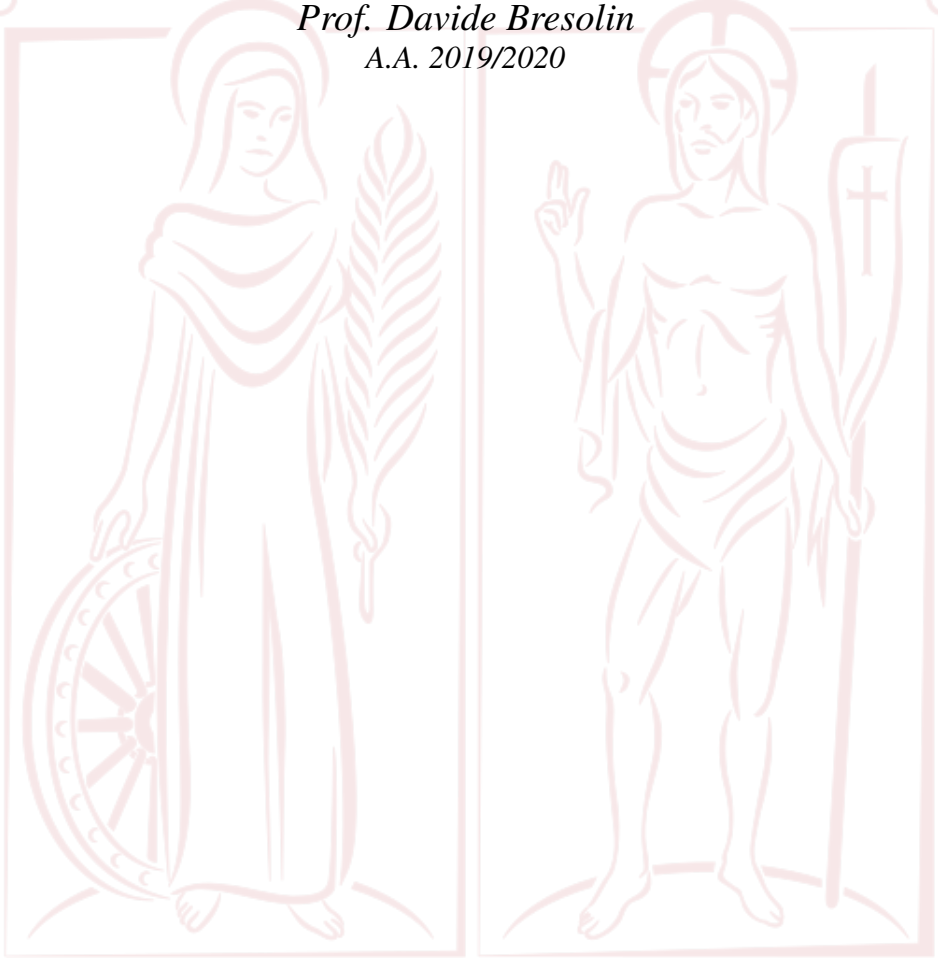
Appunti di Algoritmi Avanzati

Prof. Michele Scquizzato
Prof. Davide Bresolin
A.A. 2019/2020



Appunti a cura di Federico Brian

Indice

1 Introduzione

1.1 Programma del corso

Il corso affronterà in generale tre argomenti:

* Algoritmi per grafi:
  - algoritmi di visita di grafi, cioè come esplorare un grafo visitando tutte le sue componenti (algoritmi generici e applicazioni per problemi specifici);
  - Calcolo di minimum spanning trees (alberi di copertura del costo minimo);
  - Cammini minimi (shortest paths) su grafi esatti (tipo RO);
* Algoritmi di approssimazione:
  - Problemi intrattabili di ottimizzazione: al momento non esistono algoritmi efficienti per risolvere questi problemi, $\Rightarrow$ bisogna trovare un'ottimizzazione di una funzione obiettivo (LCS, SCS, ...) bisogna trovare qualcosa che ha un costo minimo. Quando non esiste un algoritmo che ha complessità P per risolvere il problem in modo esatto allora si passa ad un alkgoritmo di approssimazione, che restituisce l'approssimazione di una soluzione ottima secondo un certo fattore;
  - NP-completezza e riduzioni tra problemi;
  - algoritmi di approssimazione (vedi sopra) per vertex cover e traveling salesman problem;
* Algoritmi randomizzati: algoritmi che al loro interno contengono un'operazione di "lancio della moneta". Succede che per molti casi l'algoritmo più efficiente per un problema è un algoritmo randomizzato.
  - Chernoff bounds: risultati della teoria delle probabilità che ci aiutano ad ottimizzare un algoritmo, tipo dimostrare che un algoritmo fa delle cose a caso ma con altissima probabilità produce un output corretto;
  - analisi di Quicksort;
  - algoritmi randomizzati per il problema di taglio minimo (delle aste).

Per ogni argomento sono previste 12 ore di teoria + 4 ore di laboratorio di implementazione su un dataset di medie dimensioni.

1.2 Modalità esame

* **Teoria**: prova scritta che si tiene nelle normali sessioni d'esame
  - un paio di domande di natura teorica sul programma (e.g. applica questo algoritmo visto in classe su questo input e dimmi il risultato), "**domande veloci a cui si risponde abbastanza velocemente**";
  - un paio di esercizi che richiederanno di risolvere un certo un problema dando un algoritmo e analizzandolo (un po' più corposi);

- Pratica: esecizi di laboratorio.
  - Due modalità possibili:
    - Durante il corso: un gruppo di massimo tre studenti implementa gli algoritmi visti nei laboratori e consegna il codice (in un linguaggio di programmazione a scelta) ed i risultati ottenuti **entro l'inizio del laboratorio successivo**;
    - Dopo il corso: progetto personale (implementare qualche algoritmo visto in classe e farlo eseguire su un dataset di medie dimensioni).

Variazioni modalità d'esame causa COVID-19

- **Parte teorica:**
  - **Domande**: delle domande la cui risposta è veloce, cose abbastanza semplici che vanno a verificare una conoscenza base. Cosa tipica: conoscenza di un certo algoritmo, *i.e.* "Applica questo algoritmo su questo grafo $G = (V, E)$ e scrivi il risultato che ritorna".
  - **Esercizi**: risoluzione di problemi, circa $\frac{2}{3}$ del voto finale. Occorre imparare e saper applicare alcuni algoritmi fondamentali + imparare a sviluppare algoritmi per problemi nuovi che sono presentati. Fare l'analisi di un certo algoritmo.

2 Algoritmi per grafi

2.1 Introduzione

2.1.1 Definizioni preliminari

Def. 2.1. ***(Grafo)*** *Modo di rappresentare relazioni che esistono tra coppie di oggetti.*
Grafo $G = (V, E)$*:*

- $V \equiv$ *insieme di nodi (vertici);*
- $E \equiv$ *collezionie di lati (coppie di nodi).*

Def. 2.2. ***(Grafo diretto)*** *un grafo si dice diretto se ogni lato* $(u, v) \in E$ *è una coppia ordinata*[1] *(*$u \to v$*), altrimenti si dice non diretto* $(u - v)$*.*

N.B. La definizione ammette:

- la presenza di **lati multipli tra due vertici** perché *E* è definito come una collezione, non un insieme;
- lati **self loop** come ad esempio la coppia (u, u)

Def. 2.3. ***(Grafo semplice)*** *Un grafo semplice è un grafo senza lati multipli e senza self loop.*

Italiano	Inglese
nodi/vertici	nodes/vertices
lati di un grafo non diretto	edges
lati di un grafo diretto	arcs

Tabella 1: Analogie tra termini italiani ed inglesi.

Def. 2.4. ***(Lato incidente - Nodi adiacenti)*** *Dato un lato* $e = (u, v) \in E$*, si dice che* e *è incidente su* u *e* v *e che* u *e* v *sono adiacenti.*

Def. 2.5. ***(Vicini di un vertice)*** *I vicini di un vertice* v *sono tutti i vertici* u *tali che* $(v, u) \in E$

Def. 2.6. ***(Grado di un vertice)*** *Il grado di un vertice* $v \in V$ $(degree(v))$ *è il numero di lati di* E *incidenti su* v*.*

Def. 2.7. ***(Cammino semplice)*** *Un cammino semplice è una sequenza di nodi distinti* $u_1, u_2, ..., u_k$ *e* $(u_i, u_{i+1}) \in E \ \forall \ 1 \leq i < k$*.*

N.B. l'aggettivo *semplice* indica che i vertici sono tutti distinti, altrimenti si parla solo di cammino. Ad esempio 5,1,8,7,6,1,4 **non è semplice**.

Def. 2.8. ***(Lunghezza del cammino)*** *La lunghezza del cammino è data dal numero di lati* (u_i, u_{i+1})*, ovvero* $k - 1$*.*

Def. 2.9. ***(Ciclo semplice)*** *Un ciclo semplice è un cammino semplice* $u_1, u_2, ..., u_k$ *con* $u_1 = u_k$*.*

Def. 2.10. ***(Lungezza del ciclo)*** *La lunghezza del ciclo è quella del cammino* $u_1, u_2, ..., u_k$*.*

[1] cioè c'è un ordine: prima u e dopo v

N.B. Senza l'aggettivo *semplice* si ammette la presenza di vertici ripetuti, oltre ai due estremi. Esempio: 6,1, 3,4,1,6.

Def. 2.11. ***(Sottografo)*** *Un sottografo è un grafo $G' = (V', E')$ con $V' \subseteq V$, $E' \subseteq E$, e tale che i lati di E' incidono solo su nodi inV' (prendo un po' di nodi e dei lati su quei nodi).*

Def. 2.12. ***(Spanning subgraph)*** *Un sottografo di copertura (spanning subgraph) è un sottografo $G' = (V', E')$ con $V' = V$.*

Def. 2.13. ***(Grafo connesso)*** *Un grafo $G = (V, E)$ si dice connesso se $\forall\ u, v \in V\ \exists$ un cammino $u, ..., v$, cioè che inizia in u e termina in v. Non ci possono essere nodi isolati.*

Def. 2.14. ***(Grafo disconnesso)*** *Un grafo $G = (V, E)$ si dice disconnesso se non è connesso, ovvero se $\exists\ u, v$ (vertici) $\in V$ per i quali $\nexists$ un cammino $u, ..., v$.*

Def. 2.15. ***(Componenti connesse)*** *Le componenti connesse di un grafo $G = (V, E)$ sono una partizione di G in sottografi $G_i = (V_i, E_i)$ per $1 \leq i \leq k$ tali che:*

- *$G_i = (V_i, E_i)$ è connesso, per $1 \leq i \leq k$;*
- *$V = V_1 \cup V_2 \cup ... \cup V_k$ (partizione di V);*
- *$E = E_1 \cup E_2 \cup ... \cup E_k$ (partizione di E);*
- *$\forall\ i \neq j\ \nexists\ e \in E$ tra V_i e V_j, cioè non esiste una connessione diretta tra i due sottografi G_i e G_j. (?)*

Osservazioni:

- $\{G_i : 1 \leq i \leq k\}$ sono sottografi connessi massimali, cioè che è un sottografo connesso a cui non si può più aggiungere altri nodi ed ottenere ancora un grafo conneso;
- G connesso $\Rightarrow\ k = 1$. Potrebbe anche esserci un grafo che ha solo nodi e nessun lato. Se non c'è nessun lato, allora per definizione ci sono tante componenti connesse quanti sono il numero di nodi, quindi in questo caso k è il numero di nodi in G;
- la partizione di G in componenti connesse è univoca.

Esempio

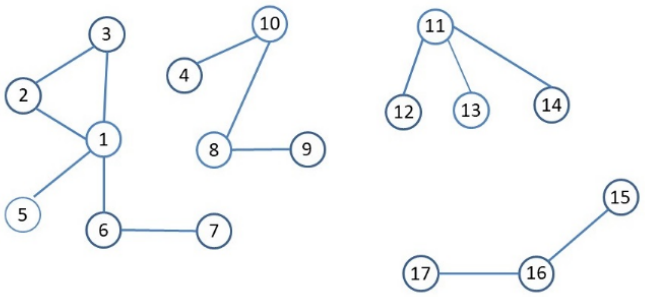


Figura 1: Grafo $G = (V, E)$.

Nella grafo G le componenti connesse sono 4 con:

- $V_1 = \{1, 2, 3, 5, 6, 7\}$;
- $V_2 = \{4, 8, 9, 10\}$;
- $V_3 = \{11, 12, 13, 14\}$;
- $V_4 = \{15, 16, 17\}$.

Def. 2.16. ***(Albero radicato)*** *Un albero radicato (rooted tree) è un grado $G = (V, E)$ tale che:*

- $\exists\ r \in V$ *detto* ***radice****;*
- $\forall\ u \in V, u \neq r\ \exists!\ p(u)$ ***(padre** di u)* $\in V$ *e si ha che* $E = \{(u, p(u))\ :\ u \in V, u \neq r\}$
  (per ogni nodo u non radice esiste il suo padre $p(u)$. Ogni nodo u è collegato direttamente con suo padre $p(u)$);
- $\forall\ u \in V,\ u \neq r,\ p(...p(u)) = r$
  (andando di padre in padre si raggiunge r).

Def. 2.17. ***(Albero libero)*** *Un albero libero (free tree) è un grafo $G = (V, E)$ connesso e senza cicli.*

Def. 2.18. ***(Foresta)*** *Una foresta (forest) è un grafo $G = (V, E)$ senza cicli, ovvero un insieme di aberi liberi disgiunti. Si noti che un albero è anche una foresta ma non vale il viceversa.*

Def. 2.19. ***(Spanning tree)*** *Uno spanning tree di un grafo G è uno spanning subgraph connesso e senza cicli (free tree). Può esistere solo se G è connesso.*

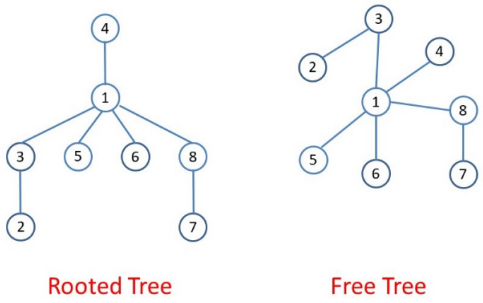


Figura 2: Esempi di *rooted tree* e *free tree*.

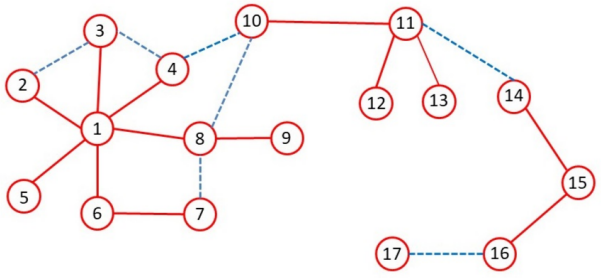


Figura 3: Spanning forest (vertici e lati in rosso).

Osservazione 1: I concetti di albero radicato e albero libero sono **equivalenti**: ogni albero radicato è un grafo connesso senza cicli e ogni albero libero può essere visto come albero radicato scegliendo opportunamente una radice e determinando di conseguenza la relazione padre-figlio.

Osservazione 2: se i nodi 16 e 17 fossero connessi, si tratterebbe di un unico spanning tree.

2.1.2 Primitive importanti

Def. 2.20. ***(Traversal)*** *Esplorazione sistematica del grafo (e.g. crawling).*

Def. 2.21. ***(Connettività)*** *Verifica se il grafo è connesso (e.g. reti wireless).*

Def. 2.22. ***(Identificazione delle componenti connesse)*** *(e.g. reti wireless).*

Def. 2.23. ***(Identificazione di minimum spanning tree)*** *(e.g. broadcast efficiente).*

Def. 2.24. ***(Identificazione di cammini minimi)*** *(e.g. navigatore).*

Def. 2.25. ***(Stima della distanza media/massima)*** *(e.g. Facebook).*

2.1.3 Notazione

Dato un grafo $G = (V, E)$:

* $n = |V|$ denota il numero di nodi;
* $m = |E|$ denota il numero di lati.

2.1.4 Proprietà dei grafi

Sia $G = (V, E)$ un grafo **non diretto semplice** con $|V| = n$ vertici e $|E| = m$ lati. Valgono le seguenti proprietà:

* $\sum_{v \in V} degree(v) = 2m$;
  (Dim.) Ogni lato è contato esattamente 2 volte nella sommatoria.
* $m \leq \binom{n}{2}$, e quindi $m \in \mathcal{O}(n^2)$;
  (Dim.) Dato che il grafo è semplice, E è un sottoinsieme di tutte le $\binom{n}{2}$ possibili coppie di vertici.
* se G è un albero $\implies m = n - 1$;
  (Dim.) Consideriamo G come un rooted tree (grazie all'equivalenza tra rooted tree e free tree) $\implies$ E reppresenta relazioni padre-figlio, che sono $n - 1$ (ogni nodo non radice ha un unico padre).
* se G è connesso $\implies m \geq n - 1$;
  (Dim.) Si consideri il seguente ciclo:
  while($\exists$ ciclo C)
  elimina da G un lato di C
  Si ha che:
  - *alla fine di ogni iterazione, G è connesso;*
  - *alla fine del ciclo, G è connesso e senza cicli (un **free tree**) con $m' = n - <\leq m$ lati.*
* se G è senza cicli (cioè una foresta) $\implies m \leq n - 1$.
  (Dim.) Si supponga che G sia composto da $k \geq 1$ alberi disgiunti, e che l'i-esimo albero abbia n_i vertici ed m_i lati. Quindi si ha che $n = \sum_{i=1}^{k} n_i$ ed $m = \sum_{i=1}^{k} m_i$. Ma dato che $m_i = N - i - 1$, in quanto si tratta di alberi, si ha che

$$m = \sum_{i=1}^{k} m_i = \sum_{i=1}^{k} (n_i - 1) = n - k \leq n - 1$$

2.1.5 Rappresentazione di grafi

Sia $G = (V, E)$ con n vertici ed m lati. Se non diversamente specificato, assumeremo che i vertici siano associati agli interi $1, 2, ..., n$.

Lista di vertici L_V: $\forall\ v \in V,\ L_v[v]$ contiene tutte le informazioni su v;

Lista di lati L_E: $\forall\ e = (u, v) \in E, L_E[e]$ contiene tutte le informazini su e e il collegamento a $L_V[u]$ e $L_V[v]$.

Per permettere un accesso più diretto ai lati si usa una delle due strutture seguenti, in aggiunta alle strutture di base L_V e L_E:

* **Liste di Adiacenza (Adjacency List)**: $\forall\ v \in V$ si ha una lista $l(v)$ di puntatori ai lati (elementi di L_E) incidenti su v.

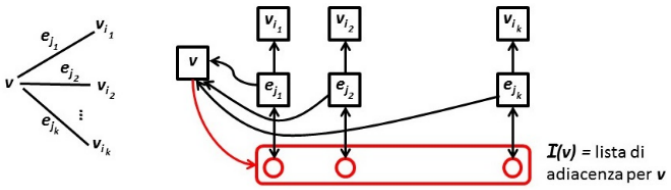


Figura 4: Liste di adiacenza.

L'uso delle liste di adiacenza, che è il più frequente, consente l'accesso veloce ai vicini di un vertice e richiede, occupando *spazio lineare*, $\mathcal{O}(n+m)$ *nella taglia del grafo.*

* **Matrice di Adiacenza (Adjacency Matrix)** A: matrice $n \times n$ tale che i vertici sono in corrispondenza 1-1 con righe e colonne e

$$A[i_1, i_2] \doteq \begin{cases} null & \text{se } (i_1, i_2) \notin E \\ \text{puntatore a } e = (i_1, i_2) \in L_E & \text{se tale lato esiste} \end{cases}$$

L'uso della matrice di adiacenza consente di testare velocemente la presenza di un lato e accedere alle sue informazioni, ma richiede spazio quadratico nei vertici che può risultare superlineare nella taglia del grafo. Per questo motivo è una rappresentazione utilizzata soprattutto nel caso di grafi densi (con numero di lati quadratico nei vertici).

3 Visite di grafi e loro applicazioni

Def. 3.1. ***(Graph traversal)*** *Un graph traversal è una procedura sistematica per esplorare un grafo $G = (V, E)$ a partire da un vertice $s \in V$ visitando tutti i vertici del grafo. Ogni graph traversal deve avere in input almeno: un grafo e il vertice di partenza.*

Esistono generalmente due algoritmi per l'esplorazione di un grafo:

* **Depth-First Search (DFS)**: dopo la visita di un vertice si visita un vicino, poi un vicino del vicino, ecc. Visita un grafo in modo aggressivo: si va in profondità, poi torna indietro, e così via;
* **Breadth-First Search (BFS)**: dopo la visita di un vertice si visitano tutti i vicini prima di passare ai vicini dei vicini: modalità non aggressiva che procede per livelli del grafo.

Osservazioni:

* Come nel caso delle viste degli alberi, DFS e BFS costituiscono dei *design pattern*, un building block, in cui l'operazione di visita può essere opportunamente istanziata in modo tale da risolvere specifici problemi: connettività, identificazione di spanning tree, ...;
* Anche scorrendo le liste L_V e L_E si ottiene un'esplorazione completa del grafo. Tuttavia la non sistematicità di tale esplorazione la rende poco sfruttabile per la risoluzione di problemi.

Idea:

* L'idea alla base dei due algoritmi è che, dato il vertice di partenza, quando c'è da scegliere quale dei vicini da visitare, si predilige per primo quello con `ID` minore;
* **DFS:** è aggressivo, quindi una volta visitato un vicino passo al vicino del vicino. Va giù in profondità;
* **BFS:** visito tutti i vicini di un nodo, poi scendo in profondità.

3.1 Depth-First Search

C'è più di un modo per scrivere DFS: noi lo faremo in modo elegante tramite un algoritmo ricorsivo.

