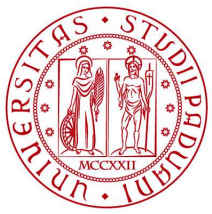

Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Dipartimento di ingegneria dell'informazione

Corso di laurea in ingegneria dell'informazione

Algoritmi esatti e approssimati per lo Hierarchical Clustering

Relatori

Prof. Andrea Alberto Pietracaprina

Prof. Geppino Pucci

Laureando

Riccardo Zattra

Anno Accademico

2022/2023

29 settembre 2023

Abstract

Lo scopo di questa tesi è cercare di colmare, almeno in parte, la poca organizzazione del materiale legato allo Hierarchical Clustering che tutti noi possiamo reperire liberamente in rete. Nel fare ciò, sarà data una panoramica generale sullo Hierarchical Clustering che porterà successivamente ad individuare la nicchia sulla quale questo lavoro vuol far chiarezza. Verranno quindi presentate le tre linkage-function più usate e i relativi algoritmi con le relative implementazioni per risolvere il problema dello Hierarchical Clsutering. In seguito a questo verrà esaminato e studiato un recente lavoro presentato al NIPS 2019 riguardante l'introduzione di approssimazioni nel risolvere il problema di hierarchical clustering al fine di migliorare la complessità temporale degli algoritmi.

Indice

1 Introduzione

Come riportato nel seguente libro [1] il clustering è un insieme di tecniche per l'analisi dei dati. Tutte queste tecniche si basano su misure della similarità/dissimilarità. In molte di queste le misure di similarità/dissimilarità rappresentano la distanza tra una coppia di punti in uno spazio multidimensionale. Il risultato, la bontà di quest'ultimo e quindi anche la validità, dipende dalla scelta della metrica, cioè da come viene calcolata la distanza. Gli algoritmi di clustering raggruppano oggetti in base alla loro distanza e questa determina quindi l'appartenenza o meno ad un dato insieme. I metodi di clustering si basano principalmente su due approcci:

1. Metodi aggregativi (bottom-up): In questo caso tutti gli elementi vengono considerati inizialmente come singoli cluster e poi, ad ogni iterazione dell'algoritmo, vengono uniti i due cluster più vicini. Il numero di unioni può essere fissato a priori ed andrà a determinare il numero massimo di iterazioni dell'algoritmo oppure si continuerà fino a che non si otterrà un unico cluster.

2. Metodi divisivi (top-down): All'inizio tutti gli elementi formano un unico cluster che verrà diviso in cluster più piccoli dall'algoritmo. Una divisione viene effettuata quando provoca un aumento della similarità tra gli oggetti che appartengono ai due cluster nati dalla divisione. Si continua fino a che non si arriva ad ottenere un numero prefissato di cluster.

Gli algoritmi vengono classificati, inoltre, in base alla possibilità di avere o meno un elemento appartenente a più cluster, pertanto si dividono in algoritmi per:

1. Clustering esclusivo: ogni elemento può essere assegnato ad uno e un solo cluster. Quindi l'intersezione tra tutti i cluster (che matematicamente possono essere visti come degli insiemi) deve essere vuota. Questo viene anche detto hard clustering.

2. Clustering non esclusivo: Un elemento può appartenere a più cluster diversi con gradi di appartenenza diversi. Questa tecnica viene anche detta soft clustering o fuzzy clustering.

Si tiene conto inoltre del tipo di algoritmo utilizzato per dividere lo spazio (altra suddivisione degli algoritmi):

1. Clustering center-based (detto anche non gerarchico o k-clustering), in cui per definire l'appartenenza ad un gruppo o meno viene utilizzata una distanza da un punto rappresentativo (es. centroide, mediode). Tutto questo viene fatto fissando il numero di gruppi al quale fermare l'esecuzione dell'algoritmo. Gli algoritmi che appartengono a questa categoria sono tutti derivati dal noto algoritmo di clustering k-means.

2. Clustering gerarchico (Hierarchical Clustering): con questo metodo gli algoritmi costruiscono una gerarchia di cluster visualizzabili graficamente tramite una rappresentazione ad albero detta dendrogramma. Con questa rappresentazione grafica si possono visualizzare i passi di accoppiamento e divisione dei cluster.

Il tema della tesi è lo Hierachical Clustering agglomerativo (o aggregativo). Il significato di queste parole è stato spiegato qui sopra. Gli algoritmi che verranno visionati e studiati sono algoritmi che sfruttano un approccio aggregativo (come si vede dal nome) di tipo non esclusivo. Come già ripetuto la bontà del clustering dipende dalla scelta della metrica (che specifica come calcolare effettivamente la distanza) e dalla funzione, che a seguito del calcolo delle distanze, seleziona i due cluster da fondere. Queste funzioni vengono anche dette linkage-function. Nel seguito si indicheranno due cluster generici con C_i, C_j e si indicherà il centroide di un cluster con $\mu(C_i)$ inoltre verrà indicato un clustering con la seguente notazione $C = \{C_1, ..., C_l\}$ (con i pedici che possono cambiare). Nello Hierarchical Clustering si usano prevalentemente le seguenti funzioni di similarità:

1. Single-link (Single Linkage):

$$D(C_i, C_j) = \min_{\substack{x \in C_i \\ y \in C_j}} d(x, y)$$

dove la distanza tra cluster viene calcolata come la minima tra elementi appartenenti ai due cluster presi in considerazione.

2. Average-link (Average Linkage):

$$D(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x \in C_i} \sum_{y \in C_j} d(x, y)$$

in questo caso la distanza tra due cluster è la media delle distanze dei singoli punti.

3. Complete-link (Complete Linkage):

$$D(C_i, C_j) = \max_{\substack{x \in C_i \\ y \in C_j}} d(x, y)$$

dove la distanza tra cluster viene calcolata come la massima tra elementi appartenenti ai due cluster presi in considerazione.

4. Ward's method:
Il Ward's method è un metodo generale per la risoluzione di un problema in maniera ottimale. Nel nostro caso, cioè nello Hierarchical Clustering, il Ward's method viene anche detto Ward's minimum variance method. Questo perchè Ward's suggerisce di scegliere la coppia di cluster da unire come quella coppia che provocherà un aumento minimo della varianza dei punti all'interno del cluster. In formule:

$$D(C_i, C_j) = \sum_{x \in C_i, y \in C_j} \frac{d^2(x, y)}{|C_i| + |C_j|} - \sum_{x, y \in C_i} \frac{d^2(x, y)}{|C_i|} - \sum_{x, y \in C_j} \frac{d^2(x, y)}{|C_j|}$$

Questa formula se espressa mediante centro del cluster e cardinalità è uguale a :

$$D(C_i, C_j) = \frac{d^2(\mu(C_i), \mu(C_j))}{\frac{1}{n_a} + \frac{1}{n_b}}$$

dove il centro di un cluster si calcola come segue:

$$\mu(C_i) = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

In tutti i punti qui sopra $d(x, y)$ rappresenta una distanza qualunque tra punti in uno spazio metrico.

In precedenza si è parlato del dendrogramma per la rappresentazione grafica dei risultati. Quest'ultimo può essere pensato come un albero in cui ogni nodo interno rappresenta una fusione effettuata dall'algoritmo e le cui foglie corrispondono a tutti i singoli punti dati in input. Il dendrogramma può essere disegnato su un piano bidimensionale. Usiamo l'asse x per rappresentare tutti i punti sui quali vogliamo eseguire il clustering e usiamo invece l'asse y come una normale retta reale per plottare (con delle linee orizzontali) le distanze alle quali sono avvenute le fusioni tra cluster. Queste linee orizzontali vengono dette "combination similarity". Se le combination similarity delle fusioni sono in ordine crescente (dal basso verso l'alto) allora si parla di metodo monotono.
Di seguito viene riportata un'immagine per chiarire quanto appena detto.

