Appunti Analisi e Progetto di Algoritmi

Domande e Risposte

Capitolo 1 – Tecnica Greedy

1) Descrivere la tecnica Greedy

La tecnica Greedy è un paradigma di progettazione di algoritmi utilizzato per risolvere problemi di ottimizzazione. La caratteristica principale della tecnica Greedy è quella di effettuare scelte vincenti sui sottoproblemi da risolvere, ossia prende decisioni passo dopo passo, selezionando in ogni istante l'opzione che sembra essere la migliore in quel momento, senza controllare che il procedimento complessivo porti alla soluzione ottimale globale.

2) Cosa si intende per sistema di indipendenza? Fornire degli esempi

Data la coppia (E, F). Essa è definita sistema di indipendenza se:

→ E è un insieme finito ($|E| < \infty$)

→ F è una famiglia di sottoinsiemi di E ($F \subseteq P(E)$) tale che $\forall A \in F, \forall B \subseteq A \rightarrow B \in F$

I sottoinsiemi di sottoinsiemi sono a loro volta ammissibili.

I possibili esempi di sistema di indipendenza sono:

Esempio 1

→ E finito;

→ $F = P(E)$

Esempio 2

→ $E \leq V$ spazio vettoriale, insieme finito di vettori.

→ $F = \{A \subseteq E | \sum_{i=1}^{|A|} v_i A_i \neq \vec{0}\}$

Esempio 3

→ E insieme degli archi di $G = (V, E)$ non orientato

→ $F = \{A \subseteq E | (V, A) \text{ è } una\ foresta\ e\ un\ sottografo\}$

3) Definire un problema di massimo associato ad un sistema di indipendenza pesato

Sia ora $\mathbb{R}^+$ l'insieme dei reali positivi e sia la funzione peso $w: E \rightarrow \mathbb{R}^+$. Tale funzione può essere estesa ai sottoinsiemi di E ponendo $\forall A \subseteq E, w(A) = \sum_{x \epsilon A} w(x)$.

Quindi si ha:

→ Istanza: un sistema di indipendenza $\langle E, F \rangle$ e una funzione peso $w: E \rightarrow \mathbb{R}^+$.

→ Soluzione: un insieme $M \in F$ tale che $w(M)$ sia massimo.

L'algoritmo Greedy è definito dalla seguente procedura:

```
Procedure MIOPE
begin
    S := ∅
    Q := (l_1, l_2, ··· , l_n)
    Q := SORT(Q)
    for i = 1, 2, ..., n do
        if S ∪ Q[i] ∈ F then S := S ∪ Q[i]
end
```

Risulta possibile concludere che la procedura MIOPE richiede al più un tempo $T(n) = O(n\,logn + nC(n))$.

4) Quando un sistema viene definito indotto?

Sia (E,F) un sistema di indipendenza e sia $C \subseteq E$. Si definisce sistema indotto (o sottosistema) (C,F_c) la coppia in cui $F_c = \{A \in F \mid A \subseteq C\}$ da C.

5) Cos'è il massimale?

Sia (E,F) un sistema di indipendenza, $A \in F$ è massimale solo se $\forall e \in E \setminus A$, $A \cup \{e\} \notin F$, cioè non è estendibile. Se $A \in F$ è massimale in (E,F), allora $A \cap C$ è massimale in (C,F_c).

Capitolo 2 – Matroidi e Teorema di Rado

6) Cosa si intende per matroide?

Un sistema di indipendenza (E,F) è un matroide se e solo se vale la seguente proprietà di scambio: $\forall A,B \in F \wedge |B| > |A| \wedge \exists b \in B \setminus A : A \cup \{b\} \in F$
Per ogni coppia, deve esistere un elemento che se aggiunto da B ad A, ciò che si ottiene è ancora parte del sistema di indipendenza.

7) Enunciare e dimostrare il teorema che definisce un matroide

Teorema: Sia $\langle E,F\rangle$ un matroide, solo se $\forall C \subseteq E$, tutti gli insiemi massimali di $\langle C,F_C\rangle$ hanno la stessa cardinalità.
Dimostrazione: Per assurdo $|B| > |A|$, quindi A è estensibile e non massimale. Quindi $|A| = |B|$.

8) Enunciare il teorema di Rado

Sia $\langle E,F\rangle$ un sistema di indipendenza, allora $\langle E,F\rangle$ è un matroide se e solo se $\forall w: E \to \mathbb{R}^+$, la $\texttt{Greedy\_Max}(E,F,w)$ risolve il problema di massimo associato, ovvero calcola un sottoinsieme massimale di peso associato.

9) Dimostrare il teorema di Rado

Sia $\langle E,F\rangle$ sia un matroide e sia $w: E \to \mathbb{R}^+$ tale che $\texttt{Greedy\_Max}(E,F,w)$ risolve il problema di massimo associato.

Sia $S = \{a_1, a_2, \dots, a_p\}$ con soluzione $w(a_1) \geq w(a_2) \geq \dots \geq w(a_p)$.
Per un qualunque sottoinsieme $S' \in F$ massimale in F si vuole dimostrare che $w(S) > w(S')$.
Siccome S' è massimale ed $\langle E, F\rangle$ è un matroide, vale che $|S| = |S'| = p$.
Sia $S' = \{b_1, b_2, \dots, b_p\}$ con soluzione $w(b_1) \geq w(b_2) \geq \dots \geq w(b_p)$ e si assume che $S \cap S' = \emptyset$. Considerando i 2 seguenti casi mutualmente esclusivi:
1) $\forall i \in \{1, \dots, p\}\, w(a_i) \geq w(b_i) \rightarrow w(S) \geq w(S')$
2) $\exists i \in \{1, \dots, p\}\, w(a_i) < w(b_i)$. Sia k il più piccolo indice i, vale allora che: $w(a_k) < w(b_k) \leq w(b_{k-1}) \leq w(a_{k-1})$, ma per $i = k - 1$ non vale $w(a_i) < w(b_i)$
Sia $C = \{e \in E |\, w(e) \geq w(b_k)\}$. Si avrebbe che:
→ $b_1, \dots, b_k \in C$
→ $a_1, \dots, a_{k-1} \in C\ ma\ a_k \notin C$
Pertanto $|S' \cap C| > |S \cap C|$ ed è una contraddizione in quanto $|S' \cap C| = |S \cap C|$ dato che se $\langle E, F\rangle$ è un matroide, allora lo è anche $\langle C, F_C\rangle$. Si ha dimostrato il caso 1.
Per dimostrare il caso 2 implica il caso 1 è sufficiente dimostrare che il negato del caso 1 implica il negato del caso 2. Supponendo che $\langle E, F\rangle$ non sia un matroide, allora $\exists A, B \in F_C$ massimali con $|A| > |B| = p$. Si definisce $w: E \rightarrow \mathbb{R}^+$, $\forall e \in E$ si avrebbe

$$w(e) = \begin{cases} p + 2 \\ p + 1 \\ 1 \end{cases}$$

Vale $w(e) = p + 2$ se $e \in B$. Vale $w(e) = p + 1$ se $e \in B \setminus A$ e 1 altrimenti.
Si ha $w(B) = p * (p + 2) = p^2 + 2p$ e si ha B come massimale. Vale che $w(A) = (p + 1)^2 = p^2 + 2p + 1$. Ne consegue che `Greedy_Max` non sta calcolando l'ottimo perché A e un candidato migliore di B.

Capitolo 3 – Insiemi disgiunti

10) Cos'è una struttura dati per insiemi disgiunti? Definire formalmente quali sono le operazioni principali su una struttura dati per insiemi disgiunti

Una struttura dati per insiemi disgiunti è una collezione $S = \langle s_1, \dots, s_k\rangle$ di insiemi disgiunti. Ogni insieme è identificato da un rappresentante, uno tra i membri dell'insieme stesso. Le operazioni supportate sono:
→ `makeSet(x)`: crea un nuovo insieme con un singolo membro;
→ `union(x, y)`: unisce i due insiemi contenenti x e y;
→ `findSet(x)`: ritorna il rappresentante dell'insieme in cui x è contenuto.

11) Illustrare i possibili modi con cui è possibile rappresentare una struttura dati per insiemi disgiunti e complessità relative

Risulta possibile rappresentare una struttura dati per insiemi disgiunti in due modi:

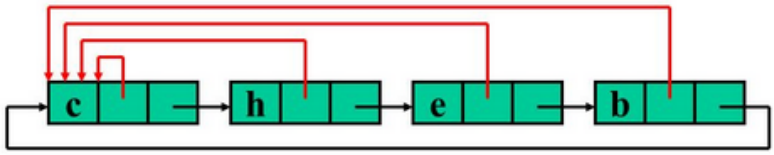
→ Lista: in cui ogni nodo è costituito da: valore, puntatore alla testa, puntatore a next, puntatore alla coda (solo il rappresentante).

Complessità:

a) `makeSet(x)`: $O(1)$

b) `union(x,y)`: $O(n)$

c) `findSet(x)`: $O(1)$

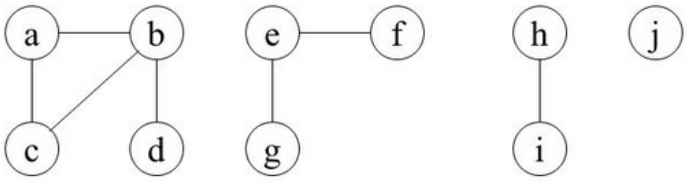


→ Foresta: in cui gli elementi di ogni albero hanno solo un puntatore al proprio genitore. La radice contiene in più un puntatore al rappresentante.

a) `makeSet(x)`: $O(1)$

b) `union(x,y)`: $O(1)$

c) `findSet(x)`: $O(n)$



12) Descrivere le differenze tra euristica unione pesata, euristica unione per rango e compressione dei cammini

Euristica unione pesata: Ogni lista conterrà anche un attributo con la sua lunghezza, così che nella `union(x, y)`, la lista più corta viene aggiunta a quella più lunga. Allora m operazioni vengono fatte in $O(m + n \cdot log\ n)$: dato che un singolo elemento può aggiornare il suo rappresentante $log\ n$ volte e che ci sono n elementi, arriviamo a tale tempo.