* Algoritmo ricorsivo che, a partire da un vertice s, visita tutti i vertici e i lati della **componente connessa** $C_s \subseteq G$ contenente s, toccando tutti i vertici e i lati di C_s;
* Oltre alle strutture di base L_V e L_E assumiamo di usare le liste di adiacenza che inducono un ordine di visita dei vicini di ogni vertice. Useremo questo ordine per determinare l'ordine di visita dei vertici;
* Ogni vertice v ha un campo $L_V[v].ID$ che vale 1 se v è stato visitato, 0 altrimenti;
* Ogni lato e ha un campo $L_E[e].label$ che vale null se e non ha ancora etichetta, oppure riporta una delle due etichette `DISCOVERY EDGE` o `BACK EDGE`;
* In un qualsiasi istante dell'esecuzione dell'algoritmo diremo che un vertice u è *discoverable* da un vertice v se esiste un cammino da v a u fatto di vertici non ancora visitati.

Algoritmo DFS(G, v) (prima invocazione: $v = s$) con s vertice di partenz (*start*)
Input: grafo $G = (V, E)$ non diretto, vertice $v \in V$ non visitato
Output: tutti i vertici discoverable[2] da v visitati e i lati incidenti su di essi etichettati come `DISCOVERY` o `BACK EDGE`.

[2] visitabili

3.1.1 Algoritmo DFS

//visita il vertice v e lo segnalo settando il campo ID a 1
$L_V[v].ID \leftarrow 1$;
//per tutti i lati incidenti a v
forall $e \in G.incidentEdges(v)$ **do**
  //se quel lato non è stato ancora etichettato
  if($L_E[e].label = null$) **then**
    //considera il nodo opposto a v su quel lato
    $w \leftarrow G.opposite(v, e)$
    //se questo nodo non è stato ancora visitato
    if($L_V[w].ID = 0$) **then**
      //aggiorno l'etichetta del lato come DISCOVERY EDGE e...
      $L_E[e].label \leftarrow$ `DISCOVERY EDGE`
      //...lo visito
      `DFS`(G, w)
    //se quel vicino (w) era già stato visitato, allora lo etichetterò come BACK EDGE
    else $L_E[e].label \leftarrow$ `BACK EDGE`

Osservazioni:

* Un lato che è stato marchiato come `DISCOVERY EDGE` significa che è stato utilizzato per "scoprire" un altro nodo, mentre un lato marchiato come `BACK EDGE` è stato utilizzato per fare "marcia indietro" nel processo di visita di tutti i nodi;
* `incidentEdges`(v) restituisce un iteratore ai vicini di v;
* `opposite`(v, e) restituisce il vertice di e opposto a v;
* Si assume che nella prima invocazione `DFS`(G, v) tutti i vertici siano non visitati ($L_V[v].ID = 0 \ \forall \ v \in V$) e tutti i lati non etichettati ($L_E[e].label = null \ \forall \ e \in E$);
* In una generica invocazione `DFS`(G, v) alcuni vertici della componente connessa di v possono essere già visitati e alcuni già etichettati.

3.1.1.1 Analisi

Ricapitolando:

* $G = (V, E)$ grafo non diretto, $s \in V$
* $C_s \subseteq G$ componente connessa di G contenete s.

Proposizione: si supponga di eseguire `DFS`(G, v) quando nessuno dei vertici/lati di C_s è stato visitato/etichettato. Alla fine dell'esecuzoine si ha che:

1. Tutti i vertici di C_s sono visitati e tutti i lati di C_s sono etichettati come `DISCOVERY` o `BACK EDGE`;
2. I `DISCOVERY EDGE` formano uno spanning tree T di C_s radicato in s e chiamato **DFS Tree**.

Dimostrazione (proposizione 1):

Si noti anzitutto che un vertice u viene visitato solo quando si invoca `DFS(G,u)`. Per assurdo, supponiamo che esista un vertice $v \in C_s$ che non sia stato visitato. Dato che v ed s sono nella stessa componente connessa, deve esistere un cammino

$$s(= u_0) - u_1 - u_2 - \cdots - u_\ell(= v),$$

con $u_j \in C_s \ \forall j$. Dato che sicuramente s è visitato, se v non è visitato, nel cammino deve esistere un primo vertice non visitato u_i. Ma questo è un assurdo perché `DFS(`G, u_{i-1}`)` deve essere eseguita e, trovando u_i non visitato, al suo interno invocherebbe `DFS(`G, u_i`)`.
Chiaramente, se `DFS` è invocata su tutti i vertici di C_s, allora tutti i lati incidenti su tali vertici (ovvero tutti i lati di C_s) devono essere alla fine etichettati come `DISCOVERY` o `BACK EDGE`.

Dimostrazione (proposizione 2):
Dal punto precedente sappiamo che `DFS(`G, v`)` è invocata su tutti i vertici $v \in C_s$. È facile vedere che su ciascun vertice viene invocata una sola volta e che $\forall\ v \in C_s$, con $v \neq s$, deve esistere un vertice u tale che il lato (u, v) esiste e viene etichettato come `DISCOVERY EDGE` e `DFS(`G, v`)` viene invocata da `DFS(`G, u`)`. In questo caso diciamo che v viene "scoperto" da u e definiamo u padre di v.
Di conseguenza, $\forall\ v \in C_s$, con $v \neq s$ esiste un unico padre e, risalendo di padre in padre, si risale attraverso le invocazioni di `DFS` indietro nel tempo e ci si può fermare solo su `DFS(`G, s`)`.
$\Longrightarrow$ i `DISCOVERY EDGE` formano un **rooted tree** con radice in s che tocca tutti ivertici di C_s, ed è quindi uno spanning tree. □

3.1.1.2 Complessità

Definiamo:
n_s = numero di vertici in C_s
m_s = numero di lati in C_s

Albero della ricorsione per DFS(G, s):

- I nodi corrispondono alle invocazioni DFS(G, v), esattamente un'invocazione $\forall$ vertice $\in C_s$;
- Il costo attribuito al nodo associato a DFS(G, v), escludendo le invocazioni ricorsive al suo interno, è $\Theta(degree(v))$.

$$\Longrightarrow \text{ Complessità } \in \Theta\left(\sum_{v \in C_s} degree(v)\right) \in \Theta(m_s)$$

perché la somma dei gradi di ogni nodo è proporzionale al numero di lati.

Osservazione: poiché C_s è connesso, $m_s \geq n_s - 1 \Rightarrow m_s \in \Omega(n_s)$.
Corollario: Se $G = (V, E)$ è connesso, `DFS(`G, s`)` ha complessità $\Theta(m),\ \forall\ s \in V$.
Osservazione: Sia T lo spanning tree di C_s, radicato in s, costituito dai lati etichettati come `DISCOVERY EDGE` dopo l'esecuzione di `DFS(`G, s`)`. Supponiamo che durante una delle varie chiamate ricorsive `DFS(`G, v`)`, fatta durante l'esecuzione dell'algoritmo, si etichetti un lato (v, w) come `BACK EDGE`. Allora w deve essere un antenato di $v \in T$ in base alle seguenti considerazioni:

- Per etichettare (v, w) come `BACK EDGE` deve essere $L_V[w].ID = 1$ e quindi DFS(G, w) deve essere stata già invocata;
- Tuttavia DFS(G, w) deve essere iniziata ma non conclusa, altrimenti (v, w) avrebbe già un'etichetta diversa da `null` quando DFS(G, v) lo esamina;

* Ne consegue che esiste una sequenza di vertici $w = w_1, w_2, ..., w_k = v$ tali che DFS(G, w_{i+1}) è invocata direttamente da DFS(G, w_i) $\forall\ 1 \leq i < k$, e quindi $(w_1, w_2)(w_2, w_3), ..., (w_{k-1}, w_k)$ è un cammino di discovery edge da w a v, ovvero w è antenato di v.

Questa osservazione giustifica la locuzione `BACK EDGE`.

3.1.1.3 Visita di tutto il grafo

Si noti che l'algoritmo DFS(G, s) visita solo la componente connessa di s. Il seguente pseudocodice può essere utilizzato come *design pattern* per estendere la visita a tutto il grafo, nel caso esso non sia connesso.

for $v \leftarrow 1\ to$ n **do**
  $L_V[v].ID \leftarrow 0$;
for $v \leftarrow 1\ to$ n **do**
  if($L_V[v].ID = 0$) **do**
    `DFS(`G, v`)`

Analisi

Sia c il numero di componenti connesse di G. Si osservi che:

* Nel secondo ciclo for, l'invocazione `DFS(`G, v`)` verrà fatta *esattamente* c volte su vertici di componenti connesse distinte, dato che ogni invocazione `DFS(`G, v`)` imposta il campo `ID` di ciascun vertice della componente connessa di v a 1;

* Sia m_j il numero di lati della j-esima componente connessa $\forall\ 1 \leq j \leq c$, e si noti che $m = \sum_{j=1}^{c} m_j$. Il costo aggregato di tutte le invocazioni di `DFS` è uguale a $\mathcal{O}\left(\sum_{j=1}^{c} m_j\right) = \mathcal{O}(m)$, mentre il costo delle altre operazioni fatte dall'algoritmo è $\mathcal{O}(n)$.
  $\Longrightarrow$ La complessità è $\mathcal{O}(n + m)$, non solo di $\mathcal{O}(n)$. Considerate di avere un grafo con 100 nodi e 4 lati. La complessità di questo algorimo non è proporzionale a 4 perché devo visitare tutti i nodi. La complessità sara $\mathcal{O}(100 + 4)$.

3.1.2 Applicazioni

3.1.2.1 Connettività

Dato $G = (V, E)$ vogliamo determinare il numero di componenti connesse (se tale numero è 1, allora G è connesso).

Sia `DFS(`G, v, k`)` una modifica[3] di `DFS(`G, v`)` che sostituisce l'istruzione $L_V[v].ID \leftarrow 1$ con $L_V[v].ID \leftarrow k$.

Il seguente algoritmo calcola il numero di componenti connesse di G e assegna ai vertici di ciascuna componente uno stesso identificativo.

`DFS(`G, v, k`)`
for $v \leftarrow 1\ to$ n **do**
  $L_V[v].ID \leftarrow 0$;
$k \leftarrow 0$

[3] *arricchimento*

for $v \leftarrow 1$ *to* n **do**
  if$(L_V[v].ID = 0)$ **do**
    $k \leftarrow k + 1$
    `DFS(`G, v`)`
return k

Analisi

Sia c il numero di componenti connesse di G. Si osservi che:

* Ragionando come nell'analisi precedente vediamo facilmente che nel secondo ciclo for l'invocazione `DFS(`G, v`)` verrà fatta esattamente c volte su vertici di componenti connesse distinte. Di conseguenza, alla fine del ciclo si avrà $k = c$ ($k = 1$ se G è connesso).
  $\Longrightarrow$ L'algoritmo è corretto

* sempre ragionando analogamente all'analisi precedente si dimostra facilmente che la complessità è $\mathcal{O}(n + m)$.

3.1.2.2 Spanning tree

Vogliamo trovare uno spanning tree di un grafo $G = (V, E)$ connesso. È sufficiente eseguire `DFS(`G, s`)` a partire da qualsiasi vertice s e restituire i discovery edge come lati dello spanning tree.
La correttezza è conseguenza immediata dell'anaisi di `DFS` fatta in precedenza, mentre la complessità è $\mathcal{O}(m)$ dato che si invoca `DFS` una sola volta.

3.1.2.3 $s - t$ connectivity

Dato $G = (V, E)$ e due vertici $s, t \in V$ vogliamo determinare, se esiste, un cammino da s a t.
L'algoritmo è il seguente:

* $\forall\ w \in V$ utilizziamo un campo aggiuntivo $L_V[w].parent$;

* Modifichiamo `DFS(`G, v`)` in odo che quando etichetta un lato (v, w) come `DISCOVERY EDGE` imposti $L_V[w].parent$ a v (ovvero v è padre di w nell'albero dei discovery edge;

* Eseguiamo `DFS(`G, s`)`. Alla fine, se t non è stato visitato si dice in output che non esiste un cammino da s a t, altrimenti partendo da t e risalendo di padre in padre si costruisce il cammino e lo si restituisce in output.

L'algoritmo è corretto (vedi dimostrazioni precedenti) ed ha complessità $\mathcal{O}(m_s)$ dove m_s è il numero di lati nella componente connessa di s.

3.1.2.4 Ciclicità

Dato $G = (V, E)$ vogliamo determinare un ciclo, se esiste. L'algoritmo è il seguente:

* $\forall\ u \in V$ usiamo un campo aggiuntivo $L_V[u].parent$ e $\forall\ e \in E$ usiamo un campo aggiuntivo $L_E[e].ancestor$;

* Facciamo le seguenti modifiche a `DFS(`G, v`)`:

  – quando etichetta un lato (v, u) come `DISCOVERY EDGE` imposta $L_V[u].parent$ a v;

  - quando etichetta un lato $e = (v, w)$ come `BACK EDGE` imposta $L_E[e].ancestor$ a w, per rappresentare il fatto che w è antenato di v nell'albero dei discovery edge (come osservato in precedenza).

* Eseguiamo la `DFS` su ciascuna componente connessa del grafo (come per il calcolo de numero di componenti connesse);

* Eseguiamo un ciclo su tutti i lati. Appena si trova un lato $e = (v, w)$ etichettato come `BACK EDGE` e con $L_E[e].ancestor = w$ antenato di v, si costruisce un ciclo aggiungendo a (v, w) i lati incontrati risalendo da v a w di padre in padre, e lo si restituisce in output. Se invece non si trova alcun `BACK EDGE` si restituisce in output che il grafo è aciclico.

La complessità è $\mathcal{O}(n + m)$ dato che si invoca la `DFS` una volta $\forall$ componente connessa.

3.1.3 Proprietà dimostrate

Abbiamo finora dimostrato la seguente proposizione: dato un grafo $G = (V, E)$ con $|V| = n$ e $|E| = m$, i seguenti problemi possono essere risolti in tempo $\mathcal{O}(m + n)$ usando la `DFS`:

* testare se G è connesso;

* evidenziare le componenti connesse di G;

* trovare uno spanning tree di G, se G è connesso;

* trovare un cammino tra due vertici s e t, se esiste;

* trovare un ciclo, se esiste.

Gli **esercizi** relativi a questa parte del corso si trovano in §A.1.1 e §A.1.2.

3.2 Breadth First Search

* Algoritmo iterativo che a partire da un vertice s "visita" tutti i vertici della componente connessa $C_s \subseteq G$ contenente s, toccando tutti i vertici e i lati di C_s e partizionando i vertici in livelli L_i in base alla loro distanza i da s;
* Oltre alle strutture di base L_V e L_E assumiamo di usare le liste di adiacenza che inducono un ordine di visita dei vicini di ogni vertice;
* Ogni vertice v ha un campo $L_V[v].ID$ che vale 1 se v è stato visitato, 0 altrimenti;
* Ogni lato e ha un campo $L_E[e].label$ che vale `null` se e non ha ancora etichetta, oppure riporta una delle due etichette `DISCOVERY EDGE` o `CROSS EDGE`.

3.2.1 Algoritmo BFS

Input: grafo grafo $G = (V, E)$ non diretto, vertice $s \in V$ non visitato
Output: tutti i vertici in C_s visitati e i lati etichettati come `DISCOVERY` o `CROSS EDGE`.

```
BFS(G,s)
//visita il vertice s
L_V[s].ID ← 1
//crea una collezione L_0 contenente s
i ← 0
//fintanto che esistono nodi in quella lista
while (!L_i.isEmpty()) do
    //crea una collezione di vertici L_{i+1} vuota.
    //per tutti i vertici della lista L_i...
    forall v ∈ L_i do
        //...considero tutti i lati incidenti a quel vertice v
        forall e ∈ G.incidentEdges(v) do
            //se un lato non è ancora stato etichettato...
            if(L_E[e].label = null) then
                //...considero il suo vertice opposto
                w ← G.opposite(v,e)
                //se w non è ancora stato visitato...
                if (L_V[w].ID = 0) then
                    //...dò a quel lato la label di DISCOVERY EDGE
                    L_E[e].label ← DISCOVERY EDGE
                    //lo visito
                    L_V[w].ID ← 1
                    //inserisco w nella lista L_{i+1}
                    L_{i+1}.insert(w)
                //il nodo rappresenta un passaggio tra diversi "livelli"
                //(tra L_i, L_{i-1} oppure tra L_I, L_{i+1})
                else
                    L_E[e].label ← CROSS EDGE
    i ← i + 1
return
```

3.2.1.1 Analisi

Il grafo $G = (V, E)$ non diretto, $s \in V$
$C_s \subseteq G$ la componente connessa di G contenente s.
Proposizione: si supponga di eseguire `BFS(G,s)` quando nesuno dei vertici/lati di C_s è stato visitato/etichettato. Alla fine dell'esecuzione si ha che:

1. tutti i vertici di C_s sono visitati e tutti i lati di C_s sono etichettati come `DISCOVERY` o `CROSS EDGE`;
2. i `DISCOVERY EDGE` formano uno spanning tree T di C_s radicato in s e chiamato **BFS tree**;
3. $\forall\ v \in L_i$ il cammino in T da s a v ha i lati e qualsiasi altro cammino in G da s a v ha $\geq i$ lati ($\Rightarrow i \equiv distanza(s, v)$);
4. se $(u, v) \in E$ e $(u, v) \notin T$ ($\Rightarrow (u, v) \equiv$ `CROSS EDGE`) gli indici dei livelli di u e v differiscono al più di 1

Dimostrazione (Proposizione 1)

Analoga a `DFS`.

Dimostrazione (Proposizione 2)

Analoga a `DFS`.

Dimostrazione (Proposizione 3)

Si consideri un cammino $P : s = u_0 - u_1 - \cdots - u_i = v$ dove $u_j \in L_j$ viene "scoperto" da u_{j-1}, $\forall\ j\ :\ 1 \leq j \leq i$. Quindi (u_{j-1}, u_j) è un `DISCOVERY EDGE` e, di conseguenza, P è un cammino di T.
Supponiamo per assurdo che esista in G un cammino

$$P'\ :\ s = z_o - z_1 \cdots - z_t = v$$

con $t < i$ (più breve), si avrebbe che:

$$\begin{aligned} s = z_0 &\in L_0 \\ z_1 &\in L_1 \\ \vdots \quad &\quad \vdots \\ z_t = v &\in L_1 \text{ o } \cdots \text{ o } L_t \end{aligned}$$

$\Rightarrow v \notin L_i$: assurdo. □
In sintesi, quello che si ha è che il primo nodo sta nel livello zero, così via finché il nodo t-*esimo* sta in qualche lista a cavallo tra L_1 ed L_t. Se questo fosse vero, allora abbiamo che $v \notin L_i$, perché abbiamo che $t < i$. Ma questo è un assurdo perché abbiamo ipotizzato che $v \in L_i$.

Dimostrazione (Proposizione 4)

Per assurdo, se $u \in L_i$ e $v \in L_{i+k}$ con $k > 1$, v sarebbe scoperto a partire da u e quindi si avrebbe $v \in L_{i+1}$ e non $v \in L_{i+k}$: assurdo. □

3.2.2 Complessità

La complessità di **BFS** è lineare nel il numero di lati della componente connessa.
Definiamo n_s il numero di vertici in C_s ed m_s il numero di lati in C_s. Supponiamo di rappresentare ogni L_i tramite una lista:

* $\forall\ v \in C_s$ viene eseguita *esattamente 1 iterazione* del primo ciclo **forall** ed esattamente $degree(v)$ iterazioni del secondo ciclo **forall**;
* ciascuna iterazione del ciclo **forall** richiede tempo $\Theta(1)$;
* tutti gli accessi alle L_i richiedono tempoo $\Theta(1)$.

$\Rightarrow$ complessità di `BFS(G,s)` $\in \Theta(m_s)$.

Corollario: se il grafo $G = (V, E)$ è connesso, `BFS(G,s)` ha complessità $\Theta(m)\ \forall\ s \in V$.

3.2.3 Applicazioni

Dato il grafo $G = (V, E)$ con $|V| = n$ ed $|E| = m$, in modo del tutto analogo a quanto fatto con la `DFS` possiamo usare la `BFS` per risolvere i seguenti probemi in tempo $\mathcal{O}(n + m)$:

* Visitare tutto G (anche ne caso non sia connesso);
* Determinare il numero di componenti connesse di G;
* Se G è connesso, trovare uno spanning tree di G.

3.2.3.1 Cammini minimi

Dato il grafo grafo $G = (V, E)$ e due vertici $s, t \in V$ vogliamo determinare, se esiste, un cammino di lunghezza minima da s a t. L'algoritmo è il seguente:

* $\forall\ u \in V$ usiamo un campo aggiuntivo $L_V[u].parent$;
* Modifichiamo `BFS(G,s)` in modo che quando etichetta una lato (v, u) come `DISCOVERY EDGE` imposti $L_V[u].parent$ a v (ovvero v è padre di u nell'albero dei `DISCOVERY EDGE`);
* Eseguiamo `BFS(G,s)`. Alla fine, se t non è stato visitato si dice in output che non esiste un cammino da s a t, altrimenti, partendo da t e risalendo di padre in padre si contruisce il cammino e lo si restituisce in output

```
BFSShortestPath(G,s)
forall (v ∈ V)
    L_V[v].parent ← null
//visita il vertice s
L_V[s].ID ← 1
//crea una collezione L_0 contenente s
i ← 0
//fintanto che esistono nodi in quella lista
while (!L_i.isEmpty()) do
    //crea una collezione di vertici L_{i+1} vuota.
    //per tutti i vertici della lista L_i...
    forall v ∈ L_i do
        //...considero tutti i lati incidenti a quel vertice v
        forall e ∈ G.incidentEdges(v) do
            //se un lato non è ancora stato etichettato...
            if(L_E[e].label = null) then
                //...considero il suo vertice opposto
                w ← G.opposite(v,e)
                //se w non è ancora stato visitato...
                if (L_V[w].ID = 0) then
                    //...dò a quel lato la label di DISCOVERY EDGE
                    L_E[e].label ← DISCOVERY EDGE
                    //il campo padre di w è v: è ora possibile risalire
                    L_V[w].parent ← v
                    //lo visito
                    L_V[w].ID ← 1
                    //inserisco w nella lista L_{i+1}
                    L_{i+1}.insert(w)
            //il nodo rappresenta un passaggio tra diversi "livelli"
            //(tra L_i, L_{i-1} oppure tra L_I, L_{i+1})
            else L_E[e].label ← CROSS EDGE
    i ← i + i
return
```

Analisi

È facile vedere che l'algoritmo è corretto ed ha complessità $\mathcal{O}(m_s)$, dove m_s è il numero di lati nella componente connessa di s.

