Esercizio. Progettare un algoritmo che dato un grafo orientato $G = (V, E)$, due vertici $v_1, v_2 \in V$, ed un numero naturale K, restituisce *true* se esiste un ciclo di lunghezza minore o uguale a K che include sia v_1 che v_2, altrimenti restituisce *false*.

Soluzione. Un ciclo che include sia il vertice v_1 che il vertice v_2 è composto da un cammino da v_1 a v_2, seguito da un cammino da v_2 a v_1. Quindi, il ciclo di lunghezza minima che include sia il vertice v_1 che il vertice v_2 avrà lunghezza pari alla somma della distanza di v_2 rispetto a v_1 (minima lunghezza di un cammino da v_1 a v_2) e della distanza di v_1 rispetto a v_2 (minima lunghezza di un cammino da v_2 a v_1). Per risolvere questo problema l'Algoritmo 1 semplicemente controlla se la somma delle due distanze è inferiore o uguale a K. Per calcolare la distanza, viene utilizzata una visita in ampiezza BFS che si interrompe quando si visita il vertice destinazione. Nel caso pessimo, ovvero quando la destinazione non è raggiungibile, tali BFS devono visitare l'intero grafo e quindi il costo computazionale risulta essere in $O(n + m)$, con $n = |V|$ e $m = |E|$, assumendo implementazione del grafo tramite liste di adiacenza.

Algorithm 1: ControlloCiclo(Graph $G = (V, E)$, Vertex v_1, Vertex v_2, Nat K) $\rightarrow$ Bool

```
return (Distanza(G, v1, v2)+Distanza(G, v2, v1)) ≤ K

Algorithm Distanza(Graph G = (V, E), Vertex s, Vertex t) → Int
Queue q ← new Queue()
for x ∈ V do
    x.distance ← ∞
s.distance ← 0
q.enqueue(s)
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if w == t then
            return u.distance + 1
        if w.distance == ∞ then
            w.distance ← u.distance + 1
            q.enqueue(w)
return ∞
```

Esercizio. Progettare un algoritmo che dato un grafo orientato $G = (V, E)$ stampa uno qualsiasi dei suoi cicli, se ne esiste almeno uno, altrimenti stampa la stringa `"grafo aciclico"`.

Soluzione. La ricerca di cicli in grafi orientati si può effettuare utilizzando una visita in profondità (DFS) in quanto esiste un ciclo se e solo se durante la DFS si visita un nodo che ha tra le sue adiacenze un nodo già visitato, ma non ancora chiuso (quindi marcato "grey" secondo la terminologia usata a lezione). Il ciclo sarà costituito dai vertici che vanno da tale vertice "grey" al vertice che si sta correntemente visitando. Per poter stampare tale sequenza di vertici si può tenere traccia dell'albero di visita tramite il campo *parent* che indica per ogni vertice visitato il vertice "padre" in tale albero. Per interrompere la DFS al momento del ritrovamento del ciclo si può utilizzare una variabile globale *loop*; tale variabile è inizializzata a $false$ e viene settata a *true* al momento del ritrovamento del ciclo. Ogni chiamata ricorsiva della chiamata in profondità viene fatta solo se non si è già trovato il ciclo, quindi solo se $loop = false$. Al momento del ritrovamento del ciclo, si salvano il vertice iniziale e finale del ciclo in due variabili globali *start* e *end*. Al termine della visita, se si è trovato il ciclo si stampa il ciclo stampando prima il cammino da *start* a *end* (nell'albero di visita) e poi stampando di nuovo il vertice iniziale *start* (per chiudere il ciclo).

Tale soluzione è riportata come Algoritmo 2; il costo computazionale di tale algoritmo, nel caso pessimo, è il medesimo della visita in profondità, ovvero $O(n+m)$, con n numero dei vertici e m numero degli archi, assumendo implementazione del grafo tramite liste di adiacenza.

Algorithm 2: CicloInGrafoOrientato(Graph (V, E))

```
// Inizializzazione variabili globali
Bool loop = false
Vertex start, end
// Inizializzazione marcatura e indicazione dei parent dei vertici
for v ∈ V do
    v.mark ← white ; v.parent ← NULL
// Esecuzione della DFS
for v ∈ V do
    if (not loop) and (v.mark = white) then
        DFSvisit(v)
// Si stampa il ciclo se è stato trovato
if loop then
    printPath(start, end)
    print(start)
else
    print("grafo aciclico")

DFSvisit(Vertex u)
u.mark ← grey
for v ∈ u.adjacents do
    if v.mark = grey then
        loop ← true ; start ← v ; end ← u
    if (not loop) and (v.mark = white) then
        v.parent ← u
        DFSvisit(v)
u.mark ← black

printPath(Vertex s, e)
if s = e then
    print(s)
else
    printPath(s, e.parent)
    print(e)
```