Euristica unione per rango: L'idea è di fare in modo di unire un albero "corto" in subordine alla radice di uno lungo piuttosto che il contrario. Ad ogni nodo è associato un rango, ovvero il limite superiore per l'altezza del nodo, vale a dire il numero di archi del cammino più lungo fra sé ed una foglia.

Compressione dei Cammini: Visto che lo scorrimento dell'albero comporta

maggiore inefficienza a livello di complessità computazionale, risulta possibile modificarlo intanto per migliorare i metodi `findSet(x)` future.

```
findSet(x):
     if x != P(x)
          P(x) = findSet(P(x))
     return P(x)
```

Ad ogni nodo viene assegnato come parent (indicato per comodità P) il rappresentante del proprio parent e questo permette di modificarlo per ottenere una struttura che migliorerà le `findSet(x)` future.
L'unione per rango porta ad un tempo $O(m \cdot log\ n)$. Se aggiungiamo la nuova `findSet` arriviamo ad un tempo $O(m \cdot \alpha(m,n))$, $\alpha \leq 4$ ovvero un tempo lineare.

Capitolo 4 – MST (Minimum Spanning Tree)

13) Definire formalmente cos'è un albero di copertura minimo (MST)

Dato un grafo non orientato $G = (V, E)$, dove:
→ V è l'insieme dei vertici;
→ E è l'insieme degli archi;
→ ogni arco $e \in E$ ha un peso associato $w(e)$, dove $w: E \to \mathbb{R}$ è una funzione che assegna un peso reale a ogni arco.
Si definisce albero di copertura minimo di G (o MST – Minimum Spanning Tree), la coppia (V, T) con $T \subseteq E$ tale che T formi un albero che copre tutti i vertici V e (V, T) è aciclico e connesso. La somma dei pesi degli archi deve essere minima, cioè $min \sum_{e \in T} w(e)$.

14) Illustrare l'approccio greedy generico (fornire la procedura GENERIC – MST) per determinare un MST mostrando il principio di funzionamento (cosa fa la procedura ad ogni iterazione, proprietà invariante di ciclo, nozione di arco sicuro, ecc…). Come si trova un'arco sicuro? Definizioni di: taglio, arco che attraversa un taglio, taglio che rispetta un sottoinsieme di archi, arco leggero.

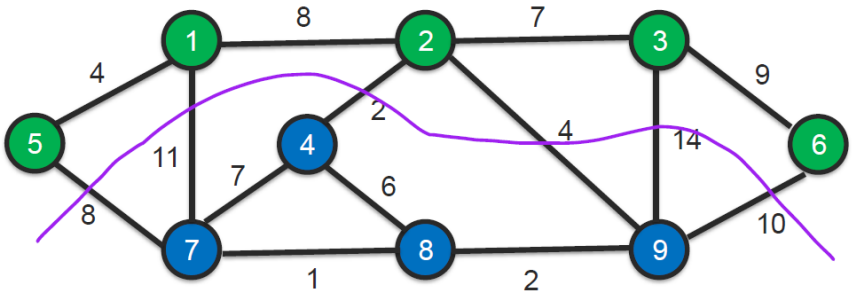
Prima di fornire la descrizione dell'approccio greedy generico per determinare la MST, risulta necessario definire l'arco sicuro. Dato $A \subseteq T$, un arco (u, v) è sicuro se $\{(u, v)\} \cup A \subseteq T$.
L'approccio greedy generico che determina un MST è il seguente:
→ inizializza un insieme A vuoto;
→ aggiunge ad ogni passo un arco (u, v) sicuro;
→ l'algoritmo termina non appena $A = T$, ovvero $G_A = (V, A)$ è MST.

Si illustra la seguente procedura:

```
GENERIC-MST(G,W)
    A ← ∅
    while |V|-|A| > 1 do
        trova arco (u,v) sicuro per A
        A = A U {(u,v)}
    return G_A = (V,A)          //MST
```

… cioè le componenti connesse di G_A diventano una sola (un solo albero) e quindi |V|-|A| diventa uguale a 1

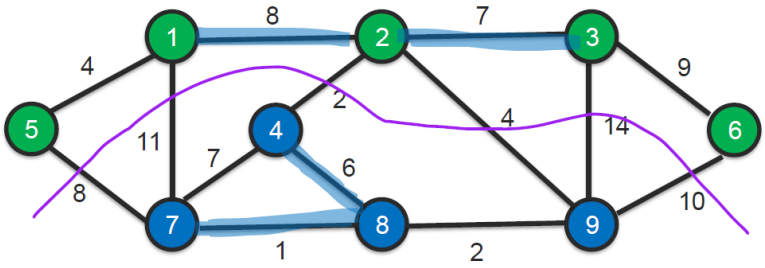
→ Dal momento che ogni iterazione aggiunge ad A l'arco sicuro, l'insieme A si mantiene aciclico durante le iterazioni (è sempre sottoinsieme di T per definizione di arco sicuro).
→ Ad ogni iterazione, $G_A = (V, A)$ è una foresta composta da un numero di alberi (componenti connesse) pari a $|V| - |A|$.
→ Inizialmente G_A è composta da $|V|$ alberi (i singoli vertici), dal momento che A è vuoto e quindi $|A|$ è uguale a 0.
→ Ogni iterazione riduce di 1 il numero di componenti connesse perché l'arco sicuro (u, v) che viene aggiunto va a collegare due componenti distinte di G_A.
→ Non appena la componente connessa è una sola (cioè diventa MST) l'algoritmo termina.
→ Il numero di iterazioni è pari a $|V| - 1$. Infatti ogni iterazione aggiunge (uno alla volta) gli archi di MST e il numero di archi di un albero è pari a $|V| - 1$.
Si definisce taglio una una partizione di V in due insiemi V' e $V - V'$.
Dato un arco (u, v), esso attraversa il taglio $(V', V - V')$ se $u \in V'$ e $v \in V - V'$.



V' = {1,2,3,5,6}
V-V' = {4,7,8,9}

archi che attraversano il taglio

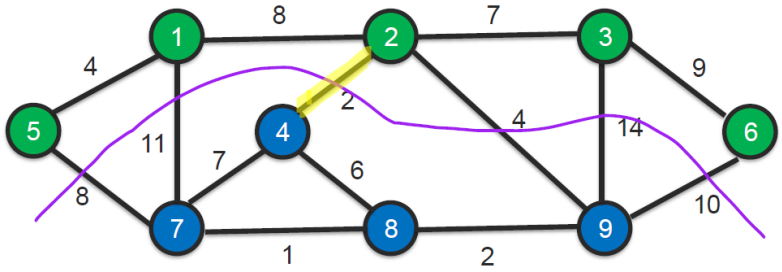
Un taglio $(V', V - V')$ rispetta un insieme A di archi se nessun arco di A attraversa $(V', V - V')$.



V' = {1,2,3,5,6}
V-V' = {4,7,8,9}

rispetta A = {(1,2), (2,3), (7,8), (4,8)}

Un arco (u, v) è leggero rispetto al taglio $(V', V - V')$ se (u, v) è l'arco di peso minimo che attraversa $(V', V - V')$.



V' = {1,2,3,5,6}
V-V' = {4,7,8,9}

(4,2) → arco leggero

15) Enunciare e dimostrare il teorema dell'arco sicuro

Dati:

→ $G = (V, E)$, grafo non orientato, connesso e pesato;

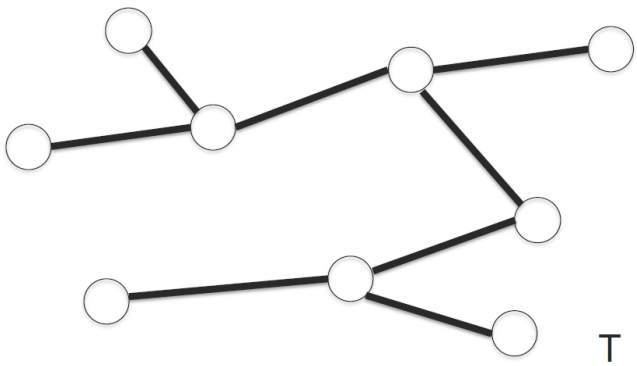
→ $A \subseteq T$, sottoinsieme degli archi di MST;

→ un taglio $(V', V - V')$ che rispetti A

allora un arco (u, v) è sicuro per A, se (u, v) è leggero rispetto a $(V', V - V')$.

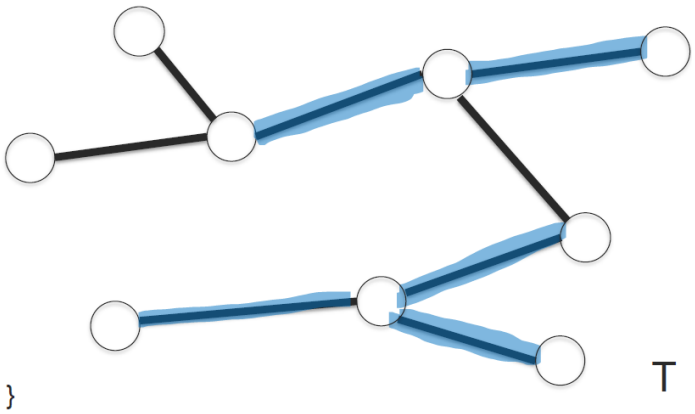
Dimostrazione:

Sia T l'insieme degli archi di un MST



T

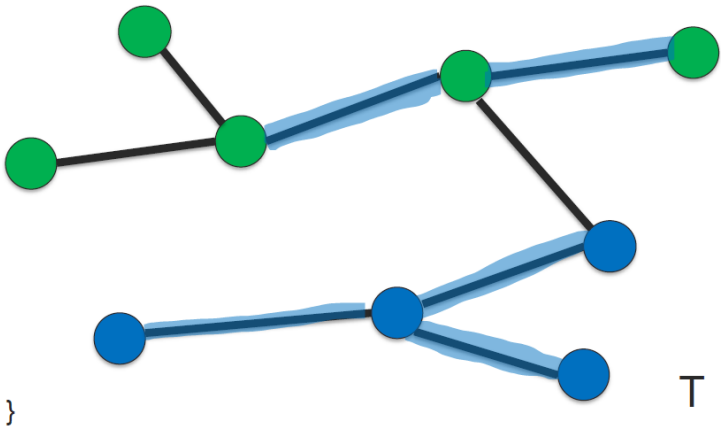
Sia A un suo sottoinsieme.