3.2.3.2 Ciclità

Dato il grafo grafo $G = (V, E)$ vogliamo determinare un ciclo, se esiste. Si osserva che esiste un ciclo se e solo se, visitando tutto G con l'algoritmo `BFS`, uno dei lati viene etichettato come `CROSS EDGE`.

```
FindCycle(G,s)
for i ← 1 to n do
    L_V[v].ID ← 0
BSD(G,s)
forall e ∈ E do
    if (L_E[e].label = CROSS EDGE) do
        return true
return false
```

3.2.4 Proprietà dimostrate

Abbiamo finora dimostrato la seguente proposizione: dato un grafo $G = (V, E)$ con $|V| = n$ e $|E| = m$, i seguenti problemi possono essere risolti in tempo $\mathcal{O}(m + n)$ usando il `BFS`:

* testare se G è connesso;
* evidenziare le componenti connesse di G;
* trovare uno spanning tree di G, se G è connesso;
* trovare un cammino tra due vertici s e t, se esiste;
* trovare un ciclo, se esiste.

4 Minimum Spanning Tree

(Cap. 23 Cormen). In molte applicazioni dobbiamo interconnetere un insieme di oggetti nel modo più economico possibile. Per esempio, componenti in un circuito elettronico usando la minor quantità di cavo elettrico → calcolo di un *Minimum Spanning Tree (MST)*, uno dei problemi computazionali più importanti e più studiati in informatica.

Def. 4.1. ***(Grafo pesato)*** *Un grafo pesato è un grafo $G = (V, E)$ a cui è associata la funzione $w : E \to \mathbb{R} \mid w(u, v) =$ costo del lato (u, v). Più grande è il peso, più grande è il costo che devo pagare per effettuare uno spostamento su quel lato.*

Def. 4.2. ***(Problema di MST)*** *Si definisce il problema di MST come segue:*

- *in **Input** viene dato:*
  - *un grafo $G = (V, E)$ non diretto, connesso e pesato.*
- *in **Output** bisogna restituire:*
  - *uno **spanning tree** T che tocca tutti i nodi di G che deve **minimizzare** la sommatoria*

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

cioè la somma di tutti i pesi di tutti i lati dell'albero.

Un paio di precisazioni:

- MST è un'abbreviazione di **Minimum-weight Spanning Tree**, cioè albero di copertura a peso minimo, quindi da non confondere con *e.g.* il minumo numero di lati, perché TUTTI gli spanning tree hanno lo stesso numero di lati e noi vogliamo quello che pesa meno.
- se il grafo non è connesso si pala di **minumum spanning forest** → un MST $\forall$ componente connessa.

Perché parlare di MST?

Nell'ambito delle **reti**, l'idea è di determinare una *backbone*, ovvero un sottografo che connetta tuti i nodi della rete e che sia di costo minimo.

Nell'ambito del **machine learning**, il calcolo del MST è un building block per acuni algoritmi di *clustering*.

Nell'ambito della **Computer Vision**, del **Data Mining**. Viene anche, spesso, usata come *subroutine* in alcuni approximation algorithms, ad esempio per il problema del TSP, del Commesso Viaggiatore, ...

4.1 Difficoltà di determinare un MST

Il numero possibile di soluzioni candidate, nel caso peggiore, è **esponenziale** nella taglia dell'input.

→ ad esempio il grafo completo[4] ha esattamente n^{n-2} spanning trees diversi. Quando $n \geq 50,\ n^{n-2}$ è maggiore del numero stimato di atomi nell'universo conosciuto. Come trovare un ago in un pagliaio.

Sorprendentemente, questo problema si può risolvere in tempo quasi lineare. Non solo: questi algoritmi utilizzano l'approccio *greedy*, sono quindi semplici da capire ed implementare in pratica.
2 algoritmi diversi, **Prim** e **Kruskal**.

4.2 Algoritmo generico per MST

Invariante

Ad ogni iterazione, A è un sottoinsieme di lati di un qualche MST. Facciamo $n - 1$ volte un ciclo in cui aggiungiamo ad uno ad uno i lati. Al termine di ogni iterazione quell'insieme di lati è un sottoinsieme di lati di un qualche MST. Quindi esiste un MST nel grafo tale per cui il sottoinsieme A è un sottoinsieme di lati di quel MST.
Ad ogni iterazione, viene aggiunto un lato che non viola l'invariante ⇒ questo lato viene detto "safe" for A.

Algoritmo

`Generic-MST`
$A \leftarrow \emptyset$
while A does not form a spanning tree
    find an edge (u, v) that is safe for A // parte cruciale
    $A \cup \{(u, v)\}$
return A

Def. 4.3. ***(Taglio)*** *Un taglio $(S, V \setminus S)$ di un grafo $G = (V, E)$ è una partizione nei due insiemi $\{S\}$ e $\{V \setminus S\}$ di V.*

Def. 4.4. ***(Attraversamento)*** *Un lato (u, v) attraversa il taglio $(S, V \setminus S)$ se $u \in S$ e $v \in \{V \setminus S\}$ o viceversa.*

Def. 4.5. ***(Rispetto del taglio)*** *Un taglio rispetta un insieme A di lati se nessun lato di A attraversa un taglio.*

Def. 4.6. ***(Light Edge)*** *Dato un taglio, un[5] lato di peso minimo che attraversa un lato si chiama "light edge"*

Trovare un lato *safe*

Teorema 4.1. *Sia un grafo $G = (V, E)$ non diretto, connesso e pesato. Sia $A \subseteq E$ e incluso in un qualche MST di G. Sia $(S, V \setminus S)$ un taglio che rispetta A. Sia (u, v) un light edge per $(S, V \setminus S)$. Allora (u, v) è safe per A.*

[4] avente tutti gli $\binom{n}{2}$ lati possibili
[5] potrebbero esserci due lati con lo stesso peso

Come spesso si usa nelle dimostrazioni degli algoritmi greedy, utilizziamo un "argomento di scambio", un "cut and paste". Questa tecnica fa finta di prendere una soluzione ottima e la trasforma pezzo dopo pezzo nella soluzione che è ritornata dall'algoritmo. Si dimostra quindi che il costo delle due soluzioni è uguale e quindi anche la soluzione ritornata dall'algoritmo è ottima.

Dimostrazione. Partiamo da una soluzione ottima. Sia T un MST che include A. Assumiamo che $(u, v) \notin T$. Si costruisce un nuovo MST T' che include $A \cup \{(u, v)\}$, dove (u, v) è safe per A perché T' è un MST.
Aggiungendo (u, v) a T si crea un ciclo, perché T è uno ST. Per ipotesi, (u, v) attraversa $(S, V \setminus S)$.
$\Rightarrow$ ci deve essere un altro lato in T che attraversa il taglio (se non ci fosse vorrebbe dire che T non era uno ST).
Sia $(x, y) \in T$ un lato che attraversa il taglio $(S, V \setminus S)$. Per ipotesi, $(S, V \setminus S)$ rispetta A, quindi non lo partiziona.
$\Rightarrow (x, y) \notin A$ e togliendo (x, y) ottengo un nuovo ST $T' = T \setminus \big\{(x, y)\big\} \cup \big\{(u, v)\big\}$ che include $A \cup \big\{(u, v)\big\}$ come desiderato.
Per dimostrare che T', è anche minimum-weight, si utilizza la terza ipotesi del teorema. Entrambi (u, v) e (x, y) attraversano il taglio ma il light edge tra i due è (u, v):
$\Rightarrow w(u, v) \leq w(x, y) \Rightarrow w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.
Ma T è un MST $\Rightarrow w(T) \leq w(T') \Rightarrow w(T) = w(T')$. □

4.3 Algoritmo di Prim

(1957). Dato il grafo $G = (V, E)$ ed un nodo qualsiasi di partenza $s \in V$, definiamo:

`Prim(G,s)`
$X \leftarrow \{s\}$
$A \leftarrow \emptyset$
//finché esiste un lato che collega X con l'esterno
while $\exists\ (u, v) \in E$ con $u \in X$ e $v \notin X$ **do**
    //considero tutti i lati di questo tipo e prendo quello di costo minimo
    $(u^*, v^*) \leftarrow light\ edge$
    //aggiungo il nodo esterno ad X
    $X \leftarrow X \cup \{v^*\}$
    //aggiungo il lato appena trovato ad A
    $A \leftarrow A \cup \{(u, v)\}$
//A è ora un MST
return A
Questo algorimo greedy "cresce" uno spanning tree da un nodo di partenza s aggiungendo un light edge a lui collegato alla volta.

4.3.1 Complessità

La complessità di Prim dipende da come viene implementato. Il passo $(u^*, v^*) \leftarrow light\ edge$ è scritto troppo ad alto livello per poterne capire la complessità. Assumiamo di volerlo fare in modo naïve, qual è la complessità di questo algoritmo assumendo che il grafo sia rappresentato con liste di adiacenza?
La risposta corretta è $\mathcal{O}(m \cdot n)$: nel caso peggiore ogni nodo del grafo è direttamente connesso con qualsiasi altro nodo, quindi ad ogni iterazione delle $n - 1$ necessarie (altrimenti non sarebbe ST) occorre confrontare tutti gli m lati per trovare il light edge.

4.3.2 Prim con heap

L'algoritmo di Prim è quindi polinomiale nella taglia dell'input, quindi efficiente (invece di una complessità esponenziale per eseguire una ricerca esaustiva su tutti i ST).
Si può fare di meglio?
Osservazione chiave: la soluzione banale fa ripetutamente il calcolo di un minimo attraverso una ricerca esaustiva: sto ripetutamente facendo un'operazione secondo modalità *brute force*, cioè ripetitivamente. Bisogna migliorare qui.
Velocizzando il calcolo di un minimo, velocizziamo automaticamente anche l'implementazione. La struttura **Heap** ha come scopo il supportare il calcolo veloce di un minimo.

Dal Codice d'Oro del Bravo Algoritmico:

> "*Quando un algoritmo fa ripetutamente lo stesso calcolo, occorre cercare la struttura dati adatta che vi permette di velocizzare quel calcolo.*"
>
> Michele Scquizzato - 20 Marzo 2020

Uno heap permette operazioni di:

* **Insert:** aggiunge un oggetto allo heap;
* **Extraction:** toglie dallo heap un oggetto con chiave più piccola;
* **Delete:** dato un puntatore ad un oggetto, lo toglie dallo heap;
* **Min:** restituisce il minore tra gli oggetti contenuti nello heap senza toglierlo.

A parte dell'operazione `Min` che viene eseguita in $\mathcal{O}(1)$, le altre operazioni hanno tutte complessità $\mathcal{O}(\log n)$.
Scriviamo quindi un'implementazione efficiente dell'algoritmo di Prim:

`Prim(G,s)`
for each $u \in V$ **do**
    //key[u] rappresenta il peso minimo di un qualsiasi lato che connetta u all'albero
    $key[u] \leftarrow +\infty$
    //$\pi[u]$ *rappresenta il nodo padre di* u *nell'albero che sto costruendo*
    $\pi[u] \leftarrow$ `NULL`
//aggiorno il valore di $key[s]$ *a 0*
$key[s] \leftarrow 0$
//creo una coda di priorità Q *con tutti i nodi di* G
$Q \leftarrow V$

//finché Q non è vuota...
while $Q \neq \emptyset$ **do**
    //...estraggo il minimo
    //nel primo ciclo, u sarà sicuramente s perché è l'unico nodo con peso pari a 0
    //mentre gli altri hanno tutti peso $+\infty$
    //l'invariante che stiamo mantenendo è che u sia light edge per $\big(V \setminus Q, Q\big)$
    $u \leftarrow$ `extractMin(Q)`
    //dobbiamo controllare tutti i nodi adiacenti ad u e prendere il lato con peso minore
    for each v *adjacent to* u **do**
        if $v \in Q$ **and** $w(u,v) < key[v]$ **then**
            //sto includendo il lato v
            $\pi[v] \leftarrow u$
            //aggiorno la chiave con il peso del lato che connette v all'albero, cioè(u,v)
            $key[v] \leftarrow w(u,v)$

Durante l'esecuzione dell'algorimo, implicitamente $A = \big\{\big(v, \pi[v]\big) \mid v \in V \setminus \{s\} \setminus \{Q\}\big\}$

Complessità

* La complessità dell'algoritmo è determinata dal ciclo while, eseguito n volte. Ogni volta fa una `extractMin` che costa $\mathcal{O}(\log n)$. Quindi il costo totale dovuto alle sole operazioni di `extractMin` è $\mathcal{O}(n \log n)$.
* C'è anche un ciclo **for** che $\forall$ nodo considera i nodi adiacenti ad esso. Essi sono in quantità uguale al grado di quel nodo, e lo si fa per tutti i nodi; quindi $\sum_v degree(v) = 2m$. Quindi $\mathcal{O}(2m) = \mathcal{O}(m)$.
* $\forall$ volta che si è all'interno del ciclo **for** viene determinato se $v \in Q$, che si fa in $\mathcal{O}(1)$ ma soprattutto aggiorno $key[v]$ con il peso del lato (u,v), cioè il light edge. Questa operazione comprende 2 operazioni: `delete` + `insert`, che costano in totale $\mathcal{O}(2 \log n) = \mathcal{O}(\log n)$.
  $\Rightarrow$ Il costo totale delle operazioni interne al ciclo **for** che si evince è quindi $\mathcal{O}(m \log n)$.

Il **costo totale** di Prim utilizzando uno heap è $\mathcal{O}(n \log n + m \log n) = \mathcal{O}(m \log n)$.

Implementando l'algoritmo di Prim con un **Fibonacci heap**, la sua complessità scende ulteriormente: $\mathcal{O}(m + n \log n)$.

4.4 Algoritmo di Kruskal

(1956). Implementa il `Generic-MST` ponendo A come foresta, mentre il lato safe è un light edge che connette due componenti distinte della foresta.
Perché vedere un altro algorimo per trovare un MST?

* È un algoritmo famoso e semplice;
* È veloce come quello di Prim, sia in teoria che in pratica (se implementato propriamente)
* Ci dà la possibilità di studiare una nuova struttura dati: il **disjoint-set** (*a.k.a.* ***union-find***).

```
Kruskal(G)
A ← ∅
sort edges of G by cost
for each e ∈ E, in ascending order of cost do
    if A ∪ {e} is acyclic then
        A = A ∪ {e}
return A
```

Piccola ottimizzazione: fermare il ciclo quando A ha $n - 1$ lati.

Domanda da esame tipica di teoria: viene dato un grafo e viene chiesto di eseguire l'algoritmo di Kruscal e/o Prim e restituire l'output dell'esecuzione.

Complessità

Dipende da come viene implementato. Banalmente può essere implementato con una complessità di $\mathcal{O}(mn)$ perché:

* il sorting dei lati (ad esempio con `MergeSort`) viene eseguito in tempo $\mathcal{O}(m \log n)$ = $\mathcal{O}(m \log m)$;
* ciclo for: m iterazioni, in ognuna si valuta la ciclicità o meno del lato in esame e su n vertici → $\mathcal{O}(mn)$.

4.4.1 Implementazione con union-find

L'algoritmo di Kruscal si può implementare con complessità $\mathcal{O}(m \log n)$ come Prim, utilizzando la struttura dati **union-find**. Qual è l'operazione che viene ripetuta più spesso? La cycle-check, ovvero controllare se il lato e comporterebbe un ciclo nell'albero di partenza.

La struttura dati union-find gestisce **insiemi disgiunti** (partizioni) di oggetti: significa che un oggetto può stare in uno solo dei sottoinsiemi degli insiemi disgiunti presenti nell'union-find.

Le operazioni supportate sono:

* **Initialize:** dato un insieme X di oggetti, crea una struttura dati union-find dove X viene partizionato in tanti sottoinsiemi quanti sono gli oggetti x che contiene
  $\Rightarrow \mathcal{O}(n)$
* **Find:** permette di, dato un oggetto x , ritornare il nome dell'insieme che contiene x
  $\Rightarrow \mathcal{O}(\log n)$
* **Union:** permette di unire due insiemi x, y in un singolo insieme. Se $x \equiv y$ l'operazione non ha nessun effetto
  $\Rightarrow \mathcal{O}(\log n)$

L'idea alla base dell'algoritmo è di usare union-find per tenere traccia delle componenti connesse nella soluzione attuale. $A \cup \{(v, w)\}$ crea un ciclo $\Leftrightarrow v, w$ sono già nella stessa componente connessa.

```
Kruskal(G)
A ← ∅
//inizializzo union-find
U ←Initialize(V)
sort edges of E by cost
for each edge e = (v, w) in ascending order of cost do
    //se i vertici del lato si trovano in due componenti connesse diverse
    if Find(U,v) ≠ Find(U,w) then
        //∄ (v, w) in A quindi si può aggiungere
        A ← A ∪ {(v, w)}
        //Bisogna aggiornare la union-find
        Union(U,v,w)
return A
```

Complessità

- **inizializzazione:** $\mathcal{O}(n)$ + $\mathcal{O}(m \log n)$ (va bene anche $\mathcal{O}(m \log m)$;
- $2m$ operazioni di **Find:** $\mathcal{O}(m \log n)$;
- $n - 1$ operazioni di **Union:** $\mathcal{O}(n \log n)$;
- resto: aggiornamento di A $\mathcal{O}(m)$

$\Rightarrow \mathcal{O}(m \log n)$.

Implementazione di union-find

Array la quale è visualizzabile come insieme di **alberi diretti**. Ogni elemento dell'array ha un campo `parent(x)` che contiene l'indice nell'array dell'oggetto x.

Esempio 1.

index of x	`parent(x)`
1	4
2	1
3	1
4	4
5	6
6	6

nodi: (indici di) oggetti

arco(x,y) $\Leftrightarrow parent(x) = y$

Quindi: un insieme di oggetti è in corrispondenza con un insieme di alberi diretto nel "parent graph". Convenzione: nome dell'insieme = root di quell'albero.

- `Initialize(U)`
  for each $i \in U$ **do**
    //inizializza i nodi a genitori di sé stessi
    $parent(i) \leftarrow i$

* **Find**:
  Find ritorna la componente connessa che contiene il nodo da cercare. L'idea dietro alla procedura `Find` è di risalire di padre in padre fino ad arrivare ad una radice che è identificabile da un self-loop:

  1. si comincia dalla posizione di x nell'array, si attraversa l'array di padre in padre fino alla radice r dove $parent(r) = r$;
  2. ritornare r.

  Def. 4.7. ***(Depth)*** *La depth di un oggetto x è il numero di lati attraversati dalla procedura* `Find(U,x)`.

  La complessità di `Find(U,x)` è proporzionale alla profondità di x.
  Caso pessimo $\stackrel{?}{\Rightarrow} \mathcal{O}(n)$ → in realtà **dipende** da come implementiamo la `Union`, cioè da come vengono uniti gli alberi man mano. In ogni caso, l'altezza massima di un albero può essere $(n-1)$

* **Union**
  Union prende in input due oggetti e deve unire i due alberi che li contengono. Il modo più semplice per farlo è far **puntare** una delle due radici ad un nodo dell'altro albero.
  Dobbiamo decidere:

  – Quale delle 2 radici resta radice?
    * L'idea è minimizzare il numero di oggetti la cui profondità aumenta (→ implementazione "**union-by-size**"). Sceglieremo questa opzione.
    * far puntare la radice dell'albero più basso a quella dell'albero più alto (→ "**union-by-rank**")
  – A quale altro oggetto far puntare l'altra radice?
    La risposta è, in generale, la profondità degli oggetti nell'albero da innestare aumenta di tanto quanto è la profondità in cui si è scelto di innestare l'albero
    ⇒ la radice deve puntare l'altra radice, così da incrementare la profondità dei nodi dell'albero da innestare solo di 1.

`Union(x,y)`:

1. Invocare `i ← Find(x)` e `j ← Find(y)` per capire in che albero stanno. Se stanno nello stesso albero non c'è nulla da fare;
2. **Se** $size(i) \geq size(j)$
   * **Allora:**
     $parent(j) \leftarrow i$
     $size(j) \leftarrow size(i) + size(j)$
     aggiornare la size tutti gli oggetti presenti nell'albero radicato in j
   * **Altrimenti:**
     $parent(i) \leftarrow j$
     $size(j) \leftarrow size(i) + size(j)$
     aggiornare la size tutti gli oggetti presenti nell'albero radicato in i

La **complessità** di `Find` (e quindi di `Union`) è $\mathcal{O}(\log n)$

Dimostrazione. Ogni oggetto x ha inizialmente $depth(x) = 0$. La $depth$ di x può aumentare solo a causa di una `Union` in cui la radice dell'albero di x nel parent graph viene attaccata ad un'altra radice. Questo succede solo quando l'albero di x viene unito ad un albero di dimensione non inferiore. Quindi quando la depth di x aumenta, la dimensione dell'albero di x quantomeno raddoppia. Siccome la dimensione di un albero è $\leq$, la $depth$ di x non può aumentare più di $\log_2 n$ volte. E dato che la complessità di `Find(x)` è proporzionale a $depth(x)$, questa è $\mathcal{O}(\log n)$. □

4.5 Algoritmi super veloci per MST

Una breve ricapitolazione di algoritmi per MST e loro complessità:

* Prim(1957)/Kruskal(1956) $\mathcal{O}(m \log n)$
* Fredman-Tarjan(1984) (Prim + Fibonacci heap) $\mathcal{O}(m + n \log n)$
* Fredman-Tarjan(1984) $\mathcal{O}(m \log^* n)$ $(\log^* 2^{65536} = 5)$
* Gabow,etc(1986) $\mathcal{O}(m \log \log^* n)$
* Chazelle(2000) $\mathcal{O}(m\, \hat{\alpha}(m,n))$ $(\hat{\alpha}(2^{65536}, 2^{65536}) \leq 4)$
* ? $\mathcal{O}(m)$

$$\log^* n := \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Gli esercizi relativi a questa parte del corso si trovano in §A.2.1

5 Shortest Path

Problema: dato un grafo $G = (V, E)$ **pesato** e **diretto**, una sorgente $s \in V$ e una destinazione $t \in V$, calcolare il cammino da s a t più corto. La lunghezza (peso) del cammino si indica con $dist(s, t)$.

