Algoritmi 2

Professore: Paul Wollan

Esercitatore: Sergio De Agostino

ven 24/02/2017 - Matching

Def. Un insieme $S \subseteq U$ si dice **massimale** o **minimale** rispetto a una proprietà P se S verifica P e $\forall\ S' \supset S$ si ha che S' non soddisfa P.

Def. Un insieme $I \subseteq V$ si dice **indipendente** se $\forall\ v, w \in I \rightarrow \{v, w\} \notin E$, cioè che non esiste un arco da v a w.

Def. Il **grado** di un vertice è il numero di archi incidenti con quel vertice.

Def. Il **vertex cover** di un grafo è definito come $\forall\ \{v, w\} \in E \rightarrow v \in V'$ o $w \in V'$.

Nel vertex cover il problema della massimalità è banale (poichè mi basta prendere tutto V), mentre è molto più complicato il problema della minimalità, cioè quello di trovare il minimum vertex cover.

Algoritmo per l'insieme massimale indipendente

```
1  V = (v_1, ..., v_n)
2  I = {v_1}
3  for (i = 2 to n)
4      if (v_1, v_t) ∉ E   ∀ v_t ∈ I:
5          I = I ∪ {v_i}
```

Complessità: usando il vettore caratteristico, dove metto 0 se il vertice v_i non appartiene all'insieme oppure 1 se ci sta, ottengo $O(|V| + (2|E| - d(v_1))) = O(|V| + |E|) = O(|Is| + |E|)$.

Def. Dato un grafo $G = (V, E)$, un insieme di archi $M \subseteq E$ si dice **matching** se $\forall\ e, e' \in M \rightarrow e \cap e' = \emptyset$, cioè se gli archi non hanno vertici in comune.

Definisco un vettore lungo $2t$, dove $t = |Is| + |E|$, dove Is sono i vertici isolati. Il vettore è fatto in questo modo $[s_0, s_0, s_1, s_1,$(coppie di vertici isolati) $\ldots v, w,$(archi) $\ldots]$.

Algoritmo per il matching massimale

```
1  M = {e_1}                    //parto con un arco a caso
2  for (i = 2 to n):
3      if (e_1) non è incluso in nessun arco di M:
4          M = M ∪ {e_i}
```

Per effettuare il controllo dentro l'if in tempo costante definisco un vettore caratteristico su V, mettendo 1 in posizione i se il vertice v_i è già coperto da un arco, altrimenti 0. Quando vado a controllare se e_i incide con qualche arco in M vado a vedere se uno dei suoi due vertici ha 1 nel vettore caratteristico.

LUN 27/02/2017 - DFS

Dfs

```
DFS(G grafo, u nodo di partenza)
        VIS <- insieme dei nodi visitati, inizialmente vuoto
        S <- stack, inizialmente vuoto
        S.push(u)
        VIS.add(u)
    while (S not empty) do:
                v <- S.top()                          // Legge il nodo in cima allo stack
        if (esiste un adiacente w di v con w non in VIS):
                        VIS.add(w)
                        S.push(w)                     //Inserisce w in cima allo stack
        else
                        S.pop()                       // Rimuove il nodo in cima allo stack
        return VIS
```

Ad ogni iterazione del `while` o viene visitato un nuovo nodo, o ne viene estratto uno dallo stack; quindi il numero di iterazioni sono al più $2n$. Ogni iterazione scansiona gli archi del nodo corrente; in totale, ogni arco viene scansionato due volte, $2m$. Quindi la complessità è $O(n+m)$.

Prop. *La dfs, partendo da un nodo u, visita tutti i nodi raggiungibili da u.*

Dim. Supponiamo per assurdo $\exists z$ raggiungibile da u, ma che non viene visitato. Significa che esiste un cammino $u_0, \ldots, u_k$, con $u_0 = u$ e $u_k = z$. Sia u_i il primo nodo non visitato dalla DFS. Vuol dire che u_{i-1} è stato visitato; ma è impossibile, poichè prima di essere buttato via dallo stack sarebbero dovuti essere visitati tutti gli adiacenti, e u_i e u_{i-1} sono adiacenti per costruzione.

Dfs ricorsiva

```
DFS(G grafo, r radice):
    P = vettore di padri inizializzato a -1
    P[r] = r
    DFS_rec(G,P,r)

DFS_rec(G grafo, P vettore padri, x vertice):
    ∀ u adiacente con x, do:
        if P[u] == -1:            //non è stato ancora visitato
            P[u] = x
            Dfs_rec(G, P, u)
```

Def. Un **albero** è un grafo connesso aciclico; tale che se ha n nodi, ha $n-1$ archi. Ogni nodo è raggiungibile da un solo cammino. Tutti gli archi sono ponti.

Def. L'**albero di visita** di una visita è il sottografo formato da tutti i nodi visitati assieme agli archi che hanno permesso di visitarli.

Si può dimostrare che un grafo non diretto di n nodi è un albero se e solo se ha esattamente $n-1$ archi.
Dobbiamo programmare l esami per n studenti durante un periodo di due settimane, in modo tale che nessuno studente deve fare 2 esami in una stessa settimana. Sol: ovviamente, se esiste uno studente che deve dare 3 esami, allora non esiste soluzione. Supponiamo allora che ogni studente deve dare esattamente 2 esami. Modelliamo un grafo in cui due esami sono collegati se ci sta uno studente che li deve fare tutti e due. Cerchiamo quindi una partizione dei vertici tale che nessun arco ha due termini nella stessa partizione. Cioè stiamo cercando una **colorazione** del grafo. Complessità: O(n +m). Prima troviamo l'albero. Poi, quando scelgo il colore della radice, allora tutti gli altri colori sono determinati. Infine facciamo un controllo su tutti gli archi vedendo se sono tutti ben colorati.

Prop. *Se esiste una tricolorazione di un grafo, raramente è unica.*

Def. Un **pozzo universale** è un nodo x in un grafo diretto tale che $\forall y \; \exists$ un arco (y,x) e $\nexists$ un arco (x,y)

Ovviamente può esistere al massimo un solo pozzo universale in un dato grafo.

Verificare se un grafo ha un pozzo universale e trovarlo

```
Al primo passo scelgo due vertici a caso.
Se non ci sono archi fra i due vertici, li elimino entrambi.
Se esiste un arco (x, y), allora elimino x
Ripeto fino a che non rimango con un nodo solo.
Alla fine verifico se quel nodo è un pozzo universale (cioè se riceve archi da tutti gli altri nodi)
```

Se uso una matrice di adiacenza, la complessità è $O(n)$.

MER 1/03/2017 - DFS 2

Nella dfs inseriamo i due seguenti contatori: $t(v)$, uguale al primo momento in cui incontriamo il nodo v (cioè quando inseriamo v nello stack), e $T(v)$, uguale all'ultimo momento in cui incontriamo il nodo v (cioè quando lo eliminiamo dallo stack). Usiamo come tempo un contatore che aumenta ogni volta che incontriamo un nuovo nodo. Osserviamo che se l'intervallo di un nodo x è $[1, n]$, allora x è la radice. Se è $[k, k]$, allora x è una foglia. Inoltre, non possono esistere due nodi $x_1[a_1, b_1]$ e $x_2[a_2, b_2]$ tali che $a_1 < a_2 < b_1 < b_2$. Definiamo come I_v l'intervallo di v. Se il nostro grafo è diretto, abbiamo che:

- Se ho un arco tale che $I_n \subseteq I_v$, allora ho un arco "all'indietro".
- Se ho un arco tale che $I_v \subseteq I_n$, allora ho un arco "in avanti".
- Se ho un arco tale che $I_n \cap I_v = \emptyset$, allora ho un arco "di attraversamento".

Prop. *Se il grafo non è diretto, non ci sono archi di attraversamento.*

Dim. Vediamo se per assurdo ci fosse un arco di attraversamento. Allora vuol dire che o I_n è stato chiuso prima di I_v, o viceversa. Assumiamo allora che ho chiuso prima n, cioè che $T(n) < t(v)$. Tuttavia, se n è stato chiuso, significa che non ci sono altri archi vicini ancora non visitati. Ma allora l'algoritmo non è stato eseguito correttamente, poichè significa che c'era un arco che non abbiamo visitato che portava a v.

Prop. *Se il grafo non è diretto, esistono solo archi all'indietro.*

Dim. Questo perchè $(u, v) = (v, u)$, e quindi non ha senso distinguere tra i casi $I_n \subseteq I_v$ e $I_v \subseteq I_n$, e quindi per convenzione diciamo che sono tutti archi all'indietro.

Prop. *In un grafo non diretto non esiste un ciclo $\Leftrightarrow$ non esistono archi all'indietro.*

Dim. Parte "solo se": Se non esistono archi all'indietro, allora tutti gli archi fanno parte dell'albero, e non ci sono cicli. Parte "se": Se non esiste un ciclo, allora il grafo è un albero e tutti gli archi fanno parte dell'albero.

