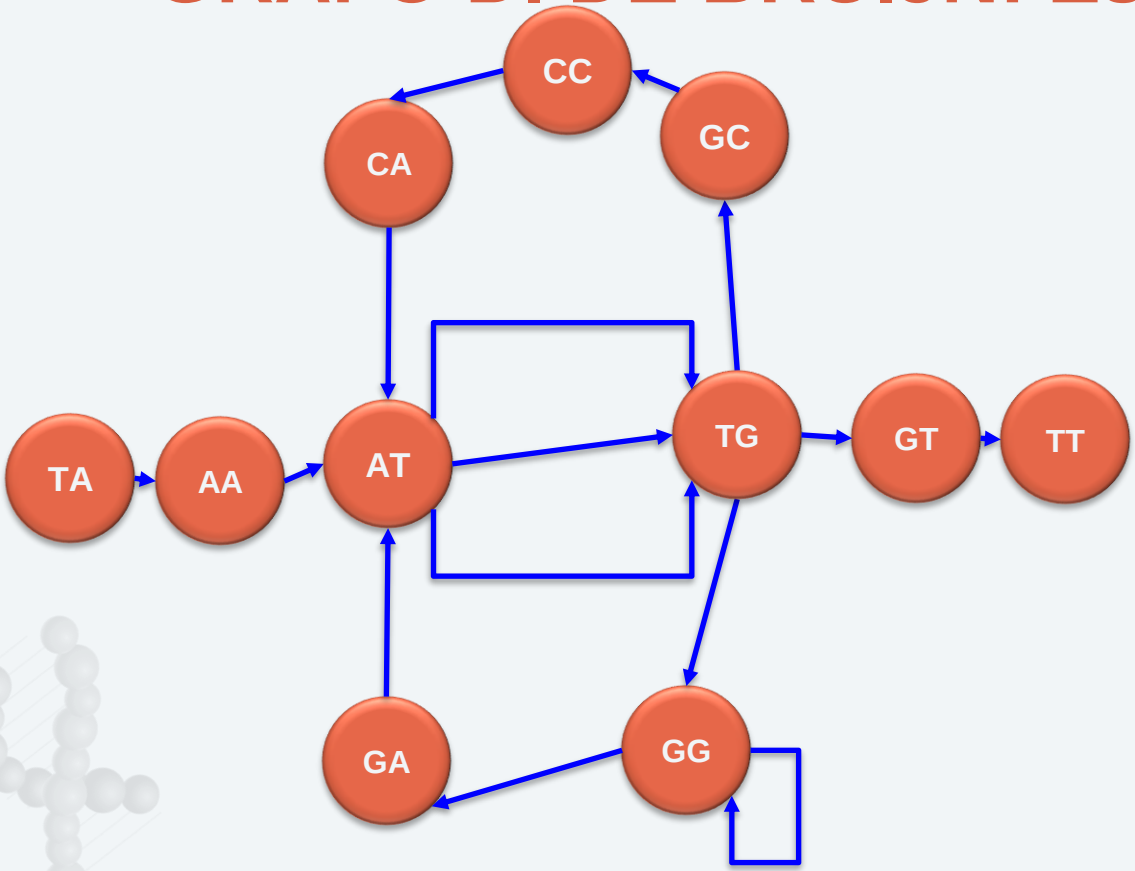
GRAFO DI DE BRUIJN: Esempio



GRAFO DI DE BRUIJN: Esempio



GRAFO DI DE BRUIJN: Esempio



TAATGCCATGGGATGTT

GRAFO DI DE BRUIJN: Errori

1. Ripetizioni nel Genoma
Le sequenze ripetute (tandem o disperse) nel genoma generano ambiguità nel grafo

Esempio: In un genoma con la sequenza ATGCGATGCGATGCG, la ripetizione di ATGCG comporterà percorsi multipli per rappresentare la stessa regione nel grafo. Questo rende difficile distinguere quante copie della ripetizione esistono e in che ordine siano disposte

GRAFO DI DE BRUIJN: Errori

2. Copertura Irregolare
Bassa copertura porta a lacune (gap) nel grafo, interrompendo percorsi che rappresentano il genoma.
Alta copertura, invece, può causare artefatti e falsi percorsi nel grafo.
Esempio: Con un sequenziamento scarso alcune regioni possono risultare non coperte e non rappresentate nel grafo. All'opposto, in un esperimento metagenomico, regioni altamente abbondanti (come i geni rRNA) possono essere rappresentate con un numero eccessivo di archi

GRAFO DI DE BRUIJN: Errori

3. Errori di Sequenziamento
Gli errori nelle read generano nodi aggiuntivi o percorsi non realistici.
Esempio: Se una sequenza reale è ATGCG, ma un errore di sequenziamento introduce ATGCA, il grafo includerà sia un nodo per ATGCG sia uno per ATGCA. Questo può complicare l'assemblaggio, soprattutto se i due nodi sono collegati da archi fasulli

GRAFO DI DE BRUIJN: Errori

4. Chimere
Il grafo può combinare erroneamente regioni non correlate del genoma, creando sequenze ibride.
Esempio: In un genoma con due regioni simili ma non identiche, come ATGCGTAC e ATGCGTGC, il grafo potrebbe unirle in una sequenza chimera come ATGCGTACGC. Questo errore è particolarmente comune in assemblaggi di metagenomi, dove più specie condividono regioni simili

GRAFO DI DE BRUIJN: Errori

5. Scelta Ambigua del Percorso
Quando esistono più percorsi possibili nel grafo, può essere difficile determinare quale sia il corretto

Esempio: Nel caso di un'inversione genomica (ad esempio AATTGGCC -> CCGGTTAA), i percorsi del grafo possono rappresentare entrambe le versioni, ma senza un'adeguata informazione aggiuntiva (come read paired-end), non è possibile scegliere il percorso corretto.

GRAFO DI DE BRUIJN: Errori

6. Limitazione dei k-mer
Un valore di k troppo piccolo aumenta l'ambiguità, rendendo difficile distinguere regioni simili.
Un valore di k troppo grande richiede una copertura maggiore per costruire un grafo robusto.
Esempio: Con k=5, due sequenze come ATGCGTAC e ATGCGTGC potrebbero condividere molti k-mer (ATGCG, TGCGT), portando a una fusione errata nel grafo. Con k=8, invece, sequenze di copertura insufficiente potrebbero non generare nodi nel grafo.