Le applicazioni principali sono:

* **Google Maps**;
* **Routing** su reti;
* **Navigazione di robot**;
* ...

5.1 Single-Source Shortest Path

Denotato con la sigla **SSSP**, il problema può essere formulato come segue:

Def. 5.1. ***(Problema di SSSP)***
Input:

* *un grafo $G = (V, E)$;*
* *un vertice sorgente $s \in V$*
* *una lunghezza $\ell_e \geq 0 \ \forall \ e \in E$*

Output:

* *$dist(s, v)$ = lunghezza totale del cammino più breve tra s e $v \ \forall \ v \in V$*

Un paio di commenti:

* Lavoreremo con grafi diretti ma gli algoritmi che vedremo funzionanp anche per grafi non diretti (con piccole modifiche "cosmetiche";
* L'assunzione $\ell_e \geq 0 \ \forall \ e \in E$ è significativa. È vera in moltissime applicazione ma non tutte!

Perché non usare BFS? Abbiamo già visto che una delle applicazioni della BFS è calcolare shortest paths...però inteso come un numero di lati del path! Questo è un caso speciale di SSSP dove ogni lato ha peso $\ell_e = 1$.
Idea: pensare ad un lato "lungo" ℓ come ad un cammino di ℓ lati ognuno lungo uno → **riduzione** da un problema ad un altro: in questo caso, da SSSP con lunghezze intere al caso speciale dove ogni lato ha $\ell_e = 1$.

* **Primo problema:**
  - lunghezze intere, quindi è ancora un caso speciale.
* **Secondo problema:**
  - La dimensione del grafo può aumentare tantissimo (la lunghezza di un lato può essere $\gg n, m$.

5.1.1 Algoritmo di Dijkstra

(1956) Molto simile all'algoritmo di Prim. Approccio Greedy.

Input: *un grafo $G = (V, E)$ diretto come lista di adiacenza, $s \in V, \ell_e \geq 0 \ \forall \ e \in E$*
Output: *len(v) = dist(s, v) $\forall v \in V$*

```
Dijkstra(G,s)
```
$X \leftarrow \{s\}$
$len(s) \leftarrow 0$
$len(v) \leftarrow +\infty \ \forall \ v \neq s$
while there is an edge v, w with $v \in X$, $w \notin X$ **do**
    $(v^* w^*) \leftarrow$ such an edge minimizing $len(v) + \ell_{(v,w)}$
    $X \leftarrow X \cup w^*$
    $len(w^*) = len(v^*) + \ell_{(v^*,w^*)}$

Correttezza

L'algoritmo di Dijkstra NON funziona con le lunghezze negative.
Invariante: $\forall \ x \in X, \ len(x)$ è la lunghezza di un camminimo da s a X.

Dimostrazione. **Per induzione su** X.

Caso Base: $|X| = 1 \Rightarrow$ banale perché $len(s) = 0$

Passo Ric: Ipotesi induttiva: invariante vero $\forall \ |X| = k \geq 1$.

- Sia $v \rightarrow w^*$ il prossimo vertice aggiunto ad X, e $(u, v) \rightarrow (v^*, w^*)$ il lato selezionato.
- Il cammino minimo da s a $v + (u, v)$ è un cammino da s a v : $\Pi(v) = \min\{len(u) + \ell_{(u,v)}\}$ dove $(u, v) : v \in X, u \notin X$.
- Si consideri un qualsiasi cammino P da s a v. Dimostriamo ora che non è più corto di $\Pi(v)$.
- Sia (x, y) il primo lato in P che attraversa X, e sia P' il sotto-cammino da s a x.
- P non è più corto di $\Pi(v)$ perché:
    * $len(P) \geq len(P') + \ell_{(x,y)}$ perché le lunghezze sono non negative;
    * $len(P') + \ell_{(x,y)} \geq len(x) + \ell(x, y)$ per ipotesi induttiva;
    * $len(x) + \ell(x, y) \geq \Pi(y)$ per definizione di $\Pi(y)$;
    * $\geq \Pi(y) \geq \Pi(v)$ perché Dijkstra sceglie v invece di y.

$\Rightarrow len(P) \geq len(P') + \ell_{(x,y)} \geq len(x) + \ell(x, y) \geq \Pi(y) \geq \Pi(v)$.

□

Complessità

La complessità di una implementazione banale è $\mathcal{O}(m \cdot n)$. Ad ogni iterazione ($\leq n - 1$), ricerca esaustiva su tutti i lati del lato (v^*, w^*).

Implementazione con Heap:
`Dijkstra(G,s)`
$X \leftarrow \emptyset$
$H \leftarrow$ empty heap
$key(s) = 0$
for each $v \neq s$ **do**
    $key(v) = +\infty$
for every $v \in V$ **do**
    `insert` v in H
while H is non-empty **do**
    $w^* =$ `extractMin(H)`
    $X \leftarrow X \cup w^*$
    $len(w^*) = key(w^*)$
    for every edge(w^*, y) **do**
        `delete(H,y)`
        $key(y) = \min\{key(y), len(w^*) + \ell_{(w^*,y)}\}$
        `insert(H,y)`

Complessità

Ci sono $\mathcal{O}(m+n)$ operazioni su heap, ognuna delle quali ha complessità $\mathcal{O}(\log n)$
$\Rightarrow \mathcal{O}((m+n)\log n)$

6 Algoritmi di approssimazione

Questo tipo di algoritmi sono per problemi intrattabili, cioè per i problemi NP-completi che non si possono risolvere con un algoritmo polinomiale. Ci si accontenta di risolvere un problema in modo approssimato, sperando di trovare una soluzione che va vicino a quella ottima.

Problema di ottimizzazione

* $\Pi : I \times S$
* $c : S \to \mathbb{R}^+$
* $\forall\, i \in I\ S(i) = \{s \in S : i \Pi s\},\ s^* \in S(i) : c(s^*) = \underset{max}{min}\ c\big(S(i)\big)$

cioè un problema di approssimazione Π (funzione da un insieme di input I a un insieme di soluzioni S, sottoinsieme del prodotto cartesiano tra input I e soluzioni S) in cui esiste una funzione di costo c che associa ad ogni soluzione un valore reale positivo. Per ogni istanza di input abbiamo un insieme di soluzioni ammissibili $S(i)$ che sono tutte quelle soluzioni tale per cui s è una soluzione per l'istanza i ($i\,\Pi\,s$). Tra tutte queste, c'è s^* che minimizza/massimizza il costo della soluzione.

Approssimazione

* $s \in S(i)$ ok se $s \neq s^*$ ma voglio:
  1. garanzia sulla **qualità** di s (fattore di approx): $\begin{cases} \text{per difetto se è un problema di massimo} \\ \text{per eccesso se è un problema di minimo} \end{cases}$
  2. garanzia sul tempo: l'algoritmo deve essere **polinomiale**

Algoritmi di approssimazione ≠ algoritmi **euristici**! Questi ultimi sono sì veloci a risolvere in modo approssimato il problema ma non danno garanzie sulla qualità della soluzione ritornata.

Le versioni di ottimo dei problemi NP-completi non hanno tutti gli algoritmi con lo stesso fattore di approssimazione (ogni problema è a sé stante ed ha una sua complessità intrinseca). Quindi questi problemi sono equivalenti dal punto di vista della risoluzione ottima (la classe NP-c è una di equivalenza rispetto alla riducibilità polinomiale) ma non sono equivalenti per quanto riguarda l'approssimabilità.

Le riduzioni non preservano la stessa approssimabilità tra problemi equivalenti.

Def. 6.1. ***(Fattore di approssimazione).*** *Sia Π un problema di ottimizzazione e sia A_π un algoritmo per Π che ritorna, $\forall\, i \in I$, $A_\pi(i) \in S(i)$. Diciamo che A_π ha un **fattore** (o rapporto) **di approssimazione** $\rho(n)$ se $\forall\, i \in I : |I| = n$:*

$$1 \leq max \left\{ \frac{c\big(s^*(i)\big)}{c\big(A_\pi(i)\big)},\ \frac{c\big(A_\pi(i)\big)}{c\big(s^*(i)\big)} \right\} \leq \rho(n)$$

dove il primo termine viene scelto in caso di un problema di massimo, il secondo in caso di un problema di minimo.

N.B.:

* problema di **minimo:** $c\big(s^*(i)\big) \geq \dfrac{c\big(A_\pi(i)\big)}{\rho(n)}$

  → lower bound esplicito sul costo della soluzione ottima

- problema di **massimo:** $c\big(s^*(i)\big) \leq \rho(n) \cdot c\big(A_\pi(i)\big)$
  → upper bound esplicito sul costo della soluzione ottima

Sarebbe bello ottenere approssimazioni del tipo $\rho(n) = 1 + \varepsilon$ con ε più piccolo possibile.

Esistono problemi per cui si può provare che $\rho(n) = \Omega(n^\varepsilon)$ cioè non si possono approssimare per un fattore inferiore a polinomiale (*e.g.* CLIQUE).

Molto più facile è chiedere $\rho(n) = 1 + \varepsilon \ \forall \ \varepsilon > 0$ (schema di approssimazione).

Def. 6.2. ***(Schema di approssimazione).*** *Uno schema di approsimazione (AS) per un problema* Π *è un algoritmo a 2 input* $A_\pi(i, \varepsilon)$ *che* $\forall\ \varepsilon$ *è un algoritmo di* $(1 + \varepsilon)$*-approssimazione.*

In altre parole: fissata una istanza i di taglia n, la qualità è ε (qualsiasi sia ε).

Def. 6.3. ***(Schema di approssimazione polinomiale).*** *Uno schema di approssimazione è polinomiale (PTAS) se* $A_\pi(i, \varepsilon)$ *è polinomiale in* $|\,i\,| \ \forall\ \varepsilon$ *fissato.*

$\Rightarrow$ si può settare quale sia la qualità dell'approssimazione, però potrei avere complessità $T_{A_{\pi,\varepsilon}}(n) = \mathcal{O}(n^{\frac{1}{\varepsilon}})\ \forall\ \varepsilon$.

Supponiamo di voler migliorare l'approssimazione di un fattore k:
$\varepsilon \to \frac{\varepsilon}{k} \Rightarrow \mathcal{O}(n^{\frac{1}{\varepsilon}}) \to \mathcal{O}((n^{\frac{1}{\varepsilon}})^k)$
cioè c'è una dipendenza *esponenziale* nella qualità dell'approssimazione. Vorrei, invece schemi polinomiali che siano nella taglia dell'input (per ϵ finito) ma anche in $\frac{1}{\varepsilon}$ quando fisso la taglia dell'input. La polinomialità deve essere rispetto a entrambi i parametri di input (*Holy Grail* dei problemi NP-c).

Def. 6.4. ***(Schema di approssimazione pienamente polinomiale).*** *Uno FPTAS è uno schema la cui complessità è polinomiale in* $|\,i\,|$ *fissato* ε*, e in* $\frac{1}{\varepsilon}$ *fissato* $|\,i\,|$.

6.1 Approssimazione di VertexCover

Spesso, per creare un algoritmo di approssimazione, si affronta il problema in modo greedy. Lo scopo di `VertexCover` è trovare un sottoinsieme di nodi di un grafo tale per cui tutti i lati siano toccati dai nodi di questo sottoinsieme. Inizialmente, si può pensare di scegliere un lato a caso e aggiungere al VC i suoi due nodi per poi non considerare più tutti i lati incidenti ad essi.

```
Approx_Vertex_Cover(G)
V' ← ∅
E' ← E
while (E' ≠ ∅) do
    let {u,v} be arbitrary edge of E'
    V' ← V' ⋃ {u, v}
    //elimino tutti i lati in E' del tipo {v, z,} e {v, w} coperti da u e v
    E' ← E' ∖ {(v, z), (v, w)}
return V'
```

L'algoritmo è corretto per costruzione. La sua complessità è $\mathcal{O}(n + m)$. Qual è la sua garanzia?

Analisi

Determiniamo la qualità della soluzione ritornata dall'algoritmo, cioè il rapporto $\frac{|V'|}{|V^*|}$.
A = insieme dei lati selezionati
A **matching**: tutti i lati sono disgiunti, cioè non c'è nessuna coppia di lati che condivide un nodo in comune.
A è **matching massimale**: comunque venga scelto un altro lato y l'unione di y con A non è più un matching.
⇒ vertex cover è un algoritmo greedy per trovare il matching massimale.

* **Lim. Inf. al costo della soluzione ottima** V^*:
  un qualsiasi vertex cover, in particolare V^*, deve coprire tutti i lati del grafo. In particolare deve coprire tutti i lati di A. Ma A è un matching (cioè tuti i lati di A sono disgiunti)
  ⇒ in V^* deve esserci almeno un nodo $\forall\ (u, v) \in A$: $|V^*| \geq |A|$

* **Lim. sup. al costo della soluzione restituita** V':
  $|V'| \leq 2|A|$ (anzi, $|V'| = 2|A|$).

$$\Rightarrow |V'| \leq 2|A| \leq 2|V^*| \Rightarrow \rho(n) = \frac{|V'|}{|V^*|} \leq 2$$

Quindi `Approx_Vertex_Cover` è un algoritmo di 2-approssimazione per vertex cover. (non potrà mai essere che questo algoritmo mi ritorni un insieme di nodi che è due volte peggio dell'insieme minimo). Quindi non potrà mai essere più grande del doppio dell'insieme minimo teorico. Non è possibile migliorare questa approssimazione (**tight approximation**).

Stato dell'arte del problema di Vertex Cover: esiste un algoritmo di 1,5-approssimazione? **No**, ad oggi non esiste. Il lower-bound è fissato a 1,36.

* esiste $2 - \theta\left(\frac{1}{\sqrt{logn}}\right)$-approximation
* vertex cover non si può approssimare meglio di un fattore $\sim 1,36$
* congettura: non si può approssimare meglio di un fattore 2

Esercizi su questa sezione si possono trovare in A.4.1, A.4.2, A.4.3.

6.2 Traveling Salesman Problem

Il problema TSP si può formulare come segue: dato un grafo $G = (V, E)$ non diretto, completo e pesato con la funzione dei costi $c : V \times V \mapsto \mathbb{N}$, si determini il **ciclo hamiltoniano**[6] di peso minimo. Si tratta di un problema NP-completo.

Inapprossimabilità per fattori ρ costanti di TSP

Diversamente dal vertex cover, per questo problema non è possibile dimostrare l'esistenza di un lower bound per un possibile algoritmo di approssimazione.
Tecnica standard: se esistesse tale algoritmo[7] allora saprei risolvere in tempo polinomiale un problema NP-hard.

[6] che tocca tutti i vertici
[7] di approssimazione

Teorema 6.1. *se $P \neq NP$, non può esistere alcun algoritmo di ρ-approssimazione per TSP con $\rho = \mathcal{O}(1)$.*

Dimostrazione. Supp.p.a. che esista un algoritmo A_ρ polinomiale di ρ-approssimazione per TSP. Dimostro come costruire A_{ham} che decide se un ciclo è hamiltoniano in tempo polinomiale.

Si parte prendendo un'istanza generica a cui voglio ridurmi: un grafo semplice e non necessariamente completo. L'output deve dire se il grafo contiene un circuito hamiltoniano oppure no.

$I = G = (V, E)$

$O = G$ contiene un ciclo hamiltoniano?
Riduzione:

$G \to G' = (V, E')$ completo

$$c(e \in E') = \begin{cases} 1 & e \in E, \text{ cioè aggiungo un peso 1 a tutti i lati che c'erano anche prima} \\ \rho|v| + 1 & \text{altrimenti} \end{cases}$$

Eseguo $A_\rho(G') \xrightarrow{output} C, c(C)$ con C ciclo e $c(C)$ costo di C.

Quello che occorre dimostrare ora è che, usando C ed il suo costo, posso dedurre se il grafo originale (G) ottiene un ciclo hamiltoniano oppure no.
Si dimostra in due passi:

1. $G \in ham \Rightarrow c(C^*) = |V| \Rightarrow A_\rho$ ritorna un ciclo C con $c(C) \leq \rho|V|$

   Se il grafo di partenza G ha un ciclo hamiltoniano, allora c'è un ciclo che passa attraverso tutti i nodi del grafo e quindi il suo costo contiene cardinalità di V lati. Quindi il costo è cardinalità di V. L'algoritmo di approssimazione può sbagliare, quindi al più può ritornare un costo pari a $\rho|V|$

2. $G \notin ham \Rightarrow C$ contiene almeno un arco non in $G \Rightarrow c(C) \geq \rho|V| + 1$

Questo ci dice che io posso usare il mio algoritmo polinomiale A_ρ per risolvere un problema che, assumendo $P \neq NP$, non ha un algoritmo polinomiale. Quindi $\nexists\ A_\rho$. □

6.2.1 Paradigma generale per dimostrare inapprossimabilità

Inapprossimabilità di un problema Π_2 entro un fattore ρ: va trovato un problema $\Pi_1 \in$ NP-c e va determinata una funzione f calcolabile in tempo polinomiale[8] tale che:

1. $x \in L_{\Pi_1}, \Pi_2\big(f(x)\big)$ ha minimo limitato da una certa funzione di costo $k(x)$.

   Cioè se x è un'istanza positiva del problema Π_1[9], allora bisogna trasformare x in un'istanza del problema Π_2[10] e questo problema ha un costo minimo limitato da una certa funzione di costo $k(x)$.

2. $x \notin L_{\Pi_1} \Rightarrow \forall\, s \in S\big(\big(f(x)\big)\quad c(s) > \rho \cdot k(x)$

   Cioè se x è un'istanza negativa del problema Π_1 allora devo dimostrare che per ogni soluzione s nell'insieme delle soluzioni alle istanze $f(x)$ il costo di questa soluzione è più grande di $\rho \cdot k(x)$.

[8] nel caso della dimostrazione precedente, la funzione f era la funzione che prendeva in input G e che dava in output un valore vero se G conteneva un circuito hamiltoniano, falso altrimenti

[9] un'istanza che soddisfa il certificato per il sì di quel problema

[10] con una funzione f

$\Rightarrow \Pi_2$ non è approssimabile entro un fattore ρ (se $P \neq NP$)

Dato che non è possibile dare un algoritmo di ρ-approssimazione per `TSP`, ci concentreremo su istanze particolari di questo problema.

6.2.2 TSP metrico

Istanza particolare TSP in cui l'input, in particolare la funzione di costo c, soddisfa la disuguaglianza triangolare:

$$\forall\, u, v, w \in V \text{ vale che } c(u,v) \leq c(u,w) + c(w,v) \Rightarrow c(< u, v >) \leq c(< u, w, v >)$$

cioè per ogni insieme di tre nodi possibili del grafo, vale che il costo di un lato (u, v) è al più il costo di un lato (u, w) più il costo di un lato $(w, v) \Rightarrow$ significa che sostanzialmente il costo del cammino per andare da u a v attraverso l'unico lato che li collega è più conveniente di passare per i lati (u, w) e (w, v). Sempre meglio viaggiare diretti piuttosto che per punti indiretti.

Chiamiamo questo problema `Triangle_TSP`. Questa restrizione del problema NP-completo si trova in P? L'esercizio relativo si trova in A.4.4.

6.2.2.1 Algoritmo di 2-approssimazione per `TRIANGLE_TSP`

Abbiamo dimostrato come `TRIANGLE_TSP` rimanga un problema NP-completo. Dobbiamo cercare in tutti i modi di approssimarlo. Fortunatamente si può approssimare bene.

Dobbiamo cercare di risolvere un problema che ci può fornire un po' di struttura per risolvere il nostro problema originale, come abbiamo fatto per vertex cover in cui abbiamo utilizzato il concetto di matching massimale. Un matching massimale A è almeno una 2-approssimazione di vertex cover. Qui come possiamo fare? Utilizzando i **MST**! Dovremo fare un po' di lavoro poiché abbiamo bisogno di un ciclo, non di un albero $\Rightarrow$ dobbiamo trasformare l'albero in un ciclo.
Devo fare la visita "Preorder" di quel MST:

```
Preorder (v)
print(v)
if internal(v) do
    for each u ∈ children(v) do
        Preorder(u)
return
```

Visita il nodo corrente e, se è interno, chiamo la ricorsione su tutti i figli in un certo ordine (lessicografico).

Idea: aggiungo alla lista preorder la radice, quindi ciò che otteniamo è un ciclo hamiltoniano del grafo originale, perché quest'ultimo è completo.

esempio:
Considera l'albero $A = [a, b, e, c, d]$. La lista preorder ottenuta è a, b, c, d, e. Non è un ciclo perché non c'è modo di collegare e ad a. L'idea è quella di aggiungere "brutalmente" il lato che manca per ottenere un ciclo. Siamo tranquilli ad aggiungere lati "a caso" perché stiamo parlando di grafi completi.

