Riassunto APA

Parte 1

Un **albero libero** è un grafo non orientato connesso aciclico.

Dato un grafo non orientato e connesso G, un **albero ricoprente** (spanning tree) di G e un sottografo di G che contiene tutti i nodi ed è un albero libero. Ossia, è un albero che connette tutti i nodi del grafo usando archi del grafo, ha quindi n nodi ed n − 1 archi.

Problema aperto se limite superiore e inferiore sono diversi, bigO diverso da omega. Può essere chiuso sia dal di sotto che dal di sopra. Se questi limiti coincidono il **problema** è **chiuso**

La **correttezza degli algoritmi ricorsivi si basa sul principio di induzione**, quindi una base e il passo induttivo, l'induzione può essere di due tipi:

- Induzione: se vale P(n), allora vale per n+1. Si assume p(n) e si dimostra per p(n+1)
- Induzione forte: se si dimostra che, se l'ipotesi induttiva vale per m, allora vale per tutti gli n < m.

La **correttezza degli algoritmi iterativi** si basa su una precondizione, una postcondizione (quello che voglio sia vero alla fine) e un invariante che si mantiene durante il ciclo. L'invariante si ottiene indebolendo la postcondizione.

Torre di Hanoi: problema intrattabile ricorsivo, la sua correttezza si basa sull'induzione aritmetica.

Spesso conviene esprimere la complessità temporale in termini di 2^k. Per Quicksort la complessità nel caso migliore è di $T(n) = \Theta(n) + T(n/2) + T(n/2)$, per $n > 1$, uguale a $0(n \log(n))$.

Per **l'analisi e la progettazione degli algoritmi ricorsivi** è lo scopo principale è trovare l'invariante perfetta, per poi trovare il passo e la condizione di controllo.

BFS, algoritmo iterativo per la visita dei grafi in ampiezza a partire da un nodo s, tutti i nodi vengono marcati come non visitati, successivamente vengono estratti via a via da una coda e si visitano tutti quelli collegati con il nodo estratto.

```
BFS(G,s) //visita nodi di G raggiungibili a partire da s
      for each (u nodo in G) marca u come bianco; parent[s] = null
      Q = coda vuota
      Q.add(s); marca s come grigio;
      while (Q non vuota)
            u = Q.remove() //u non nero
            visita u
            for each ((u,v) arco in G)
                  if (v bianco)
                        marca v come grigio; Q.add(v); parent[v]=u
            marca u come nero
```

DFS, nella dfs i nodi vengono visitati in profondità, viene usato uno stack al posto della coda, e i nodi collegati al nodo estratto vengono aggiunti in cima allo stack, successivamente vengono estratti quelli non visitati e così via. Un nodo può essere inserito nella pila più volte, al massimo quanti sono i suoi archi entranti. Questa visita può venire effettuata anche tramite un algoritmo analogo ricorsivo.

```
DFS(G)
      for each (u nodo in G) marca u come bianco; parent[u]=null
      for each (u nodo in G) if (u bianco) DFS(G,u)

DFS(G,u)
      //inizio visita
      visita u; marca u come grigio
      for each ((u,v) arco in G)
            if (v bianco)
                  parent[v]=u
                  DFS(G,v)
      //marca u come nero
      //fine visita
```

Algoritmo di Djisktra, si basa sull'idea di una visita in ampiezza dove:

- Per i nodi già visitati si ha la distanza e l'albero dei cammini minimi T.
- Per ogni nodo non ancora visitato (non ancora estratto dalla coda), si ha la distanza provvisoria, cioè la lunghezza minima di un cammino in T, più un arco.
- Si estrae dalla coda un nuovo nodo a distanza minima.
- Si aggiornano le distanze provvisorie adiacenti al nodo estratto, tenendo conto del nuovo nodo in T.

```
Djikstra(G,s)
   for each (u nodo in G) dist[u] = ∞ // tutti i nodi sono bianchi
   parent[s] = null; dist[s] = 0 // s diventa grigio
   Q = heap vuoto
   for each (u nodo in G) Q.add(u,dist[u])
   while (Q non vuota)
      u = Q.getMin() //estraggo nodo a distanza provv. minima, u diventa nero
      for each ((u,v) arco in G) //v diventa o resta grigio
            if (dist[u] + c_{u,v} < dist[v]) // se v nero falso
                  parent[v] = u; dist[v] = dist[u] + c_{u,v}
                  Q.changePriority(v, dist[v]) //moveUp
```

La correttezza è provata tramite la seguente invariante:

- Per tutti i nodi neri (già visitati) u, dist[u] = d(u), dove d(u) è la distanza minima.
- Per tutti i nodi non ancora visitati dist[u] = distanza minima provvisoria da s a u formata da tutti nodi neri tranne u.