Esercizio. Si consideri il seguente gioco per bambini che si sviluppa in un parco, su un campo di gioco che rappresentiamo tramite un grafo non orientato pesato (V, E, w). I vertici V identificano dei punti sul campo di gioco. In ogni punto si colloca un gruppo di bambini. Un arco $(v_1, v_2) \in E$ identifica la presenza in v_1 di un bambino che può spostarsi verso il punto v_2. Per spostarsi, però, il bambino richiede un numero di caramelle $w(v_1, v_2)$. La maestra dei bambini sceglie un punto iniziale s del campo di gioco, comunica ai bambini nel punto s un messaggio segreto, e gli consegna un sacchetto con K caramelle.

All'inizio del gioco solo i bambini nel punto s conoscono il messaggio segreto. Lo scopo è che tutti i bambini alla fine del gioco conoscano tale messaggio. La comunicazione del messaggio avviene attraverso passaparola, ovvero, un bambino che viene a conoscenza del messaggio nel punto v_1 può andare a comunicarlo ai bambini nel corrispondente punto v_2. Ogni bambino che si sposta da v_1 a v_2 può prendere con sè alcune caramelle, inizialmente presenti nel sacchetto della maestra, passarle poi ai bambini nel punto v_2, tenendone per sè un numero pari a $w(v_1, v_2)$. Lo scopo del gioco è quindi raggiungibile solo se il numero iniziale K di caramelle è sufficiente per permettere ad ogni bambino, che si sposta per il passaparola, di tenere per sè le proprie caramelle.

Scrivere un algoritmo che dato il grafo (V, E, w), il punto di partenza s, ed il numero iniziale di caramelle K, verifica se è possibile raggiungere lo scopo del gioco.

Cont.

Soluzione. Innanzitutto assumiamo che il grafo che rappresenta il campo da gioco sia connesso, altrimenti lo scopo del gioco sarebbe banalmente non raggiungibile. In tal caso, verificare la raggiungibilità dello scopo del gioco coincide con la verifica dell'esistenza di un albero di copertura con costo complessivo inferiore a K. Infatti, per far in modo che il passaparola raggiunga tutti i punti del campo di gioco, è necessario che vengano percorsi dai bambini dei cammini corrispondenti ad archi che ricoprono l'intero grafo. Inoltre, il numero complessivo di caramelle necessarie coinciderà con la somma dei pesi degli archi associati ai cammini effettuati.

L'Algoritmo 3 innanzitutto calcola il costo del minimum spanning tree utilizzando l'algoritmo di Kruskal. Tale costo viene memorizzato nella variabile *tot*. Successivamente, l'algoritmo controlla semplicemente se $tot \leq K$; in tal caso lo scopo del gioco è raggiungibile, altrimenti non lo è. Il costo computazionale dell'algoritmo corrisponde con quello dell'algoritmo di Kruskal, ovvero $T(n, m) = O(m \log n)$ con $n = |V|$ e $m = |E|$.

Algorithm 3: PASSAPAROLA(GRAPH $G = (V, E, w)$, VERTEX s, INT K) $\rightarrow$ BOOLEAN

```
INT tot ← 0
UNIONFIND UF
for each v ∈ V do
    UF.makeSet(v)
SORT(E, w)
for each {u, v} ∈ E do
    Tu ← UF.find(u)
    Tv ← UF.find(v)
    if Tu ≠ Tv then
        tot ← tot + w(u, v)
        UF.union(Tu, Tv)
if tot ≤ K then
    return true
else
    return false
```

Esercizio. Un grafo orientato pesato aciclico $G = (V, E, w)$ rappresenta i possibili sentieri, in un deserto, che possono essere percorsi. I vertici del grafo rappresentano incroci fra sentieri, mentre gli archi rappresentano sentieri di collegamento fra incroci. Bisogna attraversare il deserto partendo dal vertice s e arrivando al vertice t. Si parte con una riserva d'acqua iniziale K. L'attraversamento di un sentiero, rappresentato dall'arco (u, v), modifica le riserve d'acqua della quantità $w(u, v)$: tale quantità è negativa quando la riserva d'acqua diminuisce percorrendo tale sentiero, mentre è positiva quando la riserva d'acqua aumenta grazie alla presenza di piccole fonti presenti lungo il sentiero. Il problema consiste nel capire se esiste un cammino da s a t percorribile con la riserva d'acqua iniziale K, ovvero, durante il cammino le riserve non diventano mai negative. Si noti che anche se il grafo è aciclico, essendo orientato possono comunque esistere diversi cammini per raggiungere la destinazione t.