//grafo $G = (V, E)$ con funzione di costo c sui pesi
`APPROX_T_TSP (G, c)`
//nodi del grafo
$V = \{v_1, v_2, v_3\}$
//radice arbitraria usata per l'algoritmo di Prim
$r \leftarrow Random\{V\}$
//T^* è il risultato, cioè un MST
$T^* \leftarrow Prim(G, c, r)$
//eseguo l'algoritmo Preorder a partire dalla radice ottenendo una certa lista di nodi che rappresenta
//l'ordine in cui sono stati visitati i nodi. La chiamo H'
$< v_{i_1}, v_{i_2}, ..., v_{i_n} > = H' \leftarrow Preorder(r)$
//ritorno la lista ottenuta col preorder con l'aggiunta dell'ultimo arco che crea il ciclo
return $< H', v_{i_1} > = H$

Analisi della qualità della soluzione ritornata

- Il costo di H' è il più basso per la definizione stessa di MST;
- usiamo la disuguaglianza triangolare: per esempio, considerati i nodi c e d non sono collegati da un lato, anche se noi lo stiamo usando. L'idea è che il costo di (c, d) è minore rispetto al costo di fare il giro largo perché vale la disuguaglianza triangolare. Quindi in questo caso (c, d) è una *shortcut*.

Analisi del fattore di approssimazione

1. **Trovare un lim. inf. al costo della soluzione ottima H^*.**
   - Sia H^* la soluzione ottima, ovvero il MST trasformato in ciclo aggiungendo un collegamento tra l'ultimo nodo ed il primo $H' = < v_{j_1}, v_{j_2}, ..., v_{j_n}, v_{j_1} >$ di cui non so assolutamente nulla, ma so che c'è.
   - sia $H'^* = < v_{j_2}, ..., v_{j_n} >$ il **cammino** ottenuto togliendo l'ultimo nodo da H^*. Il cammino in particolare è un **albero**, o meglio, uno **spanning tree**.
     - Quindi $c(H'^*) \geq c(T^*)$ dove T^* è il MST;
     - $c(H^*) \geq c(H'^*)$ perché i costi sono ≥ 0.
2. **Trovare un lim. sup. al costo della soluzione restituita H.**
   Dobbiamo cercare di esprimere il limite sup. in funzione del costo del MST. Solo così possiamo mettere in relazione le disequazioni ed ottenere quello che vogliamo.

   Def. 6.5. ***(Full Preorder Chain).*** *Dato un albero, una full preorder chain è una lista con ripetizioni dei nodi dell'albero che indica i nodi raggiunti dalle chiamate ricorsive dell'algoritmo* `Preorder`.

   Cioè si esegue preorder prendendo nota di tutti i nodi che si toccano durante le chiamate ricorsive.

   Esempio 1. Dato un albero $T = [a, b, e, c, d]$, la sua full preorder chain è $< a, b, c, b, d, b, a, e, a >$

   Lemma: la full preorder chain contiene sempre la preorder list.

   Proprietà: in una full preorder chain ogni arco di T^* compare esattamente 2 volte.
   Limitiamo il costo del ciclo ottenuto dalla preorder list a partire dall full preorder chain, sfruttando la disuguaglianza triangolare. Dobbiamo usare la proprietà di cui sopra.
   $\Rightarrow c(f.p.c) = 2 \cdot c(T^*)$

* se si eliminano dalla f.p.c. tutte le occorrenze successive alla prima dei nodi interni (tranne l'ultima occorrenza della radice) otteniamo, grazie alla **disuguaglianza triangolare**:
  $2 \cdot c(T^*) = c(<a, b, c, b, d, b, a, e, a>) \geq c(<a, b, c, d, b, a, e, a>) \geq ...$
  $... \; c(<a, b, c, d, a, e, a>) \geq ... \geq c(H)$
  $\Rightarrow 2 \cdot c(T^*) \geq c(H)$
  $\Rightarrow 2 \cdot c(H^*) \geq 2 \cdot c(T^*) \geq c(H)$
  $\Rightarrow \dfrac{c(H)}{c(H^*)} \leq 2$

Esercizio 1. mostrare che l'analisi è *tight*, dando un esempi di grafo dove `APPROX_T_TSP` non approssima meglio di un fattore $\rho = 2$.

Consideriamo il grafo completo K_6, che è un $K_{5,5}$ con l'aggiunta di un nodo centrale, dove i lati all'esterno + i lati che collegano i nodi al nodo centrale costano 1, mentre tutti gli altri (la stella satanica senza i lati del pentacolo) costano 2.

Non è difficile vedere che:

* la soluzione ottima usa solo 6 lati di costo 1 (quindi il costo è = $n = 6$);
* `APPROX_T_TSP` usa tutti lati di costo 2, tranne 2 lati di costo 1 (quindi il costo è = $2n - 2 = 10$).

$\Rightarrow lim_{n \to \infty}$ `APPROX_T_TSP` = 2

6.2.2.2 Algorimo di $^3/_2$-approssimazione per `TRIANGLE_TSP`

Algoritmo di Christofides, 1976.

* **approccio:** usare, ancora, il concetto di MST;
* **intuizione:** cerca di vedere `APPROX_T_TSP` in un altro modo:
  - trovare un **ciclo euleriano**[11] di basso costo **raddoppiando** i lati di T^*
  - restituire il ciclo che visita i nodi di G nell'ordine della loro prima apparizione nel ciclo euleriano.

Quindi, le cose da fare sono le seguenti:

1. calcola un MST;
2. aggiungi un lato per ogni lato dell'MST;
3. calcola il ciclo euleriano ed il suo costo;
4. restituisci quel ciclo che visita i nodi del grafo originale nell'ordine della loro prima apparizione nel ciclo euleriano.

La domanda, ora, è questa: esiste un ciclo euleriano più economico?

Proprietà: un ciclo è euleriano $\Leftrightarrow$ tutti i nodi hanno grado pari.

[11] cioè un ciclo che visita tutti i lati una e una sola volta

Come si può fare? L'idea è di aggiungere a T^*[12] un **matching perfetto**[13] allora tutti i nodi nel grafo-unione ottenuto hanno grado pari → si ottiene così un ciclo euleriano.

Se si parte dal costo di quest'ultimo ciclo euleriano, si ottiene un costo finale della soluzione garantito entro un fattore $\rho = \frac{3}{2}$

Le cose che fa l'algoritmo di Christofides sono le seguenti:

1. si calcoli il MST: $T^* = (V, E^*) \leftarrow Prim(G, c, r)$;
2. sia D l'insieme dei nodi di grado dispari in T^* (da notare che $|D|$ è pari).
   ⇒ Si calcoli un perfect matching di costo minimo M^* sul grafo indotto da D;
3. il grafo $(V, E^* \bigcup M^*)$ è euleriano. Calcolarne il costo del ciclo euleriano.
4. restituisci quel ciclo che visita i nodi del grafo originale nell'ordine della loro prima apparizione nel ciclo euleriano.

Di seguito ne è riportato un esempio.

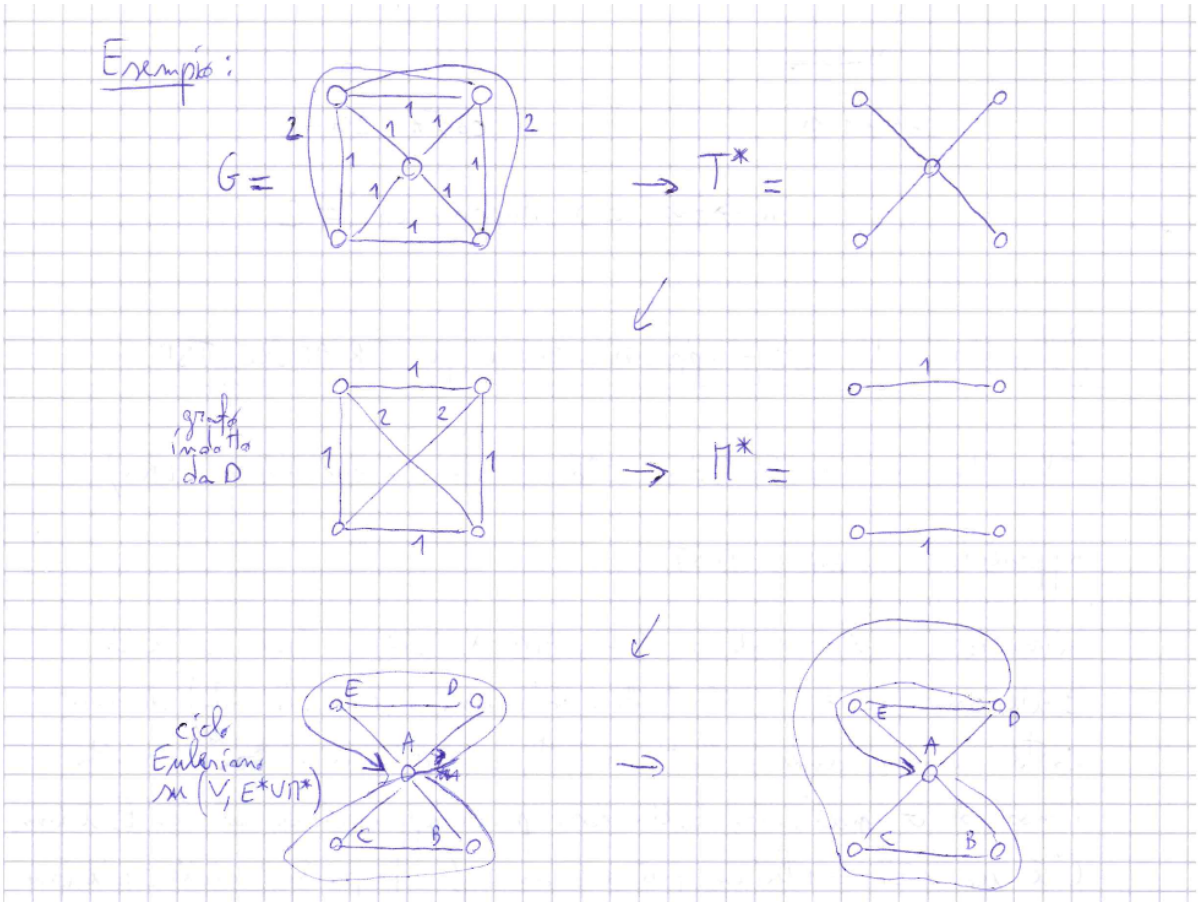


Figura 5: esempio.

[12] il MST calcolato

[13] cioè un matching massimale che include, oltre a tutti i lati, anche tutti i nodi

Domanda da esame tipica: viene dato un algoritmo di approssimazione per un certo problema e vengono chieste alcune cose, ad esempio:

* dimostrare un lower-bound al fattore di approssimazione (facile): basta pensare ad un'istanza singola dove quell'algoritmo approssima esattamente per quel fattore lì.
* dimostrare che è un algoritmo di 2-approssimazione (tipicamente viene dato un suggerimento come quello sopra)

6.2.3 Algoritmo di Held e Karp

Algoritmo esatto di programmazione dinamica, che combina i sottoproblemi di soluzioni più semplici. Si supponga che:

* si debba passare per n città, numerate da 0 a $n-1$
* il ciclo parta dalla città 0;
* dato un sottoinsieme di vertici $S \subseteq V$ ed un vertice $v \in S$, definiamo due vettori d e π come segue:
  - $d[v, S]$ è il peso del cammino minimo e parte da 0 e termina in v, visitando tutti i nodi in S;
  - $\pi[v, S]$ è il predecessore di v nel cammino minimo definito come sopra.
* alla fine dell'esecuzione $d[0, V]$ conterrà la soluzione di TSP che stiamo cercando, mentre l'informazione in π ci consentirà di ricostruire il ciclo di peso minimo.

Esempio 2. Dato un grafo $G = (V, E)$ completo con 4 nodi e con pesi così definiti:

Lato	**Peso**
$(0,1)$	4
$(0,2)$	1
$(0,3)$	3
$(1,2)$	2
$(1,3)$	1
$(2,3)$	5

Con $S = 1$:

v, S	$d[v, S]$	$\pi[v, S]$
$1, \{1\}$	4	0
$2, \{2\}$	1	0
$3, \{3\}$	3	0

Con $S = 2$:

v, S	$d[v, S]$	$\pi[v, S]$
$1, \{1,2\}$	1 + 2 = 3	2
$1, \{1,3\}$	1 + 3 = 4	3
$2, \{1,2\}$	2 + 4 = 6	1
$2, \{2,3\}$	5 + 3 = 8	3
$3, \{1,3\}$	1 + 4 = 5	1
$3, \{2,3\}$	5 + 1 = 6	2

Con $S = 3$:

v, S	$d[v, S]$	$\pi[v, S]$
$1, \{1,2,3\}$	6 + 1 = 7	3
$2, \{1,2,3\}$	4 + 2 = 6	1
$3, \{1,2,3\}$	3 + 1 = 4	1

Ora possiamo calcolare i valori di $d[0, \{0,1,2,3\}]$ e di$\pi[0, \{0,1,2,3\}]$.

$d[0, \{0,1,2,3\}] = 7$

$\pi[0, \{0,1,2,3\}] = 2$ oppure 3 (entrambe le soluzioni hanno costo 7

Possiamo quindi definire la soluzione del problema in maniera ricorsiva:

* **CASO BASE:** $\begin{cases} d[v, \{v\}] = w(0, v) \\ \pi[v, \{v\}] = 0 \end{cases} \quad (S = 1)$
* **PASSO RIC.:** $\begin{cases} d[v, S] = \min\limits_{u \in S \setminus \{v\}} \big(d[u, S \setminus \{v\}]\big) + w(u, v) \\ \pi[v, S] = \operatorname{argmin}\limits_{u \in S \setminus \{v\}} \big(d[u, S \setminus \{v\}]\big) + w(u, v) \end{cases} \quad (S \geq 2)$

Pseudocodice

Input: grafo $G = (V, E)$ non orientato completo e pesato, $S \subseteq V$ e $V \in S$
Output: Peso del cammino minimo da 0 a v che visita tutti i vertici in S

```
HK_VISIT(v,S)
//caso base, la soluzione è il peso dell'arco (v,0)
if S = {v} then
    return w[v,0]
//distanza già calcolata ritorna il valore in d
else if d[v,S] ≠ NULL then
    return d[v,S]
//passo ric: trova il minimo tra tutti i sottocammini
else
    //minimo costo corrente
    mindist = ∞
    //minimo predecessore corrente
    minprec = NULL
    //itero tra tutti i predecessori di v in S
    for all u ∈ S ∖ {v} do
        dist = HK_VISIT(u,S\{v})
        if dist + w[u,v] < mindist then
            mindist = dist + w[u,v]
            minprec = u
    d[s,V] = mindist
    π[s,V] = minprec
    return mindist
```

Complessità

È sempre NP-c, tuttavia è sempre meglio di un algoritmo brute force. L'algoritmo esegue delle chiamate ricorsive una sola volta per ogni possibile vertice e per ogni possibile sottoinsieme di S. Quindi vuol dire che il numero di chiamate ricorsive ad `HK_VISIT` che possiamo fare è lim. sup. dal numero di coppie $\{v, S\}$ utilizzare nell'indicizzazione dei nostri vettori d e π:

- $v \in V$: $n = |V|$ possibili vertici;
- S numero dei possibili sottoinsiemi di vertici: 2^n.

$\Rightarrow$ il numero di chiamate ricorsive sarà al più $n \cdot 2^n$.
Ogni chiamata ricorsiva:

- o termina immediatamente (caso base);
- o fa un'iterazione tra tutti i possibili predecessori di v, che possono essere $\mathcal{O}(n)$.

$\Rightarrow$ La complessità totale è $\mathcal{O}(n^2 \cdot 2^n)$.

Laboratorio

Provare l'algoritmo sui grafi del dataset. L'algoritmo andrà in timeout superati i 10 nodi, quindi in questo caso si chiede di interrompere l'esecuzione ed ottenere, comunque, un risultato dell'algoritmo.

Essa viene sicuramente ritornata perché viene eseguita una visita in profondita, poi comincia a fare back-tracking per andare ad analizzare tutti i possibili casi finché non trova il minimo, quello che succede è che se si interrompe l'algoritmo prima che abbia terminato l'analisi dell'albero di tutte le possibilità, vi ritroverete le situazioni in cui comunque l'algoritmo avrà trovato un certo numero di soluzioni, che tuttavia non è detto che includano la soluzione ottima. Quindi, in caso di timeout, la vostra soluzione dovrà andare a vedere il minimo delle soluzioni finora trovate. Provare con un timeout di 2 minuti.

6.3 Set Cover problem

Si tratta di una generalizzazione di Vertex Cover (o viceversa). Il problema consiste:

* data una istanza di un input generico $I = (X, F)$ in cui:
  - X è un insieme di oggetti qualsiasi, detti "**di supporto**" o "**universo**";
  - $F = \{S : S \subseteq X\} = \mathcal{B}(x)$ cioè è un sottoinsieme dell'insieme di tutti i sottoinsiemi di X, dove l'insieme di tutti i sottoinsiemi possibili di un insieme si chiama **Booleano**.
* il vincolo che voglio rispettare sempre è:

$$\forall\, x \in X\ \exists\, S \in F : x \in S \qquad \text{"F copre X".}$$

* Il problema di ottimizzazione, quindi, è il seguente:

"Determinare il più piccolo sottoinsieme di F che copra X"

Cioè determinare $F' \subseteq F : \begin{cases} 1)\ F' \text{ copre } X \\ 2)\ min\{|F'|\} \end{cases}$

Set cover, nella sua versoine decisionale $\big(I_D = <(X, F), k>\big)$ appartiene alla classe di problemi NP-c.
Set cover decisionale è così formulato:

* "Esiste un sottoinsieme di F che comprende almeno k elementi di X?"

Teorema 6.2. *Set Cover è NP-c.*

Dimostrazione. `Vertex_Cover` $<_p$ `Set_Cover`, cioè cerchiamo di dimostrare che Vertex Cover (che sappiamo essere NP-c) si riduce in tempo polinomiale a Set Cover.

$<G = (V, E), k> \quad \xrightarrow{f} \quad <(X, F), k>$

dove:
$X = E$
$F = \{S_1, S_2, ..., S_{|V|}\}$, uno per ogni nodo
$S_i = \{e : i \in e\}$, cioè l'insieme di lati coperti dal nodo e.

Dimostrare che trovare un Set Cover di taglia k equivale a tovare un Vertex Cover di taglia k.

* **CASO** $k = m$**:** dal punto di vista del problema di Set Cover, significa trovare il più piccolo sottoinsieme S^* di elementi $x \in X$ fra tutti i possibili sottoinsiemi che è possibile ottenere unendo i sottoinsiemi di S tale che, per ogni elemento $x \in X, x \in S^*$.

Ora, se X è l'insieme di archi di un grafo E allora ogni elemento $S_i \in F$ [14] non è nient'altro che la lista di adiacenza di ogni nodo $i \in V$. Allora trovare il più piccolo sottinsieme di liste di adiacenza dei nodi che contenga ogni arco $e \in E$, significa esattamente determinare l'insieme minimo di nodi che toccano tutti gli archi del grafo $\Rightarrow$ problema di Vertex Cover, che è NP-c. Allora anche $<(X,F),|X|>$ è NP-c.

* **CASO** $k < m$**:** analogo, perché la *NP-completeness* non dipende dalla taglia del grafo.

□

Un algoritmo per Set Cover

Utilizzo un approccio **greedy**: prendo il sottoinsieme che copre più elementi di X e tolgo da X gli elementi di quel sottoinsieme e ripeto finché $X = \emptyset$.

`APPROX_SET_COVER`
$U \leftarrow X$
//famiglia di sottoinsiemi che sto iterativamente costruendo
$F' \leftarrow \emptyset$
while $(U \neq \emptyset)$ **do**
  //prendo l'insieme di F che copre il maggior numero possibile di elementi in X
  sia $S \in F : |S \cap U| = \max_{s' \in F}\{|S' \cap U|\}$
  //aggiorno la lista di elementi disponibili, tolti quelli in S
  $U \leftarrow U \setminus S$
  //mantengo la coerenza togliendo gli insiemi già considerati in F
  //anche se in realtà non influenza la correttezza della soluzione ritornata
  $F \leftarrow F \setminus \{S\}$
  $F' \leftarrow F' \cup \{S\}$
return F'

Analisi

* **Correttezza**
  - Facile, basta osservare che a ogni iterazione $|U|$ decresce di almeno 1 elemento.
    In generale, per dimostrare la correttezza di un algoritmo di approssimazione basta dimostrare che la soluzione ritornata è sempre dentro all'insieme delle soluzioni ammissibili.
    $\Rightarrow \forall\ x \in U, \exists\ S' \subseteq F' : x \in S'$, cioè esiste sempre un sottoinsieme nella famiglia dei sottoinsiemi che contiene l'elemento x, $\forall\ x \in X$.
  - L'algoritmo termina quando $U = \emptyset$. L'unico problema che può sorgere è che non si esca mai dal `while`, ma ciò non è possibile poiché ad ogni ciclo U decresce, essendo $\max_{s' \in F}\{|S' \cap U|\} \geq 1$.

  $\Rightarrow$ corretto per costruzione.

* **Complessità**
  - ci sono al più $|X|$ iterazioni del ciclo `while` (caso in cui ogni insieme $S_i \in F$ contiene al più un elemento);

[14] definito come l'insieme di lati coperti dal nodo e

- ci sono al più $|F|$ iterazioni del ciclo `while` (caso in cui ogni insieme $S_i \in F$ contiene almeno due elementi)
  $\Rightarrow$ n° iterazioni $\leq min\{|X|, |F|\}$.
- $\forall$ iterazione faccio un lavoro che è $\leq |X| \cdot |F|$ perché (worst case event) devo scandire tutti gli elementi di F, ognuno dei quali può avere $|X|$ elementi. In verità $|X|$ e $|F|$ decrescono man mano che l'algoritmo va avanti.
  $\Rightarrow \mathcal{O}(|X| \cdot |F| \cdot min\{|X|, |F|\})$ che può essere cubico.
  In realtà si può implementare in $\mathcal{O}(|X| + |F|)$ usando strutture dati adeguate.