Viene estratto dalla coda il nodo u con dist[u] minima, quindi, per l'invariante (2), tale che esiste un cammino π da s a u minimo tra quelli costituiti da tutti nodi neri tranne l'ultimo. Tale cammino e allora anche il minimo in assoluto fra tutti i cammini da s a u.

Poiché l'insieme dei nodi neri risulta modificato per l'aggiunta di u, occorre ripristinare l'invariante (2): per ogni nodo v in Q, dist[v] deve essere la lunghezza del minimo fra i cammini da s a v i cui nodi sono tutti, eccetto v neri; ma adesso fra i nodi neri c'e anche u, quindi bisogna controllare se per qualche nodo v adiacente a u il cammino da s a v passante per u e più corto del cammino trovato precedentemente, e in tal caso aggiornare dist[v] e parent[v]. Complessità O((n+m)log n), n nodi m archi.

Algoritmo di Prim, il problema qua è trovare minimo albero ricoprente quindi l'albero ricoprente coi pesi degli archi minimi, l'idea è simile a Djisktra, ma si prende tra i nodi non ancora visitati, quello collegato a un nodo nero con arco di costo minimo.

L'invariante è:

- T è sottoinsieme di un MST del grafo G.
- Per ogni nodo u diverso da s non in T, dist[u] contiene la distanza minima che lo connette a un nodo nero (già visitato).

```
Prim(G,s)
   for each (u nodo in G) marca u come non visitato //necessario
   for each (u nodo in G) dist[u] = ∞
   parent[s] = null; dist[s] = 0
   Q = heap vuoto
   for each (u nodo in G) Q.add(u, dist[u])
   while(Q non vuota)
      u = Q.getMin() //estraggo nodo a minima distanza dai neri
      marca u come visitato (nero)
      for each ((u,v) arco in G)
            if (v non visitato && c_u,v < dist[v] )
                  parent[v] = u; dist[v] = c_u,v
                  Q.changePriority(v,dist[v]) //moveUp
```

La complessità dell'algoritmo è O((n+m)log n), n nodi, m archi.

Algoritmo di Kruskal, algoritmo alternativo a Prim per la ricerca di un MST, si inseriscono tutti i nodi in una foresta piena di archi e si ordinano gli archi per costo crescente, dopodiché si controllano gli archi in ordine e se non c'è già un arco che connette i nodi dell'arco estratto, allora si inserisce l'arco nell'MST.

```
Kruskal(G)
      s = sequenza archi di G in ordine di costo crescente
      T = foresta formata dai nodi di G e nessun arco
      while (s non vuota)
      estrai da s il primo elemento (u,v)
      if (u,v non connessi in T) T = T + (u,v)
      return T
```

Per fare questo tipo di operazioni, vengono utilizzate delle particolari strutture dati chiamate Union-Find, su queste strutture dati, sono possibili le seguenti operazioni:

- mkSet(a): crea un insieme formato solo dall'insieme a
- union(A, B): unisce gli insiemi A e B
- find(a): restituisce l'insieme che contiene l'elemento a

gli insiemi sono i nodi della foresta, se c'è un arco che li connette si usa la union, per vedere se due archi sono già connessi si una la find. Nell'implementazione ogni insieme è rappresentato da un suo rappresentante, la union quindi è union(a, b), con a e b rappresentanti di insiemi.

Una delle implementazioni migliori migliore viene fatta tramite la **union-by-size**, dove a seguito dell'unione si sceglie come rappresentante del nuovo insieme quello che ha cardinalità maggiore, la radice dell'albero con meno figli diventa figlia dell'altro albero. Una tecnica alternativa a quella dell'unione per dimensione, detta **union-by-rank**, e la seguente: l'unione viene effettuata scegliendo

come radice del nuovo albero quella dell'albero di altezza maggiore. Un'uteriore tecnica è la **path compression**: la find rende figli della radice tutti i nodi che incontra nel suo percorso di risalita dal nodo alla radice. Si noti che, dato che la union tra elementi generici consiste di due operazioni di find seguite dall'unione vera e propria, introducendo la compressione dei cammini nella find la introduciamo automaticamente anche nella union.

