# Strutture Dati

Elisa Stabilini

I Semestre A.A. 22/23

# Contents


**1 Prima lezione: BFS e DFS** **5**
- 1.1 Argomenti della lezione . . . 5
- 1.2 Notazione . . . 6
- 1.3 Breadth First Search . . . 6
  - 1.3.1 Proprietà . . . 7
  - 1.3.2 Costo computazionale . . . 7
  - 1.3.3 Albero di copertura minimo . . . 8
- 1.4 Depth First Search . . . 8
  - 1.4.1 Costo computazionale . . . 9
  - 1.4.2 Proprietà . . . 9

**2 Random e Quantum walks sui grafi** **11**
- 2.1 Matrice di transizione . . . 12
- 2.2 Esplorazione non-sistematica . . . 12
- 2.3 Nozioni rilevanti . . . 13

**3 Strutture dati** **15**
- 3.1 Contenitore di dati (ADT) . . . 15
  - 3.1.1 Array . . . 15
  - 3.1.2 Liste . . . 16
- 3.2 ADT . . . 16
- 3.3 Code . . . 17
- 3.4 Stack . . . 17
- 3.5 Alberi dinamici . . . 18
- 3.6 Recap pro e contro dei diversi contenitori . . . 19

**4 Hashing crittografico** **21**
- 4.1 Funzioni di Hash . . . 22

**5 Rappresentazione delle matrici** **23**
- 5.1 List Of Lists - LOL . . . 23
- 5.2 Coordinate list- COO . . . 24
- 5.3 Compressed row storage - CRS . . . 24

**6 Compressione** **27**
- 6.1 Singular value decomposition - SVD . . . 27
- 6.2 Matrix product states . . . 28

# Chapter 1

# Prima lezione: BFS e DFS

## 1.1 Argomenti della lezione

Dato un grafo, l'obiettivo che ci poniamo è quello di visitarlo in modo sistematico. Partiremo da un vertice marcato da S e assumiamo che il grafico sia connesso (tutti i vertici siano visitabili da S). Definiamo un sotto-grafo (insieme dei vertici originale e come archi un sottoinsieme -possibilmente proprio dell'insieme degli archi). è un grafo che non ha cicli, non è orientato ed è aciclico minimale (albero -¿ se rimuovo un arco, il grafo non è più connesso). Quindi facciamo le seguenti assunzioni:

- La visita parte da un nodo $s \in V$
- Tutti i vertici sono raggiungibili da s (questa seconda assunzione si può rilassare se si considera la visita sistematica dei vertici che sono raggiungibili da s.

Il risultato di questo algoritmo è un **albero di copertura del grafo**. ovvero un sotto-grafo di G che sia:

- privo di loop
- non orientato (in questo caso)
- aciclico (quindi minimale)

**Albero:** dati due nodi $u, w \in V \exists!$ un cammino tra $u, w$

**NB:** Se è definita una funzione peso, quindi il grafico è pesato, gli algoritmi di esplorazione sistematica che vedremo non garantiscono che il cammino sia minimale.

Per semplicità assumiamo che il grafo sia non orientato (assunzione non necessaria), consideriamo però che costruzioni simili si possono determinare anche per grafi orientati

Come visitare un grafo in modo sistematico (deterministico), visitare tutti i vertici raggiungibili (quindi tutti). Obiettivo costruire un *albero di copertura*, **non** è l'albero di copertura minimo (perché non sto considerando la presenza di una eventuale funzione peso. Questo è un tema importante perchè lo studio della raggiungibilità e la conta dei percorsi ci si dicono se i vertici sono raggiungibili ma non *spiegano* i percorsi; inoltre la visita casuale dei grafi può richiedere molti passi per raggiungere un nodo.

## 1.2 Notazione

Per effettuare la visita sistematica dovremo assegnare degli **attributi** ai nodi o archi del grafo. Per questo utilizzeremo la notazione $v.a$ In particolare, per tenere traccia dei vertici già visitati dall'algoritmo utilizzo i seguenti attributi

- *un*: unexplored
- *open*: incontrato almeno una volta
- *closed*: incontrato almeno una volta e tutti i vertici vicini sono stati esaminati

I due algoritmi di visita che verranno studiati sono BFS (breadth-first-search), algoritmo di visita in ampiezza e DFS (depth-first-search) algoritmo di visita in profondità. A seconda di quale dei due algoritmi si usa gli alberi di copertura sono diversi e soddisfano regole diverse.

Questi due algoritmi **hanno lo stesso costo computazionale** (non si distinguono per efficienza, entrambi richiedono dei tempi di esecuzione proporzionale alla dimensione dell'insieme dei vertici + la dimensione dell'insieme degli archi). Il costo computazionale dipende anche dalla rappresentazione che uso per il grafo:

- lista di adiacenza: costo computazionale $\Theta(|V| + |E|)$
- matrice di adiacenza: costo computazionale $\Theta(|V|^2)$

Una seconda cosa che differenzia i due approcci sono le politiche di visita, BFS si basa su una politica di visita FIFO ovvero *First In, First Out*, per questo motivo viene utilizzato con le code, o **queue**, per cui gli elementi vengono aggiunti in coda e il processo di rimozione avviene partendo dalal testa. D'altra parte invece, DFS si basa su una politica di visita di tipo LIFO, ovvero *Last In, First Out* per cui viene utilizzata con strutture di tipo **stack** ovvero pile: una volta che aggiungo un elemento alla pila lo aggiungo in cima e l'unico elemento della pila che può essere rimosso è proprio l'ultimo che ho aggiunto.

## 1.3 Breadth First Search

**Istanza**: l'input è un grafo (in generale anche orientato ma in questo caso no) **Output**: L'output è un albero di copertura (a sua volta un grafo in cui l'insieme dei vertici è sempre V mentre l'insieme degli archi è un insieme, possibilmente strettamente contenuto in E).

Funzionamento dell'algoritmo: Idea: parto da un vertice marcato S (start) e da lì in avanti esploro un intorno di lunghezza uno (tutto ciò che si trova entro un arco di distanza - anche perché ho detto che non ci sono pesi). Fatto questo esploro tutti i vertici a due archi di distanza. Questo ampliamento del raggio è simile a quello di Dijkstrat. Quindi posso scrivere un algoritmo come segue:

Anzitutto si marcano tutti i vertici come *unexplored* (all'inizio tutti i vertici **non** sono stati esplorati.

Dopo di che al vertice S assegno lo stato di *open*, chiamo T il nuovo grafico - che al termine dell'algoritmo sarà l'output) assegno S. A questo punto si prende una cosa (oggetto a cui si aggiungono oggetti

a questo punto parte un ciclo che dice che finché la coda non è vuota esegue delle istruzioni (sicuramente non è vuota)

- elimino il primo

```
for all v ∈ V do
    v.status = un;
end
s.status = open ;
T.insert(s) ;
F.enque(S) ;
while !F.empty() do do
    u = F.deque();
    for v ∈ adiacenti(u) do
        if v.status=un then
            v.status = open;
            F.enque(v);
            I.insert((u,v));
        end
    end
    u.status = closed;
end
```

Algorithm 1: BFS(G,s)

- si assegna al primo una certa variabile u
- si chiede di prendere tutto ciò che è adiacente al vertice u, per ogni vertice di questo sito ci chiediamo se lo stato del vertice v vicino a u, allora indico che lo stato è open, accodo v a alla coda F e poi inserisco nell'albero il vertice (u,v)
- a questo punto indico u come chiuso

La distanza massimo tra il primo vertice è quello più lontano è esattamente pari al numero di passi che devo fare per aggiungerlo.