Prop. *Dato un grafo diretto, con un ciclo formato dai nodi $c_0 \dots c_k$, dove c_0 è il primo nodo visitato, allora $c_1 \dots c_k$ vengono scoperti dalla dfs prima della chiusura di c_0; cioè $t(c_i) < T(c_0)$*

Dim. Assumiamo per assurdo che esista un c_i con $t(c_i) < T(c_0)$. Ma c_{i-1} è stato scoperto prima della chiusura di c_0. Il fatto che $t(c_0) < t(c_{i-1})$ e che $t(c_{i-1}) < T(c_0)$ implica che $T(c_{i-1}) < T(c_0)$; quindi l'algoritmo non è stato seguito correttamente, poichè avrei potuto esplorare l'arco che va da c_{i-1} a c_i.

Prop. *In un grafo diretto, esiste un ciclo $\Leftrightarrow$ un albero dfs ha archi all'indietro.*

Dim. Parte "solo se": Se esiste un arco all'indietro, allora vuol dire che ho un arco che torna verso la radice, e quindi ho un ciclo. Parte "se": Assumiamo che ho un ciclo, formato da $c_0, c_1 \dots c_t$ dove c_0 è il primo nodo incontrato nella dfs. Per la proposizione precedente $t(c_k) < T(c_0)$, e implica che l'arco (c_k, c_0) è un arco all'indietro.

Def. Si dice **ordine topologico** del grafo un ordine dei vertici del grafo che rispetta l'ordine dato dagli archi.

Prop. *Esiste un ciclo diretto se e solo se non esiste un ordine topologico.*

Prop. *Esiste un algoritmo che dato un grafo diretto, trova o un ordine topologico, o un ciclo diretto.*

Prop. *Dato un grafo diretto tale che ogni nodo ha almeno un arco entrante, oppure ogni nodo ha almeno un arco uscente, esiste un ciclo diretto ed esiste un algoritmo efficiente per trovarlo.*

Prop. *Ordinando i vertici in ordine decrescente rispetto al valore di T otteniamo un ordinamento topologico*

Dim. Questo vale perchè ad ogni sottoalbero, il valore massimo di T è l'ultimo a essere chiuso.

Problema: un processo in fabbrica è diviso in lavorazioni. Tra di loro, ce ne sono alcune che devono esssere completate prima di altre.

Ordine topologico di un grafo

```
1  ORDTOP(G: DAG)
2      L <- lista vuota
3      VIS: array di bool, inizializzato a false
4      for ogni nodo v di G:
5          if not VIS[v]:
6              DFS_ORD(G, v, VIS, L)
7      return L
8
9  DFS_ORD(G: DAG, v: nodo, VIS: array, L: lista)
10     VIS[v] <- true
11     for ogni adiacente w di v:
12         if not VIS[w]:
13             DFS_ORD(G, w, VIS, L)
14     L.add_head(v)
```

La complessità dell'algoritmo è $O(n + m)$.

LUN 06/03/2017 - DFS 3

Per verificare se un arco (x, y) è un ponte, basta rimuoverlo dal grafo e verificare se y è ancora raggiungibile da x.
Se volessimo trovare tutti gli archi di un grafo a forza bruta, ci costerebbe $O(m(n+m))$, poichè dovrei provare a rimuovere ogni singolo arco.
Un algoritmo più efficiente sarebbe eliminare a prescindere gli archi all'indietro di un qualunque albero dfs, dato che fanno parte di un ciclo e non possono essere ponti. La complessità questa volta diventerebbe $O(n^2)$.

Def. Un arco la cui eliminazione disconnette il grafo è detto **ponte** (un arco che non appartiene a nessun ciclo)

Def. Un nodo la cui eliminazione disconnette il grafo è detto **punto di articolazione**

Prop. *Esiste un arco che contiene $xy \Leftrightarrow$ Esiste un arco all'indietro con un solo termine in $V(Tree(Y))$.*

Prop. *In un ciclo, non ci sono ponti. In un albero, tutti gli archi sono ponti. Un arco all'indietro non può essere ponte.*

Prop. *(u, v) è ponte $\Leftrightarrow$ non esistono archi all'indietro tra il sottoalbero di v e il nodo u o gli antenati di u*

.

Prop. *Se il grafo ha almeno 3 nodi, allora almeno uno degli estremi di un ponte è un punto di articolazione. Tuttavia esistono punti di articolazione che non hanno ponti vicino.*

Prop. *Il nodo radice dell'albero dfs è un punto di articolazione $\Leftrightarrow$ la radice ha almeno due figli.*

Prop. *u è un punto di articolazione $\Leftrightarrow$ u ha un figlio v e non ci sono archi all'indietro tra i figli di v e gli antenati di u.*

DFS punti di articolazione in $O(n+m)$

```
tt <- array dei tempi inizializzato a 0
c <- 0 /* Contatore dei nodi visitati */
A <- insieme vuoto /* Insieme dei punti di articolazione */
DFS_Art(G: grafo non diretto, v: nodo, tt: array, c: contatore, A: insieme)
    c <- c + 1
    tt[v] <- c
    back <- c
    children <- 0
    for every w in adjacent(v):
        if tt[w] = 0:
            children <- children + 1
            b <- DFS_ART(G, w, tt, c, A)
            if tt[v] > 1 && b >= tt[v]:
                A.add(v)
            back <- min(back, b)
        else
            back <- min(back, tt[w])
    if tt[v] = 1 && children >= 2:
        A.add(v)
    return back
```

DFS Ponti

```
tt <- array dei tempi inizializzato a 0
c <- 0 /* Contatore dei nodi visitati */
P <- Lista vuota dei padri
DFS_Ponti(G grafo, u nodo, z nodo, tt array della prima visita, c contatore, P lista vuota di ponti):
    // il nodo z è il padre di u
    c = c + 1
    tt(u) = c
    back = c
    for every v in adjacent(u):
        if (tt[v] = 0):
            b = DFS_Ponti(G,v,u,tt,c,P);
            if (b > tt(u)):    //(uv) è un ponte poichè non sono mai tornato più indietro di u
                P.append((u,v))
            back = min(back, b)
        else if (v ≠ z):
            back = min(back, tt[v])
    return back
```

mer 08/03/2017 - Componenti connesse

Def. Una componente **fortemente connessa** di un grafo diretto è un sottografo massimale tale che esiste un cammino orientato tra ogni coppia di nodi a esso appartenenti

Def. Un grafo non diretto è connesso se $\forall x, y \ \exists$ un cammino $x \to y$

Prop. *In un grafo fortemente connesso ogni vertice ha almeno un arco che entra e almeno un arco che esce.*

Prop. *Dato un grafo direttto G e un vertice x, Se $\forall y$ esiste un cammino $x \to y$, e se $\forall y$ esiste un cammino $y \to x$, allora il grafo è fortemente connesso.*

Dim. Dati due vertici a caso u e v, allora esiste un cammino $u \to x$ e uno $x \to v$, quindi uno $u \to v$.

Troviamo un algoritmo che verifica se da x esiste un cammino per $\forall\, y$, e se $\forall y$ esiste un cammino per x.

Verifica se un grafo è fortemente connesso

```
DFS partendo da x.
Se l'albero copre tutto il grafo, allora tutti gli y sono raggiungibili.
Calcolare G^t, la trasposta di G.
DFS partendo da x in G^t.
Se l'albero copre tutto il grafo, allora x è raggiungibile da tutti gli y.
```

La complessità di questo algoritmo è $O(n+m)$, poichè a calcolare la trasposta ci vuole $(n+m)$.

Def. Il **sottografo indotto** da un insieme di vertici X è i sottografo con vertici X e tutti gli archi su i vertici X.

Def. In un grafo indiretto, le **componenti connesse** sono i massimali sottografi indotti che sono connessi.

Def. In un grafo diretto, le **componenti fortemente connesse** sono ancora i sottografi indotti massimali che sono fortemente connessi.

Def. Dato X un insieme di vertici, G/X è il grafo ottenuto dalla contrazione dell'insieme X in un singolo vertice.

Prop. *Dato U_1 e U_2 sottoinsiemi di vertici in un grafo diretto, se $G[U_1]$ e $G[U_2]$ sono fortemente connessi, allora $G[U_1 \cup U_2]$ sono fortemente connessi.*

Prop. *Dato U un sottoinsieme di vertici tale che $G[U]$ sia fortemente connesso, se G/U è fortemente connesso allora G è fortemente connesso.*

Trovare una partizione $\{U_1 \dots U_k\}$ di componenti fortemente connesse

```
Fort(G grafo diretto):
    Troviamo un ciclo C nel grafo
    if (esiste C):
        G = G/C
        v_c = il vertice ottenuto dalla contrazione G/C
        {U_1 ... U_k} = Fort(G/C)
        U'_i = {U_i se v_c ∉ U_I, altrimenti (U_i − v_c) ∪ V(c)} //dove V(C) sono i vertici del ciclo C
        return  {U'_1 ... U'_k}
    else:
        return {{v_1}, ... {v_k}}      //ritorno i singoli vertici
```

Questo algoritmo ha complessità $O(n(n+m))$, poichè al il livello massimo di ricorsione è n, e per trovare un ciclo ci metto $O(n+m)$.