A = { }

Sia $(V', V - V')$ un taglio che rispetti A.

Taglio (●, ●) che rispetta A

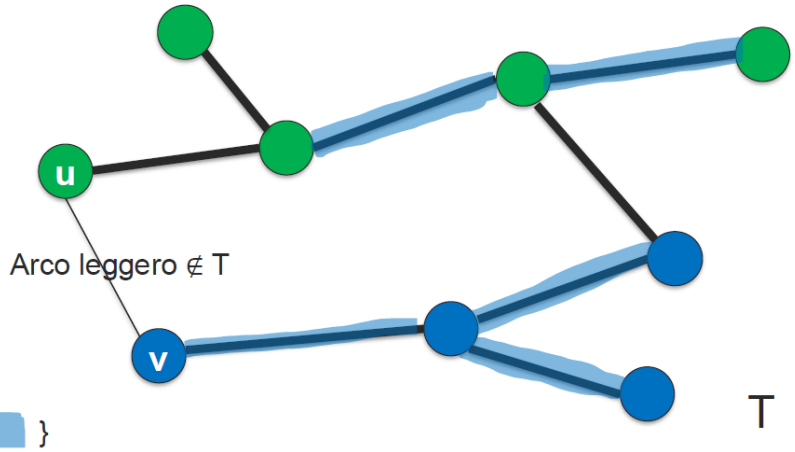


A = { }

Sia (u, v) un arco leggero per il taglio. Se T contiene (u, v) abbiamo dimostrato il teorema (dimostrazione terminata).

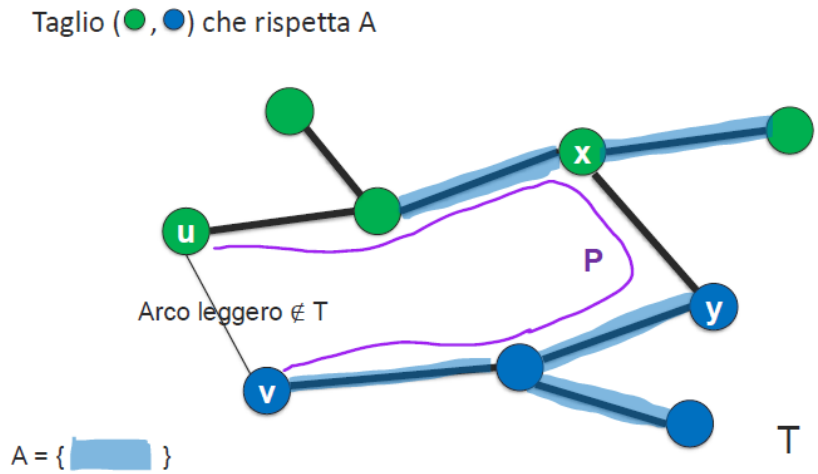
Altrimenti, si supponga che (u, v) non appartenga a T.

Taglio (●, ●) che rispetta A

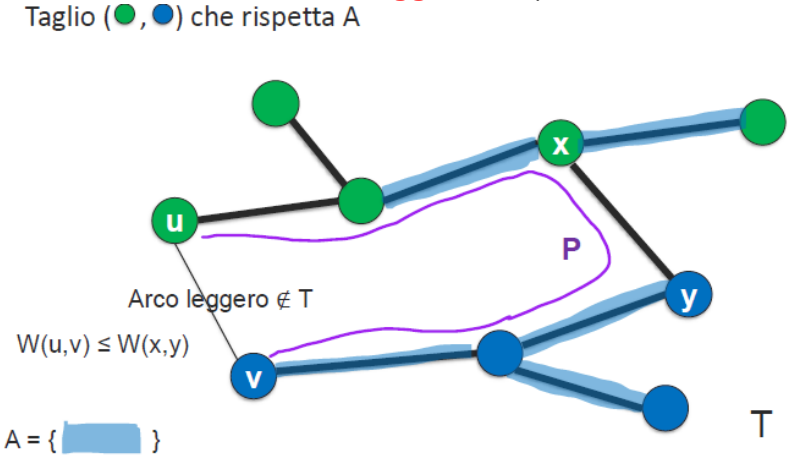


A = { }

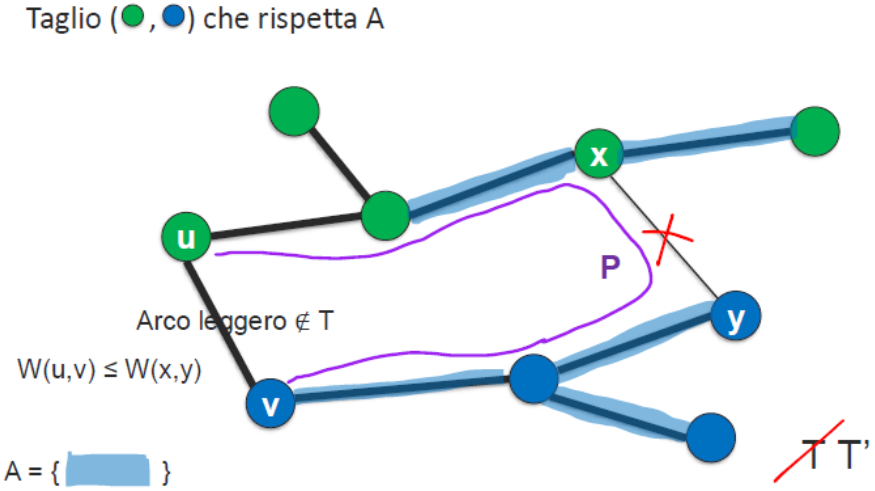
Dal momento che (u, v) è un arco che attraversa il taglio e che u e v sono connessi da un unico cammino P aciclico composto di soli archi di T (per definizione di MST), dovrà esistere un arco (x, y), del cammino P, che attraversa il taglio e che appartiene a T.



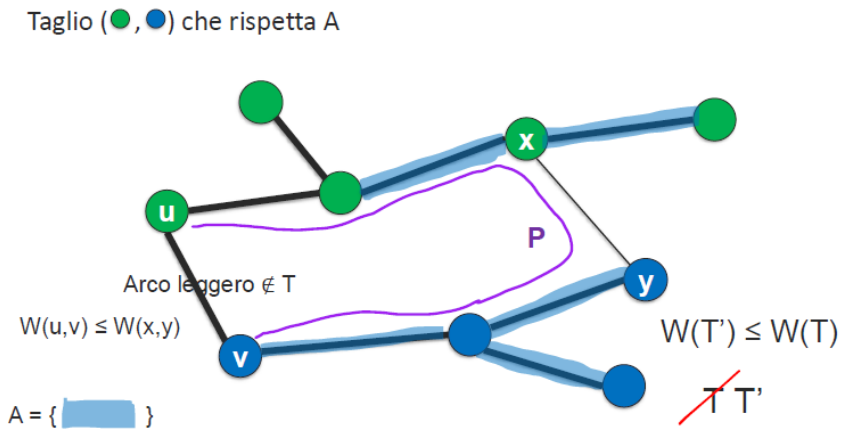
Per definizione di arco leggero, il peso di (u, v) è sicuramente ≤ al peso di (x, y).



Si consideri l'insieme T' che si ottiene sostituendo (x, y) con (u, v). Il sottografo (V, T') è sicuramente uno Spanning Tree (cioè connette tutti i vertici del grafo $G = (V, E)$).



Inoltre il peso totale $W(T')$ è dato da $W(T) - W(x, y) + W(u, v)$ che sarà ≤ $W(T)$ dal momento che $W(u, v) \leq W(x, y)$. Quindi T' produce uno Spanning Tree di costo minimo, cioè un MST.



Infine, per ipotesi l'insieme A $(\subseteq T)$ non contiene nè (u, v) nè (x, y) e, per costruzione, T' contiene tutti gli archi di T tranne (x, y). Si conclude che A risulta essere contenuto in T' e $\{(u, v)\} \cup A$ è contenuto T'. Quindi (u, v) è un arco sicuro.

16) Definire il corollario del teorema dell'arco sicuro

Dati:

→ $G = (V, E)$, grafo non orientato, connesso e pesato;

→ $A \subseteq T$, sottoinsieme degli archi di MST

→ $C = (V_C, A_C)$, componente connessa di $G_A = (V, A)$

allora un arco (u, v) è sicuro per A, se (u, v) è leggero rispetto al taglio $(V_C, V - V_C)$.

17) Cosa si intende per matroide grafico?

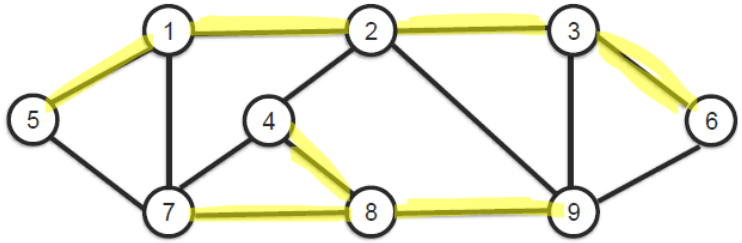
Dato $G = (V, E)$, grafo non orientato, connesso e pesato con una funzione $W : E \to R^+$, la coppia (E, F) con:

→ E, insieme degli archi di G

→ $F = \{A \subseteq E \mid A \text{ è } aciclico\}$

è il matroide grafico.

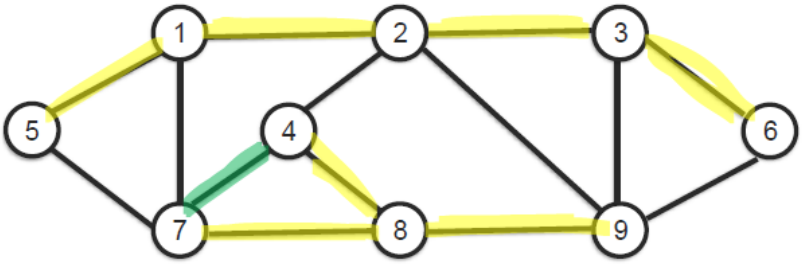
Esempio:



A = {(1,2), (2,3), (4,8), (3,6), (1,5), (7,8), (8,9)} ⊆ E

A è aciclico ⇒ A ∈ F

Controesempio:



A = {(1,2), (2,3), (4,8), (3,6), (1,5), (7,8), (8,9), (4,7)} ⊆ E

A **non** è aciclico ⇒ A ∉ F

In modo più specifico, il matroide grafico è:

→ un sistema di indipendenza (E, F) in cui $\forall A \in F, \forall B \subseteq A \rightarrow B \in F$ e se A appartiene a F, allora A è aciclico. Qualsiasi suo sottoinsieme B sarà aciclico e quindi appartiene a F;

→ nel sistema di indipendenza vale la proprietà dello scambio

$\forall A, B \in F \wedge |B| > |A| \wedge \exists b \in B \setminus A : A \cup \{b\} \in F$.