Bisogna quindi progettare un algoritmo che riceve in input un grafo orientato pesato aciclico $G = (V, E, w)$, i vertici di partenza e arrivo s e t, e la riserva iniziale di acqua K, e che fornisce in output un valore booleano: *true* se esiste un cammino da s a t percorrendo il quale le riserve non diventano mai negative, *false* altrimenti.

Soluzione. Il problema prevede di verificare se esiste un cammino *lecito* dal vertice s al vertice t. Un cammino è lecito se, considerando un contatore inizializzato con K che viene modificato ad ogni arco attraversato aggiungendo il peso di tale arco, abbiamo che tale contatore non diventa mai negativo. Se esiste un cammino lecito che collega due vertici, allora esiste un cammino lecito che massimizza il valore finale del contatore. La soluzione proposta procede calcolando tali valori massimi finali per cammini leciti, utilizzando una variante dell'algoritmo di Bellman-Ford. Usiamo l'array delle distanze $D[1..n]$ per memorizzare l'attuale approssimazione dei valori massimi finali. Inizialmente avremo $D[s] = K$, mentre $D[u] = -\infty$, per ogni altro vertice $u \neq s$ (in quanto non è ancora stato trovato alcun cammino lecito per raggiungere u). Si procede poi effettuando cicli di rilassamenti: per ogni arco (u, v) si esegue

$$newApprox = D[u] + w(u, v)$$
$$\text{if } \big(newApprox \geq 0 \wedge newApprox > D[v]\big) \text{ then } D[v] = newApprox$$

La condizione $newApprox \geq 0$ serve per controllare che il cammino sia lecito, ovvero, il contatore non diventi negativo. Dopo j fasi di rilassamento, vengono scoperti i cammini leciti di lunghezza minore o uguale a j che

Cont.

massimizzano il contatore. Essendo il grafo aciclico, la lunghezza massima per un cammino sarà $|V| - 1$, quindi dovranno essere eseguiti al più $|V| - 1$ cicli di rilassamento. Appena si trova un cammino lecito per raggiungere t, si restituisce immediatamente *true*. Se invece si terminano tutti i rilassamenti senza mai trovare un modo lecito per raggiungere t, si restituisce *false*.

Algorithm 4: AttraversaDeserto(Graph G=(V, E, w), Vertex s, t, Number K) → Boolean

```
// Assumiamo K ≥ 0 e s ≠ t altrimenti il problema sarebbe banale
n ← G.numNodi()
Number D[1..n]
for each u ∈ {1 ... n} do
    D[u] ← −∞
D[s] ← K
for i ← 1 to n − 1 do
    for each (u, v) ∈ E do
        newApprox ← D[u] + w(u, v)
        if (newApprox ≥ 0 ∧ newApprox > D[v]) then
            D[v] ← newApprox
            if v = t then
                return true
return false
```

L'algoritmo 4 riporta lo pseudocodice dell'algoritmo appena descritto. Il caso ottimo per il costo computazionale si verifica quando il primo arco considerato permette di raggiungere in modo lecito t: in questo caso il costo è $\Theta(n)$, ovvero il costo computazionale della inizializzazione dell'array D. Il caso pessimo si ha quando non esiste un cammino lecito: in questo caso si eseguono tutti i cicli, per un costo complessivo $\Theta(n \times m)$, con $n = |V|$ e $m = |E|$, in quanto il ciclo esterno viene eseguito $n - 1$ volte e quello interno m volte.

Esercizio. Progettare un algoritmo che dato un grafo orientato pesato $G = (V, E, w)$ con pesi non negativi (ovvero, $\forall v_1, v_2 \in V.\, w(v_1, v_2) \geq 0$), un insieme di possibili vertici di partenza $S \subseteq V$, un vertice di arrivo $t \in V$, ed un numero non negativo K, verifica se esiste un cammino di peso non superiore a K che va da un qualche vertice iniziale al vertice di arrivo (ovvero, l'algoritmo restituisce *true* se esiste $s \in S$ tale da avere un cammino da s a t con peso minore o uguale a K, e restituisce *false* in caso contrario).

Soluzione. Visto che i pesi sono non negativi, si può procedere ispirandosi all'algoritmo di Dijkstra. La differenza del presente problema rispetto al problema della ricerca del cammino di costo minimo da singola sorgente risolto da Dijkstra, è che in questo caso ci possono essere più possibili sorgenti. Si rende quindi necessario tener conto che tutte queste sorgenti sono a distanza 0 e che tutte devono essere prese in considerazione come possibili vertici di partenza. L'algoritmo MultiSorgente è una versione dell'algoritmo di Dijkstra che semplicemente inserisce inizialmente tutte le possibili sorgenti nella coda con priorità associando ad esse la distanza 0. La visita proseguirà quindi in ordine di distanza non decrescente rispetto ad una qualche possibile sorgente. Appena si arriva ad un vertice a distanza superiore a K, oppure si raggiunge il vertice di destinazione t, l'algoritmo termina. Se invece la visita viene completata, abbiamo la garanzia che la destinazione non risulta essere raggiungibile da alcun nodo sorgente. Il costo computazionale dell'algoritmo MultiSorgenti nel caso pessimo coincide con il costo dell'algoritmo di Dijkstra, ovvero $O(m \log n)$ dove $m = |E|$ e $n = |V|$.