GRAFO DI DE BRUIJN: Errori

7. Assemblaggio di Metagenomi
Nei metagenomi (contenenti sequenze di più specie), il grafo può mescolare regioni simili provenienti da organismi diversi.
Esempio: In un campione ambientale con batteri diversi, geni condivisi (come quelli per la resistenza agli antibiotici) possono essere erroneamente fusi in un'unica sequenza nel grafo.
8. Collasso di Sequenze Ripetute
Sequenze ripetute possono essere rappresentate come una singola copia nel grafo, perdendo informazioni importanti.
Esempio: Un gene ripetuto 10 volte in un genoma può essere collassato in una sola copia, alterando l'interpretazione biologica del genoma.

GRAFO DI DE BRUIJN: Errori

9. Difficoltà nel Gestire Sequenze Lunghe
I grafi di De Bruijn sono progettati per gestire read corti; mentre con tecnologie a lunga lettura (PacBio, Nanopore) possono diventare inefficaci.
Esempio: Con una lettura lunga 10 kb, l'approccio tradizionale richiede una segmentazione in k-mer, perdendo il vantaggio delle informazioni di contesto sulla lunghezza del read.
10. Sensibilità a Sequenze Ripetitive e Geni Omologhi
Problema: Geni omologhi (simili ma non identici) o famiglie geniche possono generare percorsi confusi.
Esempio: Nei mammiferi, i geni delle immunoglobuline condividono sequenze altamente simili. Il grafo potrebbe erroneamente collegare segmenti provenienti da geni diversi.

GRAFO DI DE BRUIJN: Errori

11. Scarsità di Informazioni Strutturali
I grafi di De Bruijn non conservano direttamente informazioni su come i read sono stati originariamente mappati, limitando la ricostruzione di strutture genomiche complesse.
Esempio: In un genoma con grandi inversioni o traslocazioni, il grafo potrebbe non riuscire a rappresentare accuratamente l'organizzazione delle regioni.

12. Complessità Computazionale
Il grafo di De Bruijn può diventare enorme e difficile da gestire, specialmente per genomi grandi o metagenomi.
Esempio: Assemblando il genoma umano (3 miliardi di basi), il grafo può richiedere terabyte di memoria e una notevole capacità di calcolo.

GRAFO DI DE BRUIJN: Errori

13. Noise (Rumore) nei Dati
Errori o variazioni nel sequenziamento introducono un numero eccessivo di nodi e archi inutili.

Esempio: In un dataset di RNA-seq, il rumore causato da trascritti parziali o errori tecnici può generare un grafo estremamente frammentato e difficilmente interpretabile.

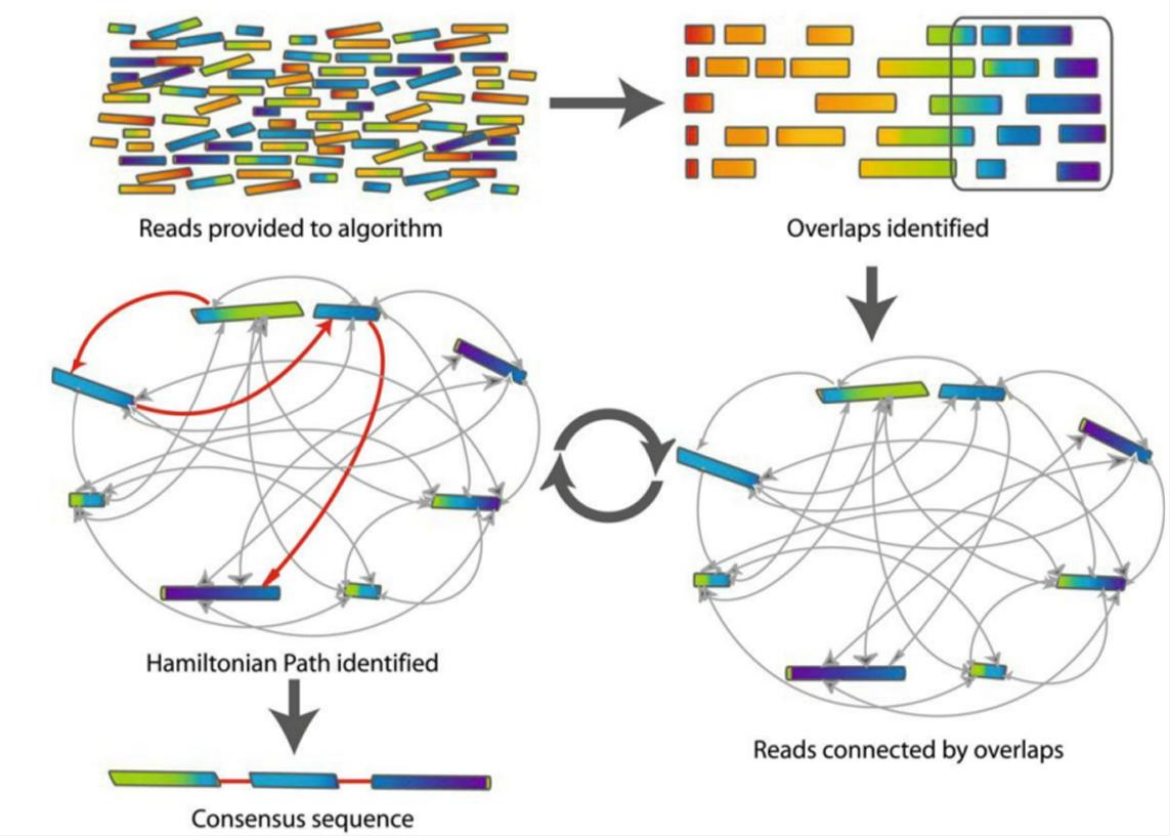
OVERLAP-LAYOUT-CONSENSUS (OLC)

OLC

L'algoritmo **Overlap-Layout-Consensus** (OLC) è utilizzato per l'assemblaggio di sequenze di DNA, in particolare con letture di sequenziamento di tipo long-read, come quelle prodotte da tecnologie PacBio o Oxford Nanopore

L'algoritmo consta di tre passaggi:
1. Sovrapposizione (Overlap)
2. Disposizione (Layout)
3. Consenso (Consensus)

OLC


Reads provided to algorithm
Overlaps identified
Hamiltonian Path identified
Reads connected by overlaps
Consensus sequence

OLC: SOVRAPPOSIZIONE (OVERLAP)

In questa fase, l'algoritmo identifica le regioni in cui le reads di sequenziamento si sovrappongono.
Poiché le reads possono essere rumorose (contenere errori), è usata una tolleranza per errori nelle sovrapposizioni
I sotto-passi sono:

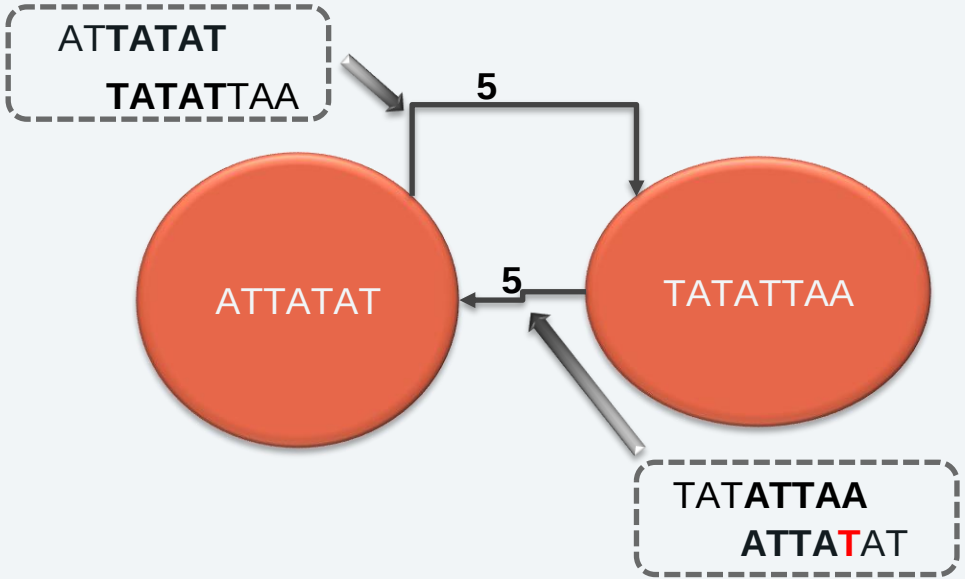
a) Ogni reads è confrontata con tutte le altre per trovare sovrapposizioni significative (di solito usando un indice per ridurre il numero di confronti). Per ciascuna coppia di letture, è calcolata una misura di similarità basata sulle sequenze nucleotidiche (tolleranza). **La tolleranza e il numero di sovrapposizioni sono dei valori fissi che non si possono eccedere**.
Questa fase si può determinare con la **Programmazione Dinamica** o la creazione di un **suffix tree**

b) Il risultato di questa fase è un grafo delle sovrapposizioni, dove ogni nodo rappresenta una reads e ogni arco una sovrapposizione significativa

OLC: SOVRAPPOSIZIONE (OVERLAP)

Reads:

TATATTAA **ATTATAT** **ATGTTAAC**
TAAATGT **ATATGT** **TGTTAACG**



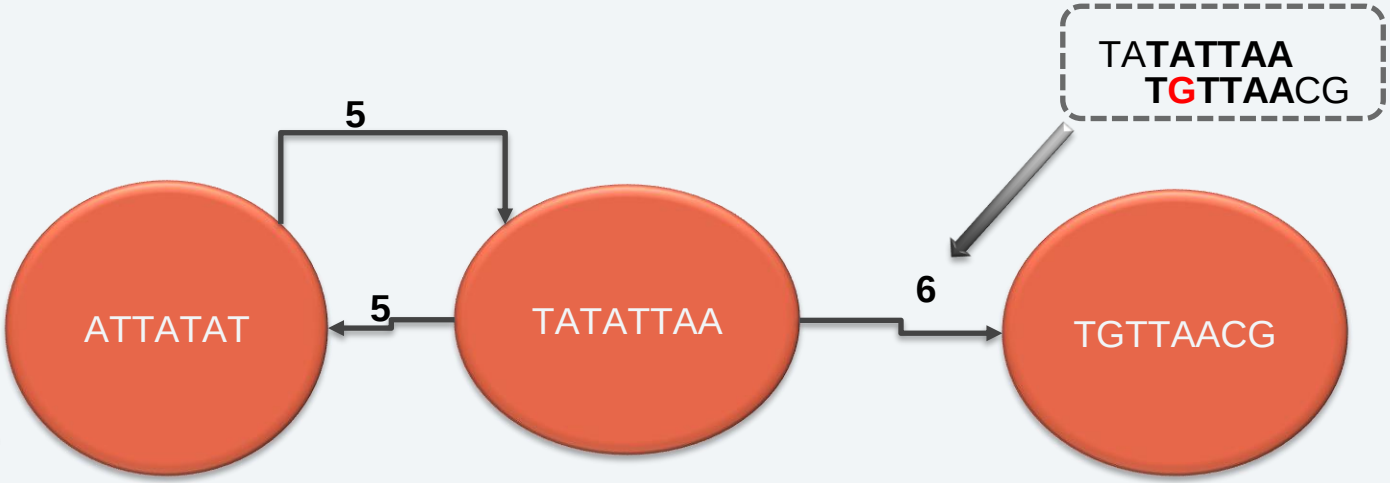
OLC<5;1>

Sovrapposizioni con almeno 5 basi uguali e un carattere non uguale

OLC: SOVRAPPOSIZIONE (OVERLAP)

Reads:

TATATTAA **ATTATAT** **ATGTTAAC**
TAAATGT **ATATGT** **TGTTAACG**



GENERAZIONE DI CONTIG
(sequenza continua (non contigua) risultante dal riassemblaggio dei piccoli frammenti di DNA generati da strategie di sequenziamento)