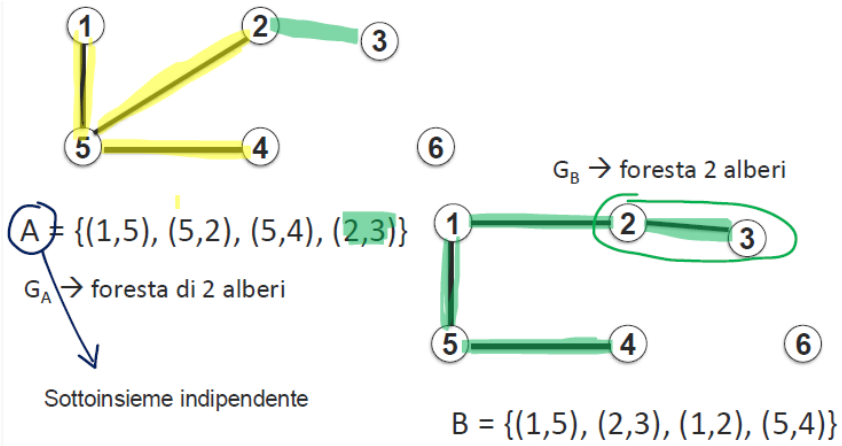
18) Enunciare il teorema di MST

Siano $A, B \in F$ tali che $|B| = |A| + 1$ in cui:

$G_A = (V, A) \rightarrow$ foresta di $|V| - |A|$ alberi;

$G_B = (V, B) \rightarrow$ foresta di $|V| - |B|$ alberi;

G_B ha un albero in meno di G_A. Esisterà quindi un arco $(u, v) \in B \setminus A$ che collega due vertici che in G_A situati in due alberi diversi. Quindi l'aggiunta di (u, v) ad A non induce un ciclo: $\{(u, v)\} \cup A \in F$.



Il sottoinsieme massimale di (E, F) è l'insieme degli archi di un albero di copertura (Spanning Tree), mentre il sottoinsieme di peso minimo è l'insieme degli archi di un albero di copertura minimo (Minimum Spanning Tree, MST).

Capitolo 5 – MST – Algoritmi di Kruskal e Prim

19) Illustrare gli algoritmi di Kruskal e Prim facendo riferimento alla procedura GENERIC-MST. Determinare inoltre le differenze tra i due rispetto all'insieme di archi che l'algoritmo va costruendo e a cosa è un arco sicuro. Determinare inoltre la complessità computazionale dei due algoritmi.

L'idea chiave dell'algoritmo di Kruskal è di selezionare gli archi più leggeri che non creano cicli, garantendo così che la MST risultante sia di peso minimo attraverso un approccio greedy. Utilizza strutture dati per insiemi disgiunti e lo pseudocodice relativo all'algoritmo di Kruskal è il seguente

```
KRUSKAL-MST(V,E,W)
        A ← ∅
        Ordino E per peso W crescente    // O(|E| log |E|)
        foreach v ∈ V do                 // O(V) operazioni MAKE_SET
                MAKE_SET(v)

        for i from 1 to |E| do           // O(E) operazioni FIND_SET e UNION   --> O((V + E)α)
                (u,v) ← ei
                if FIND_SET(u) ≠ FIND_SET(v) then
                        A ← A U {(u,v)}
                        UNION(u,v)
        return (V,A)


Tempo computazionale
O(|E| log |E| + (|V| + |E|)α)
O(|E| log |E| + (|E| + |E|)α) --> dato che con G connesso ⇒ |E| ≥ |V| - 1
O(|E| log |E| + 2|E|α)
O(|E| log |E| + |E|α)         --> α ≤ log |V|
O(|E| log |E| + E log |E|)
O(2 |E| log |E|)
O(|E| log |E|)
```

Riassumendo esso ha le seguenti caratteristiche:

→ inizia dal vertice dal peso minore;

→ attraversa un nodo una sola volta;

→ può gestire grafi non connessi;

→ più efficiente per grafi sparsi.

L'idea chiave dell'algoritmo di Prim è quella di utilizzare una coda di priorità (o heap) per mantenere traccia degli archi e dei loro pesi. Ogni vertice u ha due attributi:

→ u.key: minimo tra i valori dei pesi degli archi (x,u) tale che $x \in C$;

→ u.π: vertice che ha fornito il peso u.key

$\nexists (x,u) \in E: x \in C \Rightarrow u.key = +\infty, u.\pi = NIL$

Lo pseudocodice è il seguente:

```
PRIM-MST(V, E, W, r)
        foreach v ∈ V do                                        //O(|V|)
                v.key ← +∞
                v.π ← NIL
        r.key ← 0

        foreach v ∈ V do                                        //O(|V|)
                INSERT(Q, v)

        while Q ≠ ∅ do                                          //O(|V|) estrazioni da Q
                u ← EXTRACT-MIN(Q)                              //O(log |V|)
                foreach v ∈ adj(u) do                           //O(|E|)
                        if v ∈ Q and W(u,v) < v.key then        //O(1)
                                v.key ← W(u, v)
                                v.π ← u
                                DECREASE-KEY(Q, v, v.key)       //O(log|V|)
Complessità computazionale
O(|V|) + O(|V| log |V|) + O(|E| log |V|)
O(|V| log |V|) + O(|E| log |V|)
O(|E| log |E|) + O(E log |E|)      --> |E| >= |V| - 1
O(2 |E| log |E|)
O(|E| log |E|)
```

Riassumendo esso ha le seguenti caratteristiche:

→ inizia da qualsiasi vertice del grafo;

→ attraversa un nodo più volte;

→ gestisce grafi connessi;

→ più efficiente per grafi densi.

Capitolo 6 – Algoritmo di Dijkstra

20) Definire in modo formale l'input e l'output dell'algoritmo di Dijkstra

Input: Dati

→ un grafo $G = (V, E)$ orientato e pesato;

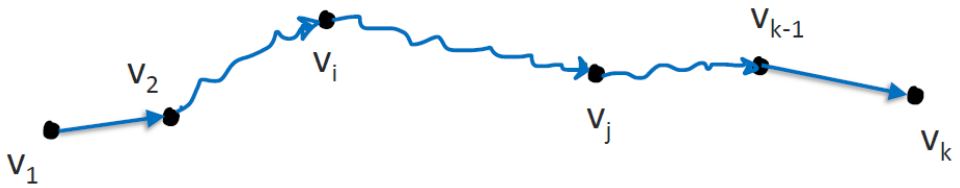
→ una funzione $W : E \rightarrow R^+$, tale che $W(u, v)$ sia il peso dell'arco (u, v);

→ $s \in V$ (sorgente).

Output: si ha l'obiettivo di ricavare $\forall v \in V$, un cammino di peso minimo dal vertice s al vertice v indicato con $\delta(s, v)$.

21) Definire la sottostruttura ottima dell'algoritmo di Dijkstra

Se $P = \langle v_1, v_2, \ldots, v_{k-1}, v_k \rangle$ è il cammino minimo da v_1 a v_k, allora $\forall$ sottocammino $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, tale che $1 \leq i \leq j \leq k$, è il cammino minimo da v_i a v_j.



Se $\langle v_1, v_2, \ldots, v_{k-1} \rangle$ è il cammino minimo da v_1 a v_{k-1}.

Si ha quindi $\delta(v_1, v_k) = \delta(v_1, v_{k-1}) + W(v_{k-1}, v_k)$

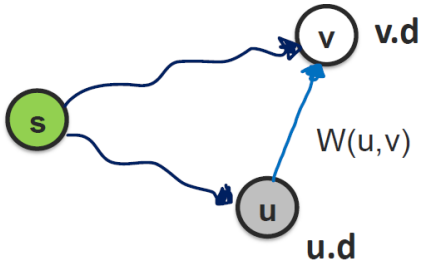
22) Illustrare la tecnica del rilassamento e come viene utilizzata all'interno dell'algoritmo di Dijkstra

Sia $G = (V, E)$, ogni vertice $v \in V$ possiede due attributi:

→ $v.d$ = limite superiore per $\delta(s, v)$ ($\delta(s, v) \leq v.d$)

→ $v.\pi$ = predecessore di v nel cammino di peso $v.d$

La tecnica del rilassamento permette di fare convergere $v.d$ al valore $\delta(s, v)$, per ogni vertice v del grafo.



Si avrebbe quindi:

```
RELAX(u, v, W)
        if v.d > u.d + W(u,v) then
                v.d := u.d + W(u,v)
                v.π := u
```

Per quanto riguarda l'inizializzazione si avrebbe:

```
INITIALIZE-SINGLE-SOURCE(V, E, s)
        foreach v ∈ V do
                v.d := +∞
                v.π := NIL
        s.d := 0
```

Lo schema generale della tecnica di rilassamento dell'algoritmo di Dijkstra è il seguente:

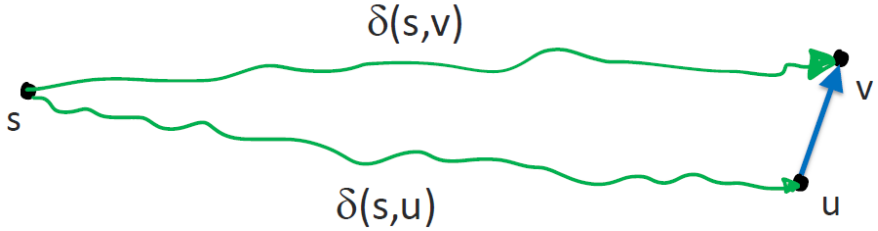
→ `INITIALIZE-SINGLE-SOURCE(G=(V,E), s)`

→ Sequenza di chiamate alla procedura `RELAX(u, v, W)`

Ogni arco è sottoposto al rilassamento una sola volta.