DAG: grafo orientato aciclico, un problema può essere dato un DAG, trovare un ordine topologico su di esso, cioè per ogni arco (u, v) => u < v. Si definiscono nodo pozzo un nodo che ha solo archi entranti, nodo sorgente che ha solo archi uscenti, è ovvio che in un DAG si hanno sempre almeno un nodo pozzo e un nodo sorgente.

Un intuitivo algoritmo per trovare un ordine topologico dato un DAG è il seguente:

- per ogni nodo in G ci si salva il numero di nodi entranti
- si aggiungono a una coda i nodi con indegree = 0
- si estrae dalla coda il nodo con indegree= 0 e si aggiunge in fondo all'ordine topologico, dopodiché si diminuisce l'indegree di tutti i nodi a cui questo è connesso, se l'indegree diventa zero, si aggiunge alla coda.

```
topologicalsort(G)
      S = insieme vuoto
      Ord = sequenza vuota
      for each (u nodo in G) indegree[u] = indegree di u //m passi
      for each (u nodo in G) if (indegree[u] = 0) S.add(u) //n passi
      while (S non vuoto) // in tutto m passi
            u = S.remove()
            Ord.add(u) //aggiunge in fondo
            for each ((u,v) arco in G)
                  indegree[v]--
                  if (indegree[v]=0) S.add(v)
      return Ord
```

Un'altra alternativa simile per risolvere questo problema è una **DFS** dove volendo si possono salvare i tempi di inizio e fine visita. Questa visita ha la seguente proprietà: per ogni (u, v) end[u] > end[v].

```
DFS(G)
      for each (u nodo in G) marca u come bianco; parent[u]=null
      time = 0
      for each (u nodo in G) if (u bianco) DFS(G,u)

DFS(G,u,T)
      time++; start[u] = time //inizio visita
      visita u; marca u come grigio
      for each ((u,v) arco in G)
            if (v bianco)
                  parent[v]=u
                  DFS(G,v)
      time++; end[u] = time //marca u come nero
      //fine visita
```

In un grafo due **nodi** si dicono **fortemente connessi** se sono mutualmente raggiungibili, in un grafo G chiamiamo componenti fortemente connesse quelle formate da nodi mutualmente raggiungibili.

Componente fortemente connessa: i sottografi massimali di G i cui nodi sono tutti fortemente connessi tra loro.

Possiamo ricavare alcune definizioni:

- In una visita in profondità il nodo avente tempo di fine visita maggiore appartiene a una componente fortemente connessa sorgente.
- In una visita in profondità, la visita DFS di una componente fortemente connessa pozzo visita solo i nodi appartenenti a tale componente.

È possibile adesso ricavare un algoritmo per trovare tutte le componenti fortemente connesse e il loro ordine topologico in un grafo orientato G.

- Si effettua una visita DFS e si salvano in ordine dal tempo di fine maggiore a quello minore, (ord).
- Si estrae via via in ordine l'ultimo nodo in ord.
- Si crea il grafo G trasposto (con la direzione di tutti gli archi invertita), adesso la prima componente estratta si troverà in una componente fortemente connessa pozzo.
- Si segnano come visitate queste componenti e si aggiunge questo insieme di nodi C all'ordine topologico.
- Si prosegue finché non si visitano tutti i nodi.

```
SCC(G)
      DFS(G,Ord) //aggiunge i nodo visitati a Ord in ordine di fine visita
      //si noti che non occorre calcolare i tempi di fine visita
      G^T = grafo trasposto di G
      Ord⊕ = sequenza vuota //ordine topologico delle c.f.c.
      while (Ord non vuota)
            u = ultimo nodo non visitato in Ord //si trova in una c.f.c. sorgente
            C = insieme di nodi vuoto
            DFS(G^T,u,C) //aggiunge nodi visitati in C
            Ord⊕.add(C) //aggiunge in fondo
      return Ord⊕
```

Programmazione dinamica, consiste nel calcolare risultati futuri basandosi su quelli attuali, un esempio classico è Fibonacci.

Un altro algoritmo di programmazione dinamica è il LongestCommonSubsequence, che consiste nel trovare sottosequenza di lunghezza maggiore date due stringhe.

L'algoritmo funziona nel seguente modo e fa uso di una matrice L.

- Se una delle due sequenze ha lunghezza nulla allora la LCS è vuota.
- Se X[i] = Y[j], allora la LCS = LCS(i-1, y-1) + (i, y)
- Altrimenti, LCS(i,j) è la più lunga tra LCS(i-1, y) e LCS(i, y-1).