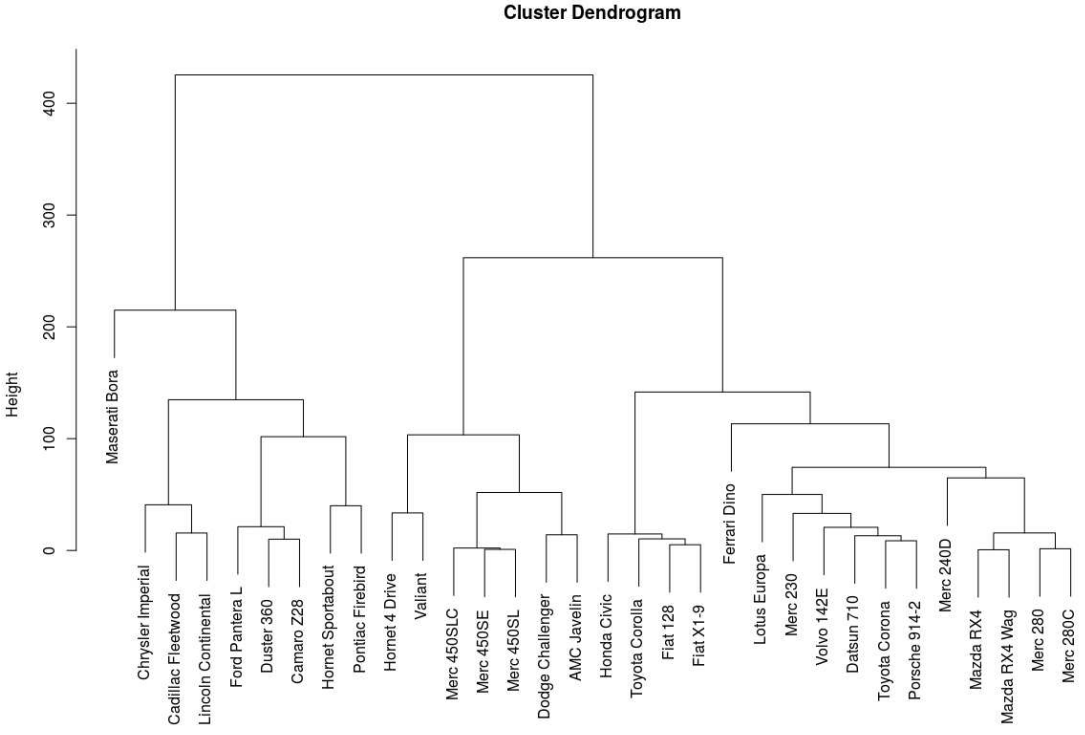

Figura 1.1: Esempio di dendrogramma

2 Single linkage Hierarchical Clustering

Per quanto concerne il Single-Linkage, si parte da una situazione iniziale in cui tutti i dati (che rappresentiamo matematicamente come punti di $\mathbb{R}^d$) costituiscono singolarmente un cluster. Ad ogni iterazione dell'algoritmo vengono uniti i due cluster più simili secondo la Single-Linkage function. La scelta della metrica è specificata a priori. L'algoritmo continua l'esecuzione fino a che tutti i punti non appartengono ad unico cluster.
Nel seguito del capitolo verrà presentato l'algoritmo naive per lo Hierarchical Clustering Single-Linkage. Verrà presentato lo pseudocodice e verranno discussi i metodi utilizzati. Successivamente verrà effettuata l'analisi della complessità temporale e spaziale di questo algoritmo. A seguito di questo si passerà allo studio degli algoritmi di Prim e Kruskal prima del quale verrà data anche una panoramica sulle strutture union-find (utili per questi algoritmi). Successivamente verrà presentato un algoritmo che risolve il problema dello Hierarchical Clustering Single-Linkage basato sugli algoritmi di Prim e Kruskal. Verrà poi analizzata la complessità temporale e spaziale dell'algoritmo. Per finire verrà presentato un algoritmo per la costruzione di un dendrogramma con Kruskal.

2.1 Naive-algorithm

[2] Il seguente algoritmo risolve il problema di clustering agglomerativo usando una matrice detta matrice di prossimità che indichiamo con D. Si tratta di una matrice quadrata nxn con n che rappresenta il numero di punti dello spazio $\mathbb{R}^d$. La matrice contiene al suo interno tutte le distanze tra tutte le possibili coppie di cluster presenti. Descritta quindi la costruzione si deduce che la matrice sarà necessariamente simmetrica (come conseguenza della proprietà di simmetria delle distanze) e avrà

le entrate sulla diagonale tutte nulle come conseguenza del fatto che rappresentano la distanza tra un cluster e se stesso. Ai raggruppamenti viene assegnato un indice $m \in [0, n-1]$ che indicherà il "livello" al quale è avvenuta la fusione tra due cluster.

Vediamo ora gli step che l'algoritmo esegue per risolvere una generica istanza:

1. Si considerino tutti i punti come singoli cluster aventi tutti livello $L(0) = 0$ e indice $m = 0$
2. Si trovi la coppia di cluster più vicina, indichiamola con C_i e C_j che avrà distanza
$$d(C_i, C_j) = \min_{\substack{x \in C_i \\ y \in C_j}} d(x, y)$$
3. Si incrementi l'indice che tiene traccia del livello, quindi $m = m + 1$. Si uniscano in un unico cluster i cluster C_i e C_j e si assegni il livello al quale è avvenuta la fusione con $L(m) = d(C_i, C_j)$
4. Si aggiorni la matrice di prossimità 'D' cancellando righe e colonne corrispondenti ai cluster C_i e C_j e aggiungendo una riga e una colonna relativa al nuovo cluster ottenuto dalla fusione dei due precedenti che indichiamo con (r, s). La distanza tra il nuovo cluster (C_i, C_j) e i vecchi cluster C_k è definita come: $d((C_i, C_j), C_k) = min(d(k, C_i), d(k, C_j))$
5. Si torni al secondo punto fintanto che i punti non appartengono ad un unico cluster.

Si veda lo pseudocodice nella pagina seguente.

2.1.1 Descrizione dei metodi

Di seguito viene data una descrizione dei metodi e delle variabili utilizzate nell'algoritmo sotto presentato:

1. Evaluate_proximity_matrix():
   Metodo che costruisce la matrice di prossimità sopra descritta considerando ogni punto come un singolo cluster.

Algorithm 1 Naive algorithm

Input
$X \subset \mathbb{R}^d$, dataset di n punti
Output: Dendrogramma del clustering

```
1: D = evaluate_proximity_matrix()
2: for all x ∈ X do make_node(x)
3: repeat
4:     (i,j) = find_most_similar_cluster(D)
5:     n= make_and_set_node(i, j, D(i,j)/2);
6:     update_proximity_matrix();
7: until D.dimension() > 1
8: return n
```

2. Il ciclo for all crea i nodi foglia del dendrogramma che per costruzione sono tutti i punti $x \in X$.

3. find_most_similar_cluster():
   Metodo che cerca i due cluster più vicini all'interno della matrice di prossimità.

4. make_and_set_node():
   Metodo che crea il nodo del dendrogramma.

5. update_proximity_matrix():
   Metodo che aggiorna la matrice di prossimità a seguito della fusione di due cluster.

Si noti che il livello di fusione tra due cluster è implicito nella costruzione del dendrogramma, per questo non è importante tenere traccia del valore di m.

