Domande di ASD

Nahir Pereira – v.pereira1@studenti.unipi.it

Anno Accademico 2023-24

Indice

1 Definisci cosa significa riducibilità polinomiale

Siano Π_1, Π_2 problemi decisionali (problemi che ammettono come risposta sì o no) definiti su Σ alfabeto. Diremo che Π_1 è polinomialmente riducibile a Π_2 ($\Pi_1 \ltimes \Pi_2$) se $\exists\ T$ algoritmo deterministico polinomiale tale che $\forall x \in \Sigma^*, x \in \Pi_1 \iff T(x) \in \Pi_2$; ovvero T è una "funzione" che trasforma l'algoritmo di risoluzione di Π_1 in un algoritmo di risoluzione di Π_2 e l'algoritmo T opera in tempo polinomiale rispetto alla dimensione di x.

2 Dimostra che il problema hamiltoniano/ciclo euleriano si riduce polinomialmente al problema del commesso viaggiatore

Innanzitutto descriviamo i due problemi:

- **problema hamiltoniano (chiamato anche criterio di eulero o HAM)**. Sia $G = (V, E)$ grafo. Un ciclio euleriano è un cammino in G che tocca tutti i nodi una volta sola. Perhcé sia possibile trovare tale cammino G dev'essere connesso e tutti i nodi devono avere grado pari.
- **problema del commesso viaggiatore (chiamato anche TSP)**. Date n città ho la matrice $M[n][n]$ delle distanze fra una città e l'altra voglio trovare il cammino minimo con distanza k.

A questo punto posso dimostrare $\Pi_1 \propto \Pi_2$:

- **Trovo la trasformazione** $T(x)$. Prendiamo un nuovo grafo $G' = (V, E')$ dove ogni coppia di nodi $u, v \in V$ è connessa da un arco e diamo i valori $d_{uv} = 0$ se $u = v$; $d_{uv} = 1$ se $u, v \in E$; $d_{uv} = 2$ se $u, v \notin E$.
- **HAM** $\implies$ **TSP** Se G ammette un ciclo hamiltoniano allora ho un ciclo hamiltoniano anche in G' che visita ogni vertice una volta sola e ha costo pari a n.
- **TSP** $\implies$ **HAM** Supponiamo che esista una soluzione al problema TSP in G', cioè un ciclo che visita ogni vertice con costo n, dato che il costo è n, il ciclo deve essere costituito solo da archi che appartengono a E, ovvero archi in G. Quindi, un ciclo del TSP in G' corrisponde a HAM in G.

3 Parlami della visita DFS e l'albero DFS e fai un esempio

```cpp
#include <iostream>
#include <vector>
using namespace std;

void visitaDFS(int u, bool visitato[], vector<int>& adj) {
    if (visitato[u]) {
        return;
    }

    visitato[u] = true;

    for (int x : adj[u]) {
        visitaDFS(x, visitato, adj);
    }
}

```

L'algoritmo DFS esplora il grafo partendo da un nodo, visitando i nodi adiacenti in profondità (ovvero, prosegue lungo un ramo finché non raggiunge un nodo che non ha nodi adiacenti non ancora visitati) e, quando non ci sono più nodi da esplorare lungo un ramo, torna indietro ed esplora gli altri rami. L'albero DFS è un sottoinsieme del grafo in cui la radice è il nodo iniziale della visita, i figli sono quelli visitati direttamente dal padre durante l'esplorazione DFS.

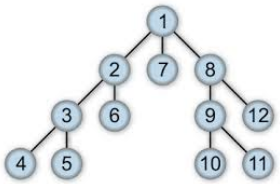


Figure 1: Albero DFS numerato con l'ordine della visita DFS

4 Sia R relazione di equivalenza sui nodi di $G = (V, E)$ grafo | $u \sim_R v$ se esiste un cammino da u a v e viceversa. Trova le classi di equivalenza e costruisci il grafo che ha per nodi le classi di equivalenza di R e dire cosa succede se questo nuovo grafo ha un ciclo