Vale inoltre la proprietà della disuguaglianza triangolare, ossia:

$\forall (u, v) \in E \rightarrow \delta(s, v) \leq \delta(s, u) + W(u, v)$



Risulta, inoltre, importante considerare che $v.d$ non cambia più non appena raggiunge il valore $\delta(s, v)$.

Per induzione si ha:

a) Caso base: Inizializzazione

$s.d = 0 = \delta(s,s)$

$v.d = +\infty > \delta(s,v)$, se $\exists$ un cammino da s a $v \neq s$

$v.d = +\infty = \delta(s,v)$, se $\nexists$ un cammino da s a $v \neq s$

b) Passo induttivo

Ipotesi: $v.d \geq \delta(s,v)\ \forall v \in V$

Tesi: $v.d \geq \delta(s,v)\ \forall v \in V$, dopo $RELAX(x,y,W)$

Risulta sufficiente dimostrare che:

E' sufficiente dimostrare che $y.d \geq \delta(s,y)$

$y.d = x.d + W(x,y) \geq \delta(s,x) + W(x,y) \geq \delta(s,y)$

```
if y.d > x.d + W(x,y) then
    y.d ← x.d + W(x,y)
    y.π ← x
```

In caso di assenza del cammino da s a v si ha sempre $v.d = \delta(s,v)$ e per la proprietà del limite superiore si ha $v.d \geq \delta(s,v) = +\infty$ dopo ogni rilassamento.

Dopo la chiamata `RELAX(u, v, W)` si avrà:

$v.d \leq u.d + W(u,v)$

23) Descrivere in generale l'algoritmo di Dijkstra

Dati due insiemi:

→ insieme S dei vertici v per cui $v.d = \delta(s,v)$;

→ insieme $V - S$ dei vertici v per cui $v.d \geq \delta(s,v)$.

Viene estratto da $V - S$ il vertice u con il minimo valore dell'attributo d e viene aggiunto a S. Si ha quindi:

→ $u.d = \delta(s,u)$

→ $u.\pi$ è predecessore di u in un cammino minimo da s a u

Per ogni $v \in adj(u)$, esegui $RELAX(u,v,W)$

$V - S$ è vuoto $\Rightarrow v.d = \delta(s,v)\ \forall v \in V$.

Viene utilizzata una coda di priorità Q in cui:

→ vengono contenuti i vertici di $V - S$;

→ viene estratto da Q il vertice con il minimo valore dell'attributo d cammino minimo da s a v;

→ all'inizio, Q contiene tutti i vertici del grafo;

→ la sorgente s è il primo vertice estratto;

→ l'esecuzione termina quando Q è vuota.

```
DIJKSTRA(V,E,W,s)
    S ← ∅
    Q ← ∅
    INITIALIZE-SINGLE-SOURCE(G=(V,E),s)
    foreach v ∈ V do
        INSERT(Q, v)

    while Q ≠ ∅ do
        u ← EXTRACT-MIN(Q)
        S ← S U {u}
        foreach v ∈ adj(u) do
            RELAX(u,v,W)
            if v.d è diminuito then
              DECREASE-KEY(Q, v, v.d)
```

24) Fornire la procedura di ricostruzione dell'algoritmo di Dijkstra

```
Procedura ricostruisci_cammino(v)
   if v.π ≠ NIL or v = s then
      if v.π ≠ NIL then
         ricostruisci_cammino(v.π)
      print(v)
```

25) Enunciare il teorema e dimostrare la correttezza dell'algoritmo di Dijkstra

Teorema: Al termine dell'esecuzione si ha $v.d = \delta(s,v)$ per ogni vertice v del grafo. Per induzione: Caso base

$v.d = \delta(s,v)$ se $v = s$

Per induzione: Passo induttivo

Ipotesi: $v'.d = \delta(s,v') \; \forall v' \in S$

Tesi: $v.d = \delta(s,v) \; con \; v = ExtractMax(Q)$

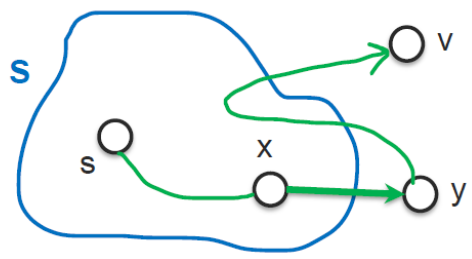
Dimostrazione

Sia S, l'insieme dei vertici già estratti, e v, il vertice estratto da Q.

Vale che $v.d \leq u.d \; \forall u \in V - S \Rightarrow v.d \leq y.d$

Si suppone per assurdo che v sia il primo vertice estratto da Q, tale che $v.d \neq \delta(s,v)$. Deve esistere, allora deve esistere qualche cammino da s a v (altrimenti $v.d = \delta(s,v) = +\infty$).

Data la seguente figura:



Sia quindi:
P → cammino minimo da s a v;
y → primo vertice di P non appartenente a S;
x → predecessore di y in P (appartiene a S)
v è il primo vertice estratto da Q tale che $v.d \neq \delta(s,v) \Rightarrow x.d = \delta(s,x)$ quando è stato aggiunto a S. Vale quindi che (x,y) è stato rilassato nell'istante in cui x è stato aggiunto a $S \Rightarrow y.d = \delta(s,y)$. Il cammino da s a y è sottocammino di $P \Rightarrow y.d = \delta(s,y) \leq \delta(s,v) \leq v.d$
Dal limite superiore $\delta(s,v) \leq v.d$ → $\delta(s,v) = v.d$

Capitolo 7 – Introduzione a NP completezza e riduzioni

26) Differenza tra problemi trattabili o facili e problemi intrattabili o difficili

I problemi vengono definiti trattabili o facili se tali problemi sono risolvibili con algoritmi in tempo polinomiale, mentre problemi vengono definiti intrattabili o difficili se tali problemi non sono risolvibili con algoritmi in tempo polinomiale e richiedono un tempo super – polinomiale.

27) Fornire degli esempi di differenze tra problemi trattabili o facili e problemi intrattabili o difficili

Vi sono alcuni esempi:
→ Cammini minimi e cammini massimi: con archi di peso sia positivo che negativo risulta possibile trovare i cammini minimi da una sorgente unica in un grafo orientato $G = (V,E)$ nel tempo $O(|V| * |E|)$. Risulta invece difficile trovare il cammino semplice massimo tra due vertici, così anche come solo determinare se un grafo contiene un cammino semplice con almeno un dato numero di archi è un problema NP completo.
→ Ciclo euleriano e ciclo hamiltoniano: un ciclo euleriano di un grafo orientato connesso $G = (V,E)$ è un ciclo che attraversa ciascun arco di G una sola volta, sebbene possa visitare un vertice più di una volta. Risulta possibile determinare se un grafo ha un ciclo euleriano in un tempo di appena $O(|E|)$. Un ciclo hamiltoniano di un grafo orientato $G = (V,E)$ è un ciclo semplice che contiene tutti e i soli vertici di V una sola volta. Determinare se un grafo orientato ha un ciclo hamiltoniano è un problema NP-completo.
→ Soddisfacibilità di formule 2-CNF e soddisfacibilità di formule 3-CNF: una formula booleana contiene variabili i cui valori sono 0 o 1, connettivi booleani come $\wedge$ (AND), $\vee$ (OR) e $\neg$ (NOT), e parentesi. Una formula booleana è soddisfacibile se esiste un'assegnazione di valori 0 e 1 per le sue variabili che la

rende pari a 1. Formalmente, una formula booleana è nella forma normale k-congiuntiva o k-CNF, se è espressa come l'AND di clausole OR di k variabili o delle loro negazioni.
Esempio: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$
Esiste un algoritmo con tempo polinomiale per determinare se una formula 2-CNF è soddisfacibile, mentre determinare se una formula 3 – CNF è soddisfacibile è un problema NP-completo.

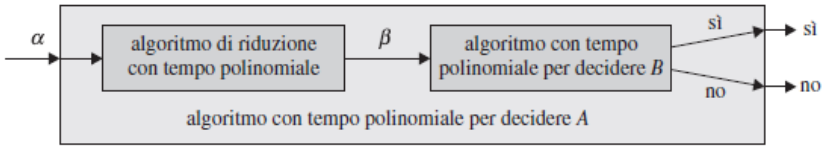
28) Definire la differenza tra classi NP, P e NP – completo

La classe P è costituita da problemi che sono risolvibili in tempo polinomiale, ossia problemi che possono essere risolti nel tempo $O(n^k)$ per qualche costante k, dove n è la dimensione dell'input del problema.
La classe NP è costituita da problemi che sono "verificabili" in tempo polinomiale. Ciò significa che con un certificato di una soluzione, risulta possibile verificare che il certificato è corretto in tempo polinomiale nella dimensione dell'input del problema. Un problema qualsiasi in classe P è anche in classe NP, perché se un problema è in P allora è possibile risolverlo in tempo polinomiale senza avere un certificato.
La classe NP – completo è costituito da tutti quei problemi in classe NP e che sono "difficili" da risolvere.

29) Quali sono le tecniche utilizzate per identificare i problemi NP completi?

L'identificazione di un problema NP completo si basa su tre concetti fondamentali:
→ Problemi di decisione: in cui la risposta è semplicemente 0 o 1.
→ Problemi di ottimizzazione: in cui ogni soluzione ammissibile ha un valore associato, e si vuole trovare la soluzione ammissibile con il valore migliore.
→ Riduzione: si considera un problema di decisione, per esempio A, che si vuole risolvere in tempo polinomiale e si definisce istanza del problema l'input di un particolare problema. Si suppone ora che ci sia un altro problema di decisione B risolvibile in tempo polinomiale. Si definisce algoritmo di riduzione una procedura che trasforma un'istanza α di A in qualche istanza β di B con le seguenti caratteristiche:
→ la trasformazione richiede un tempo polinomiale;
→ le soluzioni sono le stesse, ossia la soluzione per α è 1 se e soltanto se anche la soluzione per β è 1.
Formalizzando, i passaggi dell'algoritmo di riduzione sono:

→ data un'istanza α del problema A, utilizziamo un algoritmo di riduzione con tempo polinomiale per trasformarla in un'istanza β del problema B;
→ si esegue sull'istanza β l'algoritmo di decisione con tempo polinomiale per il problema B;
→ si utilizza la soluzione per β come soluzione per α.