**Nota:** BFS si estende naturalmente a grafi non connessi. Quello che si ottiene in questo caso è una foresta di grafi non connesi.

### 1.3.1 Proprietà

: la coda F contiene per definizione tutte e solo i vertici con stato open (per definizione perché vengono aggiunte solo se non sono ancora state chiuse), ovvero i vertici visitati almeno una volta. Prima del while (prima dell'inizializzazione la coda è vuota

Mano mano che si esplora un grafo si può anche capire quanto lontano è ciascun vertice dal vertice di partenza (posso ad esempio definire un attributo d a un vertice), la prima volta che fisso un vertice. Posoo capire anche la distanza di termini di pasti dal vertice di partenza. In sostanza, considerando che il BFS produce un albero si potrebbe anche operativamente memorizzare l'albero indicando per ogni nodo chi è il predecessore. Ad esempio, posso pensare ad un algoritmo del tipo

### 1.3.2 Costo computazionale

Ho un ciclo while e all'interno un for, la lunghezza del ciclo for dipende dal numero di vertici che ho nel grafo, ne ho —V—, quindi la complessità in tempo del ciclo for è $\mathcal{O}(|V|)$ (perchè ogni operazione di coda, come enque o deque richiede un tempo

dell'ordine del'unità). Inoltre ogni elemento della lista di adiacenza di ciascun nodo viene esplorato una sola volta, con un tempo di $\mathcal{O}(|E|)$, quindi il tempo complessivo è $\mathcal{O}(|V| + |E|)$. NB fino a qui posso ignorare il costo in tempo della condizione if. Inoltre si osservi che, se avessimo usato una matrice di adiacenza, anzichè una lista di adiacenza il tempo sarebbe $\mathcal{O}(|V|^2)$

### 1.3.3 Albero di copertura minimo

Questo teorema restituisce l'albero di copertura MINIMO (non minimale)

**Teorema** Sia G=(v,E) un grafo (diretto o non diretto) e sia $s \in V$. Allora, durante la sua esecuzione BFS scopre tutti i nodi del grafo G e, una volta terminato, $v.d = \delta(s, v) \forall v \in V$; dove $\forall u, v \in V, \delta(u, v)$ è il cammino più corto che unisce i due nodi u e v (si usa $\delta(u, c) = +\infty$ se v non è raggiungibile da u. Inoltre $\forall v \neq s : \delta(s, v) < +\infty$, uno dei percorsi più brevi tra s e $v.\pi$ è

- il percorso più breve tra s e $v.\pi$
- segmento dell'arco $(v.\pi, v)$

la delta mi introduce una topologia nel grafo perché mi dà la nozione di distanza (all'inizializzazione tutti i vertici vengono posti come a distanza più infinito e tale rimane tale la distanza, quindi significa che il grafo non è connesso)

**NB:** BFS si estende naturalmente ai grafi non connessi, in questo caso quello che si ottiene è una foresta di alberi di copertura.

## 1.4 Depth First Search

Nessuno mi obbliga di scegliere un strategia specifica per l'esplorazione di un grafo, mentre BFS va per larghezza, allontanamento successivo dalla distanza di partenza, DFS cerca di allontanarsi il più possibile dal punto di partenza. Ogni nodo scoperto diventa quindi immediatamente punto di partenza per una esplorazione sistematica.

Innanzitutto, per evitare di esplorare più volte un vertice utilizziamo nuovamente gli stessi attributi aggiuntivi che abbiamo definito prima: open, closed, unexplored. L'input dell'algoritmo è un grafo $G = (V, E)$ e un vertice di start $s \in V$, l'output è, anche in questo caso un albero di coeprtura.

Per prima cosa scrivo una sort di routine di supporto che mi restituisce l'albero di copertura: Dopo di che scrivo il vero e proprio algoritmo

```
for v ∈ V do
    v.status = un;
    T=empty tree;
    DFS(s,G,T);
end
```

Algorithm 2: DFSTree (G,s)

Quindi l'algoritmo funziona come segue: suppongo inizialmente di avere albero vuoto, tutti i vertici unexplored e poi parto con DFS. A questo punto, dato un vertice V (DFS prende in ingresso il grafo, l'albero di copertura e un certo vertice

```
v.status = open;
for w in adj(v) do
    if w.status = un then
        T.insert(w);
        DFS(w,G,T);
    end
end
v.status=closed;
```

Algorithm 3: DFS(G,V,T)

v assegnato, non necessariamente il vertice di partenza). Se lo stato del vicino è unexplored allora inserisce quel vicino nell'albero e poi richiama ricorsivamente DFS a partire dal vertice V che ha appena incontrato.

Si nota che questo algoritmo ricalca un po' l'idea di prima, prendo quel punto che passo come argomento come punto di partenza per una ricerca sul grafo. Se V è open quello che faccio è andare a prendere un vicino e poi prendo w come punto di partenza ricorsivamente. Quando esco da questo for dico che V è closed.

Ciascuno utilizza uno stack, quando ci sono le funzioni ricorsive implicitamente si crea uno stack per cui si accumulano cose in cima che poi sono le prime ad essere utilizzate o affrontate. Mentre in una coda il primo inserito è il primo servito, al contrario di quanto avviene uno stack il primo inserito è l'ultimo servito.

### 1.4.1 Costo computazionale

Il numero di chiamate di DFS è proprio pari alla cardinalità di V, ciascuna chiamata aggiunge all'albero un numero di vertici pari alla lista di adiacenza di V, dopo di che ho un secondo ciclo for, di conseguenza, il costo computazionale complessivo di DFS è $\Theta(|V|+|E|)$. Anche in questo caso, l'uso di matrici di adiacenza aumenta il costo computazionale del for a —V— e di conseguenza il costo computazionale complessivo dell'algoritmo diventa $\Theta(|V|^2)$. La lista di adiacenza è minimale per il grafo, mentre la matrice di adiacenza occupa sempre lo stesso spazio quindi non è minimale.

### 1.4.2 Proprietà

Posso dotare il mio vertice di attributi aggiuntivi:

- v.d: registra il tempo di visita di v
- v.$\pi$: registra il predecessore di v
- v.f: registra il tempo di chiusura di V

Posso modificare `DFSTree` e `DFS` come segue:

Quando il mio oggetto finisce di essere trattato si segna un punto di chiusura, quindi per ogni nodo mi segno l'istante in cui è stato scoperto e l'istante in cui è stato chiuso. Questa informazione è utile per il seguente teorema.

**Teorema** In una visita in profondità di un grafo G=(V,E), per ogni coppia $u, v \in V$ vale esattamente una selle seguenti condizioni:

1. $[u.d, u.f] \cap [v.d, v.f] = \emptyset \iff$ in `DFSTree` u non è discendente di v e v non è discendente di u.

```
time = 0;
for v ∈ V do
    v.status = un;
    v.π = null;
    T=empty tree;
    DFS(s,G,T);
end
```

Algorithm 4: DFSTree (G,s)

```
time = time +1;
v.status = open;
v.d = time;
for w in adj(v) do
    if w.status = un then
        w.π = v;
        T.insert(w);
        DFS(w,G,T);
    end
end
v.status=closed;
time = time +1;
v.f = time;
```

Algorithm 5: DFS(G,V,T)

2. $[u.d, u.f] \subset [v.d, v.f] \iff$ u è discendente di v
3. $[v.d, v.f] \subset [u.d, u.f] \iff$ V è discendente di u

E' interessante perché la storia delle aperture e chiusure di un nodo costituisce un'espressione ben formata nel senso che, se indico la scoperta di un nodo con *(u* e la chiusura di un nodo con *u)* le parentesi sono opportunamente annidate. Si parla in questi casi di ordinamento topologico (teoria dei processi).