Si costruisce una matrice LCS con m+1 righe ed n+1 colonne. In base alla definizione induttiva data sopra, la prima riga e la prima colonna vanno riempite con la sequenza vuota, di lunghezza 0. In ogni casella non occorre memorizzare tutta ` la LCS ma basta mettere:

- se si ha lo stesso carattere sulla riga e sulla colonna, una "freccia diagonale", aumentando di 1 la lunghezza rispetto alla casella puntata dalla freccia

- se sulla riga e sulla colonna ci sono due caratteri diversi, una freccia verso la cella di lunghezza maggiore fra la contigua sopra e la contigua a sinistra (se hanno uguale lunghezza si sceglie per esempio quella sopra).

Seguendo le frecce è possibile ricostruire la lcs, mentre nella cella in basso a sinistra sarà sempre contenuta la lunghezza di tale cella.

```
LCS(X,Y)
      for (i = 0; i <= m; i++) L[i,0] = 0
      for (j = 0; j <= n; j++) L[0,j] = 0

      for (i = 1; i <= m; i++)
            for (j = 1; j <= n; j++)
                  if (X[i] = Y[j])
                        L[i,j] = L[i−1,j−1] + 1; R[i,j] = frecciaDiagonale
                  else if (L[i,j−1] > L[i−1,j])
                        L[i,j] = L[i,j−1; R[i,j] = ←
                  else L[i,j] = L[i−1,j; R[i,j] = ↑
```

La complessità è O(nm), n colonne, m righe della matrice.

Algoritmo di Floyd Warshall, serve per trovare la distanza minima tra un nodo a tutti gli altri in un grafo orientato pesato, ammette archi di costo negativo ma non cicli. Diverso da Djisktra perché voglio saperlo per tutti i nodi.

L'idea la seguente. Costruisco una matrice con righe e colonne corrispondenti ai nodi, e per ogni nodo inizialmente mi segno la distanza ai nodi direttamente connessi a loro con un arco. Successivamente per ogni nodo guardo se esiste un cammino di costo minore passando per un altro nodo v, e in caso aggiorno le distanze, eseguo questo algoritmo finché non ho esaurito tutti i nodi del grafo. Se vogliamo sapere anche il cammino e non il costo salviamo anche la matrice dei predecessori, quindi a quale nodo bisogna andare per prendere il cammino di costo minimo.

Chiamiamo k-vincolato un cammino che passa da 1 a k, esclusi gli estremi, chiamiamo $d^k(x, y)$, la distanza k vincolata dal nodo x al nodo y. Possiamo esprimere d^k in funzione di d^{k-1}:

- $d^k(x, y) = \min\{d^{k-1}(x, y), d^{k-1}(x, k) + d^{k-1}(k, y)\}$

Infatti, dato un cammino minimo (semplice) k-vincolato da x a y, si hanno due casi:

- non passa per k, quindi e anche un cammino minimo k − 1-vincolato
- passa per k, quindi (essendo semplice) e composto da un cammino k − 1-vincolato da x a k, e da un cammino k − 1-vincolato da k a y.

```
FloydWarshall(G)
      for each (x,y : nodi in G)
            D[x,y] = 0 se x=y, c_{x,y} se x ≠ y ed esiste arco (x,y), ∞ altrimenti
            Π[x,y] = x se x ≠ y ed esiste arco (x,y), null altrimenti
      for (k=1; k <= n; k++)
            for (x,y : nodi in G)
                  if (D[x,k] + D[k,y] < D[x,y])
                        D[x,y] = D[x,k] + D[k,y]
                        Π[x,y] = Π[k,y]
      return D,Π
```

Dato un albero di cammini minimi $G_{\Pi,x}$, possiamo ottenere un cammino minimo da x a y nel modo seguente:

```
shortest_path(Π,x,y)
      if (x=y) return x
      else if (Π[x,y] = null) return ... [non esiste cammino]
      else return shortest_path(Π,x,Π[x,y]) · y
```

Teoria della NP completezza:

- **P:** problemi risolvibili in tempo polinomiale.
- **NP:** (non deterministic polinomial time) problemi verificabili in tempo polinomiale o per cui esiste un algoritmo non deterministico che li risolve.
- **NP-HARD:** un problema L è np-hard se ogni problema Q in NP è polinomialmente riducibile a L.
- **NP-COMPLETO (NP-C):** un problema è np-completo se appartiene a np-hard e appartiene alla classe NP, non tutti i problemi np-hard sono np-completi.
- **Problema (astratto):** è una relazione $P \subseteq I \times S$, dove I è l'insieme degli input (o istanze del problema) e S è l'insieme delle (possibili) soluzioni.
- **Problema (astratto) di decisione:** P e un problema (astratto) in cui ogni input ha come soluzione vero oppure falso, ossia $P : I \rightarrow \{T, F\}$.
- **Problema concreto:** P è un problema il cui insieme di istanze (input, credo) è l'insieme delle stringhe binarie, ossia $P : \{0, 1\}^* \rightarrow \{T, F\}$.
- Un problema astratto P puo essere rappresentato in modo concreto tramite una **codifica**, ossia una funzione iniettiva: $c : I \rightarrow \{0, 1\}^*$
- La teoria della complessita computazionale si concentra sui problemi di decisione, in quanto in genere, per ogni problema di altro tipo P, e possibile dare un problema di decisione P_d tale che risolvendo il primo si sappia risolvere il secondo, **In altri termini il problema di decisione P_d è "più facile" del problema originale P, quindi se proviamo che P_d e "difficile" indirettamente proviamo che lo è anche P.**

Quindi, se troviamo un algoritmo che risolve un problema NP-C in tempo polinomiale allora dimostreremmo che NP=P, perché tutti i problemi NP sono riducibili a un problema NP-C e i problemi NP-C sono riducibili polinomalmente tra di loro.

SAT è NP-C, il problema SAT consiste nel verificare se data una formula booleana esiste una combinazione di valori che la rende vera.

3SAT è NP-C, in quanto ovvio sotto insieme di SAT, infatti considera formule del tipo (x1 or x2 or x3) and (x4 or x5 or x6).

Una clique (cricca), in un grafo è un insieme di nodi mutualmente collegati tra di loro, il problema della clique consiste nel vedere se in un grafo esiste una clique di grado k. Questo problema è NP-C in quando 3SAT è riducibile in questo problema:

- Un nodo per ogni occorrenza di letterale in una clausola, quindi 3k nodi, una clausola in un problema 3SAT è del tipo (x1 or x2 or x3).
- Per ogni nodo in clausole diverse che non siano una la negazione dell'altra, esiste un nodo tra le due clausole

La formula 3SAT è soddisfacibile se nel grafo G esiste una clique di k nodi.

- Se la formula è soddisfacibile, Esiste un arco che collega ogni coppia di questi nodi, perché sono in clausole diverse e non sono uno la negazione dell'altro, altrimenti non potrebbero essere contemporaneamente veri.
- Se G contiene una clique di k nodi, sappiamo che la clique contiene esattamente un nodo per ogni clausola, siano $l_1, \ldots, l_k$ i corrispondenti letterali. Scegliamo un'assegnazione di valori alle variabili tale che $l_1, \ldots, l_k$ risultino veri. Con questa assegnazione ogni clausola è soddisfatta, e quindi l'intera formula.

Domande orale parte 1

Complessità di problemi, problemi aperti e chiusi:

- La complessità di problemi si esprime tramite tre classi, bigO(f(n)), omega(f(n)) e theta(f(n)). La prima indica che la complessità (spaziale o temporale) di un algoritmo cresce al più come f(n), omega indica che la complessità del problema da un certo input in poi, cresce almeno come f(n), theta invece indica che la complessita sarà compresa tra una due costanti moltiplicative di f(n), quindi cresce come f(n).
- Un problema algoritmico (come ad esempio l'ordinamento tramite confronti), ad un certo periodo nel tempo, può risultare aperto o chiuso. Un problema può essere chiuso ma non viceversa. Viene definito con limite superiore la classe temporale che indica che un problema può avere complessità massima O(f(n)). Con limite inferiore indichiamo che un problema ha complessità almeno theta(f(n)). Un problema si dice chiuso se i due limiti coincidono, aperto altrimenti. Per esempio, nel problema dell'ordinamento per confronti, se conoscessimo solo insertion sort e quicksort, avremmo come limite superiore n^2 e inferiore ovviamente n perché dobbiamo scorrere almeno tutti gli elementi, il problema risulterebbe quindi aperto, per chiuderlo possiamo farlo sia dal di sotto che dal di sopra. Dal di sopra quindi trovando un algoritmo più efficiente nel caso peggiore oppure dimostrando che il limite inferiore è più alto.

