Il problema dei Cammini Minimi

corso di laurea in **Matematica**

Informatica Generale, Lezione **23 (a)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

L'importanza dei cammini minimi

È un problema fondamentale in numerose **situazioni** in cui sono **coinvolte le reti**, ad esempio:

- distanze **stradali/ferroviarie/aeree.**
- ottimizzazione **distribuzione di energia**
- ottimizzazione **instradamento** dei **pacchetti in Internet**.

Spesso è necessario risolvere questo problema in **pochi millisecondi** su **reti con migliaia o milioni di nodi** (ad esempio i router che instradano i pacchetti, tenendo conto **dell'attuale carico della rete**).

Considereremo anche **valori negativi** possono anche rappresentare costi, penalità, perdite etc.

Non dimenticate, infine, che è tipico il caso in cui **problemi di natura diversa** possano venir risolti trovando come **modello un problema di cammini minimi** (questo in particolare giustifica il caso – meno intuitivo – di pesi anche negativi).

Definizione del problema

Un cammino c da u a v ($c : u \rightsquigarrow v$) in un grafo **orientato** $G = (V, E)$ è una sequenza di nodi $c = v_0, v_1, \ldots, v_k$ tali che $(v_i, v_{i+1}) \in E$, $v_0 = u$ e $v_k = v$. $\boldsymbol{C(u, v) = \{ c \mid c : u \rightsquigarrow v \}}$ è l'insieme di **tutti i cammini** da u a v.

Sia $w : E \rightarrow \mathbb{R}$ una funzione peso. Il **peso di un cammino $w(c)$** è definito come la **somma** dei pesi degli archi $\sum_{i=0, \ldots, k-1} w(v_i, v_{i+1})$.

[Occasionalmente possono venire considerate **altre misure**, come il **massimo costo** di un arco (**bottleneck**), il prodotto etc.]

Il peso (o costo) di un **cammino minimo** da u a v è definito come:

$\boldsymbol{\delta(u, v) = \min_{c \in C(u, v)} w(c)}$ se v è raggiungibile da u, e $\boldsymbol{+\infty}$ altrimenti.

Si considera in genere, il problema dei cammini minimi da **sorgente unica**: dato un nodo s, il problema è trovare $\delta(s, v)$, per ogni $v \in V$.

È intuitivo che questo problema si risolva con **la stessa complessità** di determinare il cammino minimo **tra una coppia** (fissata) **di nodi**.

Un problema collegato è determinare i cammini minimi **tra tutte le coppie di nodi**, che si può risolvere con n problemi di cammini minimi a sorgente unica o con algoritmi specifici.

Proprietà di sottostruttura

Le **proprietà di sottostruttura** (che permettono di costruire soluzioni a un'istanza di un problema a partire dalle soluzioni di istanze più piccole dello stesso problema) sono un ingrediente fondamentale di due tecniche algoritmiche conosciute come **algoritmi greedy** (o **golosi**) e **programmazione dinamica**.

Proposizione: Sia $c = v_0, v_1, \ldots, v_k$ un cammino di peso minimo tra v_0 e v_k. Allora ogni sotto-cammino $c_{i,j} = v_i, \ldots, v_j$ è un cammino di peso minimo tra v_i e v_j.

Dim: Assumiamo per assurdo che ci sia un cammino di costo minore $c'_{i,j}$ tra v_i e v_j tale che $w(c'_{i,j}) < w(c_{i,j})$. Per la proprietà additiva però avremo che $w(c') = w(c_{0,j}) + w(c'_{i,j}) + w(c_{j,k})$ sarebbe strettamente minore di $w(c) = w(c_{0,j}) + w(c_{i,j}) + w(c_{j,k})$ il che contraddice l'ipotesi che c sia un cammino di peso minimo. □

I due algoritmi che vedremo per la soluzione di cammini minimi da sorgente unica sono uno **goloso** (algoritmo di **Dijkstra**) e l'altro di **programmazione dinamica** (algoritmo di **Bellman-Ford**).