Ordinamento dei vertici di un grafo t. c. se uv appartiene all'insieme degli archi allora u viene prima di v nell'ordinamento.

# Chapter 2

# Random e Quantum walks sui grafi

I grafi sono abbastanza pervasivi, ci sono moltissime realtà che possono essere modernizzate con grafi. Per i fisici i grafi si trovano abbastanza naturalmente, ad esempio le interazioni tra oggetti possono essere modellati con grafo. Ci sono delle leggi, anche fisiche, che sono espresse tramite grafi.

Consideriamo un grafo non orientato $G(E, V)$, che per il momento supponiamo, per smeplicità, non pesato e sia —V—=N e V = $v_1, ..., v_N$. Consideriamo la sua matrice di adiacenza $A_G$ e consideriamo anche una matrice che è la matrice **dei gradi dei vertici** (numero di archi che insistono su un vertice, sono tanti quanti i primi vicini del verso assegnato) $D_G$.

Si definisce quindi la matrice laplaciana del grafo

$$L_G = D_G - A_G \tag{2.1}$$

ovvero la differenza tra la matrice dei gradi e di adiacenza. $L_G$ è una matrice reale e simmetrica, quindi è semi-definita positiva (gli autovalori sono tutti non negativi, e ammette anche lo 0).

Viene chiamata matrice laplaciana andando a vedere la matrice di Newton per la conduttività del calore. Guardando le slide della lezione $u_i(t)$ è la quantità di calore nel sito i-esimo, k la conduttività (qualsiasi cosa sia, non per forza questo da un punto di vista fisico). La somma su j di $A_ij$ è il grado del vertice in pratica. Quindi in questo modo si determina il flusso di calore uscente dall'i-esimo sito. Questa legge può essere vista come un'equazione di Kirchoff o di equazione di conservazione. La matrice laplaciana aiuta molto a **stabilire le proprietà del grafo a cui si riferisce**. Proprio per il fatto che abbiamo una equazione di continuità, autovettore 1 è autostato di autovalore 0 (la matrice è singolare).

**proprietà** Andando a vedere la matrice laplaciano si può ottenere l'insieme dei **vertici connessi**. La molteplicità algebrica di 0 (autovalore) indica il numero di componente connesse nel grafo. Inoltre le componenti connesse formano matrici diagonali a blocchi (previa rinumerazione eventuale).

## 2.1 Matrice di transizione

Il nostro focus è però L'esplorazione *casuale* del grafo ovvero la dinamica di una particella, *walker*, che si muova sul grafo seguendo un processo stocastico determinato dalla topologia del grafo.

Definiamo la matrice di transizione come il prodotto della matrice di adiacenza e dell'inverso della matrice di grado, ovvero

$$T_G = A_G \cdot D_G^- 1 \tag{2.2}$$

per cui questa matrice vale

- 0 sulla diagonale
- $P_{ij} = \frac{1}{deg(v_i)}$ se $i \neq jeA_{ij} \neq 0$
- 0 altrimenti

Questa matrice è una **matrice stocastica** quindi

$$P_ij = [T_G]_{ij} = P(v_i \to v_j) \text{ e } \sum_i P_{ij} = 1 \forall j \tag{2.3}$$

Gli elementi della matrice di transizione sono delle probabilità che dicono qual è la probabilità partendo da un sito i di arrivare ad un sito j o viceversa (sulle slide indico con il ket la posizione di un camminatore che esplora il grafo). Se a questo ket applico la matrice di transizione ottengo un vettore che ha tutte componenti non nulle per tutti i punti in cui mi posso spostare. In questo caso sto pensando (dato che non ci sono pesi)la probabilità di transire ad un sito adiacente è equivalente per tutti i siti adiacenti.

Inoltre, per ovvi motivi vale

$$T_G^k |i\rangle = P(v_i \to v_j) \text{in k passi} \tag{2.4}$$

Una matrice è stocastica perché se sommo gli elementi lungo una colonna ottengo sempre 1. Quindi per come è definita la matrice di transizione ad un passo non posso rimanere dove sono ma devo per forza spostarmi da qualche parte. Una matrice con queste è detta matrice stocastica. Date che i è il punto di partenza e la somma su i mi dà 1 allora devo muovermi per forza.

Questo succede **per come è definita la matrice di transizione** . Ovviamente tutto questo può essere esteso anche a grafi pesati, in questo caso il camminatore salta in nodi vicini con una probabilità che dipende dal nodo di arrivo.

## 2.2 Esplorazione non-sistematica

Il processo di visita casuale di un grafo è un processo stocastico Markoviano ovvero **senza memoria** a tempi distinti e su uno spazio delle configurazioni discreto. Nello specifico è lo spazio delle configurazione è $S = v_1, v_2, ...v_N$ oovero i vertici del grafo. Si definisce una famiglia di funzioni ad un parametro di variabili casuali (sequenza di variabili casuali $\{X_i : S \to \mathbb{Z}^+, i = 0, 1, 2...\}$ La probabilità che al tempo i il walker si trovi nel sito k è $P(X_i = k)$.

Mi chiedo quale sia la probabilità di essere al tempo n+1 (riprendi gli appunti sul perché un moto Browniano è markowiano). è un processo anche stazionario perché l'ampiezza di probabilità di transizione non varia nel tempo. Processo stocastico Markoviano stazionario (guarda appunti di galli fino all'equazione di Chapman e Kolmogrov).

## 2.3 Nozioni rilevanti

Il nostro grafo viene quindi visitato in modo casuale attraverso un processo stocastico che, nel nostro caso è definito a partire dalla topologia (struttura del grafo). Ovviamente a questo punto il nostro obiettivo non è più quello di determinare un albero di copertura ma è più relativa all'esplorazione **generica** di configurazioni. Introduco a questo punto alcune definizioni: definisco la probabilità di transire da i a j in n passi imponendo che non si passi mai da j prima del termine (posso anche traslare nel tempo perché tanto il processo è stazionario). Quindi:

$$f_{j,i}^{(n)} = P(X_{m+n} = j, X_{n+\nu} \neq j, 1 \leq \nu < n | X_m = i) \tag{2.5}$$

Inoltre dico che un sito è raggiungibile se ci vuole un tempo finito per raggiungerlo (la probabilità diventa 1 entro un tempo finito):

$$f_{j,i}^{*} = \sum_{\nu=1}^{+\infty} f_{ji}^{(\nu)} m_{ji} = \sum_{\nu=1}^{+\infty} \nu \cdot f_{ji}^{(\nu)} < +\infty \tag{2.6}$$

Inoltre

$$\sigma_{ji}^{2} = \sum_{\nu=1}^{+\infty} (\nu - m_{ji})^2 f_j i^{(\nu)} \tag{2.7}$$

dove $m_{ji}$ è il tempo medio di primo passaggio e $m_{ii}$ è il tempo medio di ricorrenza

Questo tipo di modellizzazione dell'attraversamento dei grafi richiede delle nozioni differenti da quelle deterministiche e la definizione di grandezze che sono probabilistiche e quindi richiedono un approccio differente.