Algoritmo di Dijkstra:

- È un algoritmo con lo scopo di trovare il cammino minimo da un nodo a tutti gli altri.
  L'algoritmo è una variazione di una visita in profondità dove:
  - Per ogni nodo visitato si ha l'albero T dei cammini minimi
  - Per ogni nodo non ancora visitato si ha la istanza provvisoria da s a u, formata da tutti nodi visitati tranne l'ultimo.
  - Si estrae un nodo a distanza provvisoria minima (che risulta essere la distanza definitiva)
  - Si aggiornano tutte le distanze provvisorie dei nodi adiacenti al nodo estratto tenendo conto del nuovo arco.

L'invariante è composta da due parti

1. Per ogni nodo già visitato dist[u] = distanza minima da s a u
2. Per ogni nodo non ancora visitato dist[u] = distanza provvisoria formata da tutti nodi neri tranne l'ultimo.

L'invariante 1 si mantiene perché estraiamo dal grafo con distanza provvisoria minima k, formata da tutti nodi neri tranne l'ultimo, se questa non fosse la distanza minima significherebbe che esiterebbe un altro un altro cammino da s a u che però deve passare per altri nodi non neri eccetto quelli visitati ma se deve passare per altri nodi non visitati passa necessariamente per il nodo estratto che ha distanza minima k quindi questo non sarebbe possibile.

L'invariante 2 adesso però va aggiornata, in quando adesso u fa parte dell'albero T; dobbiamo aggiornare tutte le distanze ai nodi non ancora visitati in caso queste siano minori della distanza precedente.

La complessità è O((n+ m)logn), perché abbiamo n estrazioni dallo heap (costo logn) e per ogni estrazione, nel ciclo interno ogni arco viene estratto una volta e si ha un eventuale move up; il costo totale è quindi O(log n log m) = O((n+m)logn).

Definizione di minimo albero ricoprente, algoritmi di Prim e Kruskal:

- Con minimo albero ricoprente di un grafo non orientato pesato si intende un albero passante per tutti i nodi avente n-1 archi e somma dei pesi minima. Per calcolarlo si utilizzano due algoritmi, Prim e Kruskal.
- L'algoritmo di Prim è simile a DJisktra ma per ogni nodo non ancora in T dove T è un MST, si estrae quello a distanza minima da un nodo qualsiasi già visitato in T
- Si aggiornano le distanze provvisorie da un nodo non nero a un nodo in T.

L'invariante è simile a DJisktra e formata da due parti:

- T è sottoinsieme di un MST
- Dist[u], per ogni nodo non in T, dist[u] è la distanza minima che connetta il nodo u non visitato all'MST

L'invariante si mantiene in modo molto simile a Djisktra e la complessità è esattamente la stessa O((n+m) log n).

L'algoritmo di Kruskal invece, crea una foresta contenente tutti i nodi tranne gli archi, poi ordina gli archi in ordine crescente per peso. Viene estratto l'arco (a,b) di peso minimo, tra quelli da estrarre, si verifica allora che i nodi a e b non siano già collegati, in caso si aggiunge l'arco (a, b) al MST.

- Per effettuare questo tipo di operazioni, vengono utilizzati delle strutture dati chiamate union-find, dove su di esse sono possibili tre operazioni:
    - Mkset(a): crea l'insieme formato dal solo insieme a
    - Union(a, b): unisce gli insiemi i cui rappresentati sono a e b
    - Find(a): restituisce l'insieme contenente l'elemento a

Per implementare esistono diverse tecniche le più convenienti sono la union-by-size e la union-by-rank, entrambe hanno costo O(log n) per la union e per la find.

La complessità dell'algoritmo è O(m log m) se implementato con struttura dati union find, questo perché la complessità per ordinare tutti gli archi è O(m log m), inoltre è possibile dimostrare che la complessità per n mkset, 2m find e n-1 union è O((n+m) log* m), dove però log* è una funzione che cresce molto lentamente, praticamente costante da un certo punto in poi.

Definizione di ordinamento topologico, i due algoritmi per calcolarlo

L' ordinamento topologico di un grafo orientato aciclico (DAG) è definito come, se esiste l'arco (a,b) in G allora a < b.

Definiamo come nodo sorgente un nodo con soli archi uscenti e pozzo uno con soli archi entranti; in un DAG esistono sempre almeno un nodo pozzo e un nodo sorgente.