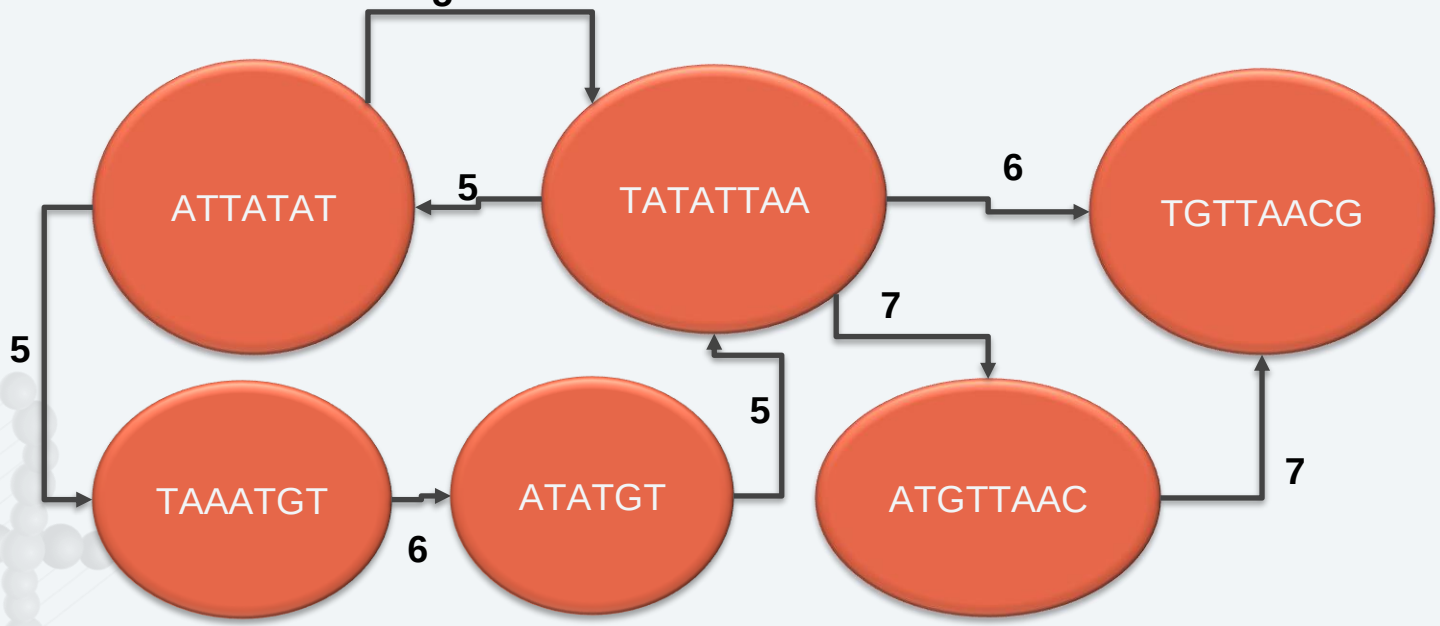
OLC: SOVRAPPOSIZIONE (OVERLAP)

Reads:

TATATTAA
TAAATGT

ATTATAT
ATATGT

ATGTTAAC
TGTTAACG



OLC: DISPOSIZIONE (LAYOUT)

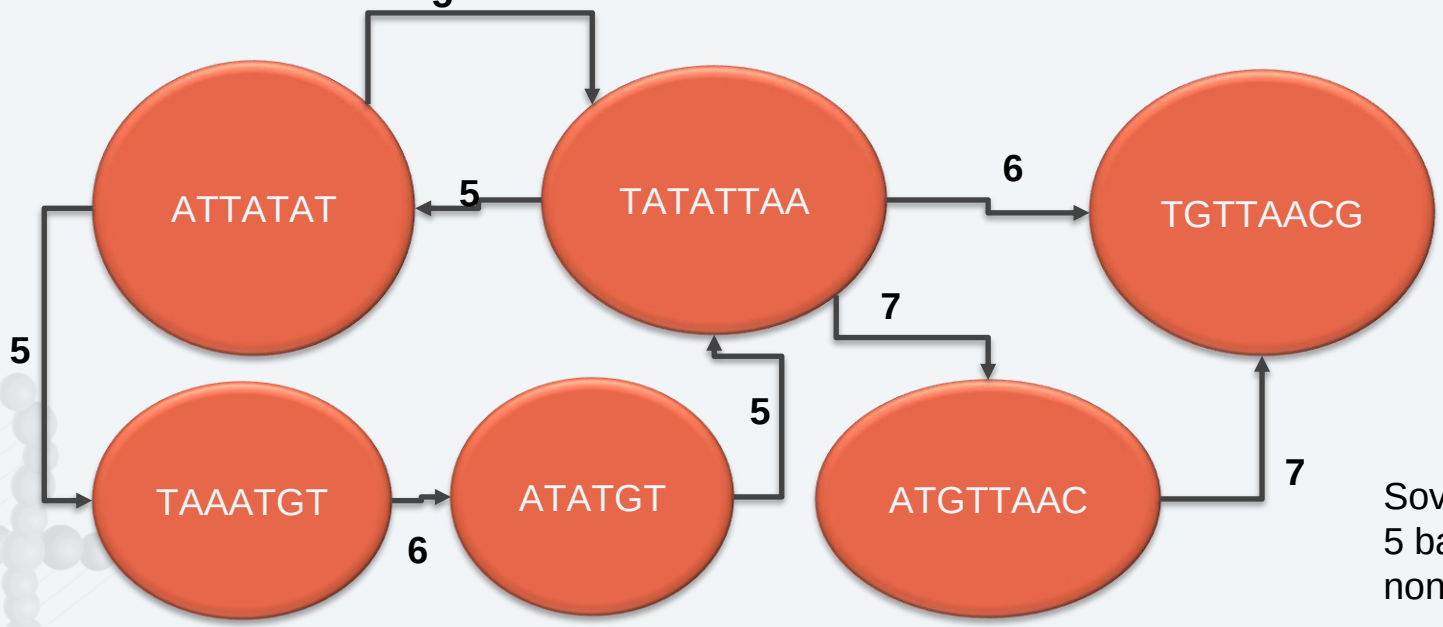
La fase di layout costruisce un ordinamento delle letture in relazione al grafo delle sovrapposizioni

L'obiettivo è **determinare il percorso ottimale** attraverso il grafo per rappresentare l'ordine in cui le letture si susseguono nel genoma originale.
Questo è un problema complesso (simile al problema del commesso viaggiatore), quindi sono utilizzati approcci euristici per trovare una soluzione pratica.
Il risultato di questa fase è un consenso preliminare delle letture, basato sulle regioni sovrapposte.