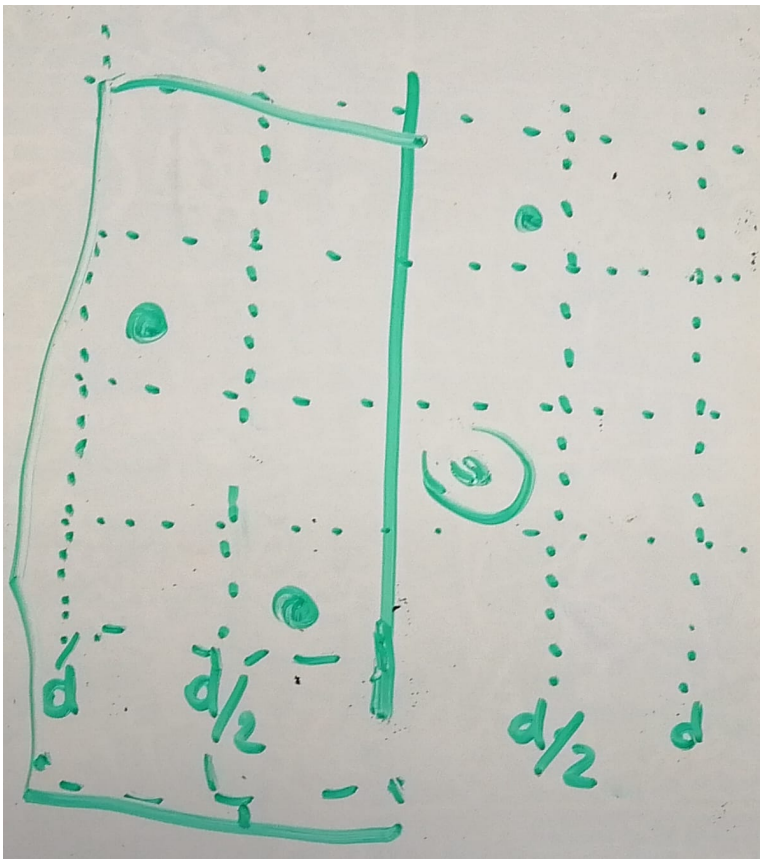
Le classi di equivalenza di R corrispondono alle componenti fortemente connesse del grafo (ovvero l'insieme dei nodi tali che ogni nodo può essere raggiunto da ogni altro nodo nel sotto-grafo).
Costruiamo $G' = (V, E')$ in cui i nodi solo le classi di equivalenza di G. Gli archi sono determinati così:

- Se in G siano $u \in C_1$ e $v \in C_2$ se c'è un arco che collega u e v in G allora ci sarà un arco fra C_1 e C_2 in G'.
- Se in G' c'è un arco fra C_1 e C_2 allora esiste un cammino fra un $u \in C_1$ e un $v \in C_2$.

Se in G' esiste un ciclo allora esiste una sequenza di componenti fortemente connesse tali che esistono cammini tra di esse. Ma quindi in G ci sono cammini tra i nodi delle componenti coinvolte in questo ciclo che formano un ciclo di nodi tra di esse.

5 Spiega il problema della coppia di punti più vicina

Siano n punti in un piano. Voglio trovare la coppia più vicina. Divido il piano in due tale che io abbia $\frac{n}{2}$ punti a sinista e $\frac{n}{2}$ punti a destra. Ricorsivamente, divido tutto il piano. A quel punto, chiamo d_s la minima distanza nell'insieme a sinistra e d_d la minima distanza dell'insieme a destra. Sia $d = \min(d_s, d_d)$. A questo punto gli unici punti che potrebbero essere più vicini di d sono quelli vicino la linea di separazione. Dunque chiamo 0 la linea di separazione. Prendo la parte di piano $[-d, d]$ e calcolo la distanza fra i punti presenti in quella parte di piano creando una griglia di quadrati $\frac{d}{2} \times \frac{d}{2}$. Per ogni punto controllo se ci sono punti a due quadrati di distanza in ogni direzione e se ci sono calcolo la distanza.



6 Parla dell'equivalenza SAT e 3-SAT

Dati $x_1, ..., x_n$ booleani definisco:

- letterali, ovvero x_i, $\neg x_i$;
- clausule, ovvero unione di letterali;
- formule, ovvero intersezione di clausule.