Riassumendo, la riduzione ha l'obiettivo di risolvere il problema A utilizzando la facilità di B.

30) Definire il concetto di algoritmo di verifica

Si definisce algoritmo di verifica un algoritmo A con due argomenti, dove un argomento è una normale stringa di input x e l'altro argomento è una stringa binaria y detta certificato. Si dice che un algoritmo A con due argomenti verifica una stringa di input x se esiste un certificato y tale che $A(x, y) = 1$.
Il linguaggio verificato è:
$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* : A(x, y) = 1\}$

31) Definire la classe di complessità NP

La classe di complessità NP è la classe dei linguaggi che possono essere verificati da un algoritmo polinomiale.
$L = \{x \in \{0,1\}^* : \exists y : |y| = O(|x|^c) : A(x, y) = 1\}$
Diremo che l'algoritmo A verifica il linguaggio L in tempo polinomiale.

32) Definire formalmente i seguenti concetti: riducibilità di un linguaggio L_1 ad un linguaggio L_2 e funzione di riduzione f

Un linguaggio L_1 è riducibile in tempo polinomiale a un linguaggio L_2 ($L_1 \leq_p L_2$), se esiste una funzione calcolabile in tempo polinomiale $f: \{0,1\}^* \rightarrow \{0,1\}^*$ e $\forall x \in \{0,1\}^* \rightarrow x \in L_1 \; sse \; f(x) \in L_2$
La funzione f è detta funzione di riduzione.

33) Definire problema NP – Completo

Un linguaggio $L \subseteq \{0,1\}^*$ è NP – Completo se:
→ $L \in NP$
→ $L_1 \leq_p L \; \forall L_1 \in NP$

34) Definire in modo formale il problema di clique o cricca

Una clique in un grafo $G = (V, E)$ è un sottoinsieme di vertici $V' \subseteq V$ tale che

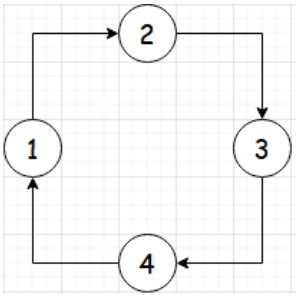
ogni coppia di vertici in V' è connessa da un arco. In altri termini una clique è un sottografo completo di G. La dimensione di una clique è il numero di vertici che contiene. Il problema della clique è un problema di ottimizzazione che consiste nel trovare una cricca di dimensione massima in un grafo. La definizione formale è $CLIQUE = \{(G,k): G \text{ è } un\ grafo\ con\ una\ cricca\ di\ dimensione\ k\}$.

Esempio:

Sia $G = (V,E)$

$V = \{1,2,3,4,5\}$

$E = \{(1,2),(2,3),(3,4),(1,4)\}$



35) Definire in modo formale il problema della copertura dei vertici

Una copertura di vertici di un grafo non orientato $G = (V,E)$ è un sottoinsieme $V' \subseteq V$ tale che, se $(u,v) \in E$, allora $u \in V' \vee v \in V'$, ovvero ciascun vertice "copre" i suoi archi incidenti, e una copertura di vertici per G è un insieme di vertici che copre tutti gli archi in E. La dimensione di una copertura di vertici è il numero di vertici che appartengono alla copertura. In termini di linguaggio formale, si definisce:

$VERTEXCOVER = \{(G,k): il\ grafo\ G\ ha\ una\ copertura\ di\ vertici\ di\ dimensione\ k\}$

36) Definire in modo formale la soddisfacibilità delle formule 3-CNF

Una formula booleana è nella forma normale 3 – congiuntiva o 3-CNF, se ciascuna clausola ha esattamente tre letterali distinti. Per esempio, la formula booleana $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$ è nella forma 3-CNF.

La prima delle sue tre clausole è $(x_1 \vee \neg x_1 \vee \neg x_2)$, che contiene i tre letterali $x_1, \neg x_1$ e $\neg x_2$. Il problema consiste nel determinare se una data formula booleana ρ in 3-CNF è soddisfacibile.

37) Enunciare e dimostrare il teorema sulla NP Completezza di 3 – CNF

Teorema: La soddisfacibilità delle formule booleane nella forma 3-CNF è un problema NP.

Dimostrazione: L'algoritmo di riduzione può essere suddiviso in tre fasi. Ogni fase trasforma progressivamente la formula di input φ avvicinandola sempre di

più alla forma 3 – CNF desiderata.

Fase 1:

Si realizza un "albero binario di parsing" per la formula di input φ, con i letterali come foglie e i connettivi come nodi interni. Si ha ad esempio:

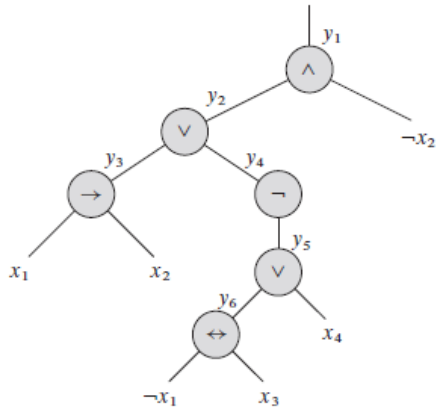
$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

Si introduce una variabile y_i per l'output di ogni nodo interno e si riscrive φ come l'AND tra la variabile della radice e una congiunzione di clausole che descrivono l'operazione di ciascun nodo.

$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ &\wedge (y_4 \leftrightarrow \neg y_5) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

Si ha il seguente albero:

Figura 34.11 L'albero corrispondente alla formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$



Notate che la formula φ' così ottenuta è una congiunzione di clausole φ'_i, ciascuna delle quali ha al più 3 letterali. L'unico requisito aggiuntivo è che ogni clausola deve essere un OR di letterali.

Fase 2:

Si converte ogni clausola φ'_i nella forma normale congiuntiva. Si costruisce quindi una tavola di verità per φ'_i valutando tutte le possibili assegnazioni alle sue variabili. Ogni riga della tavola di verità è formata da una possibile assegnazione delle variabili della clausola, insieme al valore della clausola sotto tale assegnazione. Utilizzando gli elementi della tavola di verità che valgono 0, si costruisce una formula nella forma normale disgiuntiva (o DNF), ossia un OR

di clausole AND ($\neg\varphi'_i$). Si utilizzano le leggi di De Morgan per ottenere il complemento di tutti i letterali e cambiare gli OR in AND e gli AND in OR:

$$\neg(a \wedge b) = \neg a \vee \neg b$$
$$\neg(a \vee b) = \neg a \wedge \neg b$$

Si converte la clausola $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ nella forma CNF seguente:
Si ha la DNF pari a

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

Applicando le leggi di De Morgan, otteniamo la formula CNF

$$\phi''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

equivalente a $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Fase 3:
Si trasforma ulteriormente la formula in modo che ogni clausola abbia esattamente 3 letterali distinti. La formula 3-CNF finale φ''' è ottenuta dalle clausole della formula CNF φ''.
Per ogni clausola C_i di φ'' includiamo le seguenti clausole in φ''':
→ Se C_i ha 3 letterali distinti, allora si include semplicemente come una clausola di φ'''.
→ Se C_i ha 2 letterali distinti, cioè, se $C_i = l_1 \vee l_2$, dove l_1 e l_2 sono letterali, allora includiamo $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ come clausole di φ'''. I letterali p e $\neg p$ servono solamente per soddisfare il requisito sintattico di avere esattamente 3 letterali distinti per clausola.
→ Se C_i ha un unico letterale l, allora si include
$(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ come clausole di φ'''.
Si può osservare che φ''' è soddisfacibile soltanto se φ è soddisfacibile. Risulta necessario anche dimostrare che la riduzione può essere calcolata in tempo polinomiale. La costruzione di φ' da φ introduce al più una variabile e una clausola per ogni connettivo in φ. La costruzione di φ'' a φ può introdurre al più 8 clausole in φ'' per ogni clausola di φ' dato che ogni clausola di φ' ha al più 3 variabili e la tavola di verità per ogni clausola ha al più 8 righe. La costruzione

di φ''' a φ'' introduce al più 4 clausole in φ''' per ogni clausola di φ''. Quindi, la dimensione della formula risultante φ''' e polinomiale nella lunghezza della formula originale. Ciascuna delle costruzioni può essere facilmente realizzata in tempo polinomiale.

38) Enunciare e dimostrare il teorema sulla NP Completezza di clique

Teorema: Il problema della clique è NP-completo.

Dimostrazione:

Si utilizza un sottoinsieme di vertici $V' \subseteq V$ dei vertici nella clique come certificato per G. La verifica che V' è una clique può essere effettuata in tempo polinomiale controllando che $\forall u, v \in V'$: $(u, v) \in E$.

Si procede dimostrando $\text{3-CNF-SAT} \leq_P \text{CLIQUE}$ che prova che il problema della clique è difficile. L'algoritmo di riduzione inizia con un'istanza di 3-CNF-SAT.

Sia $\varphi = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_k$ una formula booleana in 3-CNF con k clausole, dove ogni clausola C_i con $1 \leq i \leq k$ ha esattamente tre letterali distinti l_1^r, l_2^r, l_3^r.