Archi e cicli di peso negativo

Spesso è interessante considerare istanze con **archi di peso negativo**, tuttavia questi possono produrre anche cicli di **peso negativo**.

- Un ciclo di **peso positivo** non viene **mai percorso in un cammino minimo** perché peggiora il costo senza modificare la raggiungibilità.
- Un ciclo di **peso nullo** non modifica il costo di un cammino, indipendentemente se viene percorso 0, 1 o più volte: **per ogni soluzione**, ce n'è **una equivalente aciclica**.
- se il ciclo ha **peso negativo**, può essere percorso un numero arbitrario di volte, sempre diminuendo il peso del cammino. In questo caso, **il problema non è ben definito** e definiamo $\delta(u, v) = -\infty$ se andando da u a v è possibile percorrere un ciclo di peso negativo.

Osserviamo che quando il problema è ben definito, **un cammino di peso minimo è semplice** e quindi ha al più $n - 1$ **archi**.

Questo implica che la **soluzione** di un problema di cammini minimi è un **valore per ogni nodo** (distanza dalla sorgente) più **un albero di cammini minimi** radicato nella sorgente (questo dipende anche dalla proprietà di sottostruttura)

Il caso dei pesi positivi: Algoritmo di Dijkstra

corso di laurea in **Matematica**

Informatica Generale, Lezione **23(a²)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Il caso dei pesi positivi

Considerare solo pesi positivi semplifica leggermente il problema:

- **Non** è necessario **preoccuparsi del problema** dell'esistenza di **cicli di peso negativo**.
- Il peso $w(p)$ di un cammino p che è un segmento **contenuto in un cammino p'** più lungo **è sempre minore uguale a $w(p')$**.

In particolare, la seconda proprietà ci dice che se abbiamo **già determinato i cammini minimi** che portano da s a un certo **insieme di nodi M**, non possiamo migliorare la loro distanza da s con **prolungamenti dei cammini già esaminati**.

Quindi, l'insieme dei nodi di cui conosciamo la distanza minima si **può stabilizzare** in un procedimento iterativo.

Per capire se un nuovo nodo **può essere inserito in M**, è sufficiente considerare **i vicini di nodi in M**.

Algoritmo di Dijkstra: idea

L'idea base è quella di **generalizzare la BFS**, prendendo via via il **nodo più vicino** della **frontiera**.

Quindi, in ogni momento, i nodi sono partizionati in 3 insiemi:

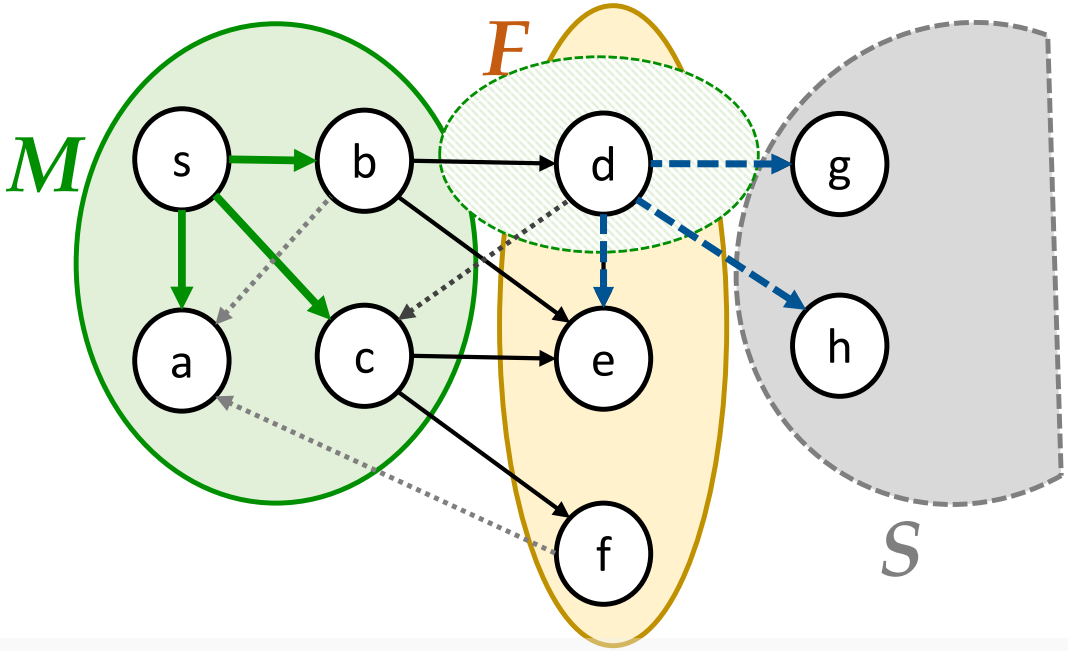
- M, nodi visitati di cui è **già nota** la **distanza minima** da s
- F, nodi della **frontiera**, già scoperti, e con una distanza "parziale" già calcolata (ma che **potrebbe ancora migliorare**)
- S, nodi **ancora sconosciuti**.