Il problema SAT è un problema NP-completo che vuole capire se una formula booleana è soddisfabile, ovvero se posso attribuire un valore tale che la formula sia vera. Il problema 3-SAT è una restrizione del problema SAT in cui ogni clausola è composta da esattamente tre letterali.
È possibile riburre polinomialmente SAT a 3-SAT. Infatti data una formula in cui ogni clausola può avere una lunghezza arbitraria possiamo applicare una trasformazione per "allineare" tutte le clausole a una lunghezza di 3 letterali:

- Se una clausola ha 1 letterale possiamo espanderla a una clausola con tre letterali; ad esempio (x_1) diventa $(x_1 \vee y \vee \neg y)$;
- Se una clausola ha 2 letterali possiamo estenderla a una clausola con tre letterali; ad esempio $(x_1 \vee x_2)$ diventa $(x_1 \vee x_2 \vee y)$, dove y è una nuova variabile ausiliaria.
- Se una clausola ha più di 3 letterali, possiamo suddividerla in clausole con 3 letterali.

Dunque ogni formula in SAT può essere trasformata in una formula con la stessa soddisfacibilità ma che appartiene a 3-SAT. Dunque SAT è riducibile in modo polinomiale a 3-SAT.

7 Parla dell'algoritmo Dijkstra e del suo costo

L'algoritmo di Dijkstra serve a trovare il cammino minimo da un nodo sorgente s agli altri nodi di un grafo pesato.

```
1  void Dijkstra(int s){
2      for(int i=0;i<n;i++){
3          prec[i]=-1;  // Inizializzazione dei precedenti
4          dist[i]=\infty;  // Inizializzazione delle distanze
5      }
6      prec[s]=s;  // Il nodo sorgente    il suo stesso
       predecessore
7      dist[s]=0;  // La distanza dal nodo sorgente a se stesso
       0
8      for(int i=0;i<n;i++){
9          PQ.Enqueue(peso,i);  // Aggiungiamo tutti i nodi nella
       coda con peso infinito
10     }
11     while(!PQ.Empty()){
12         e = PQ.Dequeue();  // Estraiamo il nodo con la distanza
       minima
13         v = e.dato;
14         for(x : adj[v]){  // Per ogni nodo adiacente di v
15             if(dist[x.key] > dist[v] + x.peso){  // Se il
       cammino trovato    pi  corto
16                 prec[x.key] = v;  // Aggiorniamo il predecessore
17                 PQ.DecreaseKey(x.key, dist[x.key]);  //
       Aggiorniamo la coda di priorit
18             }
19         }
20     }
21 }
22
```

8 Parlami della complessità di Kolmogorov con le 3 proprietà e dimostrazioni - Parlami delle stringhe random e relativo problema di indicibilità

La complessità di Kolmogorov $K(x)$ di una stringa x è la "randomicità" di x: $K_L(x)$ è la lunghezza minima del programma P nel linguaggio di programmazione L che, dato in input la stringa vuota, dà come output x. Una stringa è più "randomica" più è lungo il programma che la descrive.
La complessità di Kolmogorov di x è data da: $K(x) = \min_P\{$ lunghezza del programma P che produce $x\}$; dunque se $K(x)$ è simile alla lunghezza di x (infatti la lunghezza massima sarà sempre la lunghezza di x: il programma che stampa x carattere per carattere) allora x non è comprimibile.
La complessità di Kolmogorov ha tre proprietà principali:

- **Esistenza di una stringa casuale per ogni lunghezza**. Vogliamo dimostrare che $\forall n\ \exists x \in \{0,1\}^n |\ K(x) \geq n$. Supponiamo che S sia l'insieme delle stringhe di lunghezza n che hanno una complessità di Kolmogorov inferiore a n. Il numero delle stringhe di lunghezza n è 2^n, mentre il numero di programmi che possono generare stringhe di lunghezza inferiore a n è limitato da una funzione esponenziale più piccola, come 2^{n-1}. Infatti $|S| = 2^0 + 2^1 + \cdots + 2^{n-1} = 2^n - 1$. Quindi, la probabilità che una stringa di lunghezza n sia casuale è $1 - 2^{-c}$ con c costante.

- **Probabilità di stringhe casuali**. La probabilità che una stringa x di lunghezza n sia casuale è molto alta se $K(x)$ è almeno $n-c$ con c costante. Infatti, per quanto detto prima $|S| = 2^0 + 2^1 + \cdots + 2^{n-c-1} = 2^{n-c} - 1$. Dunque la probabilità è: $P(K(x) \geq n-c) = \frac{|S|}{|\{0,1\}^n|} = \frac{2^{n-c}-1}{2^n} = 2^{-c} - \frac{1}{2^n} \approx 2^{-c}$.