OLC: DISPOSIZIONE (LAYOUT)

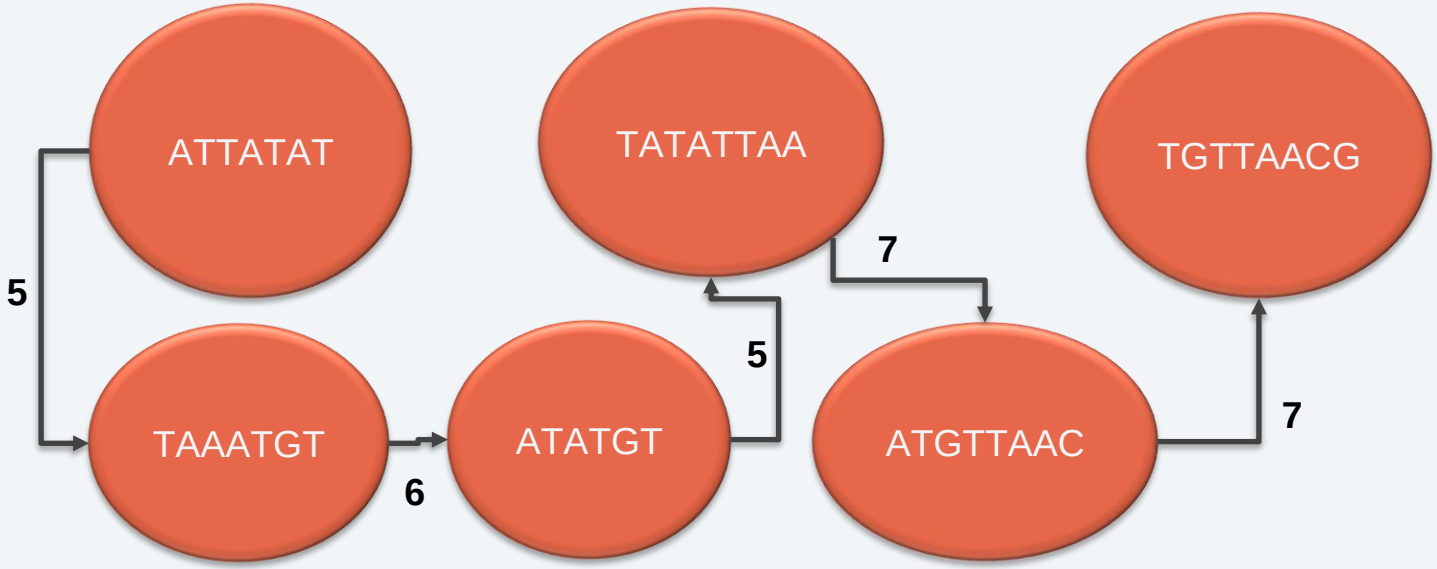
Reads:

TATATTAA **ATTATAT** **ATGTTAAC**
TAAATGT **ATATGT** **TGTTAACG**



Sovrapposizioni con almeno 5 basi uguali e un carattere non uguale

OLC: DISPOSIZIONE (LAYOUT)



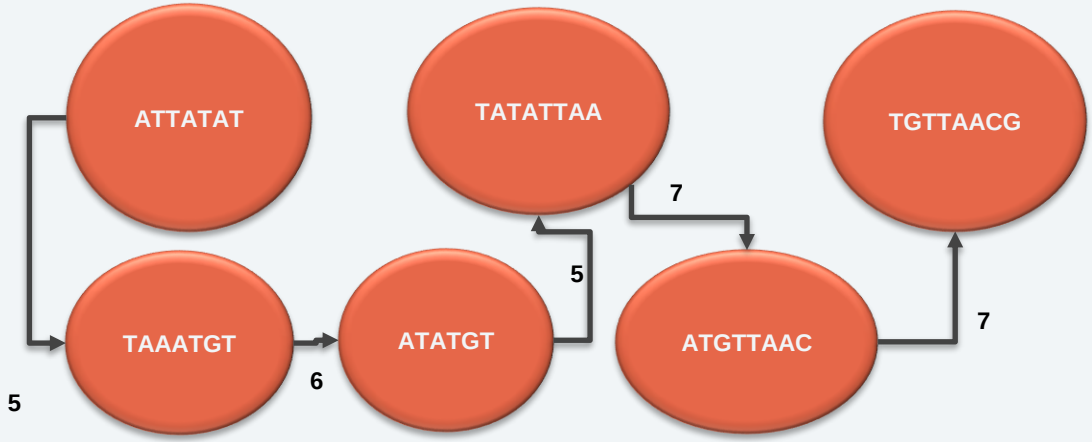
OLC: CONSENSO (CONSENSUS)

Nella fase di consenso, è prodotta la **sequenza genomica ricostruita** (consenso finale)

Le sequenze sovrapposte delle letture sono "rifinite" per generare una sequenza che massimizza l'accordo tra le reads sovrapposte

Sono corretti errori di sequenziamento o discrepanze usando metodi di allineamento multiplo (Multiple Sequence Alignment, MSA) o algoritmi specifici per il consenso

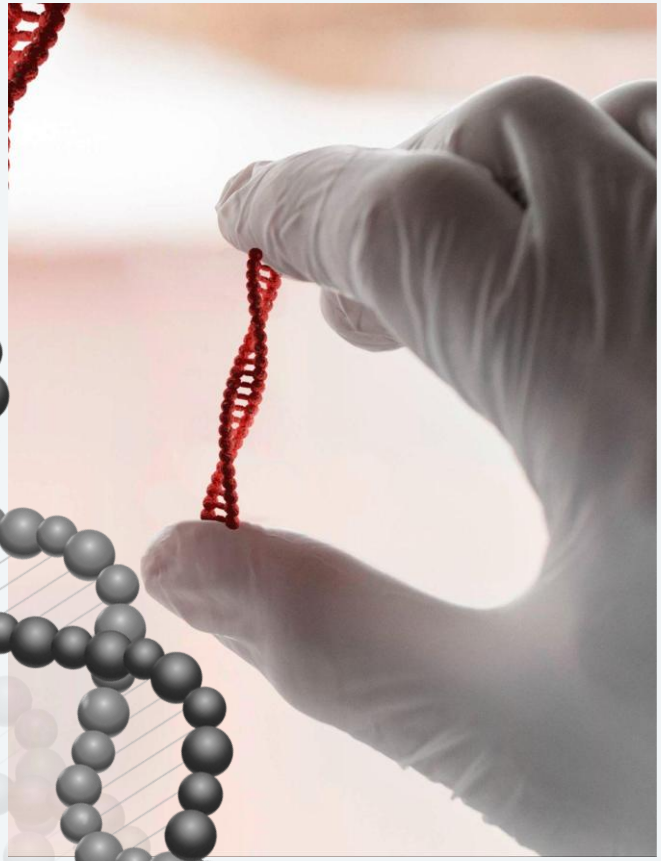
OLC: CONSENSUS



A	T	T	A	T	A	T							
		T	A	A	A	T	G	T					
			A	T	A	T	G	T					
				T	A	T	A	T	A	A			
					A	T	G	T	T	A	A	C	
						T	G	T	T	A	A	C	G
A	T	T	A	T	A	T	G	T	T	A	A	C	G

GENERAZIONE DI SCAFFOLD

sequenze ottenute dal collegamento di una serie non contigua di sequenze genomiche



03

SOFTWARE DI ASSEMBLAGGIO

SOFTWARE DI ASSEMBLAGGIO

Il sequenziamento RNA-Seq produce milioni di frammenti di sequenza (read) che rappresentano trascritti RNA.