I nodi in F **sono adiacenti ai nodi in** M: la loro distanza può migliorare quando viene inserito un nuovo nodo in V.

Tuttavia è sempre vero che $\forall u \in M, \forall v \in V \setminus M.\ d(s, u) \leq d(s, v)$.

Non è mai possibile che la distanza di un nodo possa decrescere per effetto di un detour che si scopre successivamente e che passa per nodi ancora in S: questo dipende in modo essenziale **1**) dal fatto che i pesi siano solo positivi e **2**) i nodi in M sono più vicini a s.

Dijkstra: idea & invarianti



INVARIANTI:

$\forall m \in M.\ d[m] = \delta(s, m)$

$\forall m \in M,\ \forall v \in V \setminus M.$
$\delta(s, m) \leq \delta(s, v)$

I **nodi verdi di *M*** hanno già la loro **distanza minima**, dei **nodi in *F*** conosco **la distanza usando i nodi in *M***.

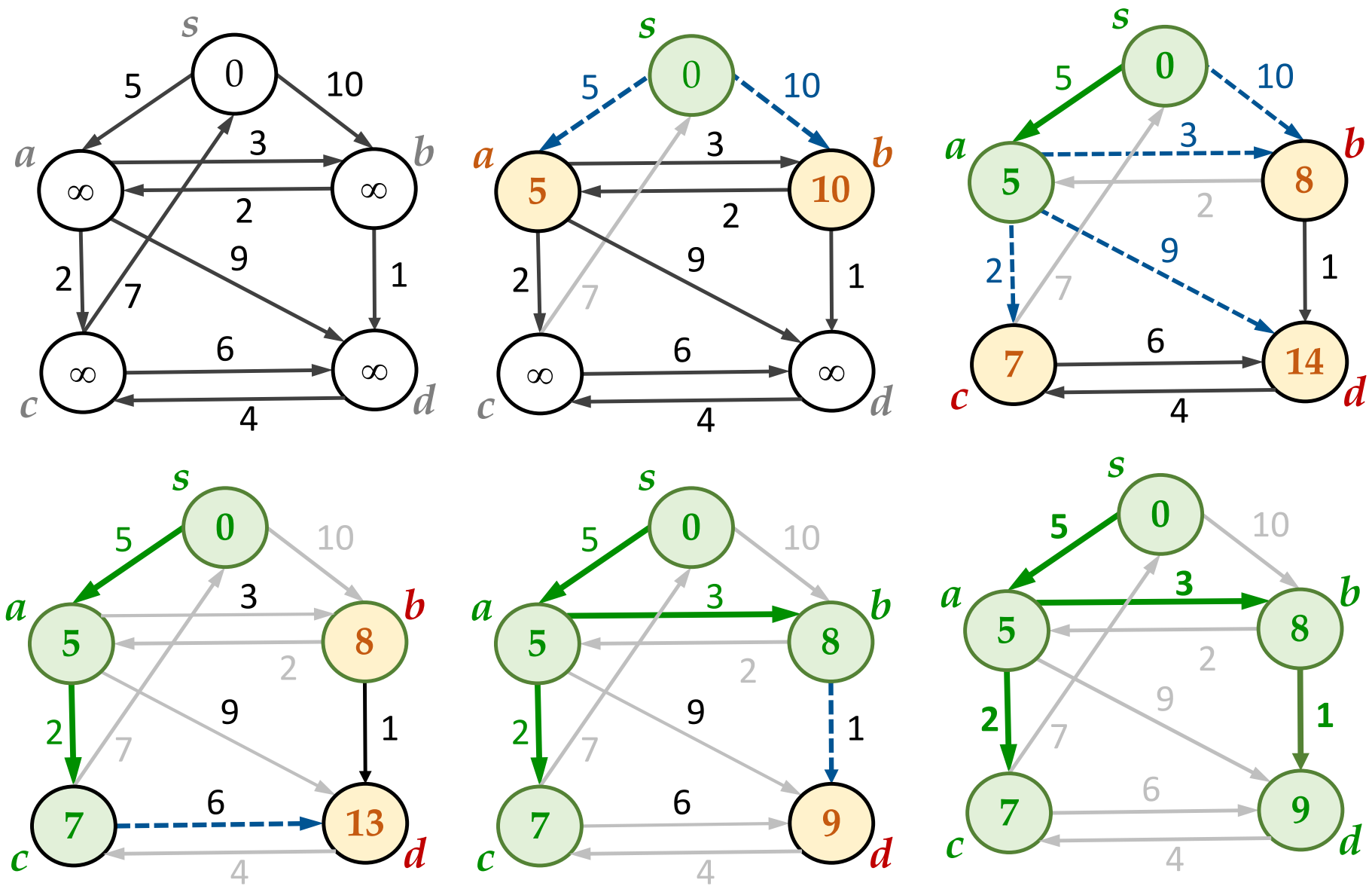
Gli archi **grigi tratteggiati** sono ininfluenti per il calcolo del minimo: ad esempio, il cammino $b \rightarrow d \rightarrow c$ **non può migliorare il peso di *c*** (perché $\delta(c) \leq \delta(d) \leq \delta(d) + w(d, c)$ per l'invariante)

Si inserisce in M il nodo a distanza minima, ad esempio d.

Si analizzano gli **archi blu tratteggiati uscenti da *d*** per aggiornare le distanze minime (esempio nodo e) o scoprire nuovi nodi (es. g e h).

Infatti, è possibile che il cammino $b \rightarrow d \rightarrow e$ abbia peso minore di $b \rightarrow e$.

Algoritmo di Dijkstra: esempio



Algoritmo di Dijkstra: pseudocodice

Gli sconosciuti sono semplicemente i nodi per cui $d[u] = \infty$ e quelli in M sono quelli che escono da F dopo esserci entrati.

Quando un nodo u entra in M, *si* considerano tutti i suoi archi uscenti e riaggiornare le **distanze dei nodi d'arrivo** con la funzione **`relax`**.