$*$ **Fattore di approssimazione:**

- Per quale funzione f è vero che la taglia della soluzione ritornata è al più funzione della taglia della soluzione ottima?
  $\Rightarrow |F'| \overset{?}{\leq} f(|F^*|)$
  Dimostreremo che $\dfrac{|F'|}{|F^*|} \leq \lceil \log_2 n \rceil + 1$, con $n = |X|$.

Proprietà: se (X, F) ammette cover con $|F| \leq k$, allora $\forall\ X' \subseteq X$, (X', F) ammette cover con $|F| \leq k$. *(cioè se posso coprire tutto X con k sottoinsiemi, a maggior ragione posso coprire un sottoinsieme di X con k sottoinsiemi)*

Idea: cercare di limitare il numero di cicli da eseguire in modo tale che l'insimeme degli elementi residui si svuoti il prima possibile.
$U_0 = X$
U_i = universo residuo alla fine della i-esima iterazione.
Sia $|F^*| = k$ la cardinalità della soluzione ottima

Lemma 6.3. *Dopo le prime k iterazioni, l'universo residuo si è almeno dimezzato.*
Cioè $|U_k| \leq \dfrac{n}{2}$.

Dimostrazione. $U_k \subseteq X$. Allora U_k ammette cover con $\leq k$ insiemi, ancora tutti in F (non ancora selezionati dall'algoritmo). È giusto perché dopo k iterate, l'universo residuo ha al più tanti elementi quanti sono gli insiemi non ancora selezionati. Chiamo questi insiemi $T_1, T_2, ..., T_k \in F$. L'unione di tutti i T_i copre quindi tutto U_k, l'universo residuo che mi rimane dopo k iterate.

Applichiamo il principio del **pigeonhole**: dato un insieme di elementi dove c'è una relazione d'ordine, c'è sempre almeno un elemento il cui valore è superiore al valore medio[15].
$\Rightarrow \exists\ \overline{T} : |U_k \cap \overline{T}| \geq \dfrac{|U_k|}{k}$, quindi esiste un sottoinsieme che copre un valore maggiore o uguale alla media dei valori.
$\Rightarrow$ in ciascuna delle prime k iterazioni copro almeno $\dfrac{|U_k|}{k}$ nuovi elementi.
$\forall\ 1 \leq i \leq k$ sia $S_i \subseteq F$ (l'insieme selezionato) con la proprietà che $|S_i \cap U_i| \geq |T_j \cap U_i|\ \forall\ 1 \leq j \leq k$, cioè che la cardinalità della sua intersezione con l'universo residuo è almeno grande come le cardinalità degli insiemi T_j non selezionati. *(ovvio, a ogni interazione seleziono l'insieme con maggior cardinalità).*

[15] Data una casa-rifugio per piccioni con n nidi, se i piccioni sono $n + 1$ vuol dire che almeno un nido è occupato da due piccioni.

Ma allora questa proprietà vale anche per $\overline{T}$, quindi posso scrivere $|S_i \cap U_i| \geq |\overline{T} \cap U_i| \geq |\overline{T} \cap U_k| \geq \frac{|U_k|}{k}$.
Quindi dopo le prime k iterazioni ho coperto $\frac{|U_k|}{k} \cdot k = |U_k|$ elementi. Ma k è l'insieme degli elementi che rimangono nell'universo residuo dopo k iterazioni, quindi è $\frac{|X|}{2} = \frac{n}{2}$. □

Uso il lemma per dimostrare il fattore di approssimazione dell'algoritmo. Essendo greedy, posso vederlo come un algoritmo ricorsivo che sceglie un sottoinsieme per poi ripetersi ricorsivamente sull'universo residuo.

- Dopo le prime k iterazioni, l'universo residuo si è almeno dimezzato: $|U_k| \leq \frac{n}{2}$;
- Dopo $k \cdot i$ iterazioni: $|U_{k \cdot i}| \leq \frac{n}{2^i}$

$\Rightarrow$ n° di iterazioni necessarie è $\lceil \log_2 n \rceil \cdot k + 1$. Il $+1$ serve per coprire l'eventuale ultimo elemento che rimane da coprire.
$\Rightarrow |F'| \leq \lceil \log_2 n \rceil \cdot |F^*| + 1$ cioè $\frac{|F'|}{|F^*|} \leq \lceil \log_2 n \rceil + 1$.
Questo è il migliore algoritmo possibile per Set Cover: un'approssimazione migliore di $\log n$ è impossibile, se $P \neq NP$.

7 Algoritmi Randomizzati

Algoritmi che possono fare scelte casuali. Perché fare scelte casuali?

Esempio 1. Randomized QuickSort.
Randomizzare la scelta del pivot[16] "rompe" l'istanza cattiva, cioè il caso peggiore, quando l'array da ordinare è già ordinato. Come faccio a "rompere" questo caso?

- prendendo come pivot un elemento a caso.

Se la scelta del pivot è casuale ad ogni chiamata ricorsiva, è molto difficile che si prenda sempre l'elemento sfortunato (come accade se il pivot viene scelto progressivamente).
$\Rightarrow T_{QS} = \mathcal{O}(n^2) \rightarrow \mathbb{E}[T_{RQS}(n)] = \mathcal{O}(n \log n)$
Con $\mathbb{E} \equiv$ aspettazione, cioè un qualcosa che si verifica "in media".

Esempio 2. Verifica di identità polinomiali.
Dati due polinomi, occorre verificarne l'uguaglianza.

$$\underbrace{(x+1)(x-2)(x+3)(x-4)(x+5)}_{H(x)} \equiv \underbrace{x^6 - 7x^3 + 25}_{G(x)}$$

- **soluzione ovvia:** trasformare $H(x)$ in forma canonica $\sum_{i=0}^{6} c_i x^i$ e poi verificare se tutti i coefficienti di tutti i monomi sono uguali.
- **una soluzione più veloce:**
  1. scelgo un intero random λ;
  2. calcolo $H(\lambda)$;
  3. calcolo $G(\lambda)$;
  4. se $H(\lambda) = G(\lambda)$
     allora ritorna **sì**
     altrimenti ritorna **no**

  Si tratta di un algoritmo corretto?
  Esempio: $\left.\begin{array}{l}\lambda = 2\\ H(2) = 0\\ G(2) = 33\end{array}\right\} H(x) \not\equiv G(x)$

  Ma se troviamo $H(x) \equiv G(x)$?
  Esempio: controlla se $x^2 + 7x + 1 \equiv (x+2)^2$
  $\lambda = 2$: $4 + 14 + 1 = 19$, $4^2 = 16$ $\Rightarrow$ sono $\not\equiv$
  $\lambda = 1$: $1 + 7 + 1 = 9$
  $\downarrow$ $3^2 = 9$ $\Rightarrow$ sono $\equiv$ $\Rightarrow$ l'algoritmo ritorna YES, ma è sbagliato!
  "cattiva" scelta di λ

[16] elemento scelto ad ogni chiamata ricorsiva, in relazione al quale si ordinano a sx gli elementi $\leq$ e a dx gli elementi $\geq$

L'algoritmo restituisce la risposta sbagliata solo se λ è una radice del polinomio $f(x) = G(x) - H(x) = 0$.
In particolare, se scelgo $\lambda \in [0, 100d]$ con $d = \max\{degree\big(f(x)\big)\}$, allora:

- la probabilità che l'algoritmo sbagli è:
$$Pr(algorithm\ fails) \leq \frac{d}{100d} = \frac{1}{100}$$

Tutto sommato l'algoritmo non è così pessimo perché sbaglia solo l'1% delle volte. Posso ritenermi soddisfatto? NO.

Come ridurre la probabilità di errore?

* Faccio girare l'algoritmo n volte. Se ritorna `YES` tutte le n volte allora restituisco `YES`, altrimenti `NO`.
Perché questo è un ottimo algoritmo?
$$Pr(algorithm\ fails) \leq \underbrace{\frac{1}{100} \cdot \frac{1}{100} \cdot \ldots \cdot \frac{1}{100}}_{n \text{ volte}}$$
$$\leq \frac{1}{100^n}$$
Esempio:
$$n = 10 \Rightarrow Pr(algorithm\ fails) \leq \frac{1}{100^{10}} = 10^{-20}$$
$$< 2^{-64}$$
Anche facendo girare l'algoritmo "solo" 10 volte, la probabilità che questo algoritmo sbagli è una probabilità strettamente minore a 2^{-64}, che è meno probabile di un errore causato da una radiazione cosmica che ne vada ad alterare qualche bit! [cit. D. Knuth]

7.1 Catalogazione degli algoritmi randomizzati

Algoritmi che:

* non sbagliano mai → algoritmi **Las Vegas**
* possono non essere sempre corretti → algoritmi **Monte Carlo**

7.1.1 Algoritmi Las Vegas

Algoritmi che il 100% delle volte restituiscono una soluzione corretta.
$$\forall\ i \in I,\ A_R(i) = s : (i, s) \in \Pi$$

con:

* $i \in I$ istanza di input
* $A_R(i) = s$ algoritmo random che, applicato all'istanza di input i, produce una soluzione s tale che la coppia (i, s)...
* ...appartiene a $\Pi \subseteq I \times S$, dove Π rappresenta il problema decisionale.

Cioè s è una soluzione dell'istanza i del problema Π.
N.B.: s non è sempre la stessa $\forall\ i$

Qui la randomizzazione è sfruttata anche nell'analisi della complessità dell'algoritmo:

* $\forall\ n, T(n)$ è una variabile aleatoria di cui si studia:
  - $\mathbb{E}\big[T(n)\big]$ oppure
  - $Pr\big(T(n) \geq c \cdot f(n)\big) \leq \dfrac{1}{n^k}$ "alta probabilità".
    Se si riesce a dimostrare questo, si dice che l'algoritmo ha complessità $\mathcal{O}(f(n))$ con alta probabilità.

  Lo spazio di probabilità corrisponde alle scelte casuali operate dall'algoritmo, da non confondere con l'analisi probabilistica di un algoritmo deterministico, dove lo spazio di probabilità coincide con la distribuzione degli input.

7.1.2 Algoritmi Monte Carlo

Data un'istanza di input, è possibile che l'output dell'algoritmo su quell'istanza sia una soluzione s che non è una soluzione corretta.

$$\exists\ i \in I,\ A_R(i) = s : (i, s) \notin \Pi$$

Con questo tipo di algoritmi, quello che si studia è $Pr\big((i, s) \notin \Pi\big)$ in funzione di $|i| = n$. Anche qui si ha una famiglia di variabili aleatorie **binarie** che rappresentano il fatto che l'algoritmo sia corretto oppure no (una variabile $\forall$ input).

Anche il tempo $T(n)$ potrebbe essere una variabile aleatoria.
Questi algoritmi (per problemi decisionali) si dividono tra:

* `ONE-SIDED`: si sbaglia solo su una sola risposta (sì/no)
  Fa giuste tutte le istanze del SÌ ma può sbagliare le istanze del NO
* `TWO-SIDED`: si sbaglia in entrambe le risposte
  Può sbagliare sulle istanze del SÌ e può sbagliare anche sulle istanze del NO

Def. 7.1. ***(Alta Probabilità).*** *Dato $\Pi \subseteq I \times S$, un algoritmo A_Π ha complessità $T(n) = \mathcal{O}(f(n))$ con altà probabilità*[17] *se $\exists\ c, d > 0$ tali che $\forall\ i \in I, |i| = n, Pr(A_\Pi(i)$ termina in $\geq c \cdot f(n)$ passi$) \leq \dfrac{1}{n^d}$*

Cioè:

* l'algoritmo A_Π ha complessità $T(n) = \mathcal{O}(f(n))$ con alta probabilità;
* la probabilità che quell'algoritmo "sfori" questa complessità è bassissima $\left(\dfrac{1}{n^d}\right)$;
* la probabilità che l'algoritmo termina con al più quel numero di passi è altissima $\left(1 - \dfrac{1}{n^d}\right)$.

Def. 7.2. ***(Correttezza con alta probabilità di un algoritmo).*** *Dato $\Pi \subseteq I \times S$, un algoritmo A_Π è corretto con alta probabilità (whp) se $\exists\ d > 0$ tali che $\forall\ i \in I, |i| = n, Pr((i, A_\Pi) \notin \Pi) \leq \dfrac{1}{n^d}$.*

La caratterizzazione whp è sempre più potente di quella al caso medio.
Esercizio A.5.1

[17] with high probability w.h.p

Algoritmo Monte Carlo whp per il problema del minimum cut

[David Karger, 1993]. Problema duale al problema del maximum flow (che si potrebbe risolvere usando `MaxFlow`). Al momento non esistono algoritmi deterministici che fanno meglio.

Risolveremo in realtà un problema più generale, cioè **min-cut** su **multigrafi** (cioè i grafi non semplici, quindi in cui ci sono più lati tra 2 nodi senza self-loops).

Def. 7.3. ***(Multiinsieme).*** *Collezione di oggetti con ripetizioni.*

- $S = \{\{v : v \in S\}\}$
- $\forall\ v \in S\ m(v) \in \mathbb{N} \setminus \{0\}$ *con* $m(v)$ = *molteplicità di* v*, cioè quante copie di* v *ci sono in* S*.*

Def. 7.4. ***(Multigrafo non orientato).*** *multigrafo* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *tale che:*

- $\mathcal{V} \subseteq \mathbb{N}, \mathcal{V}$ *finito;*
- $\mathcal{E}$ *è un multiinsieme a elementi del tipo* $\{u, v\} : u \neq v$ *(no self loops).*

Remark: un grafo semplice grafo $G = (V, E)$ è anche un multigrafo. Il viceversa non è vero.

Def. 7.5. ***(Cammino).*** *In un multigrafo, un cammino è una sequenza di nodi dove* $\forall\ u, v \in \mathcal{V}$ $\Rightarrow \exists\ (u, v) \in \mathcal{E}$.

Cioè esiste almeno un lato che connette ogni coppia di nodi in $\mathcal{V}$.

Def. 7.6. ***(Connettività).*** *Un multigrafo è connesso se* $\forall\ u, v \in \mathcal{V} \Rightarrow \exists$ *un cammino che li connette.*

Def. 7.7. ***(Taglio).*** *Dato un multigrafo* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *connesso, un taglio* $\mathcal{C} \subseteq \mathcal{E}$ *è un multiinsieme di lati tale che* $\mathcal{G}' = (\mathcal{V}, \mathcal{E} - \mathcal{C})$ *non è connesso.*

Equivalentemente, si può dire che $\mathcal{G}'$ *ha almeno due componenti connesse.*

Cioè un taglio è un insieme di lati che disconnette il grafo di partenza, creandone almeno due componenti connesse. Se il grafo è già sconnesso, un taglio valido è $\emptyset$ (che è anche il min-cut con cardinalità minima). Stiamo sempre parlando di multigrafi connessi perché, se non lo fossero, il taglio minimo (min-cut) sarebbe sempre l'insieme vuoto $\emptyset$, perché un taglio è un sottoinsieme dei lati che è un multiinsieme di lati tale che se io tolgo quei lati da $\mathcal{E}$, questo grafo non è connesso. Il problema è scegliere il più piccolo insieme di lati che disconnette il multigrafo.

7.1.2.1 Algoritmo di Karger

- Scelgo a caso un lato;
- "contraggo" i due nodi relativi eliminando tutti i lati incidenti su entrambi;
- ripeto finché restano solo 2 nodi: restituisco i lati tra quei 2 nodi.

Ciò funziona con probabilità bassissima, che però può essere amplificato ripetendo questo processo molte volte.

Def. 7.8. ***(Contrazione).*** *Dato un multigrafo* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *ed* $e = \{u, v\} \in \mathcal{E}$ *, la contrazione di* $\mathcal{G}$ *rispetto ad* e*,* $\mathcal{G}/_e = (\mathcal{V}', \mathcal{E}')$ *è il multigrafo con* $\mathcal{V}' = \mathcal{V} \setminus \{u, v\} \cup \{z_{u,v}\}$ *con:*

- $z_{u,v} \notin \mathcal{V}$ *super-nodo che nasce dalla fuzione tra* u *e* v*;*

* $\mathcal{E}' = \mathcal{E} \setminus \big\{\big\{\{x,y\} : (x = u) \vee (x = v)\big\}\big\}$
  $\cup \big\{\big\{\{z_{u,v}, y\} : (\{u,y\} \in \mathcal{E}) \vee (\{v,y\} \in \mathcal{E}), (y \neq u) \wedge (y \neq v)\big\}\big\}$

Cioè quando contraggo due nodi elimino il lato tra loro e creo un nuovo nodo che li comprende ed eredita tutti i loro vecchi lati incidenti.
Si osservi che:

* $|\mathcal{V}'| = |\mathcal{V}| - 1 \quad (= |\mathcal{V}| - 2 + 1)$
* $|\mathcal{E}'| = |\mathcal{E}| - m(e) \leq |\mathcal{E}| - 1$ (sparisce almeno il lato che collega i due nodi da contrarre)

Ora dimostreremo che in $\mathcal{G}/e$ la cardinalità del mincut non decresce.

Proprietà: $\forall$ taglio $\mathcal{C}'$ di $\mathcal{G}/e$ $\exists$ un taglio $\mathcal{C}$ di $\mathcal{G}$ della stessa cardinalità.
cioè $\{|\mathcal{C}'| : \mathcal{C}' \text{ taglio di } \mathcal{G}/e\} \subseteq \{|\mathcal{C}| : \mathcal{C} \text{ taglio di } \mathcal{G}\}$
$\Rightarrow$ |taglio minimo in $\mathcal{G}/e$| $\geq$ |taglio minimo in $\mathcal{G}$|

Dimostrazione. Determina il taglio $\mathcal{C} \in \mathcal{G}$ corrispondente a $\mathcal{C}' \in \mathcal{G}/e$, con $e = \{u, v\}$ sostituendo ogni lato $\{z_{u,v}, y\} \in \mathcal{C}'$ con il corrispettivo $\{u, y\} \vee \{v, y\}$.

* $|\mathcal{C}'| = |\mathcal{C}|$
* $\mathcal{C}$ è un taglio di $\mathcal{G}$?
  $\mathcal{C}'$ è un taglio in $\mathcal{G}/e = (\mathcal{V}', \mathcal{E}') \Rightarrow \mathcal{C}'$ separa $\mathcal{V}'$ in due componenti connesse. Sia $\mathcal{V}_1 \subset \mathcal{V}'$ la componente che contiene $z_{u,v}$ e sia $x \notin \mathcal{V}_1$. Allora in $\mathcal{G}/e$ ogni cammino da $z_{u,v}$ a x deve usare un lato in $\mathcal{C}'$. Cioè se si prende un nodo che sta in $\mathcal{V}_1$ ed un nodo che non sta in $\mathcal{V}_1$, deve usare un arco in $\mathcal{C}'$. Ora occorre mostrare che $\mathcal{C} \in \mathcal{G}$ disconnette u e v da x.
* Supp.p.a. che $\mathcal{C}$ **non** sia un taglio in $\mathcal{G}$, cioè che esiste almeno un cammino tra v e x dopo la rimozione di $\mathcal{C}$ da $\mathcal{E}$. Se non è un taglio, allora questo cammino tra $z_{u,v}$ e x sopravviverà alla rimozione di $\mathcal{C}' \in \mathcal{G}/e$, perché se sopravvive vuol dire che non usa lati in $\mathcal{C}$, cioè in $\mathcal{G}/e$ non usa lati in $\mathcal{C}'$. Cioè $\mathcal{C}'$ non è taglio in $\mathcal{G}/e$. Assurdo.

□

Abbiamo dimostrato che la contrazione diminuisce il numero di tagli (i tagli che scompaiono nel grafo contratto sono tutti e soli quelli colpiti dalla selezione del lato e), cioè può far sparire alcuni tagli, sicuramente quelli colpiti dalla selezione di e. Tutti gli altri sono preservati.

```
FULL_CONTRACTION
for i ← 1 to |V| = 2 do
    e ←Random(E)
    G' = (V', E') ← G/e
    V ← V'
    E ← E'
return |E|
```

```
KARGER(G, k) (k → ripetizioni di FULL_CONTRACTION)
min ← +∞
for i ← 1 to k do
    t ← FULL_CONTRACTION(G)
    if t < min then min ← t
return min
```

`KARGER` ripete `FULL_CONTRACTION` tante volte per ridurre la probabilità di errore (come visto nella verifica di identità polinomiali).

Analisi di `FULL_CONTRACTION`

* Si basa sulle probabilità condizionate.
  - **Def.:** $E_1, E_2 \subseteq \Omega$ indipendenti se $Pr(E_1 \cap E_2) = Pr(E_1) \cdot Pr(E_2)$
  - **Def.:** $Pr(E_1) > 0 \Rightarrow Pr(E_2|E_1) = \dfrac{Pr(E_1 \cap E_2)}{Pr(E_1)}$ (probabilità condizionata)
  - estensione a k eventi:
    $Pr(E_1 \cap E_2 \cap ... \cap E_k) = Pr(E_1) \cdot Pr(E_2|E_1) \cdot Pr(E_3|E_2 \cap E_1) \cdot ... \cdot Pr(E_k|E_1 \cap ... \cap E_{k-1})$
    Si dimostra per induzione su k.
    Useremo questa proprietà per valutare la probabilità che `FUL_CONTRACTION` ritorni un taglio minimo, dove E_i = i-esima contrazione che non ha toccato un lato di taglio minimo.
  - **Def.: (Grado).** Dato un multigrafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $d(v) = \left|\left\{\left\{(u, v) \in \mathcal{E} : u \in \mathcal{V}\right\}\right\}\right|$
  - **Oss.:** | min-cut | $\leq \min_{v \in \mathcal{V}} \{d(v)\}$. Perché? Ovviamente se prendo un nodo e tolgo tutti i lati incidenti a quel nodo, quello è un taglio.
  - **Intuizione:** quello che fa FC è sparare sui lati sperando di non prendere un lato di un min-cut. Quindi si spera che | min-cut | sia una piccola % di tutti i lati: questo si dimostra essere vero.
    * **Prop.:** sia multigrafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $|\mathcal{V}| = n$. Se $\mathcal{G}$ ha un min-cut di cardinalità t, allora $|\mathcal{E}| \geq t\,\dfrac{n}{2}$.