- **Problema di indicibilità di** $K(x)$. Il problema di indicibilità riguarda la difficoltà di determinare se x è casuale. Sia R l'insieme $R = \{x \in \{0,1\}^n \mid K(x) \geq n-c\}$. R è indicibile perché non esiste un algoritmo che, dato x, possa decidere in tempo finito se $K(x) \geq |x| - c$. Supponiamo che esista A_R algoritmo che verifica se $K(x) \geq |x| - c$. Sia B un algoritmo che genera tutte le stringhe di $\{0,1\}^n$ in ordine lessicografico, e $A_R(x)$ si ferma quando trova $x \in R$. Allora $K(x)$ per quella stringa è almeno $n-c$.

Tuttavia, B_n genera la stringa più piccola x_n che soddisfa $K(x_n) \geq n - c$: assurdo!

9 Definisci P, NP e NP-completo

Chiameremo P l'insieme dei problemi riducibi in tempo polinomiale. Un problema appartiene alla classe P se esiste un algoritmo che risolve il problema in un tempo esprimibile come una funzione polinomiale della dimensione dell'input $O(n^k)$, con n dimensione dell'input e k costante. Un esempio di un problema in P è l'ordinamento di una lista di numeri.
Chiameremo NP l'insieme dei problemi verificabili in tempo polinomiale, ovvero è possibile controllare se la soluzione proposta del problema è vera in tempo polinomiale. Un esempio di un problema in NP è il problema del ciclo hamiltoniano.
Chiameremo NP-completi i problemi Π tali che $\Pi \in NP$ e $\forall \Pi' \in NP \implies \Pi' \propto \Pi$. Da qui un importante risultato teorico è che se possiamo trovare una soluzione in tempo polinomiale per uno di questi problemi, possiamo risolvere tutti i problemi in NP in tempo polinomiale.
Uno dei sette problemi del millennio è "$P = NP$?" Sicuramente $P \subseteq NP$ perché se posso risolvere un problema in tempo polinomiale allora posso anche verificarlo in tempo polinomiale. Ma se $P = NP$ allora esisterebbe un algoritmo per risolvere tutti i problemi in NP in tempo polinomiale; invece se $P \neq NP$, significa che non esistono algoritmi di tempo polinomiale per i problemi NP-completi.

10 Spiega il QuickSort randomizzato - relazione tra alberi binari di ricerca e il QuickSort randomizzato

```cpp
int Distrib(vector<int>& A, int sx, int dx) {
    int pivotIndice = sx + rand() % (dx - sx + 1);
    swap(A[pivotIndice], A[dx]);

    int pivot = A[dx];
    int i = sx - 1;

    for (int j = sx; j <= dx - 1; j++) {
        if (A[j] <= pivot) {
            i++;
            swap(A[i], A[j]);
        }
    }

    swap(A[i + 1], A[dx]);
    return i + 1;
}

void QuickSortRand(vector<int>& A, int sx, int dx) {
    if (sx < dx) {
        int pivotIndice = Distrib(A, sx, dx);

        QuickSortRand(A, sx, pivotIndice - 1);
        QuickSortRand(A, pivotIndice + 1, dx);
    }
}

```

La complessità temporale media del Quick Sort è $O(n \log(n))$. Nonostante anche il Merge Sort e Heap Sort abbiano complessità $O(n \log(n))$, il Quick Sort è generalmente più veloce in pratica, sia perché è un algoritmo in place (ovvero non usa memoria aggiuntiva durante l'ordinamento, dunque a parità di tempo è migliore a livello di memoria rispetto al merge sort) con costo $O(n \log(n))$ in memoria che perché nel caso medio il pivot divide l'array circa a metà, riducendo il numero di chiamate ricorsive.
Il problema del Quick Sort è che la sua complessità nel caso peggiore è $O(n^2)$, quando l'array è già ordinato o quasi ordinato. Il Randomized Quick Sort, scegliendo un pivot casuale riduce la possibilità di trovare il caso $O(n^2)$.