Domanda tipica di esonero: può essere un arco (x, y) in un grafo diretto che è un arco in avanti per un DFS con albero di ricerca T_i partendo da r e un arco all'indietro per un altro DFS con albero T_2 partendo da r? La risposta è si, basta fornire un esempio di un grafo in cui accade questa cosa.

lun 20/03/2017 - Componenti connesse 2

Prop. *In un grafo diretto, due nodi u e v fanno parte della stessa componente fortemente connessa se esiste un cammino $u \to v$ e uno $v \to u$.*

Se $X_1, \ldots, X_k$ sono i componenti del grafo G/C, $X_i' = \{X_i \text{ se } v_c \notin X_i, \text{ altrimenti } (X_i - \{v_c\}) \cup V(C) \text{ se } v_c \in X_i\}$.
Notazione: $C(u)$: vertici della componente fortemente connessa che contiene u. $Tree(u)$: vertici del sottoalbero della DFS che parte da u. *c-radice* : u è una c-radice di una DFS se u è il primo vertice di $C(u)$ visitato nella DFS.

Prop. *Sia T l'albero di visita di una DFS. Allora:*
1) u è c-radice $\to C(u) \subseteq V(Tree(u))$.
2) $u_1, \ldots, u_k$ sono c-radice $\to Tree(u) = C(u_1) \cup \ldots \cup C(u_k)$.

Dim. Per il primo punto, basta notare che $C(u)$ è fortemente connesso, quindi tutti i nodi sono raggiungibili da u e verranno visitati in $Tree(u)$. Per il secondo punto, invece, consideriamo un nodo v che sta in $Tree(u)$ ma che non sta in nessun $C(u_i)$. Allora, la c-radice di $C(v)$, che chiamiamo w, non è in $Tree(u)$. L'unica spiegazione è che w sia un antenato di u, ma allora u e w apparterrebbero alla stessa componente, cosa che va in contraddizione col fatto che u è un c-radice.

Data un modo per determinare se u sia un c-radice, possiamo modificare la DFS per trovare le componenti fortemente connesse.

DFS per trovare le componenti fortemente connesse.

```
S = stack vuoto.
FOR ogni nodo u in G non visitato:
    DFS_SCC(G, u, S)

DFS_SCC(G grafo, u nod, S stack):
    marca u visitato
    S.push(u)
    for every v adiacente a u:
        DFS_SCC(G, v, S)
    if (u è un c-radice):
        creo lista vuota c
        do {
            w = S.pop()
            c.append(w)
        } while (w != u)
        Output c        //c contiene i nodi della componente fort. connessa
```

Prop. *un nodo u non è un c-radice $\Leftrightarrow$ nelle chiamate ricorsive della DFS con radice u viene attaversato un arco (v, w) ad un nodo w già visitato $(t(w) < t(u))$ e la componente di w non è ancora stabilita.*

Dim. *Parte solo se:* se u nodo non è un c-radice, allora nel $C(u)\exists$ un cammino $u \to a$ $\exists$ una prima volta che questo commino porta da $Tree(u)$ e torna ad un antenato di u. Siccome il componente che contiene u non è stato ancora determinato anche quello di z non è stato determinato $\to$ l'arco (v, z) è quello della conclusone (il nodo z è uguale a w).
Parte se: non può esistere un arco (v, w) tale che $t(w) < t(u)$, poichè altrimenti u non sarebbe un c-radice. Inoltre, w non può appartenere ad un altro ramo rispetto a u, altrimenti la DFS avrebbe esplorato l'arco (v, w).

DFS per trovare le componenti fortemente connesse (senza subroutine)

```
CC un array
CC[v] = {0 se v non visitato, -t(v) la prima volta che lo incontro, cc nome del componente che contiene v}
//insomma se cc[v] è positivo vuol dire che ho trovato la componente di v

DFS_SCC(G grafo, u nodo, CC vettore, c contatore nodi, nc contatore componenti, S stack):
    c = c+1
    CC[u] = -c
    S.push(u)
    back = c
    for every v adiacente a u do:
        if (CC[v] = 0):
            b = DFS_SCC(G, v, CC, s, c, nc)
            back = min(back, c)
        else if (CC[u] < 0):
            back = min(back, -CC[v])
    if (back == -CC[u]):        //cioè se back == c
        //cioè non abbiamo mai trovato un arco (v, w) dove CC[v] < 0
        //cioè u è il c-radice di C(u)
        nc = nc + 1
        do {
            w = S.pop()
            CC[w] = nc
        } while (w != u)
    return back
```

Dato che ogni nodo viene inserito una sola volta nello stack, la complessità dell'algoritmo è $O(n + m)$.

MER 22/03/2017 - BFS

Bfs

```
1  BFS(G: grafo, u: nodo)
2      P: vettore dei padri, inizializzato a 0
3      Dist: array delle distanze
4      P[u] <- u /* u è la radice dell'albero della BFS */
5      Dist[u] <- 0
6      Q <- coda vuota
7      Q.enqueue(u) /* Accoda u alla coda */
8      while Q non è vuota :
9          v <- Q.dequeue() /* Preleva il primo nodo della coda */
10         for ogni adiacente w di v do:
11             if (P[w] = 0):
12                 P[w] <- v
13                 Dist[w] <- Dist[v] + 1
14                 Q.enqueue(w) /* Accoda w alla coda */
15     RETURN P, Dist
```

La complessità dell'algoritmo si ottiene sommando il tempo necessario ad inizializzare i vettori `P` e `Dist`, al costo di aggiungere ogni nodo nella coda ($O(n)$), al tempo di eseguire tutti i cicli del `for`, che ci mette ($O(n)$).

Ricostruire percorso dal vettore dei padri

```
1  L = lista vuota
2  L.head(v)       /* Aggiunge v in testa alla lista */
3  while v <> u:
4      v <- P[v]
5      L.head(v)
6  return l        /* Ritorna in L il cammino da u a v */
```

Prop. *Per ogni $k = 0, 1, 2 \ldots$, c'è un passo dell'algoritmo BFS in cui:*
1) ogni nodo v con $d(u, v) \leq k$ è stato visitato.
2) la coda Q contiene esattamente i nodi a distanza k da u. Niente altro.

Dim. La dimostrazione è per induzione su k. Per $k = 0$, Dist[u] $= 0$ ed u è l'unico nodo nella coda. Per ipotesi induttiva, le propietà (a) e (b) valgono al tempo t per k. Dimosriamo per $k + 1$. I nodi a distanza $k + 1$, se ci sono, sono quelli adiacendi a nodi a distanza esattamente k. Nessuno di questi è stato visitato al passo t. Ci sarà un passo in cui tutti i nodi a distanza k sono stati estratti dalla coda, e tutti quelli a $k + 1$ sono stati esaminati. Oral, la coda contiene tutti i nodi a distanza $k + 1$, e tutti gli altri nodi con distanza minore sono stati visitati.

lun 27/03/2017 - Dijkstra

Def. In un grafo pesato, il **cammino minimo** tra due nodi u, v si denota come $\boldsymbol{d_G(u, v)}$

Prop. *In un grafo pesato valgono le seguenti proprietà:*
1) $d_G(u, u) = 0$.
2) $d_G(u, v) \geq 0$. *(non esistono archi con peso negativo)*
3) $d_G(u, v) \leq d_G(u, w) + d_G(w, v)$.

Dijkstra

```
1  DIJKSTRA(G: grafo, u: nodo)
2      Dist: array delle distanze, inizializzato a infinito
3      P: vettore dei padri, inizializzato a 0
4      Dist[u] <- 0
5      P[u] <- u
6      H <- min-heap inizializzato con tutti i nodi e le priorità sono i valori di Dist
7      while H non è vuoto:
8          v <- H.get_min()        /* Preleva il nodo con distanza minima */
9          for ogni adiacente w di v:
10             if Dist[w] > Dist[v] + p(v, w):
11                 Dist[w] <- Dist[v] + p(v, w)
12                 P[w] <- v
13                 H.decrease(w)       /* Aggiorna l'heap a seguito del decremento */
14     RETURN Dist, P
```

L'inizializzazione del min-heap costa $O(n)$. Estrarre dal min-heap costa $O(logn)$, e devo farlo $O(n)$ volte per il `while`. Ogni arco viene esaminato al più due volte, una sola se il grafo è diretto, e ogni volta mi costa $O(logn)$ esaminarlo. Quindi, in totale, $O(nlogn + mlogn) = O((n + m)logn)$.