      Dim.: $|\mathcal{E}| = \sum_{v \in \mathcal{V}} \dfrac{d(v)}{2}$. Supp.p.a. che $|\mathcal{E}| < t\frac{n}{2}$. Allora $\sum_{v \in \mathcal{V}} \frac{d(v)}{2} < t\frac{n}{2} \rightarrow \sum_{v \in \mathcal{V}} d(v) < tn$

      Assurdo. Questo è un taglio più piccolo di quello minimo

* Analisi di una singola iterazione di FC.
  - Sia t = | min-cut | = C
  - E_i = al passo $i-$esimo di FC[18] la scelta casuale non seleziona un arco del taglio minimo C. Si osservi che, se si considera un solo taglio minimo, sto minorando la probabilità di successo (mi va bene per vedere il caso pessimo).
  - $Pr(\text{successo di FC}^{19}) \geq Pr\left(\bigcap_{i=1}^{n-2} E_1\right)$. Questi eventi non sono indipendenti, quindi serve la formula delle probabilità condizionate.
    * $Pr(E_1) \geq 1 - \frac{t}{t\frac{n}{2}} = 1 - \dfrac{2}{n}$. Quindi la probabilità di non beccare un arco del min-cut alla prima iterazione di FC è buona. Successo whp.
    * $Pr(E_2|E_1) \geq 1 - \dfrac{t}{t\frac{n-1}{2}}$
    * $\vdots$
    * $Pr(E_i|E_1 \cap E_2 \cap ... \cap E_{i-1}) \geq 1 - \dfrac{t}{t\frac{(n-i+1)}{2}} = 1 - \dfrac{2}{n-i+1}$

  $\Rightarrow Pr(\text{successo di FC}) \geq Pr\left(\bigcap_{i=1}^{n-2} E_1\right) \geq \prod_{i=1}^{n-2}\left(1 - \dfrac{2}{n-i+1}\right) = Pr\left(\bigcap_{i=1}^{n-2} E_1\right)\dfrac{n-i-1}{n-i+1} =$

  $= \dfrac{(n-2)}{n} \cdot \dfrac{(n-3)}{n-1} \cdot \dfrac{(n-4)}{n-2} \cdot ... \cdot \dfrac{3}{5} \cdot \dfrac{2}{4} \cdot \dfrac{1}{3} = \dfrac{2}{n(n-1)} \geq \dfrac{2}{n^2}$

[18] ce ne sono $n-2$ di passi

[19] cioè non becco nessun arco di taglio minimo C

Quindi, eseguendo FC una sola volta, la probabilità di non beccare un arco appartenente al min-cut è almeno di $\frac{2}{n^2}$. Un po' bassa come probabilità, ma non così bassa se si pensa a quanti sono i tagli di un grafo (un numero esponenziale). Vedremo che è sufficientemente alta per i nostri fini, perché andremo ad amplificarla, ripetendo FC k volte.

* Dobbiamo calcolare la probabilità che le k iterazioni di FC non accumulino la taglia del min-cut
  - $Pr(\text{le } k \text{ FC non accumulano la taglia del min-cut}) \leq \left(1 - \frac{2}{n^2}\right)^k$ (evento insuccesso)
    Vogliamo che questa probabilità sia bassissima, cioè che diventi un qualcosa del tipo $\frac{1}{n^d}$.
    Quindi scelgo $k = d\frac{n^2}{2}\log n$. Dimostriamo che è sufficiente.
    * Infatti $\left(\left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2}}\right)^{\log n^d} \leq e^{-\log n^d} = \frac{1}{n^d}$.
* Quindi con probabilità almeno $\frac{1}{n^d}$ l'algoritmo di Karger accumula la taglia del min-cut.

Complessità

* $T(\text{FC}) = \mathcal{O}(n^2) \Rightarrow$ Karger = $\mathcal{O}(n^4 \log n)$. Vediamo come si può migliorare.
  - **Oss.:** $Pr(\text{errore}) = \frac{2}{n} \to \frac{2}{n-1} \to \frac{2}{n-2} \to \cdots$. Nelle prime iterazioni ho un'ottima probabilità di errore, mentre man mano che si prosegue, la probabilità aumenta.
  - L'idea è di tenere come contrazioni, comuni a tutte le ripetizioni di FC, le prime iterazioni di FC perché è inutile ripeterle. Queste sono già iterazioni molto buone e le considererò iterazioni iniziali per tutte le FC ripetute che farò in seguito.
* $\to \mathcal{O}(n^2 \log^3 n) \sim \mathcal{O}(n^2)$ dimostrando che la correttezza viene preservata, assieme all'alta probabilità.
* World record: l'algoritmo più veloce per questo problema ha $\mathcal{O}(m \log n)$ [2020].

7.2 I bound di Chernoff

* Sono di uso **standard** nell'analisi degli algoritmi randomizzati.
* Sono una versione **molto più potente** del lemma di Markov
* Vedremo che in molte analisi si può trasformare lo studio di $Pr(T(n) \geq c \cdot f(n))$ nello studio della distribuzione di una certa sommatoria di **variabili indicatore**, cioè variabili che assumono solo i valori zero e uno:
  - $X = \sum_{i=1}^{n} X_i \qquad X_i$ indipendenti.
    Si scompone il tempo $T(n)$ in tanti contributi elementari che possono esserci (o meno) a seconda delle scelte dell'algoritmo.
  - $Pr(X_i = 1) = p_i$
  - $\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} p_i \triangleq \mu \to$ media della somma delle variabili X_i.

* Studieremo la prob. che X si discosti dalla sua media:

  $Pr\big(X > (1+\delta)\mu\big) \leq \dfrac{\mathbb{E}[X]}{(1+\delta)\mu} = \dfrac{\mu}{(1+\delta)\mu} = \dfrac{1}{1+\delta}$ (disug. di Markov)

  Cioè la probabilità che un certo evento X si discosti dalla sua media è molto bassa. Strumento che permette di studiare la concentrazione di un evento attorno alla propria media.

  – La maggiorazione è un po' lasca, poco significativa. Una maggiorazione migliore mi permette di passare da analisi al caso medio alla più desiderabile analisi in hp. Il lemma di Chernoff serve proprio a ciò.

Lemma di Chernoff: Siano $X_1 ... X_n$ variabili indicatori indipendedi con $\mathbb{E}[X_i] = p_i,\ 0 < p_i < 1$.
Sia $X = \sum_{i=1}^{n} X_i$ con $\mu = \mathbb{E}[X]$.
Allora $\forall\ \delta > 0$:
$Pr\big(X > (1+\delta)\mu\big) < \left(\dfrac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$.

Coin flip: lancio di moneta *fair* per n volte.

* n coin flips $\rightarrow X_1, ..., X_n$ indipendenti
* $Pr(X_i = \text{testa}) = \frac{1}{2} \quad \forall\ i$
* $X = \sum_{i=1}^{n} X_i$ = n° di teste
* $\mathbb{E}[X] = \frac{n}{2}$. Mi aspetto di avere $\frac{n}{2}$ teste.

Qual è la probabilità di fare più di $\frac{3}{4}\ n$ teste? Più grande è n, più questa probabilità si abbassa.

Markov: $Pr\big(X > \underbrace{(1+\frac{1}{2})\mu}_{\frac{3}{2}\cdot\frac{1}{2}n=\frac{3}{4}n}\big) \leq \dfrac{\mathbb{E}[X]}{(1+\frac{1}{2})\mu} = \dfrac{\frac{n}{2}}{\frac{3}{2}\cdot\frac{n}{2}} = \dfrac{2}{3}$

$Pr\big(X > (1+\frac{1}{2})\mu\big) \leq \frac{2}{3}$

Non troppo utile. Ci dà solo un upper bound molto lasco sulla probabilità.

Chernoff: $Pr\big(X > (1+\overbrace{\frac{1}{2}}^{\delta})\mu\big) < \left(\dfrac{e^{\frac{1}{2}}}{\left(\frac{3}{2}\right)^{\frac{3}{2}}}\right)^{\frac{n}{2}} < (0.95)^n$

Questo valore è molto meglio. Per $n \nearrow \infty$ la probabilità si azzera. In ogni caso, si avvicina allo zero in maniera molto veloce.

7.2.1 Varianti del bound di Chernoff

1) $Pr\big(X < (1-\delta)\mu\big) < e^{\frac{-\mu\delta^2}{2}} \qquad 0 < \delta \leq 1$

2) $Pr\big(X > (1+\delta)\mu\big) < e^{\frac{-\mu\delta^2}{2}} \qquad 0 < \delta \leq 2e-1$

Nell'esame saranno ricordate tutti e 3 i bound di Chernoff + il lemma di Markov (se serve).

Analisi in alta probabilità di Randomized QuickSort

Randomized QuickSort: QuickSort in cui il pivot viene scelto a caso.

* In triennale abbiamo visto che $\mathbb{E}[RQS] = n \log n$ (anche con costanti basse) mentre al caso pessimo è $\mathcal{O}(n^2)$ (array già ordinato)
* Ora dimostreremo che $Pr(T_{RQS} > c \cdot n \log n) < \frac{1}{n^2}$

Si guarda l'albero delle chiamate:

* n nodi (escluse le foglie associate a $\emptyset$) $\Rightarrow \leq n$ cammini radice-foglia.
* Dimostreremo che questi cammini non sono "troppo lunghi";
  - se dimostro che l'altezza dell'albero è $\mathcal{O}(\log n)$ whp,
  - allora $T_{RQS}(n) = \mathcal{O}(n \log n)$ whp.

Chiamiamo l'evento E l'evento "scelta felice del pivot", cioè quando il pivot scelto è tra le order-statistic (scelte comprese tra $\frac{n}{4} + 1, \cdots, \frac{3}{4}n$)

* Se scelgo così il pivot, allora le due sottoistanze che vado a creare hanno taglia $\leq \frac{3}{4}n$ entrambe.
* $Pr(\text{"scelta felice"}) = \frac{3}{4}n - \frac{n}{4} - 1 + 1 = \frac{1}{2}$
  - Se ho sempre successo, i cammini non sono più lunghi di $\log_{\frac{4}{3}} n$
    perché $|S|$ dopo i successi è $\leq \left(\frac{3}{4}\right)^i n$
* Fisso un cammino radice-foglia Π. Se dimostriamo che:
  - $Pr(|\Pi| > a \log_{\frac{4}{3}} n) < \frac{1}{n^3}$ (a costante), allora
  - $Pr(\text{tutti i cammini siano} \leq a \log_{\frac{4}{3}} n) \geq 1 - Pr(\exists \text{ cammino} \geq a \log_{\frac{4}{3}} n) \geq \ldots$
  - E_i = il cammino Π_i è lungo $> a \log_{\frac{4}{3}} n$
  - $Pr(\exists \text{ almeno un cammino più lungo di } a \log_{\frac{4}{3}} n) = Pr(\bigcup_{i=1}^{n} E_i) \leq \sum_{i=1}^{n} Pr(E_i) \leq n \frac{1}{n^3} = \frac{1}{n^2}$
  - Equivalentemente: $Pr(\text{tutti i cammini siano più corti di } a \log_{\frac{4}{3}} n) \geq 1 - \frac{1}{n^2}$ (alta probabilità)
* $Pr(RQS \text{ esegue } \mathcal{O}(n \log n) \text{ confronti}) \geq Pr(\text{tutti i cammini siano} \leq a \log_{\frac{4}{3}} n)$
  $\Rightarrow T_{RQS}(n) = \mathcal{O}(n \log n)$ whp.
* Non ci resta che dimostrare che $Pr(|\Pi| > a \log_{-\frac{4}{3}} n) < \frac{1}{n^3}$
  - evento "un cammino fissato ha più di $\ell = a \log_{\frac{4}{3}} n$ nodi". Allora
  - "nei primi $\ell = a \log_{\frac{4}{3}} n$ nodi del cammino ha eseguito al più $a \log_{\frac{4}{3}} n$ scelte felici".
    Studio questo ultimo evento:
    * $X_i \quad 1 \leq i \leq a \log_{\frac{4}{3}} n$
    * $X_i = 1$ se all'i-esimo nodo la scelta è felice
    * $Pr(X_i = 1) = p_i = \frac{1}{2} \quad \forall\, X_i$
    * X_i indipendenti

- $Pr\left(\sum_{i=1}^{\ell} X_i < \log_{\frac{4}{3}} n\right)$ da limitare. $\mu = \mathbb{E}[X] = \sum_{i=1}^{\ell} \frac{1}{2} = \frac{a}{2}\log_{\frac{4}{3}} n$
- Uso il Chernoff bound 1) ma per usarlo devo determinare δ.
  - Devo scegliere δ tale che $a\log_{\frac{4}{3}} n$ sia scritta nel modo del lemma.
    - $a = 8, \delta = \frac{3}{4}$
    - $(1-\delta)\mu = \frac{1}{4}4\log_{\frac{4}{3}} n = \log_{\frac{4}{3}} n$
- $Pr\left(\sum_{i=1}^{\ell} X_i < \log_{\frac{4}{3}} n\right) < e^{-\frac{4}{2}\log_{\frac{4}{3}} n\left(\frac{9}{16}\right)} = e^{-\log_{\frac{4}{3}} n\left(\frac{9}{8}\right)} < e^{-\log_{\frac{4}{3}} n} = e^{-\frac{\log n}{\log\frac{4}{3}}} = \left(\frac{1}{n}\right)^{\frac{1}{\log\frac{4}{3}}} < \frac{1}{n^3}$

Exit polls: Approssimare la % di votanti che hanno votato per una certa opzione disponibile tra più opzioni, senza contare tutti i voti.

- Come funziona?
  - Si estraggono a caso alcuni voti, che andranno a rappresentare la soluzione approssimata. L'idea è che vadano estratti *abbastanza* voti, che mi assicuri la bontà della soluzione whp.
- Urna U con n palline, che possono essere bianche o nere.
  - Supponiamo di sapere che ci sono almeno $\alpha_{min} \cdot n$ palline bianche (dobbiamo assumerlo per certo, perché è difficile approssimare whp un evento molto improbabile). Ecco perché, sui partiti piccoli, si sbaglia molto maggiormente la % che viene fornita agli exit poll.
  - Per determinare α in modo preciso, ogni algoritmo deterministico ricihede $\Omega(n)$ perché deve assolutamente contare tutte le palline
  - Quello che si ottiene con l'approssimazione randomizzata (con qualsiasi livello di confidenza decidibile) è $\mathcal{O}(n)$. Dimostreremo che l'approssimazione ottenuta è corretta whp.
- L'algoritmo ritornerà una quantità β tale per cui la $Pr\left(\frac{\beta-\alpha}{\alpha} > \epsilon\right)$ (errore relativo) sia molto piccola (e.g $< \frac{1}{n}$), con ϵ = soglia di confidenza.
  N.B.: è uno schema di approssimazione randomizzato. Più piccolo è ϵ, più tempo ci metterà l'algoritmo, perché aumenta il numero di estrazioni che sarà necessario fare.

`APPROXIMATE_`α `(U,`ϵ`,`α_{min}`)`
//palline presenti
$n \leftarrow |U|$
//numero estrazioni da fare
$k \leftarrow f(n, \epsilon, \alpha_{min})$
$x \leftarrow 0$
repeat k times
  $p \leftarrow$ `Rnd`(U)
  if color(p) = bianco **then** $x \leftarrow x + 1$
return $\beta \leftarrow x/k$

Da notare che queste sono estrazioni con reimmissione, mentre negli exit poll le estrazioni sono senza reimm--issione. Questo non fa altro che migliorare l'appros--simazione dell'algoritmo, perché le estrazioni senza reimmissione sono sempre peggio perché hanno maggior varianza. Noi trascureremo questo particolare.
La **complessità** è $\mathcal{O}(k = f(n, \epsilon, \alpha_{min}))$, anzi, esattamente k.
Qual è il k che mi garantisce l'**alta probabilità**?

* k variabili indicatore, una per ogni estrazione di pallina dall'urna
  - $X_i = 1$ se la pallina estratta è bianca, 0 altrimenti;
* $p_i = \alpha$
* $\mu = \mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{k} X_i\right] = k\alpha \leftarrow$ la media
* Da valutare la $Pr\left(\frac{|\beta - \alpha|}{\alpha} > \epsilon\right) = Pr\left(\frac{|\frac{X}{k} - \alpha|}{\alpha} > \epsilon\right) = \boxed{Pr\left(\frac{|X - \alpha k|}{\alpha k} > \epsilon\right)}$
* Ora che abbiamo isolato la media si può applicare un Chernoff bound!
  - $\boxed{Pr\left(|X - \mu| > \epsilon\mu\right)} < 2e^{\frac{-\mu\epsilon^2}{2}} < \frac{1}{n} \qquad 0 < \delta \leq 1$
  - $2e^{\frac{-\mu\epsilon^2}{2}} = 2e^{\frac{-\alpha k\epsilon^2}{2}}$ deve diventare $< \frac{1}{n}$
  - α non lo conosco, uso invece α_{min}
    * suppongo $k = \frac{2}{\alpha_{min}\epsilon^2}\log n^2 = \frac{4\log n}{\alpha_{min}\epsilon^2} = \mathcal{O}\left(\frac{\log n}{\epsilon^2}\right)$
      · Cioè è sufficiente chiedere a ordine di $\log n$ persone per chi hanno votato, per avere un'approssimazione con alta probabilità.
    * (sostanzialmente scelgo k in modo che mi faccia sparire i parametri diversi da n per aggiungerci un $\log n^\omega$ in modo da ricondurmi ad una forma del tipo $e^{-\log n^\omega}$, che è uguale a $n^{-\omega} = \frac{1}{n^\omega}$).
    * $2e^{\frac{-\alpha k\epsilon^2}{2}} < 2e^{\frac{-\alpha_{min} k\epsilon^2}{2}} = 2e^{-\log n^2} = \frac{2}{n^2} < \frac{1}{n} \qquad (n > 2)$
* se $k \nearrow \infty$ ottengo $\frac{1}{n^d}$ con $d > 1$.

Quindi abbiamo abbassato esponenzialmente la complessità dell'algoritmo deterministico mantenendo un margine di errore relativo che, whp, è molto limitato!

Load Balancing: *(a.k.a. balls-and-bins)*

* serie di n *jobs* che arrivano uno dopo l'altro (come uno stream) e vanno assegnati ciascuno ad uno tra n processori, in modo da bilanciare il carico.
* Ambiente distribuito = non c'è controllo centrale.
* Possibile soluzione → assegnare ogni job a un processore a caso. Funziona?

La domanda è, quindi: qual è il carico massimo di un certo processore? Voglio che il carico massimo (tra tutti i carichi) sia il più piccolo possibile. Questo algoritmo soddisferà questa richiesta? Sì, abbastanza bene. Dimostreremo che il carico massimo non è ottimo, ma nemmeno troppo alto: sarà qualcosa dell'ordine di $\frac{\log n}{\log\log n}$ whp.

L'analisi in alta probabilità si fa in due fasi:

1. prima si considera un processore fissato (che in RQS significava considerare un cammino radice-foglia fissato) e poi si fa' l'analisi per quell'elemento singolo;

2. poi si applica un union bound (un bound sulla probabilità dell'unione degli eventi) per ottenere la probabilità che un processore qualsiasi abbia un carico alto, sia molto bassa.

Cominciamo:

1. Consideriamo un processore fissato:
   - $X_i = 1 \Leftrightarrow$ job è assegnato a questo processore
   - $Pr[X_i = 1] = \frac{1}{n}$
   - X_i indipendenti
   - Carico del processore = $X = \sum_{i=1}^{n} X_i$
   - $\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = n \frac{1}{n} = 1$

   Uso il lemma di Chernoff (versione originale). Sia $1 + \delta = c$
   - $Pr(X > c) < \frac{e^{c-1}}{e^c} < \left(\frac{e}{c}\right)^c$
   - Se poniamo $c = e\gamma(n) \Rightarrow \left(\frac{e}{c}\right)^c = \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)}$

     vale che $\gamma(n)^{\gamma(n)} = n \Rightarrow \gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$

     $\Rightarrow Pr(X > c) < \gamma(n)^{-2\gamma(n)} = \left(\gamma(n)^{\gamma(n)}\right)^{-2} = n^{-2} = \frac{1}{n^2}$

     Abbiamo dimostrato come un processore non abbia un carico alto con probabilità $< \frac{1}{n^2}$

2. Applichiamo l'union bound:
   - E_i = il processore i riceve più di $\Theta\left(\frac{\log n}{\log\log n}\right)$ jobs
   - $Pr(\exists$ processore che riceve più di $\Theta\left(\frac{\log n}{\log\log n}\right)$ jobs$) = Pr\left(\bigcup_{i=1}^{n} E_i\right) \leq \sum_{i=1}^{n} Pr(E_i)$
     $< n \cdot \frac{1}{n^2} = \frac{1}{n}$

   Cioè la probabilità che nessun processore riceva più di $\Theta\left(\frac{\log n}{\log\log n}\right)$ jobs è $\geq 1 - \frac{1}{n}$ ←.