2.1.2 Analisi della complessità temporale

In questa sottosezione viene analizzata la complessità temporale dell'algoritmo naive sopra presentato. Verrà esposto, ove necessario a chiarire i calcoli, un contributo alla complessità temporale per iterazione e il contributo totale.

1. evaluate_proximity_matrix():
   Il contributo totale alla complessità temporale per questo metodo, sfruttando la simmetria della matrice di prossimità è:

$$\sum_{i=1}^{n-1} ci = c \sum_{i=1}^{n-1} i = \frac{cn(n-1)}{2} \implies \Theta(n^2)$$

2. Ciclo for:
   Vengono eseguite n iterazioni ciascusa in tempo costante $\implies \Theta(n)$

3. find_most_similar_cluster():
   j =dimensione della matrice di prossimità
   i =indice di riga

   - Per iterazione:

   $$\sum_{i=0}^{j-1} j - 1 - i = \frac{j(j-1)}{2}$$

   - Totale:

   $$\sum_{j=2}^{n} \frac{j(j-1)}{2} \implies \Theta(n^3)$$

4. make_and_set_node():
   Questo metodo ha il compito di creare il nuovo nodo del dendrogramma che rappresenta il nuovo cluster (C_i, C_j). Per fare questo si deve impostare il campo «parent» dei nodi i e j e impostare i campi «left» e «right» ,del nodo appena creato, con un puntatore ad i e j. Per fare questo si necessita di un tempo $\Theta(1)$. Dato che vengono eseguite n iterazioni si ha quindi un contributo totale pari a $\Theta(n)$

5. update_proximity_matrix():
   Si può implementare una gestione della tabella che permetta di aggiornarla ,ad ogni iterazione, in tempo $\Theta(n)$ e quidi dato che questo metodo viene eseguito n volte si ha un contributo totale alla complessità pari a $\Theta(n^2)$

Dalla seguente analisi si evince che l'algoritmo ha una complessità temporale pari a $\Theta(n^3)$ che si ottiene dalla somma dei contributi totali calcolati nell'analisi dei vari metodi utilizzati.

2.1.3 Analisi della complessità spaziale

I due elementi che concorrono ad aumentare la complessità spaziale sono il dendrogramma e la matrice di prossimità.Il dendrogramma contribuisce con una complessità spaziale pari a $\Theta(n)$ mentre la matrice di prossimità contribuisce con un $\Theta(n^2)$ questo implica che la complessità spaziale totale è $\Theta(n^2)$.

2.1.4 Conclusione

Per concludere l'analisi di questo primo algoritmo possiamo affermare che è relativamente semplice ma presenta una complessità temporale e spaziale troppo alta che lo rende poco utile quando la taglia del dataset è grande.

Un algoritmo che risolve lo stesso problema computazionale, ma con complessità ridotta, si basa sull'equivalenza tra l'algoritmo di Kruskal e l'algoritmo naive sopra analizzato.

2.2 Prim and Kruskal Algorithm

Come preannunciato esiste un algoritmo che permette di risolvere il problema di Hierarchical Clustering Single-Linkage in tempo e spazio entrambi ottimali. Prima di vedere questo algoritmo è necessario avere a mente i concetti e le proprietà fondamentali relative ai grafi. Inoltre l'algoritmo di Kruskal fa uso di una struttura dati particolare, la cosiddetta struttura union-find, la cui comprensione costituisce un elemento necessario per poter comprendere l'algoritmo.

2.2.1 Una panoramica sulle strutture union-find

[3] Le strutture union-find sono utili ogni qualvolta si necessita di gestire una partizione di elementi con una collezione di insiemi disgiunti. Nella precedente affermazione è stato usato il termine "partizione" che va inteso con il relativo significato matematico. Ogni elemento può appartenere ad uno ed un solo insieme. A differenza dell'ADT Insieme non ci aspettiamo di essere in grado di iterare tra gli elementi di una collezione o di capire se un dato insieme contiene un dato elemento. Per evitare confusione tra gli insiemi disgiunti qui usati e l'ADT Insieme, chiameremo i nostri insiemi clusters. Per distinguere tra loro i clusters si fa uso del rappresentante di cluster che è un elemento qualsiasi del cluster stesso. I metodi a disposizione per interfacciarsi con questa struttura dati sono i seguenti:

1. makeCluster(x):
   Crea un cluster contenente il singolo elemento x e ritorna il suo oggetto «position».

2. union(p,q):
   Unisce i due cluster contenenti gli oggetti «position» p e q.

3. find(p):
   Ritorna l'oggetto «position» del leader del cluster a cui appartiene l'oggetto p passato come argomento.

N.B.: Un oggetto «position» è un oggetto che non contiene l'elemento ma bensì un riferimento in memoria del vero elemento. A volte può contenere anche il valore e riferimenti a successore e predecessore ,per esempio, in una lista concatenata. In generale può essere pensato come un nodo che contiene svariate informazioni. Il tipo di informazioni contenute in un nodo «position» vengono descritte di volta in volta in base al tipo di struttura dati che ne fa uso.

Vediamo adesso alcune possibili implementazioni di questa struttura dati:

1. Sequence implementation:
   Un'implementazione semplice di una struttura union-find con n elementi fa uso di una collezione di sequenze ,una per ogni cluster, dove la sequenza per il cluster A contiene solo gli elementi relativi al cluster A. Ogni elemento della sequenza è un oggetto «position» (come sopra descritto) composto di soli due riferimenti. Il primo si riferisce al vero e proprio elemento e il secondo è un riferimento all'oggetto «position» del proprio rappresentante. Il secondo riferimento del rappresentante punta a se stesso. Con questo di tipo di organizzazione i metodi makeCluster(x) e find(p) hanno un costo computazionale pari a $\Theta(1)$ perché si deve creare un oggetto position e impostare i due riferimenti o si deve accedere al riferimento rispettivamente. L'operazione di unione dei due cluster invece deve unire due sequenze e aggiornare i riferimenti di una delle due per farli puntare al rappresentante dell'altra sequenza. Si sceglie di spostare gli elementi della sequenza più piccola in quella più grande. Questo implica che la complessità del metodo union è $O(\min(n_p, n_q))$ con n_p, n_q la cardinalità dei due insiemi che si stanno unendo. Al caso pessimo questa complessità sale ad $O(\frac{n}{2}) = O(n)$ nel momento in cui i due insiemi contengono metà degli elementi ciascuno.

   Proposizione 2.1. *Effettuare k operazioni su struttura union-find con il precedente tipo di implementazione, su partizione inizialmente vuota e coinvolgendo n elementi, richiede un tempo computazionale*

$O(k + n \log n)$.

2. Tree-based implementation:
   Un'alternativa alla sopra discussa implementazione consiste nell'usare una collezione di alberi per salvare gli n elementi, dove ogni albero corrisponde ad un singolo cluster. Ogni albero viene implementato come un linked-tree i cui nodi sono oggetti di tipo «position» che contengono un riferimento all'elemento salvato e uno al nodo «parent». Per quanto riguarda la radice il riferimento riservato per il nodo «parent» punta a se stesso. Con questo tipo di implementazione il metodo find(p) si ottiene risalendo di padre in padre partendo dal nodo passato come argomento. Questo richiede un tempo computazionale al caso pessimo pari a $O(n)$ (al massimo un nodo può avere $n-1$ antenati). Il metodo makeCluster(x), ancora una volta, richiede un tempo computazionale $O(1)$ e il metodo union(p,q) può essere implementato rendendo uno dei due alberi sottoalbero dell'altro. Quindi si procede a cercare entrambe le radici dei due alberi da unire e poi si imposta il riferimento «parent» di una radice, facendo sì che punti alla restante radice. Per rendere più veloce questa implementazione aggiungiamo le seguenti due operazioni da eseguire in aggiunta al comportamento dei metodi da specifica.