Si realizza un grafo $G = (V, E)$ in cui φ è SAT solo se G ha una clique di dimensione k. Si ha $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in φ e si inserisce una tripla di vertici v_1^r, v_2^r, v_3^r. Si pone un arco tra due vertici v_i^r e v_j^s se valgono le seguenti proprietà:

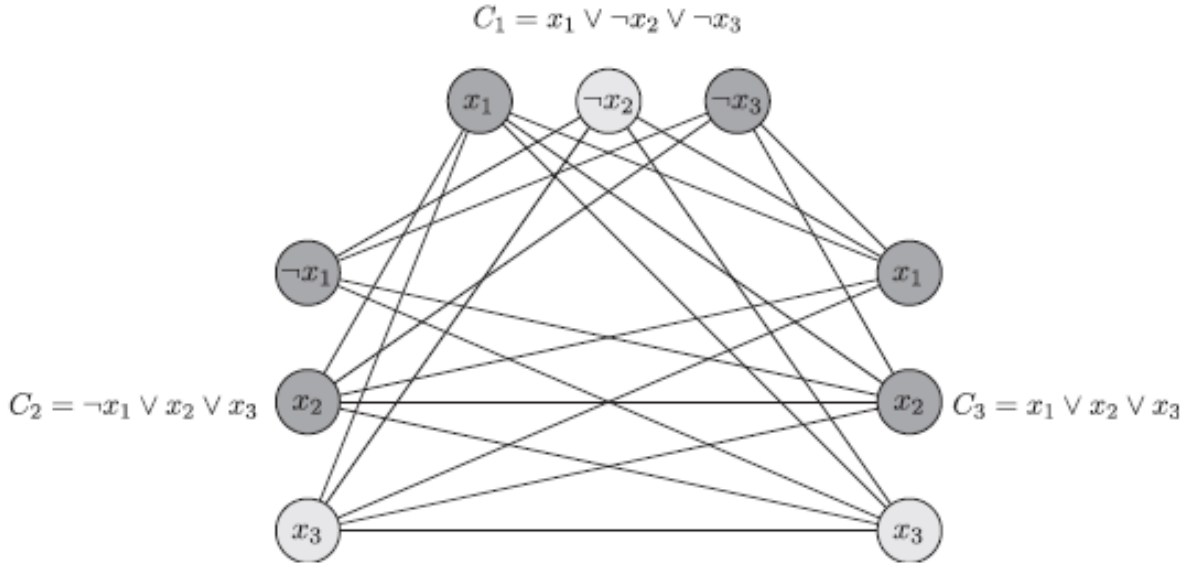
→ $r \neq s$, ovvero si trovano in triple differenti;

→ vi è una coerenza tra i letterali, ossia $l_i^r \neq \neg(l_j^s)$.

Il $G = (V, E)$ può essere facilmente calcolato in tempo polinomiale.

Ad esempio si ha

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



Per dimostrare che la trasformazione di φ è un grafo, allora si suppone che φ abbia un'assegnazione che la soddisfa. Ogni clausola C_r, allora contiene almeno

un letterale l_i^r a cui è assegnato il valore logico 1 e ciascuno di tali letterali corrisponde a un vertice v_i^r. Prendendo un letterale con valore logico 1 si ottiene l'insieme V' tale che $|V'| = k$. Se V' è una clique, allora per una coppia qualsiasi di vertici $v_i^r, v_i^s \in V'$, dove $r \neq s$, entrambi i letterali l_i^r e l_j^s sono associati a 1 dall'assegnazione di soddisfattibilità e quindi deve valere $l_i^r \neq \neg(l_j^s)$. Risulta che $(v_i^r, v_j^s) \in E$. Al contrario si suppone che G abbia una clique tale che $|V'| = k$. Si ha che V' contiene esattamente un vertice per tripla. Risulta possibile assegnare ad ogni letterale l_i^r il valore 1 in modo tale che $v_i^r \in V'$. Dato che G non contiene archi tra letterali che non sono coerenti, se ogni clausola C_r è soddisfatta, allora φ è soddisfatta.

39) Enunciare e dimostrare il teorema sulla NP Completezza di copertura dei vertici

Teorema: Il problema della copertura dei vertici è NP-completo.

Dimostrazione:

Si suppone di avere un grafo $G = (V, E)$ e $k \in \mathbb{Z}$. Si sceglie come certificato, la copertura dei vertici $V' \subseteq V$. L'algoritmo di verifica accerta che $|V'| = k$ e poi controlla, per ogni arco $(u, v) \in E$, tale che $u \in V'$ e $v \in V'$. Questa verifica può essere effettuata in tempo polinomiale. Per dimostrare che il problema di copertura dei vertici è NP-completo, risulta possibile applicare la riduzione $\text{CLIQUE} \leq_p \text{VERTEX-COVER}$. La riduzione si basa sul concetto di "complemento" di un grafo, ossia $\neg G = (V, \neg E)$, dove $\neg E = \{(u, v) \notin E\}$. L'algoritmo di riduzione riceve come input un'istanza $\langle G, k \rangle$ del problema della clique. Calcola il complemento $\neg G$ in tempo polinomiale e l'output dell'algoritmo di riduzione è $\langle \overline{G}, |V| - k \rangle$. Per completare la dimostrazione si prova che la trasformazione è una riduzione: il grafo G ha una cricca di dimensione k se e soltanto se il grafo $\neg G$ ha una copertura di vertici di dimensione $|V| - k$.

Si suppone che G abbia una cricca $V' \subseteq V \text{ con } |V'| = k$.

Si asserisce che $V - V'$ sia una copertura di vertici in $\neg G$. Allora $(u, v) \notin E$, e questo implica che almeno uno dei vertici u e v non appartiene a V', perché ogni coppia di vertici di V' è collegata da un arco E. Equivalentemente u e v non appartiene a $V - V'$. Ogni arco di $\neg E$ è coperto da un vertice in $V - V'$. L'insieme $V - V'$, che ha dimensione $|V| - k$, forma una copertura di vertici per $\neg G$. $V - V'$ è una clique di dimensione k.

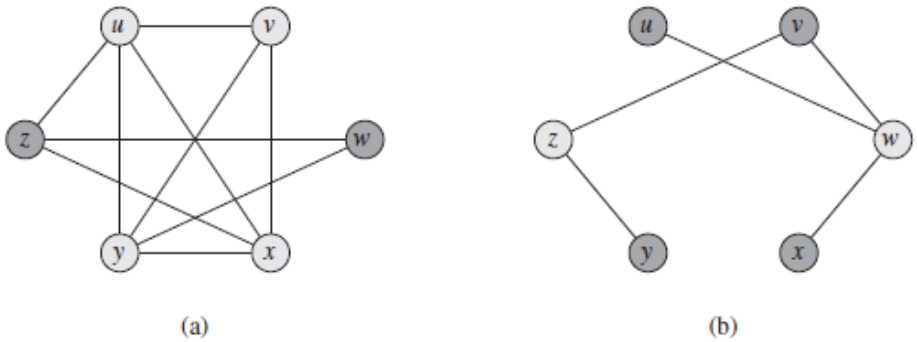


Figura 34.15 Riduzione di CLIQUE a VERTEX-COVER. (a) Un grafo non orientato $G = (V, E)$ con la cricca $V' = \{u, v, x, y\}$. (b) Il grafo $\overline{G}$ prodotto dall'algoritmo di riduzione che ha la copertura di vertici $V - V' = \{w, z\}$.

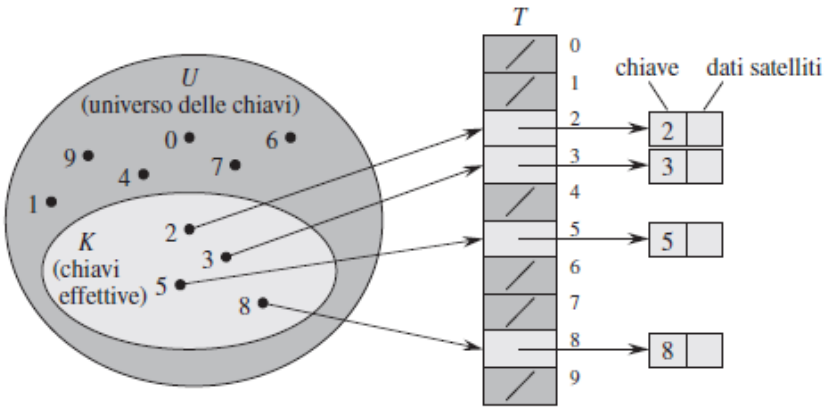
40) Definire il concetto di Independent Set

Si definisce Independent Set di dimensione k se vale la seguente proprietà: $G = (V, E)$ e sia $k \in \mathbb{Z}$. Sia $V' \subseteq V$ tale che $(u, v) \notin E$ tale che $\forall u, v \in V'$.

Capitolo 8 – Tabelle Hash

41) Descrivere la metodologia di indirizzamento aperto

L'indirizzamento diretto è una tecnica in cui si sfrutta la possibilità di esaminare una posizione arbitraria in un array nel tempo $O(1)$. Tale tecnica può essere utilizzata in modo vantaggioso quando risulta possibile allocare un array che ha una posizione per ogni chiave possibile. Si dispone dell'insieme universo di chiavi $U = \{0, 1, \ldots, m - 1\}$, dove m non è troppo grande. Per rappresentare l'insieme dinamico, si utilizza un array, indicato con $T[0, \ldots, m - 1]$, dove ogni posizione o cella corrisponde ad una chiave nell'universo U.



42) Quali sono i metodi utilizzati per l'implementazione della tecnica di indirizzamento aperto?