Cont.

Algorithm 5: MultiSorgente(Grafo $G = (V, E, w)$, Set[Vertex] S, Vertex t, Number K) $\rightarrow$ Bool

```
/* inizializzazione strutture dati                                          */
n ← G.numNodi()
Number D[1..n]
for each vertex v ∈ V \ S do
    D[v] ← ∞
MinPriorityQueue[Int, Number] Q ← new MinPriorityQueue[Int, Number]()
for each vertex v ∈ S do
    D[v] ← 0
    Q.insert(v, D[v])
/* esecuzione algoritmo di Dijkstra                                         */
while not Q.isEmpty() do
    u ← Q.findMin()
    if D[u] > K then
        return false
    else if u == t then
        return true
    Q.deleteMin()
    for v ∈ u.adjacent() do
        if D[v] = ∞ then
            /* prima volta che si incontra v                                */
            D[v] ← D[u] + w(u, v)
            Q.insert(v, D[v])
        else if D[u] + w(u, v) < D[v] then
            /* scoperta di un cammino migliore per raggiungere v            */
            Q.decreaseKey(v, D[v] − D[u] − w(u, v))
            D[v] = D[u] + w(u, v)
return false
```

Esercizio. Si consideri un grafo orientato pesato $G = (V, E, w)$ e due sottoinsiemi di vertici S e T (quindi $S, T \subseteq V$). Progettare un algoritmo che, tra tutti i possibili cammini da un vertice dell'insieme S ad un vertice dell'insieme T, stampa quello di costo minimo.

Soluzione. Considerando che dobbiamo confrontare tanti cammini minimi, risulta conveniente l'utilizzo dell'algoritmo di Floyd-Warshall per il calcolo di tutti i cammini minimi tra tutte le coppie di vertici di un grafo orientato pesato. Una volta calcolati tutti i cammini minimi, si controllano tutti i cammini fra coppie di nodi (s, t) con $s \in S$ e $t \in T$, per scegliere il più piccolo fra tutti questi. Una volta trovata la coppia (s, t) con il cammino minore, si procede a stampare il relativo cammino.

Il costo computazionale dell'algoritmo è il medesimo dell'algoritmo di Floyd-Warshall, ovvero $T(n, m) = O(n^3)$, dove n è il numero di vertici ed m il numero di archi nel grafo. Si noti infatti che le parti aggiuntive dell'algoritmo servono per ricercare i vertici *s_min* di inizio e *t_min* di fine del cammino minimo (con costo $O(n^2)$) e per stampare il cammino da *s_min* a *t_min* (con costo $O(n)$). Tali costi aggiuntivi sono inferiori in ordine di grandezza e vengono quindi assorbiti dal costo dell'algoritmo di Floyd-Warshall.

Cont.

Algorithm 6: MinimoCamminoMinimo(Grafo $G = (V, E, w)$, Set[Vertex] S, Set[Vertex] T,)

```
n ← G.numNodi()                                  // esecuzione dell'algoritmo di Floyd-Warshall
Real D[1..n, 1..n]
Int next[1..n, 1..n]
for x ← 1 to n do
    for y ← 1 to n do
        if x = y then
            D[x, y] = 0
            next[x, y] = −1
        else if (x, y) ∈ E then
            D[x, y] = w(x, y)
            next[x, y] = −1
        else
            D[x, y] = ∞
            next[x, y] = −1
for k ← 1 to n do
    for x ← 1 to n do
        for y ← 1 to n do
            if D[x, k] + D[k, y] < D[x, y] then
                D[x, y] = D[x, k] + D[k, y]
                next[x, y] = next[x, k]
Int s_min, t_min, min ← ∞          // ricerca miglior cammino da un nodo in S a un nodo in T
for s ∈ S do
    for t ∈ T do
        if D[s, t] < min then
            min ← D[s, t]
            s_min ← s
            t_min ← t
PrintPath(s_min, t_min, next)                     // stampa del cammino minimo da s_min a t_min

function PrintPath(Int u, v, next[1..n, 1..n])  // stampa cammino per l'algoritmo Floyd-Warshall
if (next[u, v] < 0) then
    errore(u e v non sono connessi)
else
    print u
    while ¬(u = v) do
        u ← next[u, v]
        print u
```

The End.