   In ogni nodo dell'albero si riserva un campo per memorizzare la dimensione del sottoalbero radicato in quel nodo e ad ogni invocazione del metodo union(p,q), tramite questo valore, si fa in modo di rendere l'albero più piccolo sottoalbero di quello più grande, aggiornando questo valore per la radice del nuovo albero. Inoltre per ogni invocazione del metodo find(p) ,per ogni posizione q che viene visitata durante il percorso di risalita verso la radice, si cambia il suo puntatore «parent» facendolo puntare alla radice. Questo diminuisce l'altezza dell'albero e pertanto riduce la complessità del metodo find(p).Così facendo si riduce anche la complessità del metodo union(p,q) che inizialmente deve cercare le radici dei due alberi a cui appartengono i nodi p e q. Il metodo find(p) se implementato ricorsivamente ha un albero della ricorsione con un numero di nodi proporzionale alla profondità di p ed essendo l'altezza di un albero la massima tra tutte le profondità di tutte le foglie, se questa diminuisce , diminuisce anche la massima profondità e pertanto la complessità al caso pessimo di questo algoritmo.

Proposizione 2.2. *Con l'implementazione appena discussa, usando anche le*

due operazioni euristiche sopra citate, eseguendo una serie di k operazioni coinvolgendo n elementi si ha un costo computazionale $O(k \log^* n)$.

N.B: $\log^* n$ è il logaritmo iterato di n ovvero il numero di volte che la funzione logaritmo deve essere applicata iterativamente al valore n prima di ottenere un numero minore o al più uguale ad uno.

2.2.2 Proprietà degli MST

Addentriamoci ora nello studio dei due algoritmi per la ricerca di un MST (minimum spanning tree). Entrambi gli algoritmi sfruttano il paradigma Greedy per cercare un MST e il fatto che questo sia sufficiente a trovare un MST è giustificato dalla seguente proposizione. Il seguente lemma è d'aiuto per comprendere meglio la proposizione.

Lemma 2.1. *Dato un albero l'aggiunta di un arco comporta necessariamente la creazione di un ciclo.*

Dimostrazione. Per definizione un albero è una struttura gerarchica che si dipana a partire da un nodo r ,detto radice, tramite delle relazioni padre-figlio. Per ogni altro nodo ,che non sia la radice, è unico suo padre e partendo da quel nodo e andando di padre in padre, si giunge sempre alla radice. Questo implica una struttura connessa, in quanto, per ogni coppia di nodi V_1 e V_2 scelti, considerando gli archi che costituiscono il cammino di risalita alla radice per V_1 e V_2, che indico con P_1 e P_2 rispettivamente, e considerando P_2 (potrebbe essere anche P_1) percorso in senso opposto (dalla radice al nodo V_2) e unendolo con P_1, la loro unione costituisce un cammino tra V_1 e V_2. Dato che questo lo si può fare per ogni coppia di nodi scelti, questo implica che l'albero è connesso per definizione. Detto questo, se aggiungiamo un arco, considerando l'unione del cammino già esistente tra i due nodi e l'arco appena aggiunto tra quest'ultimi, abbiamo creato un ciclo perché possiamo esibire un cammino che comincia e finisce nello stesso nodo. □

Proposizione 2.3. *Sia G un grafo connesso e pesato e siano V_1 e V_2 due insiemi, non vuoti, che costituiscono una partizione dei nodi di G. Chiamiamo e l'arco con peso minimo tra tutti gli archi che hanno un nodo in V_1 e l'altro in V_2 allora esiste un MST che ha e come uno dei suoi archi.*

Dimostrazione. Sia T un MST del grafo G. Supponiamo che T non contenga e e al suo posto ci sia un altro arco h come ramo di T. Essendo MST in particolare è

un albero, per il lemma precedente, aggiungendo un arco ad un albero si crea un ciclo. Detto questo, se a T aggiungiamo e creiamo un ciclo che possiamo eliminare togliendo h. L'albero così formato è un albero (perché è ancora connesso è senza cicli) e visto che $w(e) \leq w(h)$ avrà un peso minore o uguale rispetto al precedente albero che quindi sarà un MST a maggior ragione. Questo implica necessariamente che esiste un MST che contiene e. □

Passiamo ora, tenendo bene a mente il fatto appena esposto, alla descrizione del primo algoritmo per la ricerca di un MST in un grafo G.

2.2.3 Prim-Jarnik algorithm

Questo algoritmo crea un MST a partire da un nodo s qualsiasi del grafo G, che sarà la radice dell'MST. S inizialmente costituisce la nuvola di punti (cloud) C che ad ogni iterazione verrà allargata inserendoci il nodo che è in collegamento all'insieme C con l'arco di peso minimo. Si ripete questo processo fintanto che esiste almeno un nodo del grafo non ancora inserito nell'insieme di vertici C. Osserviamo che questa procedura porta alla risoluzione del problema in quanto è una pura applicazione della proposizione precedentemente dimostrata 2.3. Possiamo infatti osservare che $V_1 = C$ e $V_2 = \overline{C}$ (C complementare) e ad ogni iterazione aggiungiamo l'arco di peso minimo che connette V_1 con V_2, sicuri del fatto che questo andrà a costruire un MST del grafo G ,in quanto sicuramente (per la proposizione precedente) esiste un MST che contiene l'arco inserito.

N.B. consideriamo che ogni vertice abbia tre label, la prima v.D che salva il valore dell'arco di peso minimo, fino a quel momento incontrato, che connette il nodo con l'insiemi di vertici C. La seconda v.arch che contiene un riferimento all'arco che realizza la connessione con peso v.D. La terza v.Q che contiene un true o false per capire se v è in Q (coda con priorità, vedi pseudocodice) o meno.

Qui di seguito viene presentato lo pseudocodice e una tabella che riporta la complessità temporale dell'algoritmo di Prim-Jarnik considerando la coda con priorità implementata tramite heap e lista non ordinata:

Prim-Jarnik	Heap	Lista non ordinata
Complessità temporale	$O(m \log n)$	$O(n^2)$

Algorithm 2 Prim-Jarnik algorithm

Input
G grafo connesso, pesato, non diretto e semplice con n vertici e m archi
Output: MST (minimum spanning tree) di G

```
1: s.D=0
2: for all v ≠ s do
3: v.D = +∞
4: Inizializza l'MST T = ∅
5: Inizializza coda con priorità Q con entry (v.D, v)∀v ∈ G
6: repeat
7:     u = Q.removeMin()
8:     unisci il vertice u a T usando la label u.arch
9:     u.Q = false
10:    for all edges e' = (u, v) t.c. v è ancora in Q do
11:        if w(u, v) < v.D then
12:            v.D = w(u, v)
13:            v.arch = e'
14:            aggiorna Q
15: until Q.is_not_empty()
16: return T
```

La complessità spaziale risulta essere lineare nella taglia del grafo, quindi $O(n+m)$. La correttezza di questo algoritmo può essere provata facilmente osservando che è una semplice applicazione della proposizione 2.3. Per ulteriori informazioni [3]

2.2.4 Kruskal algorithm

Vediamo ora un altro algoritmo per la costruzione di un MST di un grafo G. A differenza del precedente algoritmo, che lo costruiva a partire da un singolo albero fino alla copertura totale del grafo, quest'ultimo mantiene molti piccoli alberi in una foresta,che ricordiamo essere un sottografo di copertura senza cicli,unendo coppie di alberi ripetutamente finché non si ottiene un unico albero che copre l'intero grafo. Inizialmente ogni nodo costituisce un albero. L'algoritmo considera ogni arco, uno alla volta, in ordine crescente di peso. Se un arco e connette due diversi alberi, questo viene aggiunto all'MST e i due alberi vengono uniti con l'ausilio di e. Se invece e connette due nodi nello stesso albero, allora l'arco viene scartato, in quanto andrebbe sicuramente a produrre un ciclo togliendo immediatamente la possibilità di ottenere alla fine un albero. Dopo aver aggiunto archi a sufficienza per formare

uno spanning tree l'algoritmo lo restituisce come output e quindi come minimum spanning tree. Pseudocodice:

Algorithm 3 Kruskal algorithm

Input
Grafo G connesso pesato e semplice con n vertici ed m archi
Output: MST (minimum spanning tree) di G

1: **for all** $v \in G$ **do**
2: C(v)=makeCluster(v)
3: Inizializza l'MST $T = \emptyset$
4: Inizializza coda con priorità Q che contenga tutti gli archi di G usando il peso dell'arco come chiave
5: **repeat**
6: (u,v) = Q.removeMin()
7: C(u) = find(u)
8: C(v) = find(v)
9: **if** $C(u) \neq C(v)$ **then**
10: inserisci (u, v) in T
11: union$(C(u), C(v))$
12: **until** $|T| < n - 1$
13: **return** T

La seguente tabella riporta la complessità temporale dell'algoritmo di Kruskal considerando la coda con priorità implementata tramite heap e lista non ordinata:

Kruskal	Heap	Lista non ordinata
Complessità temporale	$O(m \log n)$	$O(m^2)$

La complessità spaziale risulta essere lineare nella taglia del grafo, quindi $O(n+m)$. La correttezza di questo algoritmo può essere provata facilmente osservando che è una semplice applicazione della proposizione 2.3. Per approfondimenti [3]

2.3 Algoritmo di Prim e Kruskal per Single-Linkage HC

Come preannunciato il secondo algoritmo per la soluzione di un problema di Hierarchical Clustering Single-Linkage sfrutta entrambi gli algoritmi sopra presentati. Prima di vedere come usarli per la risoluzione di questo problema computazionale, osserviamo alcuni fatti importanti.

2.3.1 Analogia tra l'algoritmo Naive e Kruskal

Ricordando l'algoritmo naive, verranno elencate tutte le analogie tra quest'ultimo e l'algoritmo di Kruskal. Proviamo quindi formalmente che l'esecuzione dell'algoritmo di Kruskal, su un grafo appositamente costruito, risolve il problema di clustering. Consideriamo un grafo costituito da n vertici che sono i punti dello spazio $\mathbb{R}^d$ sui quali si vuole risolvere il problema di clustering. Consideriamo gli archi del grafo come archi con un peso (dando luogo quindi ad un grafo pesato) dove il peso di un arco rappresenta la distanza tra i due vertici che costituiscono l'arco. Detto questo si può calcolare il numero di archi m, $m = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} < n^2$, questa formula può essere ricavata pensando di numerare i vertici da 1 ad n e, ricordando la proprietà di simmetria delle metriche, si evince che il primo vertice ha $n-1$ archi che rappresentano le $n-1$ distanze da tutti gli altri punti, il secondo vertice ha $n-2$ archi che rappresentano le $n-2$ distanze da tutti gli altri punti escluso il primo e così via. Analogie:

1. Un albero libero nell'algoritmo di Kruskal corrisponde ad un cluster.
2. Gli archi del grafo sono in corrispondenza biunivoca con le entrate della matrice triangolare superiore (o inferiore), escludendo la diagonale, ricavata dalla matrice di prossimità di cui si è parlato nella descrizione dell'algoritmo naive (questo solo nella prima iterazione).
3. Inizialmente nell'algoritmo naive ogni punto dello spazio è considerato come un singolo cluster, mentre in Kruskal è un albero, ma sfruttando l'analogia del primo punto deduciamo che anche in Kruskal ogni punto è considerato come singolo cluster.

Proposizione 2.4. *L'algoritmo di Kruskal risolve il problema di Hierarchical Clustering Single-Linkage agglomerativo e lo risolve con gli stessi archi con cui costruisce l'MST del grafo.*

Dimostrazione. Per dimostrare la proposizione dimostriamo l'invariante seguente la cui prima parte costituisce esattamente l'invariante dell'algoritmo naive. $Inv(i) =$ ad ogni iterazione i-esima vengono uniti i due cluster più vicini e l'arco usato per unire i due cluster viene inserito in T che è l'MST. N.B considero $i =$ indice dell'iterazione in cui effettivamente viene inserito un arco in T quindi mentre il numero totale di iterazioni varia tra 1 e m l'indice i assume valori nell'intervallo $[1, n-1]$. Non consideriamo le altre iterazioni in quanto scartano solamente gli archi che uniscono vertici appartenenti allo stesso cluster. L'invariante è banalmente vera per ogni ite-

razione i come conseguenza della costruzione dell'algoritmo di Kruskal, sfruttando il punto primo delle analogie che ricorda che un albero libero in Kruskal è l'equivalente di un cluster. Infatti per ogni iterazione i vengono uniti i due cluster distinti più vicini in quanto si preleva dalla coda con priorità l'arco con chiave minima. Essendo la chiave della entry coincidente con il peso dell'arco, il quale coincide con la distanza tra i cluster, deduciamo che ad ogni iterazione vengono uniti i due cluster più vicini e l'arco usato per l'unione (osservando il codice) viene inserito in T. Questo dimostra che ad ogni iterazione vengono uniti i due cluster più vicini e che ogni arco usato per costruire il clustering viene anche usato per costruire l'MST. Essendo vera l'invariante per ogni iterazione in particolare sarà vera per l'iterazione finale dell'algoritmo e quindi verranno uniti gli ultimi due cluster rimanenti dando luogo ad unico cluster contenente i vertici del grafo G. □

Oltre a questo, tenendo traccia delle unioni fatte per poter costruire il dendrogramma, si ha che l'esecuzione dell'algoritmo di Kruskal porta alla soluzione del problema di clustering e in particolare si deduce che gli archi necessari alla costruzione dell'MST sono sufficienti anche ad effettuare il clustering dei punti. Sorge a questo punto una domanda: perché non possiamo usare direttamente questo algoritmo (senza Prim) per eseguire il clustering? D'altronde abbiamo appena provato che il solo algoritmo di Kruskal è sufficiente per risolvere il problema di clustering. Ricordiamo che l'algoritmo naive ha una complessità temporale $O(n^3)$ e una complessità spaziale $O(n^2)$ mentre l'algoritmo di Kruskal presenta una complessità temporale $O(m \log n) = O(n^2 \log n)$ e una complessità spaziale $O(n^2)$, ricordando che il numero di archi è $\Theta(n^2)$. Quindi usando Kruskal, implementando la coda con priorità tramite heap, si ottiene un algoritmo nel complesso più performante rispetto a quello naive, ma non troppo. Infatti esiste un altro algoritmo chiamato SLINK che porta a risolvere il problema con una complessità temporale pari a $O(n^2)$ e una complessità spaziale pari a $O(n)$.