Il primo algoritmo per trovare un ordine topologico è topological sort; in questo algoritmo iniziamo la visita di un grafo da un nodo sorgente e lo aggiungiamo all'ordine, successivamente diminuiamo l'indegree di tutti i suoi nodi figli; se l'indegree diventa 0 allora aggiungiaamo il nodo alla lista dei nodi da visitare. La complessità è O(n +m).

Il secondo algoritmo consiste sostanzialmente in una visita in profondità dove ordiniamo i nodi in ordine di fine visita; quello con il tempo di fine fisita maggiore sarà un nodo sorgente; quindi il primo nell'ordine topologico e così via. Possiamo quindi inserire a ogni fine visita i nodi uìin una sequenza ordinata.

La complessità di questo algoritmo è O(n + m); come per la DFS normale

Definizione di componenti fortemente connesse, l'algoritmo per calcolarle

In un grafo orientato, due nodi si dicono fortemente connessi se mutuallmente raggiungibili; all'interno di un grafo una componente fortemente connessa è formata da solo nodi fortemente connessi; ossia i sottografi di G massimali tale che ogni componente è fortemente connessa tra loro.

Definiamo come grafo quoziente quel grafo formato da solo componenti fortemente connesse C e C', dove esiste l'arco tra C e C' solo se esiste un arco tra una componente di C a una di C'.

Il tempo di fine visita di una componente fortemente connessa C è il massimo dei tempi di fine visita dei suoi componenti. Allora se C e C' sono due componenti fortemente connesse del grafo G, ed esiste un arco da C a C' allora il tempo di fine visita di C' è minore di quello di C.

In una DFS su un grafo G, il nodo con tempo di fine visita maggiore appartiene a un una componente fortemente connessa sorgente. Mentre in una DFS su un una componente fortemente connessa pozzo C questa visiterà solo i nodi di C.

Per trovare tutte le componenti fortemente connesse di un grafo G esiste un ovvio algoritmo:

- Si esegue una DFS su G e si inseriscono tutti i nodi in una lista in ordine di fine visita (tempo più alto all'inizio)
- Si estrae il primo nodo dalla struttura dati ed è chiaramente visibile, dalle assunzioni precedenti che questa appartiene a una componente fortemente connessa sorgente.
- Si crea il grafo trasposto di G, cioè con la direzione dei nodi inverita, adesso il nodo estratto apparterrà a una componente fortemente connessa pozzo.
- Si effettua una DFS su questa componente e marcando i nodi trovati come visitati.
- Si ripete fino a che la coda non è vuota.

La sequenza ottenuta è un ordine topologico delle componenti fortemente connesse del grafo G.

La complessità è O(n + m) perché:

- Visita del grafo G O(n+m)
- Generazione grafo trasposto O(n+m)
- Visita del grafo trasposto O(n+m)

Caratteristiche della programmazione dinamica, problema LCS e algoritmo per risolverlo, algoritmo di Floyd-Warshall

La programmazione dinamica si basa sul prenderere decisioni basandoci su risultati precedenti, per prima cosa si risolvono i problemi base, e successivamente utilizziamo questi risultati per risolvere i problemi intermedi memorizzando il risultato, fino ad arrivare alla soluzione del priblema richiesto, un esempio classico è fibonacci.

Il problema LCS consiste nel trovare la più lunga sotto sequenza in comune tra due sequenze X e Y, i casi possibili sono:

- Se una delle due sottosequenze è vuota allora la più lunga sotto sequenza in comune è vuota (lunghezza 0)
- LCS(i, j) se X[i] = Y[J] allora LCS = LCS(i-1, j-1) . (i, y), cioe la lcs della sottostringa più la lettera in comune
- Altrimenti max(LCS(i, j-1), LCS(i-1, j)) perché la LCS non potrà contenere la coppia (i, j)

Si costruisce un algoritmo tale per cui si salva in una matrice, con n+1 righe e m+1 colonne con n lunghezza di X e m di Y, la lunghezza della LCS fino a quel punto e una freccia in base al valore che si trova, se si incontra un carattere uguale tra le due sequenza, si aumenta di uno la lunghezza della LCS e si inserisce una freccia in diagonale, altrimenti una freccia che punta al più alto valore della LCS stra quella in alto o a sinistra rispetto alla cella che stiamo consiederando e si eredita anche il valore di lunghezza della LCS da quella cella. Per la prima colonna e riga invece si salva zero.

La complessità è O(n*m).

L'algoritmo di floyd warshall è un altro algoritmo di programmaione dinamica con lo scopo di trovare la dispanza minima di tutti i nodi verso tutti gli altri. Differisce da djisktra perché ha lo scopo di trovarlo per tutti i nodi e non da uno solo.