Appendices

A Esercizi

A.1 Lezione 2

A.1.1 Esercizio 1

Sia grafo $G = (V, E)$ un grafo non diretto con $k > 1$ componenti connesse. Progettare un algoritmo che aggiunga $k - 1$ lati a G per renderlo connesso e analizzarne la complessità. Si assuma di poter aggiungere a E un lato $(u, v) \notin E$ in tempo costante invocando il metodo `G.addEdge(u,v)`.

```
ConnectDFS(G, v)
for v ← 1 to n do
  L_V[v].ID ← 0;
pv ← NIL
for v ← 1 to n do
  if(L_V[v].ID = 0) do
    DFS(G, v)
    if(pv != NIL)
      G.addEdge(pv,v)
    pv ← v
```

Correttezza

L'algoritmo funziona in quanto si tratta di un arricchimento di `DFS(G,v)` in cui viene utilizzata la variabile pv che indica il nodo di partenza della componente connessa precedente C_{s-1}. Quando individuo una componente non connessa con nodo di partenza v, viene chiamato `DFS(G,v)`: in questo modo seleziono tutti i nodi in C_s ($\forall\ v \in C_s,\ L_V[v].ID \leftarrow 1$). Una volta terminato `DFS(G,v)` l'algoritmo controlla se in precedenza è già stata trovata una componente connessa C_{s-1}, con nodo di partenza pv. Se sì, procede al collegamento tra i nodi di partenza di C_s e C_{s-1} tramite la chiamata a funzione `G.addEdge(pv,v)`.

Complessità

La complessità si rivela essere la medesima di `DFS(G,v,k)`, quindi $\mathcal{O}(n + m)$.

A.1.2 Esercizio 2

Si consideri un labirinto L che ha un unico punto di ingresso s e al cui interno è nascosto il terribile Minotauro. L'ing. Teseo deve entrare in L, trovare il Minotauro, ucciderlo, e uscire da L. Trovare un'opportuna rappresentazione del labirinto come grafo e far vedere come, sfruttando l'algoritmo DFS, l'ing. Teseo può compiere con successo la sua missione. Si assuma che il labirinto sia connesso, nel senso che ogni punto di esso sia raggiungibile da s.

Ai nodi del grafo-labirinto $L = (V, E)$ possono essere aggiunte le seguenti peculiarità:

* $s \in L$ è il nodo *gateway*, cioè il nodo di entrata e di uscita dal labirinto L;
* $\forall\ v \in L$:
  - $L_V[v].hasMinotaur =$ `true` se c'è il Minotauro in quel nodo;
  - $L_V[v].hasMinotaur =$ `false` altrimenti.

  $\exists!\ v \in L\ :\ L_V[v].hasMinotaur =$ `true`, cioè c'è solo un Minotauro all'interno del labirinto L.

```
MinotaurDFS(G, v)
//segno il passaggio sul nodo v
L_V[v].ID ← 1;
//se c'è il minotauro in questo nodo...
if (L_V[v].hasMinotaur = true)
  //...lo uccido e torno indietro
  killTheMinotaur()
  return true
//altrimenti, per tutti i lati incidenti a v
forall e ∈ G.incidentEdges(v) do
  //se quel lato non è stato ancora etichettato
  if(L_E[e].label = null) then
    //considera il nodo opposto a v su quel lato
    w ← G.opposite(v, e)
    //se questo nodo non è stato ancora visitato
    if(L_V[w].ID = 0) then
      //aggiorno l'etichetta del lato come ARIADNE'S ROPE e...
      L_E[e].label ← ARIADNE'S ROPE
      //...lo visito
      killed = DFS(G,w)
      if (killed = true)
        return true
```

A.1.3 Esercizio 3

Sia il grafo $G = (V, E)$ non connesso con n vertici ed m lati. Progettare un algoritmo che conti le coppie di vertici $u, v \in V$ tali che u e v siano raggiungibili uno dall'altro tramite cammini, analizzandone la complessità. Per avere punteggio pieno la complessità deve essere $\mathcal{O}(n + m)$ (si ricordi che da un insieme di K oggetti si possono formare $\frac{K(K-1)}{2}$ coppie distinte).

L'idea è di usare un BFS modificato per determinare, per ogni componente connessa di G, la sua cardinalità K, aggiungendo il valore $K(K-1)/2$ al conteggio delle coppie di vertici raggiungibili uno dall'altro.

Supponiamo di aver modificato `BFS(G,v)` definendo una variabile `cardinality` inizializzata a 0 che viene incrementata ogni volta in cui si visita un vertice ed il cui valore viene restituito in output.

Algoritmo: `ReachablePairs(G)`
Input: grafo $G = (V, E)$ non diretto e non connesso
Output: Numero coppie $u, v \in V$ raggiungibili uno dall'altro

$count \leftarrow 0$
for $v \leftarrow 1$ **to** n **do**
    if ($L_V[v].ID = 0$) **then**
        $K \leftarrow$ `BFS(G,v)`
        $count \leftarrow count + \frac{K(K-1)}{2}$
return $count$

Analisi

La correttezza dell'algorito è immediata, e le modifiche apportate alla `BFS` non ne alterano la complessità, lasciandola intatta a $\mathcal{O}(n + m)$.

A.1.4 Esercizio 4

Sia il grafo $G = (V, E)$ diretto e connesso in cui ciascun vertice ha grado esattamente c, con $c > 2$. Si consideri l'esecuzione di `BFS(G,s)` a partire da un arbitrario vertice $s \in V$. Dimostrare per induzione su i che il livello L_i generato da `BFS(G,s)` contiene $\leq c \cdot (c-1)^{i-1}$ vertici $\forall\ i \geq 0$.

Caso Base: $i = 0$ contiene 1 solo vertice.
Passo Ric.: Fissiamo $i \geq 1$ e supponiamo, come ipotesi induttiva, che $|L_j| \leq c \cdot (c-1)^{j-1}\ \forall\ 0 \leq j \leq i$. I vertici del livello L_{i+1} sono tutti adiacenti a vertici del livelli i e poiché ciascun vertice $v \in L_i$ ha c vicini, di cui però **almeno uno è nel livello** L_{i-1}. Concludiamo che:

$$|L_{i+1}| \leq (c-1) \cdot |L_i| \leq (c-1) \cdot c \cdot (c-1)^{i-1} = c \cdot (c-1)^i$$

□

A.2 Lezione 5

A.2.1 Esercizio 1

Uniqueness of MST. Dimostrare che se i pesi dei lati sono tutti diversi allora esiste un unico MST.

Dimostrazione. Supp. p.a. che esistano 2 MST diversi A, B. Allora $\exists$ sicuramente almeno un lato che è in uno ma non nell'altro. Sia e_1 il lato di peso minore (scelta unica per via dell'ipotesi). Assumiamo

che $e_1 \in A$. B è un MST $\Rightarrow B \cup \{e_1\}$ contiene un ciclo C, che contiene e_1. A è un MST $\Rightarrow$ non può contenere cicli $\Rightarrow C$ ha un lato $e_2 \neq e_1$ che $\notin A$. Per come è stato scelto e_1 vale che $w(e_2) > w(e_1)$. Siccome $e_1, e_2 \in C$, rimpiazzando e_2 con e_1 in B si ottiene uno ST di peso inferiore di quello di B. Ma B era un MST $\Rightarrow$ assurdo. □

A.2.2 Esercizio 2

Il maximum spanning tree di un grafo è uno ST di costo max, cioè la cui somma $\sum_{e \in T} w(e)$ è max. Dare un algoritmo per questo problema che usi come procedura un agoritmo per risolvere il MST problem. Algoritmo:

* moltiplica per -1 i pesi di tutti i lati
* applica l'algoritmo di Kruskal

A.2.3 Esercizio 3

Sia il grafo $G = (V, E)$ connesso e con pesi sui lati distinti, e sia $\mathcal{T}$ l'insieme di tutti gli ST di G. Sia T' l'MST di G. Il **second-best minimum ST** è uno ST T tale che $w(T) = \min_{T'' \in \mathcal{T} \setminus \{T'\}} \{w(T'')\}$.
Mostrare che il second-best MST non è necessariamente unico. Esempio

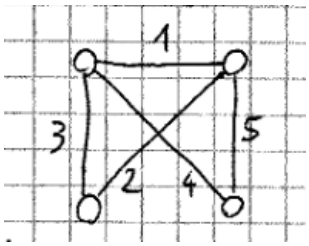


Figura 6: esempio.

C'è un unico MST di peso 7, ma 2 second-best MSTs di peso 8:

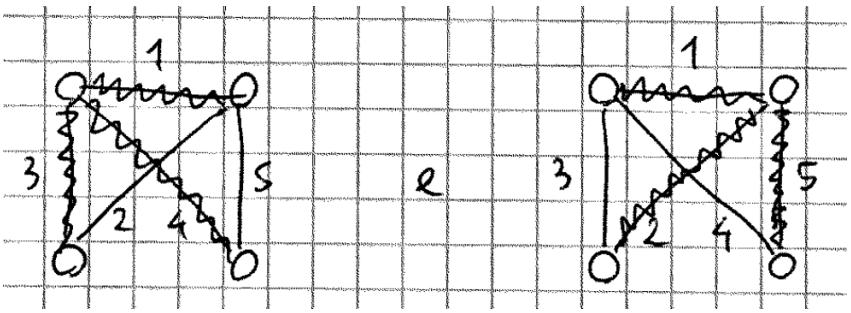


Figura 7: esempio.

A.3 Lezione 6

A.3.1 Esercizio 1

Modificare l'algoritmo di Dijkstra affinché restituisca anche i cammini minimi stessi e non solo la loro lunghezza.

`Dijkstra(G,s,f)`
$X \leftarrow \{s\}$
$len(s) \leftarrow 0$
foreach $v \in V$ **do**
    $len(v) \leftarrow +\infty$
while $e \in E : e = (v, u)$ con $v \in X \wedge u \notin X$ **do**
    $(v^*, u^*) \leftarrow$ such an edge minimizing $len(v) + \ell_{(v,u)}$
    $L_V[u^*].parent \leftarrow v^*$
    $X \leftarrow X \cup u^*$
    $len(u^*) \leftarrow len(v^*) + \ell_{(v^*,u^*)}$
return X

A.3.2 Esercizio 2

Considera un grafo $G = (V, E)$ diretto con pesi sui lati non negativi. Sotto quali condizioni esiste un unico shortest path da $s \in V$ a $t \in V$?

* Quando tutti i pesi sono interi positivi e distinti;
* Quando tutti i pesi sono potenze di 2 distinte;
* Quando vale 1. e il grafo non contiene cicli diretti

Risposta: **2**: due somme di potenze di 2 distinte non possono mai essere lo stesso numero (si pensi che i numeri siano scritti in binario). Per 1 e 3 ci sono controesempi.

A.3.3 Esercizio 3

Si consideri un grafo $G = (V, E)$ diretto con pesi sui lati non negativi. Si definisca il botteneck di un cammino come il massimo peso dei suoi lati (invece della somma dei pesi su tutti i lati). Modificare l'algoritmo di Dijkstra per calcolare, $\forall\ v \in V$, il bottleneck più piccolo su tutti i possibili cammini da uno starting node s a v. L'algoritmo deve avere complessità $\mathcal{O}(m \cdot n)$.

Risposta: rimpiazzare, nella 1a nella 3a riga del ciclo **while** dell'algoritmo di Dijkstra, $len(v) + \ell_{(v^*,w^*)}$ con $\max\{len(v), \ell_{(v,w)}\}$ e $len(v^*) + \ell_{(v^*,w^*)}$ con $\max\{len(v^*), \ell_{(v^*,w^*)}\}$.

`Dijkstra(G,s)`
$X \leftarrow \{s\}$
$len(s) \leftarrow 0$
$len(v) \leftarrow +\infty\ \forall\ v \neq s$
while there is an edge v, w with $v \in X$, $w \notin X$ **do**
    $(v^*w^*) \leftarrow$ such an edge minimizing $\mathbf{max}\{len(v), \ell_{(v,w)}\}$
    $X \leftarrow X \cup w^*$
    $len(w^*) = \mathbf{max}\{len(v^*), \ell_{(v^*,w^*)}\}$

A.4 Lezione 12

A.4.1 Esercizio 1

Un possibile miglioramento di `Approx_Vertex_Cover` è aggiungendo un solo nodo del lato scelto invece di entrambi i nodi. Dimostrare che esistono istanze (grafi) su cui questa strategia non raggiuinge nessuna approssimazione costante.

Risposta: consideriamo lo "**star graph**" (grafo con nodo centrare e collegato con tutti gli altri, che sono a loro volta collegati solo con il nodo centrale).
Se il nodo scelto è quell con identificatore ("nome") minore, e il nodo centrale ha l'identificatore più alto di tutti $\Rightarrow$ l'algoritmo modificato sceglie $n - 1$ nodi. Ma il vertex cover ottimo consiste del solo nodo centrale! □

A.4.2 Esercizio 2

Consideriamo il seguente algoritmo di approssimazione alternativo per Vertex Cover:

1. esegui una visita DFS del grafo da un vertice qualsiasi
2. restituisci i nodi non-foglia dell'albero DFS

Dimostrare che questo è un algoritmo di 2-approssimazione per vertex cover.

Risposta: quando si scrivono cose del genere, significa fare 2 cose:

1. dimostrare che ρ è al più 2;
2. dimostrare che esistono istanze per cui l'algoritmo non fa meglio (upper bound per l'algoritmo)

Dimostrazione. Quindi:

1. dato il grafo $G = (V, E)$ eseguiamo una visita DFS partendo da un vertice arbitrario. Questo ci restituisce l'insieme dei nodi interni di G, che chiamiamo V_i. I lati incidenti a V_i costituiscono un vertex cover poiché:
   - i nodi interni sono collegati tra di loro;
   - i nodi all'estremità sono collegati a loro volta con i nodi foglia.
2. ci accorgiamo che un qualsiasi matching A di V_i deve avere dimensione almeno $|V_i/2|$ dal momento che ogni vertex cover deve contenere almeno un vertice da ogni lato presente in A. Questa costituisce la soluzione ottima $|V^*|$ al problema di vertex cover, quindi:

$$\rho = \frac{|V_i|}{|V^*|} = \frac{|V_i|}{\frac{|V_i|}{2}} = 2$$

□

Risposta del prof: [*Suggerimento:* mostrare che il DFS-tree ottenuto contiene un matching opportuno e mettere in relazione in modo opportuno la taglia di questo matching con la taglia della soluzione ottima V^* e con la taglia della soluzione ritornata V'.]

Sia T un DFS-tree del grafo. Mostriamo che T contiene un matching[20] M. Nota che M deve essere piuttosto grande, visto che voglio dire qualcosa del tipo $|V'| \leq 2 \cdot |M|$ (come nell'analisi vista in classe).

[20] insieme di lati che non hanno nodi in comune

(I) dimostrare che ρ è al più 2:

(i) Sia r la radice di T e v uno dei suoi figli [tale che $L_V[v].father = r$, ndr].

* Aggiungo (r, v) in M.

(ii) $\forall$ livello $i \geq 1$ di nodi, considero i nodi di T del livello i che non sono nodi di qualche lato di M.
esempio: se una radice ha 5 figli, uno di essi è stato aggiunto al matching M mentre gli altri 4 sono i nodi che sto considerando.

* Per ognuno di questi nodi v, seleziono un figlio u e aggiungo v, u a M.
* Ripeto questa procedura fino alle foglie di T.

(iii) Per costruzione, in M ci sono tutti i nodi non-foglia di T, più (possibilmente) qualche nodo foglia di $T \Rightarrow |V'| \leq 2 \cdot |M|$. Abbiamo dimostrato che ρ è al più 2.

(iv) Ovviamente, per ogni matching qualsiasi M' del grafo, $|V^*| \geq |M'|$ [ricordo che stiamo confrontando il numero di nodi col numero di lati]. Quindi combino le due cose:

$$|V'| \leq 2 \cdot |M| \leq 2 \cdot |V^*| \Longrightarrow \frac{|V'|}{|V^*|} \leq 2.$$

(II) Dimostrare che non esistono istanze per cui l'algoritmo non fa meglio.
(dare quindi un upper bound all'algoritmo in cui il fattore ρ non può essere migliorato.)

(i) Consideriamo lo **star graph**, con nodo centrare c.
Se partiamo da uno dei nodi foglia u, quello che viene restituito da V' è l'insieme dei due nodi facenti parte del lato (u, c) quindi $V' = \{u, c\}$.

(ii) È evidente che, invece, la soluzione ottima V^* è costituita solamente dal nodo centrale c.

$\Rightarrow |V'| = 2 \cdot |V^*| \Rightarrow \rho = 2$.

A.4.3 Esercizio 3

Sia G^c il grafo complemento di G (contiene esattamente quei lati che non sono presenti in G). Se V^* è un vertex cover di G alllora vale che $V \setminus V^*$ è una clique di taglia massima in G^c (**clique:** il sottografo completo più grande che c'è). È possibile approssimare il problema clique applicando un algoritmo di approssimazione per vertex cover su G^c?

Risposta: NO. In generale le funzioni di riduzioni tra problemi non redservano l'approssimazione. In questo caso, supponiamo che la taglia della clique di taglia massima in G sia $|c_{max}| = \frac{n}{2}$. Quindi in G^c : $|V^*| = n - \frac{n}{2} = \frac{n}{2}$. Applico `Approx_VC`$(G^c) \rightarrow V'$. Potrebbe essere che $|V'| = n$ cioè $2|V^*|$.
Quindi viene restituita una clique di taglia $n - n = 0 \rightarrow$ non approssima per nessun $\rho(n)$.

A.4.4 Esercizio 4

$(P \neq NP)$

Dimostrare che `TSP` $<_p$ `TRIANGLE_TSP`, cioè che `TSP` si riduce in tempo polinomiale a `TRIANGLE_TSP`.

Dimostrazione. Supp. per assurdo che esista un algoritmo A polinomiale esatto per `TRIANGLE_TSP`.

Si definisca $< G = (V, E), c, k >$ un'istanza del problema TSP, con:

$G(V, E)$: grafo non diretto, completo e pesato con V vertici e E archi

c : funzione di costo $c : V \times V \mapsto \mathbb{N}$

k : upper-bound $\in \mathbb{N}$ del costo della soluzione al TSP.

Si riduca l'istanza generica $< G = (V, E), c, k >$ in una nuova istanza $< G' = (V, E), c', k' >$, con:

$\cdot\ c'(u, v) = c(u, v) + W$, con $W = \max_{u,v \in V} \{c(u, v)\}$,

$\cdot\ k' = k + |V| \cdot W$.

- **PARTE 1:** Si consideri $< G = (V, E), c, k >$ istanza di TSP che soddisfa la disuguaglianza triangolare di `TRIANGLE_TSP`. Allora:

$$c'(u, v) \overset{?}{\leq} c'(u, w) + c'(w, v)$$

$$c(u, v) + \max_{u,v \in V} \{c(u, v)\} \leq c(u, w) + \max_{u,w \in V} \{c(u, w)\} + c(w, v) + \max_{v,w \in V} \{c(v, w)\} \quad (1)$$

$$c(u, v) \leq c(u, w) + c(w, v) \text{ per ipotesi} \Rightarrow (1) \text{ vera solo se} \quad (2)$$

$$\max_{u,v \in V} \{c(u, v)\} \leq \max_{u,w \in V} \{c(u, w)\} + \max_{v,w \in V} \{c(v, w)\}. \text{ Vacuamente vero.} \quad (3)$$

$\Rightarrow (1)$ vera.

- **PARTE 2:** $G \in ham$ con costo $k \Leftrightarrow G' \in ham$ con costo k'

  1. $G \in ham \Rightarrow c(C^*) = |V| \Rightarrow A$ ritorna un ciclo C^* con con cardinalità $\leq |V|$ e costo pari a k. Allora anche G' dispone del medesimo circuito con cardinalità $\leq |V|$ con costo k', per ipotesi.
  2. $G \notin ham \Rightarrow C$ contiene almeno un arco $\notin G$ con $c(C^*) = k$. Anche il ciclo C' che attraversa G' contiene almeno un arco $\notin G'$, con $c'(C) = k'$. Ma $\forall\ (u, v) \in G'$, $c'(u, v) = c(u, v) + W$, con $W \geq 0 \Rightarrow$ le funzioni di costo c e c' differiscono solo dal termine W. Quindi l'algoritmo A (algoritmo polinomiale) ha risolto il problema `TRIANGLE_TSP`, che richiede un tempo non polinomiale. Pertanto, se $P \neq NP$, ciò è un assurdo. Quindi $\nexists\ A$.

□

A.5 Lezione 18

A.5.1 Esercizio 1

Ipotesi:

(i) A_Π Las Vegas (sempre corretto), con $T_{A_\Pi}(n) = \mathcal{O}(f(n))$ whp, in particolare

$$Pr\big(T_{A_\Pi}(n) \geq c \cdot f(n)\big) \leq \frac{1}{n^d};$$

(ii) A_Π ha una complessità al caso peggiore deterministico $\mathcal{O}(n^a)$, $a \leq d\ \forall\ n$

Dimostrare che $\mathbb{E}\big[T_{A_\Pi}(n)\big] = \mathcal{O}(f(n))$. Abbastanza ovvio visto che devo dimostrare il verificarsi di una cosa, che si verifica con alta probabilità, con rispetto alla sua media.
$\Rightarrow$ se qualcosa succede whp significa che succede anche in media (non è automaticamente vero il viceversa).

Dimostrazione. Uso la disuguaglianza di Markov.

Lemma di Markov: sia T una variabile aleatoria intera, non negativa e limitata $\big(\exists\, b \in \mathbb{N} : Pr\big(T > b\big) = 0\big)$.
Allora $\forall\ t : 0 \le t \le b$
$t \cdot Pr\big(T \ge t\big) \le \mathbb{E}[T] \le t + (b - t) \cdot Pr(T \ge t)$

Usiamo il limite superiore del lemma per dimostrare questo esercizio:

$$\mathbb{E} \le \underbrace{c \cdot f(n)}_{t} + \Big(\overbrace{n^a}^{b} - \underbrace{\overbrace{c \cdot f(n)}^{t}}_{\text{elim. per magg.}}\ \Big) \cdot \underbrace{\frac{1}{n^d}}_{Pr(T \ge t)} \le c \cdot f(n) + \frac{n^a}{n^d} \le c \cdot f(n) + 1 \Rightarrow \mathbb{E}[T] \le c \cdot f(n)$$

□