Prop. *L'algoritmo di Dijkstra termina sempre.*

Dim. Il `while` non può eseguire più di $n - 1$ iterazioni, poichè seleziona solo i nodi di cui non conosciamo la distanza.

Prop. *Per ogni* $h = 0, 1, \ldots k$, $Dist_h$ *è uguale a* $d_G(u, \cdot)$ *su* R_h, *cioè* $\forall x \in R_h, Dist[x] = d_G(u, x)$

Dim. Procediamo per induzione su h. Banalmente è vero per $h = 0$, poichè $R_0 = \{u\}$ e $Dist[u] = 0$. Per hp induttiva, $\forall x \in R_h$, $Dist_h[x] = d_G(u, x)$. Dimostriamo che è vero anche per $h + 1$. Sia (u, w) l'arco scelto nella $h + 1$-esima iterazione. Dimostriamo che $Dist_{h+1}[w] = d_G(v, w)$. Siccome $Dist_{h+1}[w] = Dist_h[v] + p(v, w)$, abbiamo che $Dist_{h+1}[w] \geq d_G(u, w)$. Dimostriamo anche la disugualianza inversa. Se ci fosse stato un cammino minimo C che portava a w attraverso l'arco (y, z) uscente da R_h (cioè $y \in R_h$ e $z \notin R_h$). Chiamiamo C_y la parte di quel cammino che porta a y. Siccome $p(C) \geq p(C_y) + p(y, z)$, sappiamo che $d_G(u, w) = p(C) \geq p(C_y) + p(y, z) = Dist_h[y] + p(y, z) \geq Dist_h[v] + p(v, w) = Dist_{h+1}[w]$.

Per dimostrare un algoritmo greedy, devo inanzitutto dimostrare che l'algoritmo termina in un numero finito di passi. Fatto questo, devo dimostrare per induzione la seguente proposizione: *Per ogni* $h = 0, 1, 2 \ldots m$ *esiste una soluzione ottima* SOL^* *che contiene* SOL_h. Fatto questo, sappiamo che la soluzione finale è estendibile a una soluzione ottima. Spesso, inoltre, occorre trasformare la soluzione ottima che estende la soluzione parziale per ipotesi induttiva, in un'altra soluzione ottima che estende la soluzione parziale dell'iterazione successiva.
Spesso tale trasformazione consiste in una sostituzione di un opportuno elemento della soluzione ottima con uno di quella parziale.

Per dimostrare un algoritmo greedy(2), devo procedere sempre per induzione, con Sol_k = valore di Sol dopo il k-esimo passo, e così via. Poi devo dimostrare che $\forall k$, $\exists$ una soluzione ottimale Sol^* tale che $\text{Sol}^* \geq \text{Sol}_k$. Il caso base è che Sol_0 è contenuto in ogni soluzione ottimale. Passo induttivo: assumiamo che $\exists$ una soluzione ottimale Sol^* che contiene Sol_k per dimostrare che $\exists$ una soluzione ottimale $\text{Sol}^* *$ che contiene Sol_{k+1}

lun 3/04/2017 - Kruskall

Def. Si chiama **spanning tree** di un grafo G un albero che tocca tutti i nodi di G. Se gli archi sono pesati, l'albero di costo minimo prende il nome di **minimum spanning tree**

Kruskall

```
KRUSKAL(G: grafo non diretto, pesato e connesso)
    SOL <- insieme vuoto
    while esiste un arco che può essere aggiunto a SOL senza creare cicli:
        Sia {u, v} un arco di peso minimo fra tutti quelli che possono essere aggiunti a SOL
        SOL <- SOL  {{u, v}}
    return SOL
```

Kruskall efficiente

```
KRUSKAL(G: grafo non diretto, pesato e connesso)
    SOL <- lista vuota
    E <- array contenente gli m archi di G, per ogni arco i: E[i].u, E[i].v, E[i].p
    Ordina l'array E rispetto ai pesi in modo non descrescente
    CC <- array delle componenti
    for ogni nodo u DO
        CC[u] <- u            /* Inizialmente, ogni nodo è una componente a sè stante */
    for i = 1 TO m DO
        {u, v} <- {E[i].u, E[i].v}
        If (CC[u] != CC[v]):  /* Se l'arco non crea cicli, */
            SOL.append({u, v})  /* aggiungilo alla soluzione */
            c <- CC[v]
            for ogni nodo w:     /* Fondi le due componenti */
                if (CC[w] == c):
                    CC[w] <- CC[u]
    return SOL
```

La complessità di questo algoritmo richiede $O(mlogm) = O(mlogn)$. Il `for` sugli archi richiede $O(m + n^2) = O(n^2)$. In totale, $O(mlogn+n^2)$. In realtà, se si usano strutture dati precise come il **merge-find-set**, si può raggiungere complessità $O((n+m)logn)$.

Prop. *Per ogni $h = 0, 1, \dots n-1$ esiste una soluzione ottima SOL^* che estende SOL_h*

Dim. Procediamo per induzione su h. Per $h = 0$ è banalmente vero. Per ipotesi induttiva, esiste una soluzione $SOL*$ che estende SOL_h. Dimostriamo che SOL^* contiene anche SOL_{h+1}. Diciamo che nella $(h+1)$-esima iterazione abbiamo aggiunto l'arco (u, v). Siccome gli archi di SOL^* formano uno spanning tree, aggiungendo l'arco (u, v), si formerebbe un ciclo. Siccome (u, v) non crea cicli in SOL_h, ci deve essere almeno un altro arco (x, y) nel ciclo che non appartiene a SOL_h. Questo altro arco però non crea cicli con SOL_h, poichè SOL^* contiene SOL_h e non ha cicli. Quindi l'arco (x, y) era tra gli archi che potevano essere scelti nella $(h+1)$-esima iterazione. Ma questo significa che $p(u, v) \leq p(x, y)$. Se definiamo $SOL^o = (SOL^* - (x, y)) \cup (u, v)$, notiamo che è una soluzione ottima che estende SOL_{h+1}

Prop. *La soluzione finale prodotta dall'algoritmo è ottima*

Dim. Siccome sappiamo che la soluzione finale dell'algoritmo è contenuta in una ottima, e sappiamo che entrambe hanno lo stesso numero di archi $(n-1)$, allora vuol dire che sono uguali.

Kruskall aggiungi un nodo in O(V) (on-line)

```
Quando aggiungo un arco a uno spanning tree, formo un ciclo.
Per ottenere il MST, devo levare l'arco di peso maggiore da quel ciclo.
```

mer 05/04/2017 - Prim

Prim

```
PRIM(G: grafo non diretto, pesato e connesso)
    SOL <- insieme vuoto
    Scegli un nodo s di G
    C <- {s}
    while C   V:  /* Finché 'lalbero non copre tutti i nodi di G */
        Sia {u, v} un arco di peso minimo fra tutti quelli con u in C e v non in C
        SOL <- SOL   {{u, v}}
        C <- C   {v}
    return SOL
```

Prim efficiente

```
P = 0   //vettore di padri inizializzato a 0
P[s] = s
H = inf //min heap di nodi inizializzato a 1 per s e a inf per tutti gli altri
while H non è vuoto do:
    u = getmin(H)
    for ogni adiacente v di u do:
        if v ∈ H && p(u,v) < H.costo(v) then:
            P[v] = u
            H.decrease(v, p(u,v)) //sarebbe H.set in realtà
return P
```

Complessità, $O((n+m)(logn))$. Ora dimostriamo la correttezza.

Dim. Supponiamo S_k la soluzione parziale al k-esimo ciclo del `while`. $S_1 = \{s\}$. S_k è un albero con k vertici. Dobbiamo dimostrare che per ogni k, S_k è contenuto dentro una soluzione ottimale S^*. S_1 è contenuto dentro qualsiasi soluzione ottimale. Dimostriamo per induzione, assumendo che $S_k \in S^*$ per dimostrare che esiste una soluzione ottimale S^{**} che contiene S_{k+1}. Notiamo che $S_{k+1} - S_k$ è un arco f e pssiamo assumere che $f \notin E(S^*)$. Allora $S^* + f$ ha un unico ciclo C. Siccome C ha un altro cammino da v ai vertici di s_k, esiste un altro arco f' di C che ha un estremo in S_k e l'altro estremo fuori di S_{k+1}. Ma dato che l'algoritmo ha scelto f come l'arco di peso minore che esce da S_k, allora sappiamo che $p(f') \geq p(f)$. Quindi, $S^{**} = (S^* - f') + f$ è un albero di copertura con $p(S^{**}) \leq p(S^*)$. Se S^* è ottimale, allora $p(S^{**}) = p(S^*)$, e quindi S^{**} è una soluzione ottimale che contiene S_{k+1}.