La funzione **`aggiusta`** aggiunge in F, oppure **mantiene F pronta** per la prossima **estrazione di minimo**.

```
def initSingleSource(G, s):
    forall v ∈ V[G]:
        d[v] = ∞
        p[v] = v
    d[s] = 0
```

$\Theta(n)$

```
def dijkstra(G, w, s):
    d, p = allocaV(n), allocaV(n)
    initSingleSource(G, s, d, p)
    F = {s}
    while F ≠ ∅:
        sia u, d[u] è minima in F
        F = F ∖ {u}
        forall v ∈ G.adj(u):
            if relax(u, v, d, p):
                aggiusta(F, v)
    return d, p
```

$\Theta(n)$ **minimi**

$\Theta(m)$ **aggiunte/ aggiornamenti**

```
def relax(u, v, d, p):
# ENS: aggiorna distanze e padre
# ritorna True se modifica
    if d[v] > d[u] + w[u][v]:
        d[v] = d[u] + w[u][v]
        p[v] = u
        return True
    return False
```

$\Theta(1)$

Implementazione & complessità

Anche se nello pseudocodice abbiamo trattato F come un insieme, in realtà viene usata come una **coda con priorità**, in quanto su F facciamo le seguenti operazioni:

- **estrazione di minimo** ($\theta(n)$ volte il nodo a minima distanza)
- **inserimento** di un nuovo nodo ($\theta(n)$ volte)
- **aggiornamento** delle **distanze** ($\theta(m)$ volte, una per ogni arco, per effetto di un nuovo nodo inserito in M)

Analizziamo due strutture dati per implementare la coda F:

- **vettore non ordinato**: inserzioni/aggiornamenti si fanno in $\theta(1)$ avendo l'indice del nodo, mentre le estrazioni di minimo costano $\theta(n)$. Totale $\boldsymbol{\theta(n^2) + \theta(m)}$
- **heap**: inserzioni/aggiornamenti ed estrazioni di minimo si fanno tutte in $\theta(\log n)$. Totale $\boldsymbol{\theta(n \log n) + \theta(m \log n)}$.

Se il **grafo è denso** ($m = \Theta(n^2)$) **conviene il vettore** in quanto ho che il costo è $\Theta(n^2)$ [contro $\Theta(n^2 \log n)$ con heap].

Se il **grafo è sparso** ($m = \Theta(n)$) allora **conviene lo heap**: infatti il costo è $\theta(n \log n)$ mentre con il vettore rimane $\Theta(n^2)$.

Correttezza algoritmo di Dijkstra

È sufficiente osservare che le proprietà viste prima:

$$(\mathbf{1})\ \forall m \in M.\ d[m] = \delta(s, m)$$

$$(\mathbf{2})\ \forall m \in M,\ \forall v \in V \setminus M.\ \delta(s, m) \leq \delta(s, v)$$

sono effettivamente invarianti.

Dim: (**INIT**) s è il primo nodo a entrare in s, e $d[s]$ viene inizializzato a $0 = \delta(s, s)$. Inoltre $0 \leq \delta(s, v)$, per ogni $v \in V$.

(**conservazione**) L'invariante si conserva perché se u è il nodo scelto in una iterazione, allora $d[u]$ è minima in $V \setminus M$ e corrisponde a un cammino $s \rightsquigarrow v \rightsquigarrow u$ con $s \rightsquigarrow v$ minimo e $v \in M$ con $w(s \rightsquigarrow v \rightsquigarrow u) \leq w(s \rightsquigarrow v' \rightsquigarrow u)$ per tutti gli altri cammini fatti di tutti nodi in $M \cup \{ u \}$.

Quindi un eventuale cammino c' di costo minore dovrebbe passare per un nodo $z \notin M$. Ma se $\delta(s, v) = d[z] + w(z, v)$ allora $\delta(s, v) \geq d[z]$ perché gli archi hanno peso positivo e per (**2**) $d[z] \geq \delta(s, z) \geq \delta(s, u) + w(u, v) = d[v]$ e quindi $\delta(s, v) = d[v]$, v può entrare in M e $\delta(s, v) \leq \delta(s, z) \leq d[z]$ per tutti i nodi z non ancora in M. $\square$

Algoritmi Greedy

L'algortimo di Dijkstra appartiene a una gloriosa famiglia di algoritmi: gli algoritmi **greedy** (o **golosi**).