Probabilità:

$$E[X] = \sum_{i<j} E[x_{ij}] = \sum_{i<j} Pr[x_{ij} = 1] \leq \sum_{i<j} \frac{2}{j-i+1}$$
$$\leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \sim \sum_{i} \sum_{k=2}^{n-i+1} \frac{2}{k} \sim O(n \log(n)) \quad (1)$$

La relazione tra alberi binari di ricerca e il QuickSort randomizzato è che posso porre radice dell'albero come se fosse il pivot. Questa relazione d'ordine è transitiva perché preso un nodo è come se fosse la radice di un nuovo albero, nel quick sort sarebbe il nuovo pivot per la separazione.

11 Dimmi qual è il limite inferiore di tempo per ordinare un array e dimostralo

Il tempo minimo di ordinamento di un array di n elementi è $O(n \log n)$.
Per un array di n elementi, esistono $n!$ permutazioni possibili.Ogni algoritmo di ordinamento può essere rappresentato da un albero delle decisioni, dove ogni nodo rappresenta un confronto tra due elementi dell'array. Ogni ramo dell'albero corrisponde a uno dei due esiti possibili del confronto: $x < y$ oppure $x \geq y$. Dato che ogni confronto ha solo due esiti possibili, tale albero è binario. La profondità è il numero di confronti necessari per determinare l'ordinamento corretto dell'array. Per determinare la profondità minima dell'albero, dobbiamo considerare che un albero binario di profondità h ha al massimo 2^h foglie. Per distinguere tra $n!$ permutazioni, l'albero delle decisioni deve avere almeno $n!$ foglie, ovvero voglio h tale che $2^h \geq n!$. Dunque voglio $h \geq \log_2(n!)$. Grazie alla formula di Stirling $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ da cui $\log_2(n!) \approx n \log_2(n) - n \log_2(e)$. Semplificando $\log_2(n!) = O(n \log n)$.

12 Spiega il problema della fermata

Il problema della fermata consiste nel determinare se dato A algoritmo e D input l'algoritmo termina quando eseguito su D. Definiamo quindi la funzione:

$$\texttt{termina(A, D)} = \begin{cases} \text{TRUE} & \text{se } A \text{ termina su } D \\ \text{FALSE} & \text{se } A \text{ non termina su } D \end{cases}$$

Sia poi la funzione:

$$\texttt{paradosso(x)}\{\text{while(termina(x, x));}\}$$

Noto che se `termina(x, x)` = TRUE `paradosso(x)` non termina mai; invece `termina(x, x)` = FALSE `paradosso(x)` termina.
Quindi se `termina(paradosso, paradosso)` = TRUE allora `paradosso(x)` non termina mai. Ma questo è un contraddizione, perché `termina(paradosso, paradosso)` = TRUE. Invece se `termina(paradosso, paradosso)` = FALSE allora `paradosso(x)` non termina. Ma questo è un contraddizione, perché `termina(paradosso, paradosso)` = FALSE. Allora non esiste una funzione `termina(A, D)` che possa determinare in modo corretto se un algoritmo termina su un dato input. Quindi il problema della fermata è indecidibile.

13 Spiega i diversi tipi di Hashing

L'hash è una funzione $h_{a,b} : U \to [m]; h_{a,b}(x) = (ax + b\%)\%m$. Serve soprattutto per la gestione delle strutture dati. I valori della funzione vengono inseriti in una tabella hash. Quando due elementi hanno lo stesso valore hash si dice che si verifica una collisione. Esistono diversi modi di risolvere la collisione in una tabella hash.
La lista di trabocco prevede l'uso di una struttura di dati ausiliaria per memorizzare tutti gli elementi che condividono lo stesso valore di hash. Quando si inserisce un elemento nella tabella hash, si calcola il suo hash e si verifica se c'è già un elemento nella stessa posizione: se c'è già un elemento, il nuovo elemento viene semplicemente aggiunto in coda alla lista di trabocco in quella posizione.
L'indirizzamento aperto prevede che quando si verifica una collisione si cerca una "posizione libera" direttamente all'interno della tabella stessa. Si può gestire in diversi modi ad esempio cercare la prima posizione libera successiva o cercare la prima posizione libera successiva usando una funzione quadratica.
Il cuckoo hashing è una tecnica che dà a ogni elemento due possibili posizioni nella tabella hash. Quando un elemento viene inserito nella tabella, se entrambe le posizioni sono occupate, l'elemento che occupa una di esse viene "spostato" nella sua seconda posizione. Questo processo continua ricorsivamente finché tutte le posizioni della tabella non sono assegnate correttamente.