**Raggiungibilità e random walks** La nozione di raggiungibilità ha un nuovo signiificato nel contesto dei random walks: dati due vertici $i, j$, se $f_{ji}^{*} f_i > 0$; ovvero $\exists n(i,j), n(j,i) t.c. p_{ji}^{n(j,i)} \mathbf{and} p_{ij}^{n(i,j)}$ allora $i$ e $j$ sono nella stessa classe. In particolare

- se $f_{ii}^{*} = 1$: i è ricorrente
- se $f_{ii}^{*} < 1$: i è transiente
- se $f_{ii}^{*} = 1$: e j è nella stessa classe di i allora $f_{ij}^{*} = 1$

L'insieme degli stati ricorrenti nella stessa classe è l'analogo della componente connesa del grafo. Il corrispondente processo stocastico, risolto ai vertici riccorenti nella stessa classe è detto irriducibile.

La dinamica di un walker quantistico su un determinato tipo di grafo è esponenzialmente più veloce di un camminatore classico.

# Chapter 3

# Strutture dati

Con la lezione scorsa abbiamo chiuso l'argomento dei grafi. Oggi affrontiamo qualcosa di diverso, si comincia davvero con le strutture dati.

## 3.1 Contenitore di dati (ADT)

Prendo in considerazione un archivio anagrafico (oggetto con tante schede, all'interno di ciascuna delle quali sono associate una serie di informazioni), a me non interessa tanto il contenuto della scheda anagrafica, quello che mi interessa è

- aggiungere scheda grafica
- cancellare scheda anagrafica deceduto
- trovare scheda (per diversi motivi)

Questo archivio in sostanza è una collezione di oggetti dello stesso tipo, io devo semplicemente occuparmi della loro organizzazione. Questo archivio può essere denominato contenitore

Contenitore: entità che contiene tanti elementi dello stesso tipo, di tipo generico non mi interessa di che tipo, e che questo contenitore metta a disposizione di chi lo vuole usare tre funzioni: inserimento di un oggetto, rimozione di un oggetto, ricerca di un oggetto.

Diciamo che a noi non interessa esattamente il contenuto dell'oggetto, quello che mi interessa però è che ciascuna scheda grafica sia in qualche modo univocamente identificata, ho bisogno di una chiave diversa per ciascuna scheda per poter accedere all'archivio.

### 3.1.1 Array

Quindi ad ogni oggetto associo una chiave univoca, il resto dal mio punto di vista non mi riguarda. Prima cosa che mi viene in mente per implementarlo è un array (sicuramente si può fare).

**Criticità e vantaggi** Se ho un array lungo la ricerca può non essere efficace (se l'array non è ordinato). Inoltre allargare un'array coinvolge un sacco di operazioni di copia: devo crearne uno più grande e poi trasferirci tutto il materiale da uno all'altro. L'array è una n-upla di posizioni di memoria contigue, quindi anche semplicemente spostare un dato può essere problematico. Inserimento e rimozione quindi possono essere complicati

### 3.1.2 Liste

Si potrebbe utilizzare una lista puntatori (strutture in cui esiste una testa e una coda ed ogni elemento della lista sa chi è chi lo precede e chi lo segue). Quindi in sostanza data la testa della lista posso scorrere tutta la lista fino alla coda, viceversa data la coda posso scorrere tutti gli elementi fino alla testa. Una lista di questo tipo è detta lista **doppiamente concatenata**

**Criticità e vantaggi** Inserimento ed eliminazione in questo caso sono molto meno costose rispetto a quanto succede per un array, se devo togliere un elemento della lista mi basta eliminarlo senza lasciare buchi e senza dover spostare tutti gli elementi successivi. Le stesse operazioni di inserimento cancellazione e ricerca si possono fare. Nella lista doppiamente concatenata l'inserimento prevede una sola copia, se posso accedere solo dalla testa o solo dalla coda ci impiegherò in media N/2 (mentre in un array questa fase può essere più semplice). Una volta raggiunta la posizione di rimozione anche la rimozione è molto più semplice. Il punto fondamentale di una lista però è che, quando modello un contenitore, ciò che importa non è come organizzo il contenitore, ciò che definisce un contenitore sono **le operazioni che io faccio sul contenitore**, tutto il resto è assolutamente sovrastrutturale. All'utente interessa avere a disposizione le tre possibilità, non mi interessa come queste siano effettivamente implementate

## 3.2 ADT

ADT (Abstract Data Type) in questo tipo di dato quello che importa sono i dati (che posso contenere) e le operazioni sui dati. Questo modo di ragionare è fondamentale per sviluppare un progetto di un certo spessore. Devo capire quali sono i dati fondamentali e quali sono le operazioni che posso fare sui dati. Finché chi mi realizza il prodotto mi garantisce quali operazioni posso fare a me va bene così. Questo concetto è centrale nella **programmazione ad oggetti**.

Ad esempio se penso ad una classe, quello che lo sviluppatore mi mette a disposizioni sono dei campi della classe (pubblici) e i metodi della classe. Èer C++ la programmazione ad oggetti è fondamentale. Anche nella costruzione di una funzione è importante la stessa cosa. Da una parte faccio la dichiarazione della funzione )che è l'unica cosa che interessa al cliente finale), l'implementazione della funzione è da un'altra parte.

Questo è un passaggio fondamentale sia dal punto di vista teorico che dal punto di vista concreto perché è ciò che ha reso possibile la progettazione software su larga scala.

ADT: è una coppia di dati e operazione sui dati, nell'ADT non compare da nessuna parte l'implementazione di ADT. In sostanza l'ADT definisce l'interfaccia del dato: come l'utente interagisce con il dato, non si sa nulla , questo da una grande libertà a chi sviluppa software perché finché non cambio l'interfaccia posso cambiare dietro quello che voglio.

Esistenza di un elemento fondamentale: contenitore (singola unità) identificata univocamente dalla chiave (questo e il dato) e d'altra parte ho le operazioni possibili sui dati che costituiscono un dizionario.

Quello che dobbiamo fare è cercare di capire come scrivere delle *possibili* implementazioni di ADT.

## 3.3 Code

Una **coda** è un tipo di dato astratto contenitore, è un contenitore particolare (non generico). È un contenitore e quindi deve mettere a disposizione le operazioni menzionate prima. La cosa interessante è che inserimento e cancellazione non avvengono in luoghi arbitrari ma sempre in luoghi specifici (la rimozione avviene sempre in testa, l'inserimento sempre in coda). Siccome posso inserire e rimuovere solo in posti specifici allora la gestione degli elementi della coda è abbastanza rigido, l'unico modo che ho per accedere ad un elemento per poterlo rimuovere è farlo arrivare in testa, non posso eliminare un elemento prima di un elemento arrivato in precedenza. **Ogni coda è un dizionario, non tutti i dizionari sono code**

L'implementazione di una coda può essere fatta in maniera completamente diversa da quella di un altro tipo di ADT. Quando una persona implementa una coda ha in mente un oggetto che deve crescere (deve crescere dinamicamente, a seconda delle richieste diciamo). Il contenitore che memorizza le richieste deve essere in grado di modulare la sua dimensione a seconda della necessità.

**Inserimento** Questo suggerisce a me di usare, per l'implementazione di una coda, una *lista doppiamente concatenata*. Quello che succede quindi è che il dato che avevo all'inizio deve essere incapsulato all'interno di una sovrastruttura che lo contenga, ma che insieme ad esso contenga anche delle informazioni di accesso. Prendo il dato lo inserisco in questa struttura che conterrà il dato, un puntatore al precedente e un puntatore al successivo. La specificità è che il dato viene inserito in coda. A questo punto se voglio aggiungere un dato, quello che so è che la mia tail sa di essere l'ultimo elemento della coda, quindi è facile fare sì che il puntatore al precedente assuma proprio il puntatore al precedente. A questo punto il successore è nessuno (quello che devo andare ad aggiornare è il campo del puntatore al successore a quello che prima era l'ultimo). L'inserimento di un elemento nella coda mi richiede semplicemente 3 operazioni (non un numero di operazioni proporzionali al numero di elementi nella struttura dati).