Ci sono molti problemi per cui l'approccio greedy non funziona perfettamente, come per esempio il **Minimum Vertex Cover**, cioè un sottoinsieme di vertici $S \in V(G)$, tale che ogni arco del grafo ha almeno un estremo in S, tale che S sia più piccolo possibile. Infatti, il minimum vertex cover è NP-completo, quindi non esiste nessun algoritmo efficiente.

Un matching, invece, è un insieme di archi disgiunti di un grafo (cioè non hanno estremi in comune). Il **Maximum Matching** è un matching al quale non posso aggiungere nessun arco. Attenzione: non corrisponde a un matching con il massimo numero di archi. Posso trovare un matching massimale con un algoritmo greedy: basta che continuo a provare ad aggiungere archi fino a che non posso più.

Prop. *Siccome tutti gli archi del grafo devono intersecare con un arco nel matching massimale, allora il sottoinsieme dei vertici coperto dai vertici del matching è un vertex cover.*

Prop. *Inoltre, il minimo numero di vertici che deve avere un vertex cover è al minimo il numero di archi nel matching massimale (deve prendere almeno uno degli estremi), e al massimo il doppio degli archi del matching massimale (tutti e due gli estremi di ogni arco) .*

ESERCITAZIONE

Ordinamento topologico di un DAG

```
L: lista vuota di nodi

finchè ci sono nodi non chiusi:
    visita(v);

visita(nodo v) {
    se v è aperto, return;
    se v non è chiuso:
        apri v;
        per ogni nodo adiacente w di v:
            visita(w);
        chiudi v;
        aggiungi v all'inizio di L;
}
```

VIDEOLEZIONI

Prop. *Gli archi che portano ad un nodo non visitato nella dfs formano un albero*

Dim. Per assurdo: se ci fosse un ciclo, allora visiterei due volte lo stesso nodo, ma per definizione nella dfs un nodo viene visitato solo una volta.

Prop. *Gli archi del grafo non appartenenti all'albero dfs sono archi all'indietro che collegano un nodo con il suo antenato.*

Dim. Supponiamo per assurdo ci siano due nodi x, y collegati da (x, y), e z sia antenato di entrambi. Gli archi (x, z) e (y, z) non appartengono all'albero dfs. Ma è assurdo, poichè la dfs non sarebbe terminata una volta visitato x.

Def. Un grafo si dice **bipartito** se si possono dividere i suoi nodi in due insiemi in modo che tra nodi dello stesso insieme non ci siano archi.

Def. Un grafo si dice **2-colorabile** se è possibile colorarne i nodi con soli due colori in modo che $\forall(u, v)$, u e v hanno colori distinti.

Def. Un grafo è bipartito $\Leftrightarrow$ è 2-colorabile.

ven 03/03/2017 - Stringhe

Def. Con la notazione A^* si indicano tutte le stringhe finite a caratteri nell'alfabeto A (anche quella vuota).

Def. Un dato algoritmo $g : A^* \to B^*$ è **computabile on-line** su una stringa in input S se esiste un algoritmo AL tale che AL legge S da sinistra a destra, e che quando AL legge l'i-esimo carattere di S può accedere solo all'informazione fornita dal suffisso di lunghezza k del prefisso di lunghezza i di S. (gli ultimi k caratteri precedenti)

Def. Un algoritmo on-line è un algoritmo in **tempo reale** se ha complessità lineare e impiega un tempo costante su ogni carattere letto.

Per esempio, un automa a stati finiti è un tipo particolare di algoritmo on-line in tempo reale. Un automa si denota come una sestupla $(A, B, Q, \delta, q_0, F)$. q_0 è lo stato iniziale. Q sono tutti gli stati. A è l'alfabeto di input. B è l'alfabeto di output. $\delta : Q \times A \to Q \times B^*$ è la funzione di transizione. F è l'insieme degli stati finali ammessi.

Def. Dato $S \in A^*$, e $S' \in A^*$, ho che $SS' \in A^*$ e che $(SS')S'' = S(S'S'')$. Infatti, A^* è un monoide (ha elemento neutro λ, cioè la stringa vuota). Inoltre, A^+ è definito come $A^+ = A^* - \{\delta\}$.

Una stringa S è fattorizzabile in $S = s_1|\dots|s_n$ dove $s_i \in A$.

Def. Il sottoinsieme proprio $D \subset A^*$ è un **dizionario** di fattori ristretto ad A^* se $D = A \cup \{f_i | f_i \in A^* \forall 1 \leq i \leq k\}$ tale che $|D| = k + |A|$

Date due stringhe F, S, mi chiedo se F è un fattore di S. $F = f_1 \dots f_m$, con $f_i \in A$. $|F| = m$. L'algoritmo banale, cioè quello in cui per ogni carattere di S provo a metterci F, ha complessità $O(nm)$. Voglio invece complessità lineare. Assumendo che F sia fissato e non cambi, allora devo calcolare il più lungo suffisso di $f_1 f_2 \dots f_i x$ che è prefisso di F, e lo devo fare $\forall i \forall x$, dove x è un qualsiasi carattere dell'alfabeto A. Facendolo a forza bruta, la complessità di questo precalcolo è $O(m^3|A|)$. Ma è un precalcolo che devo fare una volta sola, e quindi sopportabile. Siccome poi la risposta è solo positiva o negativa (o F è fattore oppure no), quindi non ho bisogno di un alfabeto di output. Inoltre, l'automa a stati finiti corrispondente all'algoritmo, è formalizzabile attraverso un grafo orientato. Per rappresentare questo grafo, uso una matrice di adiacenza, dove il numero di righe è la cardinalità dell'alfabeto, mentre le colonne sono la lunghezza di F.

ven 10/3/2017 - Stringhe 2

manca la prima ora della lezione del 10/3/2017, la parte finale sulla ricerca delle strighe

Def. Un **DAG** è un directed acyclic graph, cioè un albero.

zip lavora con un dizionario che è una finestra che scorre sulla stringa e si modifica mentre si legge. Inoltre, siccome non sa quali simboli dovrà usare, parte all'inizio con solo i 256 ascii, e poi in caso ne aggiunge. Ma noi assumiamo che il dizionario con cui partiamo abbia i fattori iniziali giusti .

Stringa in input $S \in A^*$, con dizionario $D = A \cup \{f_i \in A^+ | 1 \leq i \leq k\}$. Vogliamo trovare la fattorizzazione $S = s_1 \dots S_n$ on-line. Accade spesso che negli algoritmi on-line devo accontentarmi di un algoritmo greedy, che non è ottimale.

Regole per i dizionari per gli algoritmi on-line: Un dizionario D è prefisso (suffisso) se ogni prefisso (suffis so) di un fattore in D è un fattore in D.

ven 17/03/2017 - Dizionari

In realtà ci sta un modo per calcolare greedy on-line la fattorizzazione minima con un dizionario prefisso: in pratica io prendo il fattore più lungo possibile, e poi vedo quale prefisso del primo prendere per trovare un secondo fattore che può essere concatenato affinchè la loro unione sia più lunga possibile.

pseudocodice

```
1 j = 0; i = 0;
2 ripeti fino a fine stringa:
3     per k = i + 1 a j + 1:
4         sia h(k) tale che S_k ... S_h(k) è il più lungo fattore in posizione k
5         sia k' tale che h(k') è massimo.
6         S_j ... S_k'-1 è selezionato come fattore e mandato in output
7         j = k'; i = h(k');
```

Siccome per questo algoritmo devo controllare tutte le posizioni, allora la complessità è $O(nL)$, dove L è la lunghezza massima del fattore. Infatti in realtà in questo algoritmo devo trovare il fattore più lungo per ogni posizione della stringa.

Invece, se avevo un dizionario suffisso, la complessità è semplicemente $O(n)$.

Abbiamo un trie che rappresenta un dizionario (qualunque), e vogliamo calcolare la fattorizzazione greedy on-line con una macchina a stati finiti.
In pratica è un on-line senza buffer. Per fare questo modifichiamo la struttura del dizionario: lo facciamo diventare un grafo orientato che rappresenta una macchina a stati finiti (cioè devo fare un preprocessing).
Automa definito così: $M = (A, B, Q, \delta, q_0, F)$. A è l'alfabeto della stringa in input. B è $\{0, 1\}$, perchè una fattorizzazione è rappresentata da un 1 all'inizio di ogni fattore, e poi tutti 0. Q è l'insieme dei nodi della trie del dizionario. q_0 è la radice del trie. F è uguale a Q, poichè tutti gli input sono ammissibili (cioè ogni stringa è fattorizzabile dal dizionario).
Prendo una stringa fa. Prendo una fattorizzazione $g_1 \dots g_k$ della stringa fa. Ora, sia che f sia un fattore del dizionario oppure no, devo andare in un nodo w tale che $g_1 \dots g_j$ tale che j è l'indice più piccolo tale che $g_{i+1} \dots g_k$ è rappresentato da un nodo w dell'albero.
Si manda in output una stringa di bit che corrisponde alla fattorizzazione $g_1 \dots g_j, g_{j+1} g_k$.
In questo modo vado sempre avanti senza mai tornare indietro (così soddisfo la condizione della macchina a stati finiti, cioè non avere nessun buffer). Ovviamente, c'è il caso speciale della fine della stringa: fa, se a è il carattere EOF, allora in output si manda la fattorizzazione di f.
Tutto questo però se ho un dizionario qualunque; se invece ho un dizionario prefisso, avrò tutti 1 in ogni nodo della trie, e quindi tornerò sempre alla radice, non dovrò mai andare a un nodo strano w.