Si hanno i seguenti metodi a disposizione:

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] = NIL
```

Ciascun metodo a disposizione richiede tempo $O(1)$.

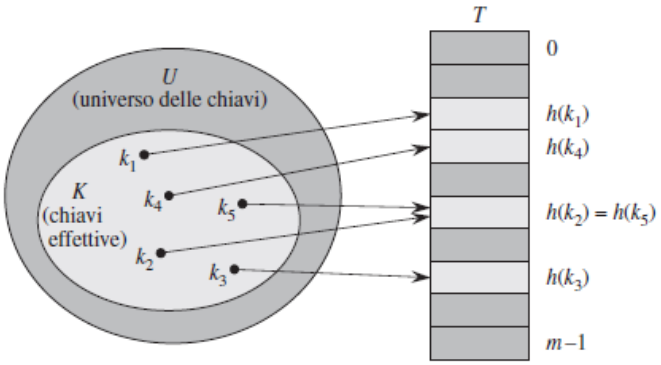
43) Quali sono gli svantaggi della tecnica di indirizzamento? Come risolvere tale problematica?

Lo svantaggio di tale tecnica è che se l'universo delle chiavi U è troppo grande, memorizzare una tavola T di dimensione $|U|$ può risultare piuttosto impraticabile, considerando la memoria disponibile in un calcolatore. Si avrebbe uno spreco di memoria se l'insieme delle chiavi memorizzate K fosse così ridotto, ossia $|K| << |U|$. Quando l'insieme K delle chiavi memorizzate in un dizionario è molto più piccolo dell'universo U di tutte le chiavi possibili, una tavola hash richiede molto meno spazio di una tavola a indirizzamento diretto. Lo spazio richiesto può essere ridotto a $\Theta(|K|)$, senza perdere il vantaggio di ricercare un elemento nella tavola hash nel tempo $O(1)$. Dato che con l'indirizzamento diretto, un elemento x con chiave k è memorizzato nella cella k, attraverso la funzione hash così definita:

$h : U \rightarrow \{0, 1, \ldots, m-1\}$

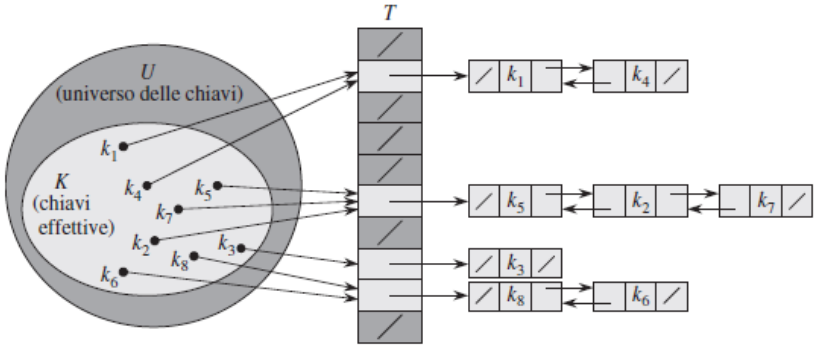
x verrà memorizzato nella cella $h(k)$.

La dimensione m della tavola hash è generalmente molto più piccola di $|U|$.



44) Descrivere in modo dettagliato il problema di collisione

La collisione è un evento che si verifica quando due chiavi vengono mappate nella stessa cella. Tale evento può essere risolto attraverso la tecnica del concatenamento, in cui si pongono tutti gli elementi che sono associati alla stessa cella in una lista concatenata.



I metodi che implementano tale risoluzione sono:

CHAINED-HASH-INSERT(T, x)
inserisce x in testa alla lista $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)
ricerca un elemento con chiave k nella lista $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
cancella x dalla lista $T[h(x.key)]$

Il tempo di esecuzione nel caso peggiore per l'inserimento è $O(1)$. Il tempo di esecuzione per la ricerca nel caso peggiore è proporzionale alla lunghezza della lista. La cancellazione di un elemento x può essere realizzata in $O(1)$.

45) Definire il coefficiente del fattore di carico

Data una tavola hash T con m celle dove sono memorizzati n elementi, si definisce il coefficiente del fattore di carico α, tale che

$$\alpha = \frac{n}{m}$$

46) Enunciare e dimostrare il teorema di hashing uniforme semplice riguardo alla ricerca senza successo

Teorema:

In una tavola hash le cui collisioni sono risolte con il concatenamento, una ricerca senza successo richiede un tempo $\theta(1 + \alpha)$ nel caso medio, nell'ipotesi di hashing uniforme semplice.

Dimostrazione:

Nell'ipotesi di hashing uniforme semplice, qualsiasi chiave k non ancora memorizzata nella tavola ha la stessa probabilità di essere associata a uno qualsiasi delle m celle. Il tempo atteso per ricercare senza successo una chiave k è il tempo atteso per svolgere le ricerche fino alla fine della lista $T[h(k)]$, che ha una lunghezza attesa pari a $\mathrm{E}[n_{h(k)}] = \alpha$. Quindi, il numero atteso di elementi esaminato in una ricerca senza successo è α e il tempo totale $\theta(1 + \alpha)$.

47) Enunciare e dimostrare il teorema di hashing uniforme semplice riguardo alla ricerca con successo

Teorema:

In una tavola hash le cui collisioni sono risolte con il concatenamento, una ricerca con successo richiede un tempo $\theta(1 + \alpha)$ nel caso medio, nell'ipotesi di hashing uniforme semplice.

Dimostrazione:

Si suppone che l'elemento da ricercare abbia la stessa probabilità di essere uno

qualsiasi degli n elementi memorizzati nella tavola. Il numero di elementi esaminati durante una ricerca con successo di un elemento x è uno in più del numero di elementi che si trovano prima di x nella lista di x. Si ha quindi:

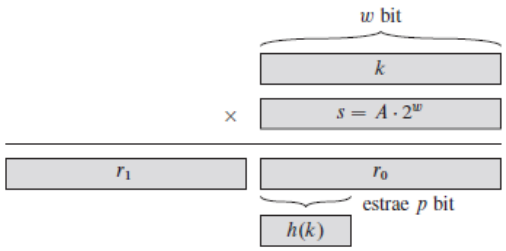
$$
\begin{aligned}
&\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&\quad = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}\left[X_{ij}\right]\right) \quad \text{(per la linearità del valore atteso)} \\
&\quad = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&\quad = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&\quad = 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&\quad = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \quad \text{(per l'equazione (A.1))} \\
&\quad = 1+\frac{n-1}{2m} \\
&\quad = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}
$$

In conclusione, il tempo totale richiesto per una ricerca con successo è $\Theta(2+\alpha/2-\alpha/2n) = \Theta(1+\alpha)$.

48) Calcolo funzione hash: descrivere il metodo della divisione

Quando si applica il metodo della divisione per creare una funzione hash, una chiave k viene associata a una delle m celle prendendo il resto della divisione fra k e m; cioè la funzione hash è

$$h(k) = k \bmod m$$



49) Calcolo funzione hash: descrivere il metodo della moltiplicazione

Il metodo della moltiplicazione per creare funzioni hash si svolge in due passi.
Il primo passo è moltiplicare la chiave k per una costante A tale che $0 < A < 1$ e si estrae la parte frazionaria di kA.
Il secondo passo è moltiplicare tale valore per m e si prende la parte intera inferiore del risultato.

$$h(k) = \lfloor m\,(k\,A \bmod 1)\rfloor$$

50) Calcolo funzione hash: descrivere la tecnica di hashing universale

L'approccio hashing universale permettere di ottenere buone prestazioni in media scegliendo casualmente la funzione hash in modo che sia indipendente dalle chiavi che devono essere effettivamente memorizzate.
Sia H un insieme di funzioni hash, H è definita universale per ogni coppia $k, l \in U$, il numero di funzioni hash $h \in H$ per cui $h(k) = h(l)$ è $\frac{|H|}{m}$. In questo caso si definisce la funzioni hash $h_{ab}\ \forall a \in \mathbb{Z}_p^* \wedge \forall b \in \mathbb{Z}_p$ utilizzando una trasformazione lineare seguita da riduzioni modulo p e poi modulo m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$$

51) Descrivere la tecnica di indirizzamento aperto

Nell'indirizzamento aperto, tutti gli elementi sono memorizzati nella tavola hash stessa; ovvero ogni cella della tavola contiene un elemento dell'insieme dinamico o la costante NIL. Quando si cerca un elemento, si esamina sistematicamente le celle della tavola finché non si trova l'elemento desiderato o finché non ci si accorge che l'elemento non si trova nella tavola. In tale tecnica, non ci sono liste né elementi memorizzati all'esterno della tavola, per cui la tavola hash può "riempirsi" al punto tale che non possono essere effettuati altri inserimenti una conseguenza è che il fattore di carico α non supera mai 1. Per effettuare un inserimento mediante l'indirizzamento aperto, si esamina in successione le posizioni della tavola hash (ispezione), finché non si individua una cella vuota in cui inserire la chiave. Al posto di procedere sequenzialmente $0, 1, \ldots, m-1$, la sequenza delle posizioni esaminate durante una ispezione dipende dalla chiave da inserire.
Si ha quindi:

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

52) Fornire lo pseudocodice dell'inserimento e ricerca tramite indirizzamento aperto

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "overflow della tavola hash"
```

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

53) Quali sono le possibili ispezioni adottate nella tecnica di indirizzamento aperto?

Si hanno:

→ Ispezione Lineare: si usa la funzione hash

$h(k,i) = (h'(k) + i) \bmod m$

dove $i = 0, 1, \ldots, m-1$.

Il problema principale di tale ispezione è l'addensamento primario: si formano lunghe file di celle occupate, che aumentano il tempo medio di ricerca.

Gli addensamenti si formano perché una cella vuota preceduta da i celle piene ha la probabilità $(i+1)/m$ di essere la prossima ad essere occupata.

→ Ispezione Lineare: si usa la funzione hash

$h(k,i) = (h(k) + c_1 i + c_2 i^2) \mod m$

Il problema principale di tale ispezione è l'addensamento secondario: la prima posizione determina l'intera sequenza, quindi vengono utilizzate soltanto m sequenze di ispezione distinte.

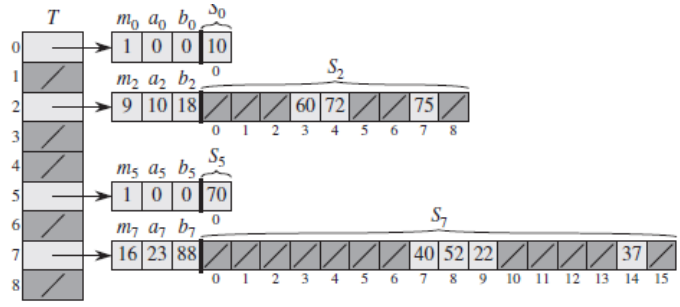
→ Doppio hashing: si usa la funzione hash

$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

dove h_1 e h_2 sono funzioni ausiliarie. Risulta necessario però considerare che $h_2(k)$ deve essere primo rispetto a m.

54) Descrivere la tecnica di hashing perfetto (o statico)

Si definisce hashing perfetto una tecnica di hashing in cui il numero di accessi in memoria richiesti per svolgere una ricerca è $O(1)$ nel caso peggiore. Si utilizza uno schema di hashing a due livelli, con un hashing universale in ciascun livello.



Il primo livello è essenzialmente lo stesso di hashing con concatenamento le n chiavi sono associate a m celle utilizzando una funzione hash h accuratamente selezionata da una famiglia universale di funzione hash. Al posto di creare una lista delle chiavi associate alla cella j si crea una tabella hash secondaria S_j con una funzione hash h_j associata. Scegliendo le funzioni h_j in modo accurato si evitano collisioni.