**Rimozione** La rimozione di un elemento (so già dove devo rimuovere un elemento, devo rimuoverlo per forza dalla testa, questo significa semplicemente prendere il campo predecessore (aggiornare il predecessore dell'elemento che segue e aggiornare ...

**Ricerca** Le operazioni di ricerca in una coda, ma in generale in una lista non sono banali. Ho due punti d'accesso: o la testa o la coda, quindi l'unico modo che ho a disposizione è prendere un elemento e farli scorrere. Anche se gli elementi nella coda fossero (e non possono esserlo per come è definita la coda) ordinati, non posso comunque accedere all'elemento che voglio. Le code infatti non vengono usate per operazioni di ricerca.

## 3.4 Stack

Lo stack è a sua volta un contenitore in cui però la politica di inserimento e rimozione è diversa da quelle di prima. Nel caso dello stack, inserimento e rimozione sono specifici del contenuto e sono entrambi fatti in testa (o in coda a seconda del punto di vista). Chiaramente deve esistere anche una operazione di ricerca.

**Implementazione** Anche in questo caso sono possibili diverse implementazioni: posso sempre usare l'array, anche in questo caso però per gli stack. Per inserire un nuovo elemento si fa sostanzialmente come prima per la coda, ho un nuovo dato che vie incapsulato nella sovrastruttura per la gestione della lista, dopo di che al campo left assegno il contenuto dell'altro elemento della lista e a destra nessuno (perché io sono il primo elemento della coda). Mi aspetto che il costo di inserimento e di rimozione non dipenda dall'altezza dello stack.

**Ricerca** Il problema in questo caso è trovare un elemento con una data chiave e poi restituirlo (generalmente quello che viene restituito non è l'elemento stesso ma un puntatore all'elemento). A questo punto ho deu possibilità a seconda che la lista o l'array sia ordinato o no, una lista ha un costo per la ricerca che è proporzionale alla sua dimensione, per un array invece il costo è pari al log della sua dimensione.

Un array ha una ricerca più efficiente se è ordinato ma è assolutamente sfavorevole per quanto riguarda inserimento e rimozione. Una lista invece può crescere in modo molto semplice ma sulla ricerca

La domanda quindi è chiedersi se esista una struttura dtai che prenda il meglio delle due cose: che sia dinamica mi consenta di effettuare rimozione e inserimento in maniera semplice e che consenta una ricerca altrettanto semplice.

## 3.5 Alberi dinamici

Quello di cui ci siamo resi conto è che in realtà l'implementazione del contenitore non è indifferente, a seconda del contesto in cui il contenitore viene creato e a seconda dell'implementazione posso avere un costo computazionale diverso. L'idea è quella di imporre una sovra-struttura su una struttura dati dinamica (la lista è una struttura dati dinamica ma senza sovrastruttura quindi completamente inefficiente).

Assumo che la chiave appartenga ad un insieme con una relazione d'ordine ben definita, supponiamo che questa chiave (quella che identifica il dato) sia composta di due interi. Devo inserire il mio elemento nella mia struttura (prevede che io inserisca un elemento a sinistra o addestra di quello che lo precede, (lo inserisco sfruttando la relazione d'ordine tra le chiavi dei singoli dati.

SI nota che qui non si ha solo la posizione destra e sinistra ma ci sono anche dei livelli. La struttura dati è dinamica, non devo fare nulla di difficile per inserire un elemento. QUanti elementi dell'albero esistente devo aggiungere per poter In generale il numero di confronti che posso fare al massimo è pari all'altezza dell'albero l'altezza dell'albero limita dall'Alto il numero di confronti che posso fare (perché non conta l'ampiezza dell'albero).

Voglio capire qual è la configurazione che minimizza l'altezza dell'albero, l'altezza più bassa dell'albero è quella che si ottiene quando **l'albero è ben bilanciato**, se questo è ben bilanciato la sua altezza sarà pari log2 (n). è chiaro che se questa fosse la configurazione allora anche la ricerca può chiedere non più di log2 n di confronti. So che se un elemento c'è e non è la radice; se c'è si troverà a destra se sto cercando qualcosa con una chiave maggiore di quella della radice.

Il caso peggiore è quello di un albero completamente sbilanciato. In media non avrò un albero completamente bilanciato però d'altra parte se i dati sono disordinati, non c'è alcun tipo di ordine pre-esistente allora in media l'albero andrà bene.

Esiste un modo per far si che gli alberi di ricerca binari vengano quasi perfettamente bilanciati. Il metodo è quello dei red-black-trees, ai nodi si aggiunge un

attributo (rosso o nero) per cui si va a garantire che l'albero dinamico sia quasi perfettamente bilanciato. Per questo motivo ovviamente le operazioni di inserimento e rimozione devono essere fatte in maniera un po' più complicata.

L'albero di ricerca binario si implementa mediante un "incapsulatore" che mi consenta di gestire la struttura dati dinamica, prevede sicuramente il dato, un puntatore a chi mi precede e poi due puntatori a chi mi segue (i miei figli). Quando si parte si parte dall'albero vuoto, un puntatore solo che non punta a niente e poi mano a mano si fa si che si aggiungano gli elementi. HO bisogno chiaramente di aggiornare ogni volta che aggiungo un elemento di aggiornare i campi necessari per cui da una radice si possa sempre passare ad una foglia e viceversa.

La cosa interessante è che dato un nodo, posso vederlo come punto di partenza di un sotto-grafo, questo rende abbastanza auto-evidente che tutti gli algoritmi di ricerca binaria su alberi sono tutti **algoritmi ricorsivi**, perché è proprio la struttura ad albero ade essere **ricorsiva**, un albero è sostanzialmente un nodo comune a due rami.

Caso base: albero vuoto allora inserisco elemento nell'albero, se l'albero non è vuoto, allora guardo la chiave e poi faccio una chiamata ricorsiva.

La rimozione di un elemento dall'albero è l'operazione più delicata, se tolgo l'elemento più piccolo del sotto-albero di destra (vedi slide)

O l'elemento è l'ultimo e quindi lo tolgo senza problemi, se non è una foglia invece devo andare a controllare che cosa posso fare (e vado a chiamare ricorsivamente rimuovi fino a che non trovo la chiave che cerco), prendo l'elemento più piccolo del sotto-albero di destra e lo sostituisco. La ricerca dell'elemento di valore minimo la farò cercando sempre nel sotto-albero di destra.

In un albero non ho un ordinamento definito però si può estrarre un ordinamento in modo molto facile utilizzando quella che si chiama visita in ordine. Stampo in ordine ciò che c'à a sinistra, stampo il nodo e poi a destra (ovviamente questo in modo ricorsivo. La visita in order quindi permette di estrarre tutti gli elementi, l'altro vantaggio è anche che ha una complessità lineare.

Per quanto riguarda gli alberi di ricerca binari se è bilanciato va tutto bene altrimenti no.

## 3.6 Recap pro e contro dei diversi contenitori

**Inserimento e cancellazione** In una lista molto bene è semplice, in un array è più complicato devo fare molte copie

**Ricerca** Non c'è modo efficienti di fare bisezione in una lista quindi il tempo di ricerca è lineare, in un array, soprattutto se ordinato è molto efficiente: il tempo può diventare logaritmico.