Per rappresentare una fattorizzazione, ogni fattore viene rappresentato come un puntatore a un fattore nel dizionario, quindi per ogni fattore servono $log(|D|)$, dove $|D|$ è la cardinalità del dizionario. Inoltre, sappiamo che in un dizionario $D = A \cup \{f_i \in A^+ \ 1 \leq i \leq k\}$, la cardinalità è $|D| = |A| + k$. Di solito, questi dizionari hanno 2^16 bit, e quindi ogni fattore richiede 16 bit per essere rappresentato. Facendo così, ho usato **codici da lunghezza variabile a lunghezza fissa** (poichè i fattori sono lunghi diversamente, e ognuno usa 16 bit per essere rappresentato).
La codifica di Huffman standard, invece, non usa fattorizzazione. Usa il dizionario banale che ha solo i carattere dell'alfabeto (tutti fattori di lunghezza 1). Lui diceva che se un carattere ha probabilità p, allora da $log(1/p)$ informazioni. Cioè se un carattere è molto frequente, allora devo rappresentarlo con pochi bit.
Questa cosa la posso estendere usando un dizionario $D = A^k$, cioè comprendente tutte le stringhe lunghe k formate da caratteri in A (in pratica usa un alfabeto formato da parole lunghe k), e usarelo stesso principio di meno informazioni per le parole più frequenti. Questo metodo si chiama **codifiche da lunghezza fissa a lunghezza variabile**.
Alcuni programmi, come `gzip`, usano un alfabeto $D = A \cup \{f_i \ 1 \leq i \leq k\}$, cioè un alfabeto formato da tutte le parole possibili. Questo metodo si chiama **codifica da lunghezza variabile a lunghezza variabile**.

Parte 2

Divide et Impera

La tecnica Divide et Impera tenta di spezzre l'istanza di un problema in istanze più piccole. Gli algoritmi che usano questa tecnica si prestano naturalmente ad essere implementati in modo ricorsivo. Per dimostrare la complessità di un algoritmo ricorsivo, spesso si può usare il **Master Theorem**:

Prop. *Data una relazione di ricorrenza* $T(n) = aT(n/b) + f(n)$ *con* $a \geq 1$ *e* $b > 1$*, se* $f(n) = O(n^c)$*, allora* $T(n) = O(n^{max(log_b a, c)})$*. Se invece* $f(n) = O(n^{log_b a} log^k n)$*, con* $k \geq 0$*, allora* $T(n) = O(n^{log_b a} log^{k+1} n)$*.*

Ad esempio, per il Merge Sort, dove la relazione di ricorrenza è $T(n) = 2T(n/2) + O(1)$, la complessità totale è $T(n) = O(nlogn)$.

Problemi famosi

- *Problemi di ordinamento come il quicksort e il mergesort*

- *Il problema del massimo sottovettore:*
Devo trovare il sottovettore di somma massima di un vettore lungo n con valori positivi e negativi. Una soluzione banale brute-force impiega $O(n^2)$. Invece noi spezziamo il problema: dividiamo il vettore in due (spezzandolo a metà), e calcoliamo il sottovettore di somma massima nella metà sinistra, quello nella metà destra, e quello a cavallo della spezzatura. Quello a cavallo della spezzatura lo calcolo trovando il prefisso massimo e il suffisso massimo che partono e finiscono nella spezzatura. Facendo così ottengo la relazione di ricorrenza $T(n) = 2T(n/2) + O(n)$, che per il Master Theorem $T(n) = O(nlogn)$.

- *Il problema della coppia di punti più vicini:*
Questo è decisamente più complicato. Abbiamo n punti distribuiti nel piano, e vogliamo trovare la coppia di punti più vicina tra loro. Una soluzione banale brute-force che esplora tutte le possibili coppie va in $O(n^2)$. Allora quello che faccio io è ordinare i punti in base alla loro coordinata x, e tracciare una linea verticale in corrispondenza del punto a metà (il punto $n/2$). Ora, se $n \leq 3$ possiamo terminare qui. Altrimenti, facciamo una ricorsione e troviamo la distanza minima tra le coppie nella parte sinistra e tra le coppie della parte destra. Ora chiamiamo δ il minimo tra queste due distanze. Tuttavia adesso ci manca da controllare le coppie di punti che stanno a cavallo della nostra linea. Queste le troviamo in poco tempo sapendo che la coppia che stiamo cercando noi non puo' essere più lontana di δ. Infatti consideriamo i punti che sono lontani al massimo δ dalla nostra linea verticale. Presi quei punti, li ordiniamo in base alla loro coordinata y, e ora consideriamo semplicemente le coppie (i, j) tali che $j > i$ e che $y_j \leq y_i + \delta$. In pratica stiamo considerando tutti i punti in un rettangolo alto δ e largo 2δ. In questo rettangolo si puo' dimostrare che ci sono al massimo 8 punti, e quindi, per ogni punto vicino alla linea verticale, devo esaminare al più 7 coppie (così facendo ho una complessità lineare nella ricerca delle coppie a cavallo della linea centrale). In sostanza, la complessità totale è data dalla relazione di ricorrenza $T(n) = 2T(n/2) + O(nlogn)$, che con il Master Theorem diventa $T(n) = O(log^2 n)$. Possiamo ottimizzare l'algoritmo imponendo che le chiamate ricorsive della parte sinistra e destra ritornano i punti ordinati in base alla coordinata y e quindi per le coppie a cavallo basta unire le due parti come nel mergesort, ottenendo complessità $T(n) = 2T(n/2) + O(n) = O(nlogn)$.

- *Il problema della mediana:*
Calcolare la mediana di una sequenza di n numeri. Una soluzione banale sarebbe ordinarli e prendere l'elemento a metà, con $O(nlogn)$. Tuttavia, possiamo fare di meglio. Inanzitutto notiamo che è un caso particolare del problema di trovare il k-esimo elemento in una sequenza non ordinata, con $k = n/2$. Risolviamo questo allora. Facciamo lo stesso approccio del quick sort: scegliamo un perno a caso e dividiamo l'array in tre (invece che due) parti, quelli più piccoli del perno, quelli uguali al perno, e quelli più grandi. La ricorsione la impostiamo in questo modo: se l'array dei più piccoli $A_<$ è lungo almeno k, allora cerco il k-esimo nell'array dei più piccoli. Altrimenti, se l'array dei più piccoli $A_<$ sommato a l'array degli uguali $A_=$ è più lungo di k, ritorno l'elemento perno. Infine, se non si verifica nessuna delle condizioni precedenti, cerco nell'array dei più grandi $A_>$, e impongo che $k = len(A_>) - (n - k)$. Ora si dimostra con metodi probabilistici che nel caso medio la complessità è lineare. (Ma nel caso peggiore che scelgo come perno l'elemento più piccolo o più grande allora la complessità diventa quadratica).

Programmazione Dinamica

La programmazione dinamica è un metodo che risolve un dato problema partendo dalle soluzioni dei problemi più piccoli dello stesso tipo del problema originale. Presenta alcune somiglianze con la tecnica Divide et Impera, ma la differenza fondamentale è che mentre nel Divide et Impera i sottoproblemi analizzati sono generalmente disgiunti, nella programmazione dinamica i sottoproblemi si sovrappongono ampiamente. L'analisi della complessità è molto semplice (infatti deriva proprio dalla definizione dell'algoritmo e della tabella di memoizzazione). Tuttavia, è spesso complicato trovare l'algoritmo corretto.

Problemi famosi

- *Il problema del file:*
Dati n file di vario peso e un disco di capacità c, trovare un sottoinsieme dei file tale che il disco venga riempito al massimo. Definizione PD: impongo che T[k][c] = massimo spazio usabile dai primi k file su un disco di capacità c. In questo modo, posso anche ricostruirmi quali file devo avere per la soluzione ottima. Se non mi serve cio', posso usare anche solo un array lungo c, ma la complessità rimane sempre $O(nc)$.

- *Il problema del resto:*
Dato un resto r da dare al cliente, e un numero illimitato di banconote di n tagli diversi, calcolare il numero minimo di banconote necessarie per dare il resto esatto. (Attenzione! Non è greedy!). Impostiamo la definizione PD: M[k][r] = minimo numero di banconote, con tagli presi tra i primi k, per formare il resto r. La complessità è proporzionale alle dimensioni della tabella, quindi $O(nr)$.