Un algoritmo greedy costruisce incrementalmente una soluzione, scegliendo ogni volta ciò che **appare favorevole** sulla base di **informazioni parziali** o **locali**, senza analizzare la struttura globale dell'istanza del problema in esame.

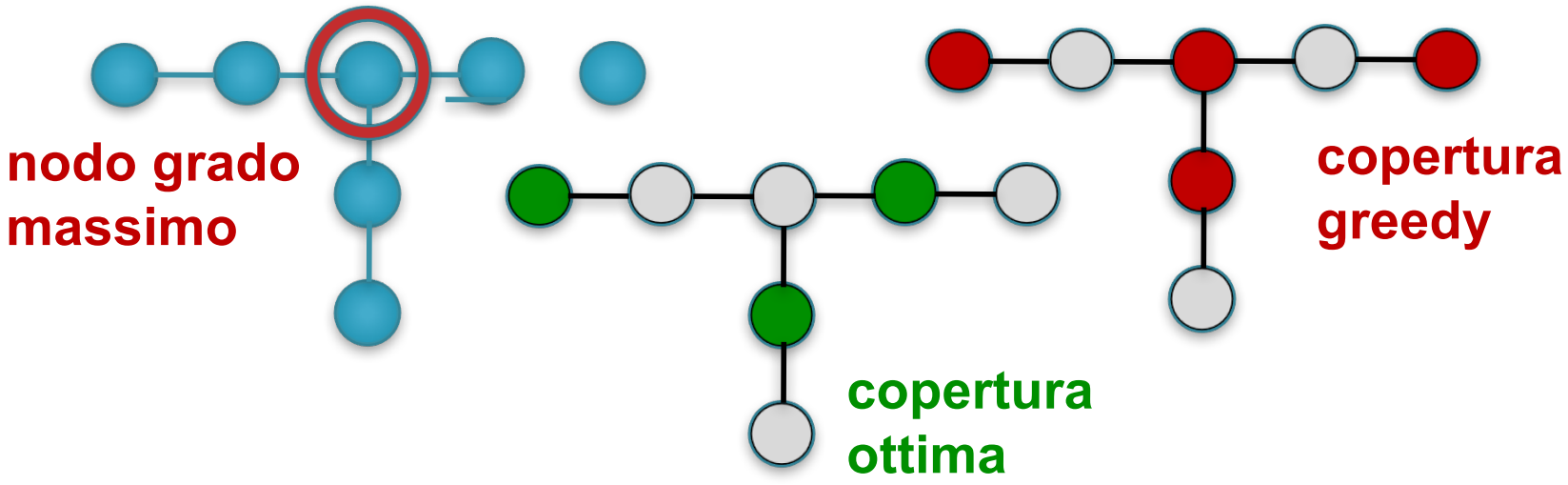
Nel caso dell'algoritmo di Dijkstra, prendiamo **sempre il nodo più vicino** e in questo caso tale strategia (o meglio, euristica) porta alla soluzione ottima.

Ci sono numerose e sofisticate caratterizzazioni dei casi in cui la strategia greedy porti necessariamente a una soluzione ottima.

Tuttavia, in **molti casi**, questa strategia **non dà risultati ottimi**.

Limiti della Strategia Greedy

Un **node covering** di un grafo $G=(V, E)$ è un sottoinsieme $V' \subseteq V$ tale che per ogni nodo $u \in V$, esiste un nodo $v \in V'$ tali che $(u, v) \in E$.

Trovare una generica **copertura** è relativamente **facile** (ad esempio prendendo $V'=V$). In generale, siamo interessati al **node covering di cardinalità minima** (o l'esistenza di uno di cardinalità fissata)

Uno potrebbe pensare di risolvere il problema con una **strategia greedy**, scegliendo iterativamente **i nodi di grado massimo** (che quindi **coprono il maggior numero di** altri **nodi**). Tuttavia questa strategia (che dà un algoritmo polinomiale, nell'ordine di $|V|^2$) **non necessariamente restituisce la copertura minima.**