# Chapter 4

# Hashing crittografico

Le tecniche di Hashing trovano applicazione in molti contesti, tra cui la sicurezza e il controllo dell'integrità. Suppongo che D sia la cardinalità dell'insieme delle chiavi e che tutti gli oggetti del dizionario abbiano chiavi diverse. Se definisco una funzione f:S-D (cardinalità di S) allora potrei fare un direzionamento diretto. In questo modo posso costruire una tabella di indirizzamento diretto, devo inserire un elemento che ha una chiave specifica, calcolo la funzione della chiave e scrivo quell'elemento nella cella individuata dal valore della funzione.

È interessante perché **in questo caso non ho il problema dei buchi**, f è biettiva quindi posso sempre andare da un elemento delle chiavi a tabella e viceversa, quindi il costo è costante, **non dipende dalla dimensione del dizionario**.

Per fare indirizzamento diretto ho bisogno di un vettore di cardinalità pari alla cardinalità dell'insieme delle chiavi. Anche in questo caso ho un trade-off in termini di memoria.

L'idea è quella di cogliere i vantaggi e bilanciare gli svantaggi dell'indirizzamento diretto, sia h una funzione che ha dominio nello spaio delle chiavi e codominio m¡¡ D (D cardinalità insieme delle chiavi) in questo modo ho poche celle nella tabella. In questo modo però la funzione non può essere biettiva, esisteranno almeno due chiavi che possono assumere lo stesso valore nel codominio, se le due chiavi hanno un valore di hash succede una **collisione**. In realtà non è un problema, la tabella è una tabella che mi dice dove devo andare a scrivere, non posso scrivere due valori però quello che posso fare è associare ad ogni entry della tabella *l'inizio di una lista che contiene tutti gli elementi associati a quel valore.*

L'inidirzzamento diretto mi porta al punto d'inizio di una lista che contiene tutti gli elementi del dizionario con lo stesso valore di hash.

Ho un vettore di dimensione m (vettore di puntatori o teste di liste) e una volta che ho questo costo, poi ho il costo di memoria associato a ciascun elemento del dizionario , quindi ho un costo che dipende linearmente dal numero di elementi nel dizionario.

**Costo** Devo prima calcolare la funzione di hash in un punto, che ha un costo costante, non dipende dal numero di elementi, il costo di copia in una lista è anche quello O(1) perché posso scegliere di inserire l'elemento nuovo in testa alla lista. Anche in questo caso il costo è costante

**Rimozione e ricerca** Per la ricerca devo cercare un elemento con chiave k, se questo elemento c'è allora sarà associato a f(k), arrivato lì dovrò far scorrere in media un numero di elementi pari Nf(k)/2 (nel caos peggiore N/2).

A questo punto, come per gli alberi binari il costo di inserimento è banale, ricerca e rimozione invece dipendono da come è strutturata la tabella. Qui però abbiamo un grado di libertà in più che è la funzione di Hash. Il consto delle operazioni di rimozione (togli dopo che hai trovato quello che cerchi) e ricerca dipende da un oggetto che è il rapporto (nel caso migliore) N/m (numero elementi dizionario su lunghezza tabella). Questo valore si chiama $\alpha$ e viene chiamato fattore di carico della tabella (load factor). Si riesce a dimostrare che ricerca e rimozione hanno un costo che è $O(\alpha = 1)$, ovvero è un costo costante. Alpha va bene solo se la funzione di hash è una funzione buona ovvero riesce a sparpagliare bene tutti gli elementi del dizionario sulla tabella (sulla lista).

**Teorema** Il costo della ricerca di un elemento presente all'interno di una tabella di Hash è $\theta(1 + \alpha)$.

La dimostrazione di questo teorema, come degli altri nell'ambito delle tabelle di hash è di tipo probabilistico (modo uniforme vale solo se la funzione di Hash risulta essere bilanciata. Quindi la ricerca di un elemento che c'è richiede un numero di operazioni che è proporzionale a $1 + \alpha$

La condizione in cui l'elemento non c'è è la situazione peggiore perché comincio a scorrere la lista e poi devo arrivare sempre fino in fondo. Questa dimostrazione si basa un un assunto fortissimo che è quello che la tabella di hash distribuisca bene tutte le funzioni sul dizionario. è evidente quindi che la scelta della funzione di hashing deve essere fatta bene, quindi il valore atteso di quella variabile $x_i$ sia esattamente 1/n

## 4.1 Funzioni di Hash

La funzione di Hash è buona se sparpaglia bene le chiavi, uan buona funzione di hash deve fare diverse cose: deve essere ben bilanciata, in ambito crittografico deve essere impossibile risalire al valore della chiave partendo dal valore di hash

$$h(k) = \quad \text{mod } m \tag{4.1}$$

Ha come dominio l'insieme delle chiavi e come codominio m però dato un valore di input so che il valore di valore di hash è una sorta di partizione dell'insieme delle chiavi abbastanza prevedibile (so cime funziona la funzione modulo). Per quanto riguarda la creazione della tablla funziona bene però dato l'hash value posso risalire al valore della chiave quindi in in realtà non funziona benissimo.

Esistono anche funzioni più complicate, può essere definita come la parte intera inferiore ecc. (vedi slide). Dal punto di vista crittografico però la funzione di hash si basa sull'idea che deve distribuire i valori delle chiavi e far si (tutela anche la tabella di hash da eventuali regolarità nei dati). Avere una funzione di hash caotica (dati due valori di chiavi molto vicini i valori di hash sono lontani, garantisce la distruzione totale di eventuali pattern di regolarità all'interno del dizionario.

Contestualmente una funzione di hash dovrebbe riuscire ad evitare il più possibile le collisioni (sempre distribuzione uniforme nei bucket). Se m è ben proporzionato rispetto a N (quindi in realtà m=f(N)) allora il costo di inserimento ricerca e rimozione.

# Chapter 5

# Rappresentazione delle matrici

Le matrici nell'informatica sono presenti in diversi ambiti: sistemi lineari di equazioni, matrice di adiacenza di un grafo, operatori hamiltoniani. Le operazioni di prodotto che possono essere fatte con le matrici sono prodotto matrice vettore o matrice per matrice. Il prodotto di matrici si può fare con l'algoritmo di strassenn oppure con matrici di adiacenza. D'altra parte sappiamo anche che il Laplaciano di un grafo associato a un semigruppo può descrivere un cammino quantistico.

Ci concentriamo ora su un grafo che sia *non intensamente connesso*, OVVERO CON $|E| << |V|^2$, in questo caso la rappresentazione tramite matrice di adiacenza risulta inefficiente. Infatti, quando si ha un **indice di sparsità** molto piccolo

$$\frac{|a_{ij} \neq 0, i, j = 1, ..., n|}{|V|^2} << 1 \tag{5.1}$$

allora è più efficiente la rappresentazione tramite lista di adiacenza. Chiaramente il problema di una rappresentazione efficiente diventa via via più importante al crescere di $|V|$.

Se prendo un grafo di n nodi so che la matrice di adiacenza è nxn, questa potrebbe essere molto vuota o molto piena a seconda delle connettività di un grafo. Esiste un modo intelligente di memorizzare una matrice con un indice di sparsità molto basso (poche connessioni?). Questo è un problema molto importante soprattutto per grafi molto grandi (eg. un milione). Ci sono diversi modi per rappresentare matrici a bande (eg. quando ho un random walk) oppure a blocchi (Fock). In sostanza ho una matrice di cui so calcolare l'indice di sparsità (o che già so che è sparsa) e voglio un modo intelligente per memorizzarlo.