2.3.2 Miglioriamo le performance

Notiamo che ad aumentare la complessità temporale del solo algoritmo di Kruskal è il valore di m che in particolare è il massimo possibile in quanto il grafo necessariamente,vista la costruzione,è un grafo semplice. Quindi per ridurre il costo computazionale dell'algoritmo di Kruskal è necessario ridurre il numero di archi che costituiscono il grafo. Dalla dimostrazione della correttezza dell'algoritmo si evince che gli archi di un minimum spanning tree sono sufficienti per permettere il cluste-

ring, quindi perché dobbiamo lavorare con tutti gli archi quando bastano solo quelli che formano un MST? Da qui nasce l'esigenza di usare l'algoritmo di Prim che ci permette di trovare un MST e poi usiamo questo (l'MST) per trovare il clustering con l'algoritmo di Kruskal. Così facendo il numero di archi su cui Kruskal opera è $n-1$ (perché l'MST è un albero e un albero ha $n-1$ archi) e questo riduce la complessità di quest'ultimo. Ovviamente Prim ha il suo costo computazionale, ma analizzando l'esecuzione nel complesso vedremo che si riuscirà ad ottenere un ulteriore miglioramento delle performance che saranno identiche a quelle dell'algoritmo SLINK già citato.

2.3.3 Pseudocodice

Algorithm 4 Prim-Kruskal algorithm

Input
Grafo G connesso pesato e semplice con n vertici ed m archi, peso = distanza

Output: hierarchical clustering Single-Linkage degli n vertici che sono punti di $\mathbb{R}^d$

1: $E^{'}$ =Prim(G,s)
2: $G^{'} = (V, E^{'})$
3: **return** Kruskal($G^{'}$)

2.3.4 Analisi della complessità temporale

Avendo il numero di archi massimo per un grafo semplice, per diminuire la complessità computazionale dell'algoritmo di Prim è necessario implementare la coda con priorità tramite lista. Questa scelta porta ad una complessità temporale e spaziale rispettivamente di : $O(n^2)$ e $O(n+m)$. Dato che nel nostro caso $m = \Theta(n^2)$ si ha hanno entrambe le complessità pari a $O(n^2)$. A seguito dell'algoritmo di Prim viene eseguito l'algoritmo di Kruskal su un grafo che ha n vertici e $n-1$ archi, considerando le complessità $O(m \log n)$ e $O(n+m)$ temporale e spaziale rispettivamente si ha, nel nostro grafo ridotto, un costo computazionale pari a $O(n \log n)$ e $O(n)$ per tempo e spazio. Quindi in definitiva sommando entrambi i contributi l'algoritmo nel complesso presenta una complessità temporale $O(n^2)$ e una complessità spaziale $O(n^2)$. Questo non porta però alla complessità computazionale sperata dell'algoritmo SLINK. Per avere quel tipo di performance è necessario adattare l'algoritmo di Prim alle nostre esigenze.

2.3.5 Addattamento dell'algoritmo di Prim

Il punto debole del precedente algoritmo sta nella complessità spaziale che risulta essere di un ordine superiore rispetto all'algoritmo SLINK, quindi dobbiamo cercare di eliminare spazio utilizzato dall'algoritmo senza far variare la complessità temporale. Ricordiamo che la complessità spaziale dell'algoritmo di Prim è lineare nella taglia del grafo. L'addendo m che compare nella complessità è dovuto alla lista di adiacenza usata per accedere ai vicini di un nodo in tempo pari ad $O(degree(v))$ e per salvare la lista degli archi. Per togliere il fattore m è quindi necessario eliminare entrambe queste due strutture dati. Nel nostro caso questo è possibile perché l'arco ha un peso che rappresenta la distanza tra due vertici e quindi questo può essere determinato di volta in volta tramite una funzione $distanza(u, w)$ che può restituire la distanza e quindi il peso dell'arco in tempo $O(1)$. Ricordando lo pseudocodice dell'algoritmo di Prim 2 si nota che il ciclo for interno al ciclo while viene eseguito ad ogni iterazione su tutti gli elementi della lista (che sono i vertici non ancora inseriti nell'MST). I pesi degli archi che connettono il vertice appena inserito con i vertici nella lista sono calcolati tramite la funzione $distanza(u, w)$ e se questa restituisce un valore inferiore al valore $v.D$ questo viene sostituito. Così facendo non necessitiamo più né della lista di adiacenza né della lista degli archi ma solo della coda e dell'albero T che nel loro complesso danno un contributo alla complessità spaziale pari a $\Theta(n)$. La complessità temporale rimane inalterata in quanto il ciclo while viene eseguito n volte e per ognuna di queste iterazioni, che enumeriamo con l'indice i, eseguiamo un numero massimo di operazioni pari a $2i$, in quanto non servono più di i operazioni per estrarre la entry con chiave minima e servono esattamente i operazioni per aggiornare le label nei nodi restanti nella coda. I nodi presenti nella coda devono essere aggiornati tutti ad ogni iterazione in quanto per costruzione del grafo (in questo caso a livello teorico) ogni nodo è vicino di tutti gli altri e quindi ogni volta che andiamo ad inserire un nuovo vertice nell'MST dobbiamo aggiornare le distanze da tutti i vertici non ancora inseriti che sono proprio tutti i vertici ancora nella coda. Il ciclo while viene quindi riscritto come nella pagina seguente.

In formule si ha: $\sum_{i=1}^{n} 2i = n(n-1) = O(n^2)$ che è la stessa complessità ottenuta precedentemente. Con questa modifica si ottengono, per l'algoritmo nel suo complesso, le stesse performance dello SLINK. Ovviamente quanto fatto può essere applicato come conseguenza del fatto che stiamo risolvendo un problema di clustering con l'ausilio di un grafo e per questo sono possibili le modifiche fatte. In generale per un grafo qualsiasi questi accorgimenti non sono realizzabili e quindi questa non deve

essere intesa come una miglioria dell'algoritmo di Prim perché vale solo per il caso preso in esame.

Algorithm 5 Modifica al ciclo while dell'algoritmo di Prim

```
1: repeat
2:     u = Q.removeMin()
3:     unisci il vertice u a T usando la label u.arch
4:     for all v in Q do
5:         if distanza(u, v) < v.D then
6:             v.D = distanza(u, v)
7:             v.arch = (u, v)
8: until Q.is_not_empty()
9: return T
```

2.3.6 Costruzione del dendrogramma con Kruskal

L'algoritmo di Kruskal è stato pensato per determinare un MST di un grafo G. Usarlo per risolvere il problema di clustering è possibile, come già dimostrato, ma sono necessarie opportune modifiche. Faremo in modo di restituire il dendrogramma tramite un linked-tree che costruiremo iterazione per iterazione. Per la costruzione del linked-tree usiamo degli oggetti di tipo "position" già presentati. Ogni nodo dell'albero (sarà un oggetto «position») salverà un riferimento al padre e ai due figli (li chiameremo rispettivamente «parent»,«left»,«right»). I campi salvati nei nodi dell'albero possono essere arricchiti per avere, nel caso servissero, più informazioni. Faremo inoltre uso della struttura union-find (già usata in Kruskal) con una leggera modifica. Il rappresentante di un cluster conterrà anche un riferimento (che chiameremo «node») al nodo che rappresenta il cluster nel dendrogramma. Di seguito questa affermazione verrà chiarita con l'ausilio dello pseudocodice. Vediamo lo pseudocodice dell'algoritmo di Kruskal appositamente adattato per risolvere il problema di clustering.

Algorithm 6 Kruskal-clustering

Input
Grafo G che costituisce un MST restituito dall'algoritmo di Prim.