- *Il problema dello zaino:*
Dati n oggetti ognuno con un certo costo e un certo peso, e dato un limite C, trovare un sottoinsieme degli ogetti la cui somma dei pesi non supera C e massimizza il valore totale. (Assumiamo che tutti i pesi siano positivi). Definizione PD: Z[k][c] = massimo valore ottenibile dai primi k oggetti per uno zaino di capacità c.

Riduzioni a problemi sui grafi: alcuni problemi come *resto* possono essere ridotti a un grafo e risolti con una semplice BFS. Non solo la complessità rimane uguale, ma possiamo anche risparmiarci di costruire il grafo vero e proprio la memoria richiesta dall'applicazione è $O(r)$, cioè meno della programmazione dinamica che usa tutta la tabella (ma la dp che usa solo la riga no). Inoltre, dall'albero di visita della BFS, mi posso facilmente ricalcolare la soluzione ottima. Altri problemi, invece, come *zaino*, richiedono di trovare il cammino massimo; questa volta una semplice BFS non basta, la faccenda è più complicata (inoltre, il problema è risolvibile solo se il grafo è un DAG).

- *Longest Path in DAG:*
Dato un DAG pesato e una coppia di nodi u, v, vogliamo trovare il percorso di peso massimo da u a v. Per fare cio' ordiniamo topologicamente i nodi, in modo tale che per ogni arco (i, j), vale che $j > i$. Ora siano h e k le nuove numerazioni di u, v, con $h \leq k$. Sappiamo che un percorso da h a k puo' passare solo attraverso i nodi $h, h+1, \ldots k-1, k$. Allora per ognuno di questi nodi $h+i$ possiamo calcolarci il peso massimo di un cammino da h a $h+i$. Per la DP usiamo la tabella M[i] = peso massimo di un cammino da h a $h+i$. La complessità è $O(n+m)$, dato che per l'ordinamento topologico ci vuole $O(n+m)$ e per riempire la tabella devo esplorare tutti i nodi e tutti gli archi una sola volta. Il problema dello zaino si puo' ridurre a questo, creando un nodo per ogni coppia di capacità/valore (ogni nodo per ogni entrata della tabella Z), e un nodo speciale z rappresentante la chiusura dello zaino (ogni nodo ha un arco di peso 0 che va alla chiusura), e da ogni nodo escono al massimo tre archi (uno per prendere l'oggetto k, uno per scartarlo, uno per chiudere lo zaino), ma spesso non è vantaggioso, in quanto il grafo in questione diventerebbe enorme sulla memoria ($nC + 1$ nodi), mentre magari una tabella ci entrerebbe giusta giusta.

- *Il problema del cammino critico:*
In una certa azienda devono essere svolte delle varie attività, e ci sono delle dipendeze (i, j) che stanno a significare che l'attività i deve venire svolta prima di j. Due attività non dipendenti fra loro possono essere anche eseguite contemporaneamente. Vogliamo sapere quali sono i tempi di inizio di ogni attività e il tempo di completamento del progetto. Per risolverlo, creiamo un DAG pesato in cui ci sono n nodi, ognuno rappresentante un'attività, e il nodo 0 che corrisponde all'inizio del progetto (il nodo 0 ha un arco di peso 0 per ogni altro nodo). Un arco da i a j indica che esiste una dipendenza tra le attività i, j e il peso dell'arco è il tempo di esecuzione t_i dell'attività i. Fatto questo grafo, devo semplicemente calcolare il longest path partendo dal nodo 0. Se indico con M[i] il cammino massimo dal nodo 0 al nodo i, allora la soluzione del problema sarà $T = max\{M[i] + t_i \mid i = 1 \ldots n\}$.

- *Il problema del sistema con vincoli di differenza*
Se nelle dipendenze dell'attività imponiamo che ci siano vincoli aggiuntivi della forma $x_i - x_j \leq b$, dove x_i è il tempo d'inizio dell'attività i, e dove b puo' essere positivo, negativo oppure nullo (a indicare che l'attività j deve essere eseguita solo dopo b, al massimo entro b, oppure contemporaneamente a i). Per trovare una soluzione a un sistema di vincoli di differenza, possiamo costruire un grafo in cui per ogni vincolo c'è un arco (quindi gli archi possono avere anche peso negativo). Non è detto che venga un DAG (ci possono essere dei cicli). Se tuttavia c'è un ciclo di peso negativo, allora il sistema non ha soluzione. La soluzione (cioè i tempi di inizio delle attività) la troviamo dai cammini minimi M[1] ... M[n] dal nodo 0 ai vari nodi $1 \ldots n$. Quindi basta risolvere il problema dei cammini minimi in un grafo pesato con pesi negativi. In realtà, questi tipi di problemi sono una generalizzazione del problema del cammino critico precedente in cui b è sempre negativo. Infatti, se cerco il longest path in DAG con tutti pesi negativi ottengo un cammino minimo invece che massimo.

- *Il problema del viaggio*
Viaggio in auto con n tappe, con auto che ha c litri di serbatoio; ogni tappa ha un benzinaio con un certo costo della benzina al litro. Ora, lo possiamo rappresentare come un grafo con nC nodi e risolverlo con Dijkstra, ma ci metto $O(nC^2 log(nC))$, poichè il numero degli archi è almeno nC^2. In realtà posso vedere che il fattore $log(nC)$ puo' essere risparmiato in quanto il grafo è un DAG è usare la dinamica come nel longest path ma usando il minimo al posto del massimo, usando V[i][q] = minimo costo di un viaggio che parte dalla località i con q litri e arriva a destinazione. Siccome la tabella ha $O(nC)$ elementi, la complessità è $O(nC^2)$. Posso ridurre ancora la complessità, fino a $O(nC)$, considerando che ogni nodo ha solo due archi: uno per andare alla prossima tappa, uno per fare solo un litro di benzina (invece che scegliere fra C litri). Quindi il numero di archi è nC, e quindi la complessità è $O(nC)$.

Grafi cose

Cammini minimi in presenza di pesi anche negativi: Attenzione: se da un certo nodo sono raggiungibili cicli di peso negativo, allora non ha senso cercare cammini minimi. Usiamo la programmazione dinamica e definiamo la tabella così: M[k][v] = peso di un cammino di lunghezza al più k dal nodo sorgente alla destinazione (se non esiste un cammino lungo k allora setto a infinito) (attenzione: come lunghezza si intende il numero di archi, non il peso. Il peso deve essere minimo). Usando s come sorgente e v come destinazione, setto M[0][s] = 0 e M[0][v] = ∞. La ricorsione la impostiamo come M[k][v] = $min\{M[k-1][v], min\{M[k-1][u]+p(u,v) \mid (u,v) \in E\}\}$. Questa regola ricorsiva vale per qualunque k, anche superiore a n. Inoltre, notiamo che $M[n][v] \neq M[n-1][v]$ se e solo se esiste un ciclo di peso negativo raggiungibile da s.

Bellmann Ford

```
1  BELLMAN_FORD(G: grafo diretto e pesato, s: nodo):
2      M = tabella (n+1)xn
3      for (every node u) M[0][u] = ∞;
4      M[0][s] = 0
5      for (k = 1 TO n) {
6          for (every node v) {
7              M[k][v] = M[k-1][v]
8              for (every arco (u, v) entrante in v) {
9                  if (M[k-1][u] + p(u, v) < M[k][v]) {
10                     M[k][v] = M[k-1][u] + p(u, v)
11                 }
12             }
13             if (k = n AND M[k][v] < M[k - 1][v]) {
14                 return "Ci sono cicli negativi"
15             }
16         }
17     }
18     return riga n di M
```

L'inizializzazione della tabella prende $O(n^2)$. Il calcolo di ogni riga di M esamina al più una volta ongi arco del grafo, quindi $O(n+m)$, in totale la complessità è $O(n(n+m)) = O(nm)$. Per mantenersi i cammini minimi basta tenersi un vettore dei padri. Inoltre, non è necessaria mantenersi tutta la tabella in memoria ma solo le ultime due righe, quindi invece di usare memoria $O(n^2)$ uso memoria $O(n)$.

Problemi famosi

- *Il problema dei cammini*

Dato un grafo diretto con pesi anche negativi, vogliamo contare il numero di cammini minimi da un nodo u a un nodo v. Definizione della tabella PD: N[k][x] numero di cammini minimi da u a x di peso M[k][x] e di lunghezza al più k, dove M è la tabella otteuta con Bellmann-Ford. La soluzione del problema starà in N[n-1][v].