## 5.1 List Of Lists - LOL

Per questo metodo di rappresentazione delle matrici prendo ispirazione dalle liste di adiacenza della teoria dei grafi. Per ciascuna riga riporto solo gli elementi non nulli. In realtà le operazioni di algebra lineare su una rappresentazione di questo tipo sono un complicate. Ad esempio rende molto difficile la parallelizzazione del codice (che mi serve per far fare i calcoli in architettura multi-core). Questo tipo di approccio si usa soprattutto per **definire** le matrici, quando devo usarla per far

di conto si presta malissimo (perché se voglio accedere a un elemento devo scorrere tutta la lista)

## 5.2 Coordinate list- COO

Ho un vettore che memorizza i valori non nulli e due vettori che memorizzano uno coordinata x e uno coordinata y. In sostanza la rappresentazione COO viene fatta mediante una lista di triplette (i,j,v) dove i rappresenta l'indice della riga, j l'indice della colonna e v il valore presente nell'entrata della matrice.

In particolare i tre vettori sono di lunghezza $n \cdot n \cdot z$ dove $n \cdot n \cdot z$ è il numero di entrate della matrice che sono non nulle. Quello che devo fare è leggere insieme i valori contenuti nella stessa posizione all'interno dei tre vettori.

Anche questa rappresentazione in realtà non è utile per il calcolo.

## 5.3 Compressed row storage - CRS

La rappresentazione che viene data delle matrici più comunemente è la CRS i punti fondamentali di questo tipo di rappresentazione sono:

- non faccio nessuna assunzione a priori sulla struttura delle matrici
- solo gli elementi non nulli vengono memorizzati
- è molto più efficace quando si fanno operazioni di algebra lineare (nominate inizialmente)

Per questo tipo di rappresentazione di usano sempre tre vettori ma in modo diverso:

1. il vettore 1 (VAL) contiene tutti i valori non nulli della matrice. In particolare **elementi non nulli consecutivi di una riga vengono inseriti in posizioni adiacenti**
2. il vettore 2 (COL-IND) vettore di indici di colonna (stessa lunghezza di val, per ogni elemento non nullo memorizzo il suo indice di colonna)
3. il vettore 3 (ROW-PTR) registra la posizione in in VAL in cui inizia una riga

Quello che cambia radicalmente è la memorizzazione degli indici di riga. ROW-PTR, registra la posizione nel vettore di val che inizia una nuova riga (ptr sta per pointer) (quindi primo elemento posizione in cui inizia

Il vantaggio è che il row pointer (ROW-PTR) mi dice quanti elementi non nulli ci sono per una determinata riga e come devo associare le porzioni degli altri vettori. Row pointer è generalmente decisamente più corto (l'indice di riga viene dato per implicito).Questa rappresentazione rende ovviamente più facile la parallelizzazione perché il prodotto che devo fare è un prodotto riga colonna.

In particolare, la rappresentazione CRS ottimizza le performance se si usa una parallelizzazione (sia che sia multi-thread, sia che si tratti semplicemente di un sistema multi-processore). Inoltre ho ridotto il costo di memorizzazione rispetto a COO. Il costo della rappresentazione infatti si riduce a $2 \cdot n \cdot n \cdot z + n + 1$, dove n+1 rappresenta il numero di righe non interamente nulle.

Però per gestire l'accesso conflittuale alla stessa risorsa devo attendere che uno dei due abbia finito Problematiche di accesso sono risolte dal formato CRS,

Cin questo formato ho la garanzia che i dati sulla stessa riga siano vicini (perché impone il formato di storage). Il CRS consente una parallelizzazione eficace e corretta sena problemi di blocco dovuto al fatto che processori diversic erchino di accedere alla stessa porzione di memoria.

Il vantaggio è che nel secondo caso in pratica spezzo il singolo for di prima in due for e questo mi consente di sfruttare al meglio la parallelizzazione e quindi la potenza di calcolo che ho a disposizione.

Quello che ho fatto fino ad ora è stata sostanzialmente una sorta di compressione che ha portato la dimensione della rappresentazione da $2n^2$ a $mnz$+qualcosa. Ovviamente se la matrice è piena succede un casino oltre al fatto che è inutile ho una rappresentazione in spazio $2n^2$ più addirittura qualcosa, quindi questo tipo di "compressione" riduzione dello spazio di memorizzazione funziona solo per matrici sparse. Ovviamente questa rappresentazione ha senso di essere usata se quello a cui sono interessata è fare conti con quella matrice, se devo usarla una volta sola allora magari scelgo un'altra rappresentazione

# Chapter 6

# Compressione

Nel caso di matrici sparse, l'uso di formati intelligenti consente di ridurre notevolmente l'impronta della rappresentazione in memoria e il costo di operazioni algebriche elementari.

**NB** : E' chiaro che la sparsità di una matrice dipende dalla base scelta. Quella vista prima è una compressione, sotto opportune condizioni, senza perdita di informazione.

Quello che ci chiediamo ora è, ammettendo di poter tollerare della perdita di informazione, qul è la strategia ottimale per ridurre la dimensione di una matrice, quindi studieremo una **compressione con perdita**, gli obiettivi, per cercare di avvicinarsi all'ottimo sono quelli di minimizzare la perdita e massimizzare la compressione.

Ho una matrice quadrata normale (A=A*) una matrice di questo tipo è diagonalizzabile. Data una matrice normale esiste sempre una base in cui la matrice è diagonale. In linea di principio quindi posso chiedere sempre ricordare la matrice in forma diagonale. In questo caso però dovrei sempre portarmi dietro anche la base. Supponendo di poter tollerare la perdita di informazione mi chiedo quale sia la scelta ottimale per poter diminuire la dimensione di una matrice. La matrice è grossa ma ho bisogno di ridurla per qualche motivo, non è sparsa e i conti sono troppo onerosi. Quindi mi chiedo, data una matrice, qual è la strategia ottimale per comprimere la matrice, chiedendo quindi che la perdita di informazione sia minima fissata n (dimensione della matrice).

## 6.1 Singular value decomposition - SVD

**Teorema** Sia $A \in \mathbb{C}^{m\times n}$ (o equivalentemente $A \in \mathbb{R}^{m\times n}$) allora è sempre possibile riscrivere A come $A = USV^{\dagger}$ (o equivalentemente $A = USV^{T}$). Dove

- $U \in \mathbb{C}^{m\times m} : U^{-1} = U^{\dagger}$ ovvero, U è unitaria
- $V \in \mathbb{C}^{m\times m} : V^{-1} = U^{\dagger}$ ovvero, V è unitaria
- $S \in (\mathbb{R}^{+} \cup 0)^{m\times n}$ diagonale

Quindi $S = diag(\sigma_1, ..., \sigma_k, ...\sigma_q), q = m, n(n, m)$ con $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_k, k = rank(A)$ e $\sigma_j = 0, j > k$.

In pratica ho S diagonale e assumo che i valori di questa matrice, chiamati valori singolari, siano ordinati con ordine non crescente (decrescente o uguale))

Un operatore unitario è sempre una rotazione (La matrice S dilata le componenti, sto facendo una trasformazione lineare, non affine perché non è unita la traslazione) Se una matrice è normale si ottengono gli autovalori. Quella è la nostra matrice, A è riscritta come prodotto delle tre matrici. Ho delle matrici più piccole che preservano però la , il motivo per cui funziona è che la mia immagine ha delle forte correlazioni (se ci sono delle correlazioni significa che c'è della ridondanza e liberandomi della ridondanza posso avere una immagine simile o praticamente indistinguibile infatti dove il fondo è più rumoroso quindi più omogeneo si vede di più la sgranatura).

Quello che succede è che A è una mappa lineare generica, mentre la combinazione di $V^\dagger$, che è una rotazione, S - dilatazione - e U, altra rotazione, mi porta allo stesso risultato a cui mi porta A.