Output: Dendrogramma relativo al clustering sui vertici del grafo G

```
1: for all v ∈ G do
2:     C(v) = makeCluster(v)
3:     n = new_node()
4:     C(v).node = n
5: Inizializza una coda con priorità Q che contenga tutti gli archi di G, usando il
   peso come chiave della entry
6: repeat
7:     n = new_node()
8:     (u,v) = Q.removeMin()
9:     C(u) = find(u)
10:    C(v) = find(v)
11:    C(u).node.parent = n
12:    C(v).node.parent = n
13:    n.left = C(u).node
14:    n.right = C(v).node
15:    if (|C(u)| > |C(v)|) then C(u).node = n
16:    elseC(v).node = n
17:    union(C(u),C(v))
18: until Q.is_not_empty()
19: n.parent = null
20: return n
```

Il primo ciclo for crea i vari cluster per la struttura union-find e crea un nodo per cluster che corrisponderà ad una foglia del dendrogramma. L'inizializzazione della coda viene fatta come esigenza dell'algoritmo di Kruskal. La condizione d'uscita del while è stata cambiata in quanto essendo il grafo in input G un MST per costruire il clustering necessitiamo di tutti gli archi presenti nella coda e pertanto l'algoritmo deve continuare fino a che è presente anche un solo arco nella coda. Ad ogni iterazione vengono uniti due cluster. Si parte quindi creando il nodo che andrà a rappresentare la fusione nel dendrogramma, si rimuove l'arco che realizza l'unione dei due cluster. Poi si cercano i rappresentati dei due cluster a cui appartengono i due vertici che formano l'arco estratto. Sicuramente questi saranno diversi perché il grafo è un MST. Si impostano i padri dei nodi relativi ai due cluster con l'indirizzo in memoria del nodo appena creato e si impostano i figli del nodo appena creato con gli indirizzi in memoria dei due nodi relativi ai due cluster. Controllando la cardinalità dei due cluster (si può fare in tempo costante sfruttando la label salvata

nel rappresentante, si veda la tree based implementation) si imposta il riferimento al nodo del dendrogramma del rappresentante del cluster più grande con l'indirizzo in memoria del nodo appena creato. Questo viene fatto perché il rappresentante del cluster che risulterà dall'unione è il rappresentante del cluster più grande tra i due. Poi i cluster vengono uniti. All'uscita dal ciclo while, l'ultimo nodo creato corrisponde alla radice del dendrogramma e pertanto impostiamo il suo campo parent a null. Ritorniamo quindi la radice dalla quale è possibile avere accesso a tutto l'albero tramite una qualunque visita.

Si può facilmente verificare che la complessità temporale è la stessa dell'algoritmo di Kruskal originale e pertanto non vengono alterate le prestazioni dell'algoritmo che nel complesso risolve il problema di Hierarchical Clustering.

3 Tecniche avanzate per HC

In questo capitolo verranno descritti degli algoritmi performanti per risolvere il problema dello Hierarchical Clustering. Verrà data una descrizione di alto livello tralasciando i dettagli implementativi. Si vedrà come uno stesso algoritmo possa risolvere il suddetto problema computazionale per più linkage-function. Il tutto verrà esposto in maniera formale e precisa così da evitare di lasciare dubbi al lettore. Una panoramica generale è presente nel seguente articolo [4]. In questo capitolo verrà presentata la formula di Lance-Willliams, il concetto di NN-Graph e di NN-chain. Verrà poi esposto il cosidetto NN-chain algorithm con una prova della correttezza e analisi della complessità temporale e spaziale. Inoltre verrà studiato un algoritmo per Single-Linkage basato ancora una volta su MST, questo algoritmo viene chiamato Single-Fragment algorithm. Anche per quest'ultimo verrà data l'analisi della complessità temporale e spaziale. Per concludere verrano dati dei cenni su un altro algoritmo chiamato Multifragment algorithm.

3.1 Introduzione

Molti algoritmi possono essere presentati per la risoluzione del problema di hierarchical clsutering. Tuttavia tutti possono essere descritti informalmente come segue:

1. Determinare tutte le dissimilarità tra tutte le coppie di elementi/clusters
2. Formare il nuovo cluster a partire da i due elementi/clusters più vicini
3. Ricalcolare la dissimilarità tra il nuovo cluster formato al punto 2 e gli altri punti/cluster

4. Tornare allo step numero due fino a che tutti gli oggetti non sono in un unico cluster

Generalmente lo step numero uno richiede una complessità temporale pari a $O(n^2)$. Essendo precisi il numero esatto corrisponde a $\frac{n(n-1)}{2}$. La dimensione dello spazio in cui sono rappresentati i punti definisce chiaramente un costo temporale diverso, a seconda della dimensione, per il calcolo della dissimilarità. Questo costo computazionale però viene considerato costante per ogni insieme di punti. Per quanto concerne lo step due e tre potrebbe valere la pena di mantenere una lista ordinata di tutte le dissimilarità prese in esame. Utilizzando un qualunque algoritmo ottimo per ordinare gli elementi, dato che il numero di elementi è $\frac{n(n-1)}{2}$ e il costo computazionale dell'algoritmo di ordinamento è $O(n \log n)$ deduciamo tramite un po' di algebra che il costo temporale iniziale per ordinare le dissimilarità è $O(n^2 \log n)$ poi ovviamente vanno considerati i costi dovuti all'aggiornamento delle distanze. Se non viene fatto questo ogni esecuzione dello step due richiederà un tempo pari a $O(n^2)$ in quanto dobbiamo trovare il minimo tra $\frac{n(n-1)}{2}$ dissimilarità. Il fatto di eseguire un Hierarchical Clustering implica che verranno fatti esattamente $n-1$ raggruppamenti. Pertanto lo step due, tre e quattro verranno eseguiti esattamente $n-1$ volte. Lo step tre può essere eseguito in un tempo $O(n)$ usando la formula di Lance-Williams che verrà spiegata a seguito della conclusione di questo paragrafo. Se gli oggetti o i due cluster appena uniti hanno indice i e j e se k è un altro oggetto o un altro cluster allora la distanza tra il cluster ottenuto dall'unione di i e j e il cluster k può essere calcolata con la seguente formula:

$$d(i+j,k) = a_i d(i,k) + a_j d(j,k) + bd(i,j) + c|d(i,k) - d(j,k)|$$

i valori a_i, a_j, b e c dipendono dal tipo di linkage-function che si vuole usare.

3.2 Formula di Lance-Williams

La formula di Lance-Williams è una formula utilizzata per aggiornare la matrice delle dissimilarità (che nell'algoritmo naive abbiamo chiamato anche matrice di prossimità) usando le distanze calcolate in precedenza e con qualunque tipo di linkage-function scelta impostando opportunamente i coefficienti. Daremo i valori dei coefficienti, per le più comuni linkage-function, ricavando questa formula.

1. Single-Linkage function: Con questo tipo di linkage function la dissimilarità

tra due oggetti/clusters è definita come segue: $d(C1, C2) = \min_{\substack{x_1 \in C_1 \\ x_2 \in C_2}} d(x_1, x_2)$, detto questo l'idea è quella di trovare una formula che permetta di determinare il minimo tra due numeri. Dati due numeri $a, b \in \mathbb{R}$ il minimo tra i due può essere ricavato in questo modo: $\min(a, b) = \frac{a+b}{2} - \frac{|a-b|}{2}$. Quindi, ritornando all'algoritmo di clustering, ad ogni iterazione vengono fusi i due cluster più vicini e a seguito di questo si deve aggiornare la matrice di dissimilarità. Necessitiamo quindi di calcolare $d(i + j, k)$ per ogni cluster k rimanente. Nel caso del single linkage, essendo la distanza tra cluster espressa come la minima tra tutte le distanze tra elementi appartenenti al cluster $i + j$ e il cluster k, possiamo ottenere questo risultato sfruttando due distanze calcolate precedentemente ovvero $d(i, k)$ e $d(j, k)$ in quanto queste rappresentano la minima tra tutte le possibili distanze calcolabili tra (i, k) e (j, k) rispettivamente. Quindi per determinare la minima tra tutte le possibili distanze calcolabili tra $(i+j, k)$ basta calcolare :

$$\min(d(i,k), d(j,k)) = \frac{d(i,k) + d(j,k)}{2} - \frac{|d(i,k) - d(j,k)|}{2} =$$

$$\frac{1}{2}d(i,k) + \frac{1}{2}d(j,k) - \frac{1}{2}|d(i,k) - d(j,k)|$$

che notiamo essere una parte della formula di Lance-Williams nella forma generale sopra riportata, con $a_i = \frac{1}{2}$, $a_j = \frac{1}{2}$, $b = 0$ e $c = -\frac{1}{2}$.