L'assemblaggio RNA mira a ricostruire:
Trascritti completi: sequenze di RNA messaggero (mRNA), RNA non codificante, ecc.
Trascrittoma: l'insieme completo di trascritti espressi in un campione.

HISAT2 + StringTie, STAR + RSEM, TopHat + Cufflinks sono strumenti di assemblaggio de novo Reference-based assembly

Trinity e RNA SPAdes sono principalmente strumenti di assemblaggio de novo

BIOINFORMATICA

RNASPADES

rnaSPADES è un software utilizzato per l'assemblaggio dei trascrittomi da dati di RNA-Seq.
Questo strumento è basato sull'approccio di costruzione del grafo di De Bruijn.

RNASPADES: Algoritmo

Passo 1 - Creazione dei K-mer: rnaspades inizia suddividendo le read di RNA-Seq in k-mer, che sono sottosequenze di lunghezza fissa (ad esempio, 31 basi). Questi k-mer sono essenzialmente "pezzi" delle read che sovrappongono tra loro. La scelta di k influisce sulla sensibilità e sulla specificità dell'assemblaggio. K-mer più grandi catturano sovrapposizioni più lunghe ma richiedono più memoria

Passo 2 - Costruzione del Grafo di De Bruijn: rnaspades utilizza i k-mer per costruire un grafo di De Bruijn. Il grafo è una rappresentazione delle possibili sequenze trascrizionali presenti nei dati di RNA-Seq

RNASPADES: Algoritmo

Passo 3 - Ricerca di Contigs: Il software cerca quindi sequenze trascrizionali complete (chiamate contigs) all'interno del grafo di De Bruijn. Inizia cercando nodi con un grado di ingresso (numero di archi in ingresso) uguale a 0 (estremità di un percorso) e grado di uscita uguale a 1 (inizio di un percorso). Questi nodi rappresentano il punto di partenza di una sequenza trascrizionale.

Passo 4 - Allungamento delle Sequenze: A partire dai nodi iniziali, rnaspades allunga progressivamente le sequenze, seguendo gli archi del grafo di De Bruijn. Questo processo continua fino a quando non si raggiunge un punto di biforcazione (nodo con più di un arco in uscita) o un nodo di arresto (nodo con un grado di uscita maggiore di uno).

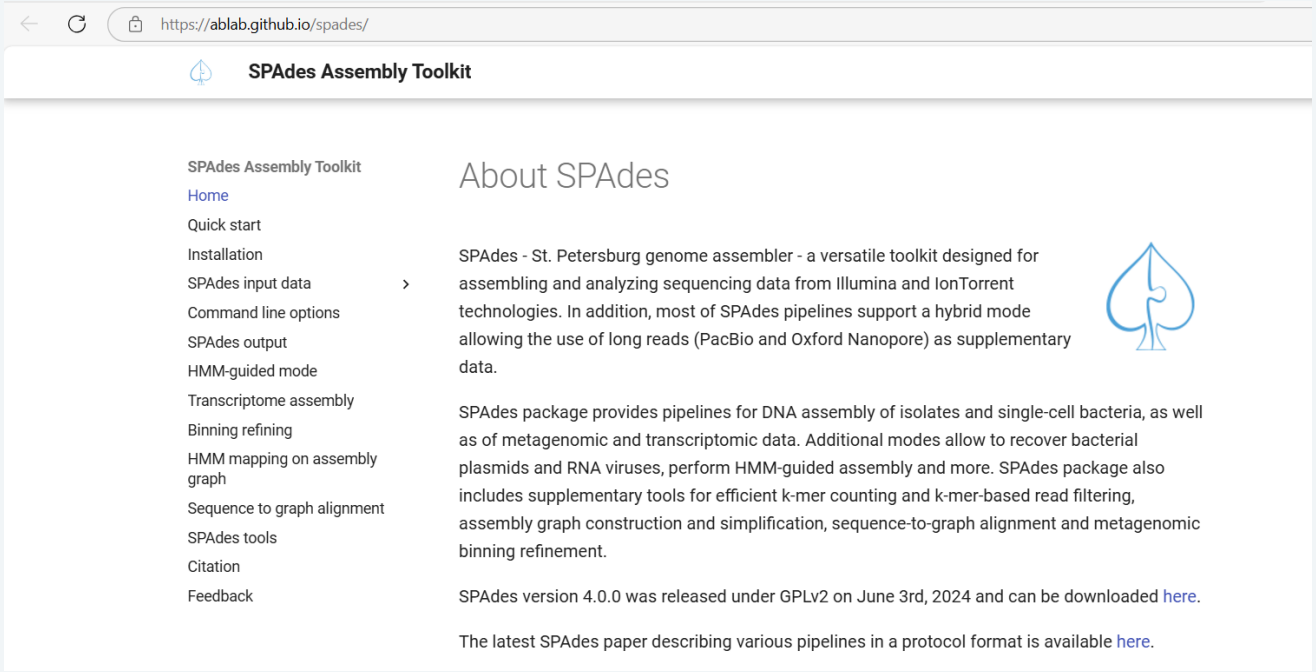
RNASPADES: Algoritmo

Passo 5 - Risoluzione delle Sovrapposizioni: Quando si raggiungono punti di biforcazione, rnaspades cerca di risolvere le sovrapposizioni e di decidere quale percorso seguire per formare sequenze trascrizionali complete. Questo passaggio è critico per gestire splicing alternativi o isoforme trascrizionali.