**Osservazione:** a differenza degli autovalori, che sono definiti solo epr matrici quadrate, i valori singolari sono definiti per qualsiasi matrice. Dalla definizione di valore singolare per una matrice quadrata A, si ha che **i valori singolari coincidono con gli autovalori *se e solo se* A è normale e semi-definita positiva**.

Ma quello che volevo io era realizzare una compressione: voglio una rappresentazione esatta o approssimata (ma se è approssimata voglio che sia il più vicino possibile allo stato di partenza). Per avere un cambio di stato si devono creare delle correlazioni a lunghissimo range. Quindi, sia

$$\tilde{A} = \tilde{U}\tilde{S}\tilde{V}^\dagger \tag{6.1}$$

dove

$$\tilde{S} = diag(\sigma_1, \sigma_2, ...\sigma_j) conj < k. \tag{6.2}$$

. Inoltre $\tilde{U}$ è una matrice $m \times j$ ovvero i primi j vettori colonna della matrice U e $\tilde{V}^\dagger$ è una matrice $j \times n$ ovvero i primi j vettori colonna della matrice V.

**Teorema** : La matrice $\tilde{A}$ costruita come sopra è la migliore approssimazione di rango $j < k$ della matrice A di rango k.

Quindi data una matrice, se ammetto la possibilità di perdere informazione la compressione ai valori singolari ammette di perdere una **piccola** quantità di informazione. La scelta di approssimazione in valori singolari funziona bene.

## 6.2 Matrix product states

Parliamo ora di una applicazione rilevante della compressione delle matrici. Consideriamo un sistema multi-corpo fisico di L entità, immagino di avere una collezione di oggetti ciascuno di dimensione d. Suppongo per semplicità che gli L oggetti siano identici, ciascuno definito su uno spazio di Hilbert di dimensione d. Prendiamo una base ortonormale locale.

**Esempio** Considero uno spazio bidimensionale, D=2, ovvero particella di spin 1/2 (considero gli autostati up e down di $sigma_z$). Prendiamo una collezione di l sistemi a d livelli e scrivo lo stato (raggio dello spazio di Hilbert). Sarà descritto da

un tensore di rango L. Assumo che un elemento della base sia il prodotto tensore di una base locale di elementi del sistema. e' chiaro che il numero di coefficienti in grado di descrivere lo stato di un sistema così scala esponenzialmente ($2^L$). Il trattamento esatto di sistemi multi-corpo è un po' complicato e soprattutto è limitato a sistemi piccoli.

Qualsiasi tensore di rango arbitrario può essere trasformato in una matrice. Dato un qualsiasi tensore posso darne una rappresentazione matriciale. Allora posso fare un reshape in particolare posso fare una SVD. Quindi dato un tensore posso trasformare prima questo tensore in una matrice e poi fare la SVD e scriverlo come prodotto di tre matrici. In sostanza, questa scrittura otteniamo che il numero posso scriverlo come prodotto di tre matrici.

Rappresentazione grafica (il rango è dato dal numero delle gambette), l'indice assume valori in un range fissato. Le somme che devo fare sono contrazioni, quando le sommo gli indici spariscono. A livello grafico quando faccio delle contrazioni tiro delle linee orizzontali che uniscono parti di grafico.

Quindi ho riscritto un tensore di rango L tramite il prodotto di 3 tensori di rango 2 Il procedimento può essere iterato. Posso ottenere il coefficiente del tensore (per ogni valore assegnato della nupla) facendo opportune operazioni di moltiplicazioni tra matrici scritte opportunamente.

Uno stato scritto in questa forma si chiama **matrix product state**, in sostanza dato uno stato posso scriverlo in forma di prodotto di matrici. Le operazioni usate sono di contrazione (somma rispetto ad un indice), per costruzione le matrici hanno sempre un indice in comune che è l'indice che le collega. Quello che mi posso chiedere è perché mi conviene fare tutte queste operazioni? Quindi posso semplificare la notazione considerando che il rango determina quante righe e colonne devo tenere (per avere comunque una dimensione esatta della matrice di partenza). Supponiamo di prendere un insieme di particelle identiche e ne faccio una partizione.

**Teorema** Dato uno stato puro di un sistema composto (da almeno due sottoinsiemi) A-B, dove A e B non hanno necessariamente la stessa dimensione, esistono sempre stati ortonormali $|i\rangle_A$ per il sottosistema A e $|i\rangle_B$ per il sottosistema B tali che

$$|\psi\rangle = \sum_i \lambda_i |i\rangle_A |i\rangle_B \tag{6.3}$$

con

$$\lambda_i \geq 0, \sum_i \lambda_i^2 = 1 \tag{6.4}$$

Quindi la dimensione dei sottospazi A e B è la stessa e i coefficienti sono reali (maggiore o uguali a zero e la loro norma è pari a 1. Questo tipo di decomposizione si chiama **decomposizione di Schmidt**, di conseguenza poi ho le basi di Schmidt, ho i coefficienti di Schmidt e il rango di Schmidt (mi dice quali sono gli autovalori non nulli).

La decomposizione di Schmidt è interessante perché supponendo di prendere uno stato statistico di cui costruisco una matrice densità.

Compressione: adesso voglio in qualche modo troncare lo stato. L'obiettivo è quello di ridurre la dimensione del sistema perdendo la minor quantità di informazione possibile. Naturalmente elimino i singoli valori più piccoli. Per ridurre la dimensionalità dello stato senza perdere troppe cose è eliminare i $\lambda_j$ più piccoli. La tessa tecnica può essere utilizzata per fornire una approssimazione di rango k di uno stato. Questo è legato alla *decomposizione di Schmidt che è espressione del*

*numero di componenti quantistiche in un sistema.* Lo stato si può scrivere come prodotto tensore di uno stato e di uno stato locale. In pratica quello che succede è che perdo il sottospazio meno rilevante.

Perdendo informazione perdo anche norma. è anche vero però che quella norma che sto perdendo è la quantità di norma minima che posso perdere per dato livello di compressione. Posso fare questo tipo di compressione su tutti i bond, se anche faccio il troncamento su tutti i bond quello che ottengo alla fine è comunque la miglior approssimazione di rango k del mio stato.

Massima correlazione tra ... di un sistema quantistico è quando ho un cambiamento di fase. Durante il cambio di fase si crea una correlazione a lunghissimo raggio. Lontano dai punti critici delle transizioni di fase, si può dimostrare che gli stati fisici tipici (che vengono effettivamente occupati dal sistema) sono stati in cui l'andamento del valore singolare decresce più velocemente di $1/x$ (nella maggior parte dei casi il decay è esponenziale), al punto critico invece va come 1/x.

In sostanza, complessità notazionale a parte, si nota che:

- è possibile ottenere in modo banale la decomposizione di Schmidt a partire dalla forma MPS di uno stato
- è possibile operare una compressione dello stato fissando il numero di valori singolari da tenere su ogni bond (ovvero il numero di $\lambda_{ij}$ che collegano ciascuna bipartizione.

Sistemi quantistici lontano dai punti critici soddisfano la cosiddetta area low. Quello che si fa tipicamente con questa rappresentazione di questi oggetti è fissare la bond-dimension. Per tutti i bon fisso la bond-dimension a k. Se la fisso a k, per ogni sistema ho matrici indicizzate da i (con ild eice di i va da 1 a j.

Quindi il numero di indici che girano in questa rappresentazione in cui ho fatto un troncamento a x scala come $x^2$ d l (molto meno di $d^l$ per l suff. grande).

Determinazione di uno stato fondamentale