Chiamiamo cammino k-vincolato, un cammino che passa per i nodi 1 ... k con k <= n, indichiamo con d^k la distanza k-vincolata da x a y. $d^k(x,y)$ è il minimo tra $d^{k-1}(x,y)$ (cammino che non passa per k) e $d^{k-1}(x,k)$ + $d^{k-1}(k,y)$.

Il funzionamento è il seguente:

- Si segnano in una matrice n * n le distanze tra un nodo e i suoi adiacenti, se questa distanza non esiste si segna infinito, se è lo stesso nodo si segna zero. Successivamente si esegue l'algoritmo analizando le distanze per ogni nodo se si passasse anche da un nodo a, se la distanza diventa minore si aggiorna la distanza. Si esegue questo passaggio per tutti i nodi.

La complessita è $O(n^3)$ perché per ogni nodo dobbiamo scorrere tutta la matrice.

Definizioni di problema di decisione, astratto e concreto, algoritmo di verifica, classi P, NP e NP-C

Si definisce problema di decisione astratto un problema per cui dato un input viene restutuito vero o falso. Con problema concreto di decisione si indicano quei problemi in cui gli input sono strighe binarie. Ad esempio trovare se esiste in un grafo un cammino di distanza minima k. Tutti i problemi sono riconducibili a problemi di decisione, se sappiamo risolvere P allora sappiamo risolvere P_d, i problemi di decisone son quindi più facili, se proviamo che questi sono più difficili allora anche il problema P è difficile.

Con algorito di verifica si intende un algoritmo che presa un istanza di un problema e un certificato, verifica se questo soddisfa l'istanza. In sintesi verifica se il certificato passato in input è valido. Con certificato intendiamo una stringa o un input per una determinata istanza che afferma che questa istanza ha soluzione affermativa.

La classe P è la classe di problemi risolvibili in tempo polinomiale.

NP è la classe di problemi verificabili in tempo polinomiale.

NP-C è la classe dove tutti i problemi appartengo a np e sono NP-HARD, cioè per ogni problema in NP è possibile ridurlo polinomialmente a il determinato problema.

Nozione di riduzione polinomiale, definizione di classe NP-C, problema P-NP

Con riduzione polinomiale si intende la "trasformazione" da un problema a un altro fatta in tempo polinomiale tramite una funzione di riduzione.In particolare se esiste una funzione f, calcolata in tempo polinomiale, tale che se x appartiene a P1 allora f(x) appartiene a P2

Il problema P-NP consiste nel dimostrare se P = NP o meno, questo può essere fatto trovando un problema NP-C che può essere risolto in tempo polinomiale oppure dimostrando che questo non è possibilie.

Esempi di problemi NP-completi e riduzioni (SAT, 3SAT, CLIQUE)

Il problema SAT consiste nel verificare se per una certa formula booleana, esiste un insieme di valori che la rende vera. Questo è un problema NP-COMPLETO, è il primo a essere stato dimostrato come tale nel 1971 da Cook.

3SAT è un caso particolare di SAT ed è quindi anche questo NP-COMPLETO, il caso di 3SAT studia formule in terza forma normmale congiuntiva, quindi formate da end di clausole formate a loro volta da tre or.

Il problema della clique o cricca, consiste nel trovare in un grafo la cricca di grado massimo, si definisce cricca un'insieme di nodi direttamente connessi tra loro. Questo problema è NP-COMPLETO e la conversione da 3SAT a clique è la seguente:

- Un nodo per ogni occorrezza di letterale in una clausola, quindi 3k nodi. Con k numero di clausole
- Un arco che connetta ogni letterale in clasuole diverse, a meno che queste non siano una la negazione dell'altra.

La formula è quindi soddisfacibile se e solo se il grafo contiene una clique di k nodi

- Se la formula è soddifacibile esiste un assegnazione di variabili che rende vera tutte le clausole(almeno un letterale per ogni clausola risulta vero), quindi esiste un arco che connette ogni coppia di questi nodi in quanto per rendere vere tutte le clausole non possono essere uno la negazione dell'altra.
- Se il grafo contiene una clique di k nodi, prendiamo i corrispondenti letterali e assegnamo a essi valori tali per cui tutti risultano veri, questo è possibile perché non può essere che siano uno la negazione dell'altra; con queste assegnazioni ai letterali la formula di partenza risulta vera.