Passo 6 - Output delle Contigs: Alla fine, rnaspades produce un insieme di contigs, che rappresentano sequenze trascrizionali complete o parziali. Questi contigs possono essere utilizzati per studiare l'annotazione dei geni, l'espressione genica e altre analisi biologiche

RNASPADES: Sito

https://github.com/ablab/spades
https://ablab.github.io/spades/



RNASPADES: Installazione

È sufficiente caricare il file:
https://github.com/ablab/spades/archive/refs/heads/main.zip

Dipendenze
Python

RNASPADES: Esempio

rnaSPADES è un assemblatore progettato per costruire trascrittomi a partire da dati RNA-Seq. È parte del pacchetto SPAdes e può essere utilizzato per assemblaggi de novo di RNA. Ecco un esempio di utilizzo:

PE reads

rnaspades.py -1 reads_1a.fastq,reads_1b.fastq -2 reads_2a.fastq,reads_2b.fastq -o output_dir

Parametro utile –t n (specifica i numeri di threads)

In output_dir si trova il file **transcript.fasta**

TRINITY

Trinity è un programma informatico in grado di assemblare trascritti (cioè, RNA messaggeri) in assenza di un genoma di riferimento, il che lo rende particolarmente utile per studiare organismi per i quali non esistono genomi di riferimento annotati

Trinity è adatto quando l'obiettivo principale è identificare isoforme trascrizionali, cioè diverse varianti di una sequenza trascrizionale per lo stesso gene.

Trinity è sensibile alle sovrapposizioni tra le read e può gestire bene le sequenze con splicing alternativo o complessità trascrizionale.

TRINITY: Algoritmo

L’algoritmo di Trinity è suddivisibile in tre parti:

Parte I - Inchworm

Input: Letture di RNA-Seq (in formato FASTQ).

Metodo:

Ordina i k-mer (sequenze di lunghezza k) estratti dalle letture.

Costruisce un grafo basato sui k-mer, in cui i nodi rappresentano i k-mer e gli archi rappresentano le sovrapposizioni.

Si focalizza sui cammini unici (circuiti hamiltoniani) nel grafo per estendere i contigs in modo deterministico.

Output: Contigs primari che rappresentano trascritti altamente espressi o sequenze core comuni.

TRINITY: Algoritmo

Parte II - Chrysalis

Input: Contigs prodotti da Inchworm.

Metodo: Suddivide i contigs in cluster di trascritti (basati su similarità) che rappresentano potenziali geni. Per ciascun cluster, costruisce un grafo de Bruijn per rappresentare varianti e connessioni tra trascritti. Identifica possibili percorsi alternativi per rappresentare isoforme di trascritti o trascritti correlati.

Output: Grafi parziali che rappresentano le strutture trascrittomiche locali.

TRINITY: Algoritmo

Parte III - Butterfly

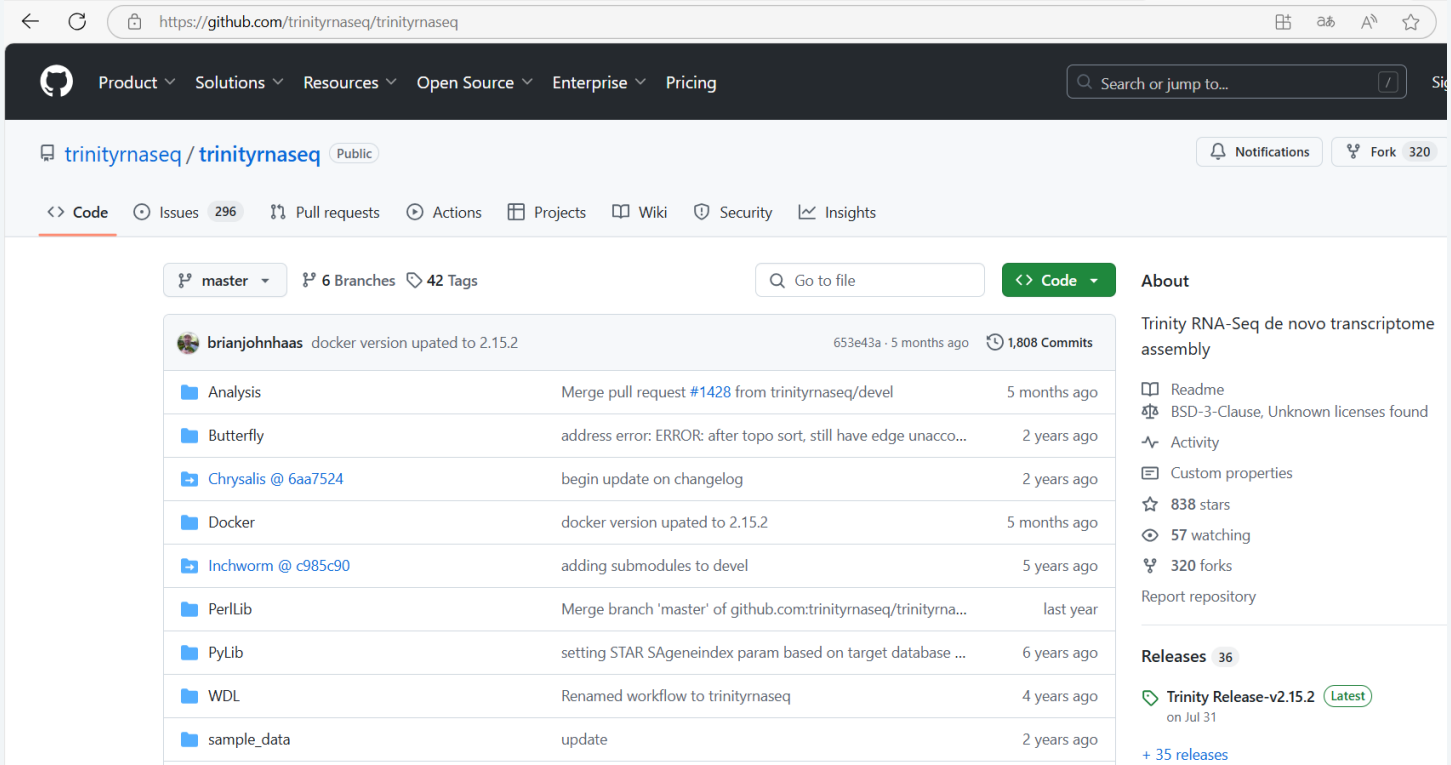
Input: Grafi di sovrapposizione generati da Chrysalis.

Metodo: Estrae percorsi dai grafi de Bruijn per rappresentare isoforme o trascritti completi.
Tiene conto della copertura delle letture per discriminare tra percorsi plausibili.
Risolve cicli o biforcazioni nel grafo basandosi su dati di espressione e informazioni di supporto dalle letture RNA-Seq.

Output: Sequenze di trascritti finali assemblati (in formato FASTA).