Cammini minimi tra tutte le coppie di nodi: Gli algoritmi che conosciamo sono tutti inefficienti per il calcolo di tutti i cammini minimi da un nodo a un altro, infatti Dijkstra ci metterebbe $O(n(n+m)logn)$ (assumendo solo pesi positivi). Bellmann Ford, invece, impiega $O(n^2m)$. Tuttavia, anche qui possiamo usare la dinamica per risparmiare molti conti. Allora imposto la mia tabella così: F[i][j][k] = minimo peso di un cammiino da i a j con nodi intermedi in $1 \dots k$, e se non ci sta nessun cammino di questo genere, allora setto a infinito. La relazione di ricorrenza è F[i][j][k + 1] = $min\{F[i][j][k], F[i][k+1][k] + F[k+1][j][k]\}$, poichè per andare da i a j posso passare o non passare per il nodo $k+1$. Nel primo caso, il cammino è F[i][j][k]. Nel secondo, è la somma tra il percorso per arrivare da i a $k+1$ e da $k+1$ a j.

Floyd-Warshall

```
1  FLOYD_WARSHALL (grafo diretto e pesato):
2      F: tabella nxnx(n+1) inizializzata a ∞
3      for (i = 1 TO n) F[i][i][0] = 0
4      for (every edge (i, j)) F[i][j][0] = p(i, j)
5      for (k = 1 TO n) {
6          for (i = 1 TO n) {
7              for (j = 1 TO n) {
8                  F[i][j][k] = F[i][j][k - 1]
9                  if (F[i][k][k - 1] + F[k][j][k - 1] < F[i][j][k]) {
10                     F[i][j][k] = F[i][k][k - 1] + F[k][j][k - 1]
11                 }
12             }
13         }
14     }
15     return F[·][·][n]
```

La complessità dell'algoritmo è banalmente $O(n^3)$. Anche la memoria è $O(n^3)$, ma può essere riscritto usando una tabella di dimensioni $O(n^2)$. Se non ci sono cicli negativi, da questo algoritmo risulterà che per ogni i, F[i][i] = 0. Altrimenti, se ci sono cicli negativi, accadrà che per qualche i F[i][i] < 0.

Sequenze

Problemi famosi

- *Il problema del massimo sottovettore*
Dato un vettore A lungo n formato da interi sia positivi che negativi, voglio trovare il massimo sottovettore in $O(n)$. Semplicemente imposto la tabella M[i] = massima somma di un sottovettore di A che termina in i. A questo punto, la relazione di ricorrenza è M[i] = $max\{A[i], A[i] + M[i-1]\}$. Complessità $O(n)$, memoria si arriva anche a $O(1)$.

- *Il problema dei prezzi*
Dato un vettore P di n interi positivi in cui P[i] rappresenta il prezzo di una certa merce nel giorno i, vogliamo sapere qual è il giorno i in cui conviene comprare e il giorno j in cui conviene vendere. Nella tabella mi salvo M[k] = minimo prezzo del prefisso P$[1 \dots k-1]$. In questo modo, mi tengo il minimo prezzo incontrato finora. La regola di ricorrenza è banalmente M[k] = $min\{P[k-1], M[k-1]\}$. Per trovare la vendità massima devo scorrermi tutto k e ogni volta trovo il prezzo massimo con P[k] - M[k - 1]. Complessità $O(n)$, memoria volendo $O(1)$.

- *Il problema della massima sottosequenza comune*
Anche conosciuto come LCS (longest common subsequence). Poniamo come n e m le lunghezze delle due sequenze. Una ricerca esaustiva richiederebbe tempo $O(2^n + 2^m)$. Ma noi usiamo la dinamica impostando come tabella L[i][j] = massima lunghezza di una sottosequenza comune a x$[1 \dots i]$ e a y$[1 \dots j]$. La nostra regola di ricorrenza diventa quindi L[i][j] = $L[i-1][j-1] + 1$ se $x[i] = y[j]$, altrimenti $max\{L[i-1][j], L[i][j-1]\}$. Complessità $O(nm)$.

- *Edit Distance*
Date due stringhe x e y, una lunga n e l'altra m, vogliamo sapere quante operazioni di inserimento, eliminazione e sostituzione sono necessarie per ottenere y da x o viceversa. Come nel problema precedente, usiamo la dinamica e ci teniamo la tabella E[i][j] = edit distance dei prefissi x$[1 \dots i][1 \dots j]$. La regola di ricorrenza è E[i][j] = $E[i-1][j-1]$ se $x[i] = y[j]$, altrimenti $1 + min\{E[i-1][j-1], E[i-1][j], E[i][j-1]\}$. Come prima, complessità $O(nm)$.

- *Il problema senzaspazi*
Data una parola formata da n caratteri e un dizionario contenente tutte le parole che prende in input una stringa e ritorna 0 o 1 se quella stringa è o non è una parola. Determinare se la parola iniziale è formata da una concatenzione di altre parole. Per risolverlo in $O(n^2)$ mi salvo in un array W[i] = 1 se il prefisso $1 \dots i$ è formato da una concatenazione di parole oppure 0.

- *Matrix Chain Multiplication*
Ho una sequenza di matrici da moltiplicare. Il modo in cui uso la proprietà associativa è fondamentale e potrebbe ridurre di molto i calcoli necessari (ricordiamo che una matrice è moltiplicabile per un'altra solo se il numero di colonne sue è uguale al numero di righe della seconda). Quello che mi salvo nella mia tabella è P[i][j] = minimo numero di moltiplicazioni per il prodotto $M_i \times \dots \times M_j$. La relazione di ricorrenza è P[i][j] = $min\{P[i][k] + P[k+1][j] + m_i m_{k+1} m_{j+1} \mid k = i, \dots, j-1\}$, dove $m_x m_{x+1}$ sono le dimensioni della matrice M_x. In questo modo mi analizzo tutti i modi di mettere le parentesi nel prodotto delle matrici.

- *Il problema della massima sottosequenza crescente*
Lo dovresti già sapere bene. S[i] = massima lunghezza di una sottosequenza crescente di A$[1 \dots i]$ il cui ultimo elemento è A[i]. La ricorrenza diventa quindi S[i] = 1 se A[i] è il minimo di A$[1 \dots i]$, altrimenti $max\{S[j] \mid 1 \leq j < i \wedge A[j] < A[i]\} + 1$. Complessità $O(n^2)$.

- *Il problema della parola palindroma*
Lo dovresti già sapere bene. Trova la sottostringa massima palindroma. P[i][j] = *true* se $j - i \leq 1$ e $s[i] = s[j]$; P[i][j] = $P[i+1][j-1]$ se $j - i > 1$ e $s[i] = s[j]$; P[i][j] = $false$ altrimenti. Complessità $O(n^2)$.

Backtracking

Prop. *Sia n la lunghezza delle sequenze da generare, sia $O(f(n))$ il tempo richiesto dalla visita di un nodo interno e sia $O(g(n))$ il tempo richiesto dalla visita di una foglia. Se vengono visitati esclusivamente nodi che portano a sequenze generate, allora il tempo richiesto dalla visita è $O((nf(n)+g(n))S(n))$, dove $S(n)$ è il numero di sequenze generate.*

Un algoritmo che esplora tutte le permutazioni di una sequenza ha come complessità $O(nn!)$. Un algoritmo che esplora tutti i k-sottoinsiemi di un insieme impiega $O(n\binom{n}{k})$. Un algoritmo che esplora tutte le 3-colorazioni di un grafo impiega $O(n3^n)$. Un algoritmo che esplora tutti i possibili sottoinsiemi di un insieme ha come complessità $O(n2^n)$.

Problemi famosi

- *Il problema della cassaforte*

Una cassaforte ha una combinazione formata da n cifre decimali la cui somma ha valore k. Generare tutte le possibili combinazioni in $O(nC)$, dove C è il numero di possibili combinazioni.

- *Il problema della 3-colorazione*

- *Il problema dei cicli Hamiltoniani*

Dato un grafo G, determinare se esiste un ciclo Hamiltoniano. Un ciclo Hamiltoniano non è altro che un ciclo che passa per tutti i nodi del grafo. Un modo non molto difficile è considerare tutte le permutazioni dei nodi e vedere se esisteno tutti gli archi che collegano una certa permutazione di nodi.

- *Il problema dello zaino*

Generazione k-sottoinsiemi

```
1  COMB(n: cardinalità, k: card. sottoinsieme, l: numero di 1, h: lungh. prefisso, S: k-sottoinsieme):
2      if (h = n) print(s)
3      else {
4          if (l >= k - n + h + 1) {
5              S[h + 1] = 0
6              COMB(n, k, l, h + 1, S)
7          }
8          if (l < k) {
9              S[h + 1] = 1
10             COMB(n, k, l + 1, h + 1, S)
11         }
12     }
```

Generazione permutazioni

```
1  PERM(n: lungh. permutazione, h: lungh. prefisso, S: permutazione):
2      if (h = n) print(S)
3      else {
4          E: array lungo n inizializzato a 0
5          for (i = 1 TO h)  E[S[i]] = 1
6          for (i = 1 TO n) {
7              if (E[i] = 0) {
8                  S[h + 1] = i
9                  PERM(n, h + 1, S)
10             }
11         }
12     }
```