TRINITY: Sito

https://github.com/trinityrnaseq/trinityrnaseq



TRINITY: Installazione

È sufficiente scaricare il file da:
https://github.com/trinityrnaseq/trinityrnaseq/releases
Entrare nella cartella e digitare il comando
Make
Installare i plugin
make plugins

Dipendenze:
bowtie2
jellyfish
salmon
Samtools
python

TINITY: Esempio

Si deve eseguire

```
Trinity --seqType fq --max_memory 50G
        --left condA_1.fq.gz,condB_1.fq.gz,condC_1.fq.gz
        --right condA_2.fq.gz,condB_2.fq.gz,condC_2.fq.gz
--CPU 6
```

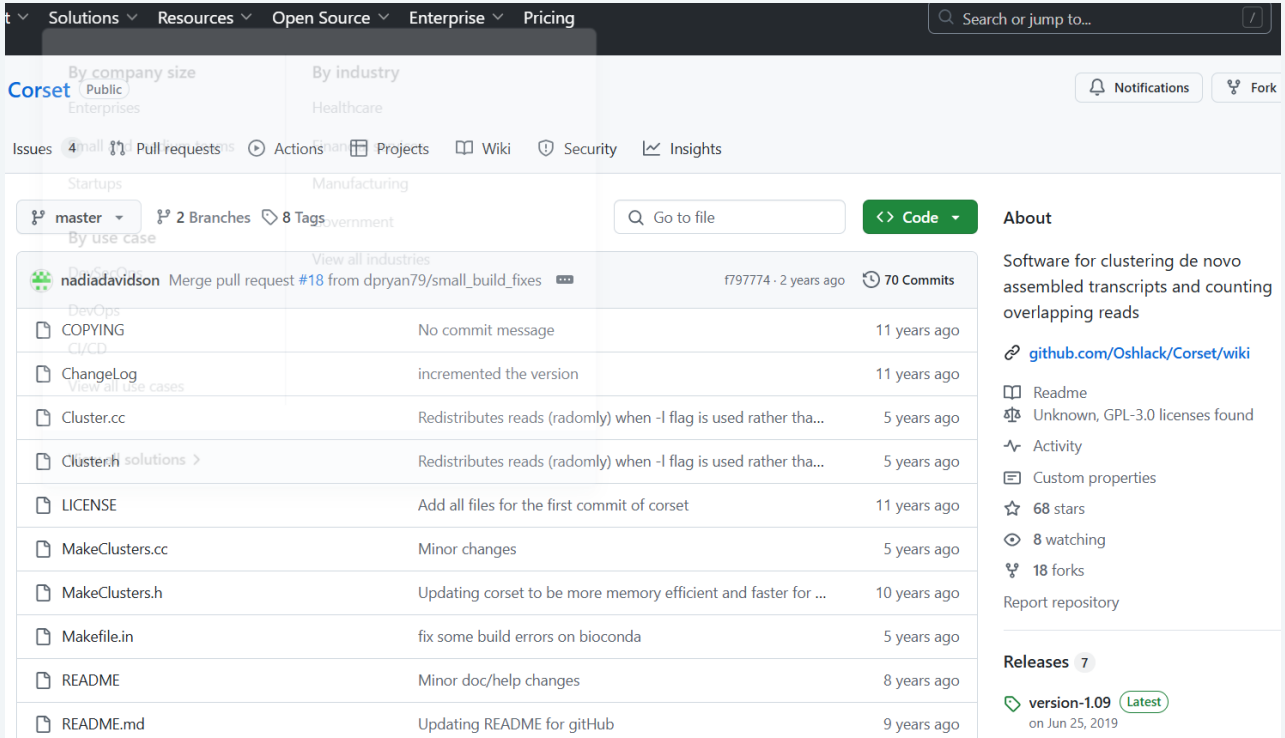
Dove
--seqType specifica il tipo di file
--max_memory indica la quantità di memoria massima da utilizzare
--left i file FASTQ forward
--right i file FASTQ reerse (i file vanno elencati nello stesso ordine dei forward)
--CPU n numero di core/thread utilizzati

CORSET

Quando si utilizzano assemblatori come Trinity o SPAdes, i risultati possono includere molteplici trascritti per gene.
Corset organizza questi trascritti in gruppi (cluster) che rappresentano singoli geni o unità funzionali simili.
Ciò è particolarmente utile in organismi senza genomi di riferimento, dove non è possibile mappare i trascritti direttamente al genoma.

CORSET: Sito

https://github.com/Oshlack/Corset



CORSET: Installazione

È sufficiente scaricare il file da:
https://github.com/Oshlack/Corset/releases/download/version-1.09/corset-1.09-linux64.tar.gz

Se si scaricano i sorgenti da compilare (https://github.com/Oshlack/Corset/archive/refs/tags/version-1.09.zip) entrare nella cartella e digitare il comando
./configure

Dipendenze:
Samtools

CORSET: Esempio

Esempio di utilizzo di Corset
Immaginiamo un esperimento di RNA-Seq su un organismo senza genoma di riferimento:

1 Assemblaggio dei trascritti:
Esecuzione di Trinity/RNASpades per assemblare i trascritti, ottenendo un file FASTA di trascritti assemblati.

2 Mappatura delle read:
Mappare le read di RNA-Seq sui trascritti assemblati usando un mapper come Bowtie2 o Salmon, ottenendo file BAM.

3 Esecuzione di Corset:
Passare i file BAM e il file dei trascritti assemblati a Corset.
Corset genera:
Cluster di trascritti (file con ID dei trascritti raggruppati per cluster). È un file FASTA.
Dati di conteggio aggregati per ogni cluster

CORSET: Esempio

RNASPADES
python ./SPAdes-4.0.0-Linux/bin/rnaspades.py -t 48 -o /ASSEMBLY/specie_spades_k_auto
--dataset SPECIE/specie_dataset.yaml --tmp-dir /ASSEMBLY/TMP_SPADE/ --only-assembler

SALMON
salmon quant --index /SPECIE/INDEX_SALMON/ARTEMIA_indici --libType A -1
SPECIE/TRIMMED/specie_forward.fq -2 /SPECIE/TRIMMED/specie_reverse.fq
--output SPECIE/SALMON/SALMON_specie --dumpEq

CORSET Passo 1
./corset corset -i salmon_eq_classes /SPECIE/SALMON/SALMON_*/aux_info/eq_classes.txt -f true
CORSET Passo 2
python fetchClusterSeqs.py -i /SPECIE/TRANSCRIPTS/specie_spades_transcript.fasta -o
/SPECIE/TRANSCRIPTS/specie_corset_transcript.fasta -c /SPECIE/CORSET_DATA/clusters.txt

Grazie!

Domande?

franco.liberati@unitus.it
deb.scienceontheweb.com