2. Average-Linkage function: In questo caso la dissimilarità tra due oggetti/cluster è definita come segue,

$$d(C_1, C_2) = \frac{1}{|C_1|}\frac{1}{|C_2|}\sum_{x_1 \in C_1}\sum_{x_2 \in C_2} d(x_1, x_2)$$

A seguito di una fusione tra due oggetti/clusters i e j la distanza tra questo nuovo cluster e un qualsiasi altri oggetto/clusters esistente è (tramite un pò di algebra):

$$\frac{|i|d(i,k)}{|i| + |j|} + \frac{|j|d(j,k)}{|i| + |j|}$$

Ancora una volta notiamo che questa è una parte della formula di Lance-Williams nella forma generale sopra riportata, con $a_i = \frac{|i|}{|i|+|j|}$, $a_j = \frac{|j|}{|i|+|j|}$, $b = 0$ e $c = 0$.

Tramite questi esempi notiamo che ci sono una serie di termini che si ripetono nell'espressione del calcolo della dissimilarità. Pertanto per dimostrare la formula

di Lance-Williams può essere sufficiente trovare la formula di aggiornamento per il calcolo delle dissimilarità per ogni tipo di linkage function, considerare quindi una sola volta tutti i termini che compaiono nelle singole formule ricavate caso per caso e unificarle con una formula unica tramite una combinazione lineare.

3.3 NN-Graph

Per comprendere gli algoritmi che andremo a presentare è necessario capire cos'è un NN-graph (nearest neighbour graph). E' definito come un insieme di punti p i cui archi diretti $(p, NN(p))$ sono tali che $NN(p)$ sia il punto più vicino a p. Per ogni punto p ci sono tre casi. Il primo prevede che p abbia come punto più vicino q il quale a sua volta ha come punto più vicino g e così via, il secondo caso si ha nel momento in cui p ha come vicino q che ha a sua volta un punto h che è un suo vicino reciproco. Con "vicino reciproco" si intende un punto p che ha come vicino più vicino q e q ha come vicino più vicino p e questo costituisce anche il terzo caso.

3.3.1 NN-chain

Definizione 3.1. Una sequenza del tipo: $i, j = NN(i), k = NN(j), ..., q = NN(p), p = NN(q)$ viene detta *NN-chain*.

Elenchiamo qui di seguito alcune proprietà utili:

Proposizione 3.1. *La dissimilarità tra coppie di punti adiacenti in una NN-chain è decrescente.*

Dimostrazione. Consideriamo la seguente NN-chain $p_1, p_2 = NN(p_1), ..., p_{n-1} = NN(p_{n-2}), p_n = NN(p_{n-1})$ e supponiamo per assurdo che le dissimilarità tra coppie di punti adiacenti siano decrescenti a parte per la coppia (p_{i-1}, p_i). Quindi $d(p_{i-2}, p_{i-1}) < d(p_{i-1}, p_i)$. Essendo p_i il vicino più vicino di p_{i-1} si ha che $d(p_{i-1}, p_i) = \min d(p_{i-1}, x)$ e questo è assurdo in quanto per la precedente disuguaglianza si ha che $d(p_{i-2}, p_{i-1}) < d(p_{i-1}, p_i)$. □

Proposizione 3.2. *I due punti finali di una NN-chain costituiscono una coppia RNN (reciprocal nearest neighbour)*

Dimostrazione. Consideriamo la seguente NN-chain $p_1, p_2 = NN(p_1), ..., p_{n-1} = NN(p_{n-2}), p_n = NN(p_{n-1})$ e supponiamo per assurdo che la coppia (p_{n-1}, p_n) non

sia una coppia RNN. Questo implica che p_n, che è l'ultimo punto della NN-chain ha come vicino più vicino un punto diverso da p_{n-1}. Quindi esiste un punto q tale che $d(p_n, q) < d(p_n, p_{n-1})$ questo porta ad un assurdo perchè si avrebbe un ulteriore punto nella NN-chain contraddicendo il fatto che p_n è l'ultimo □

Proposizione 3.3. *Una NN-chain non può contenere un ciclo*

Dimostrazione. Consideriamo la seguente NN-chain $p_1, p_2 = NN(p_1), ..., p_{n-1} = NN(p_{n-2}), p_n = NN(p_{n-1}), p_1 = NN(p_n)$ che supponiamo per assurdo contenere un ciclo (infatti comincia in p_1 e finisce in p_1).Per la proposizione 3.1 si ha che $d(p_n, p_1) < d(p_1, p_2)$ il che porta ad un assurdo perchè p_2 non può essere un NN(p_1). □

3.4 NN-chain algorithm

Passiamo ora alla descrizione di un algoritmo per Hierarchical Clustering basato su NN-chain. Verranno analizzate le varie complessità e in particolare la correttezza. Per i dettagli si rimanda al seguente articolo [4]

Algorithm 7 NN-chain algorithm

Input
$\{x_i\}_{i=1}^n$ con $x_i \in \mathbb{R}^d$

Output: HC (Hierarchical Clustering) dei punti in input

1: Partendo da un punto x_i generico cosruire una NN-chain
2: Unisci i due punti RNN della NN-chain e sostituiscili con il centro del clsuter
3: Continua una NN-chain dall'ultimo punto rimasto nella catena a seguito della fusione oppure costruisci un'altra NN-chain da un punto/cluster arbitrario se i due punti/clusters uniti al punto precedente erano gli unici a formare la NN-chain
4: Ritorna al secondo step fino a che non rimane un solo punto

3.4.1 Prova della correttezza

Per provare la correttezza di questo algoritmo si deve far vedere che la gerarchia di cluster prodotta è identica a quella prodotta da un algoritmo basato su paradigma Greedy. Ovviamente si accetta il fatto che la gerarchia sarà si la stessa, ma la fusione dei cluster verrà eseguita in ordine diverso e questa peculiarità costituirà anche la

chiave per ottenere una complessità in tempo e spazio ottimale. La correttezza di questo algoritmo si basa sui seguenti risultati:

Lemma 3.2. *La coppia di cluster più vicina costituisce una coppia di cluster RNN .*

Dimostrazione. Siano (C, D) i due cluster più vicini tra i rimanenti questo implica per definizione che il vicino più vicino di C è D e viceversa. Pertanto per definizione sono RNN. □

Proposizione 3.4. *Un algoritmo per clustering ,basato su paradigma Greedy, unisce ,ad ogni iterazione, una coppia di cluster RNN.*

Dimostrazione. Un algoritmo per il clustering basato su paradigma Greedy unisce ,iterazione per iterazione, i due cluster più vicini ,che per il precedente lemma, sono una coppia RNN. Questo implica che vengono uniti , iterazione per iterazione, coppie di cluster RNN. □

A questo punto nasce spontanea una domanda: visto che un algoritmo per HC unisce ad ogni iterazione due clusters che formano una coppia RNN ,si può evitare di cercare la coppia più vicina e cercare invece le coppie RNN? In generale la risposta è negativa. Tuttavia se ,per la linkage function usata, vale la cosiddetta ≪reducibility property≫ (proprietà di riducibilità) allora la risposta alla domanda esposta risulta essere affermativa. Diventa quindi possibile sostituire la ricerca di un minimo con la ricerca di una coppia RNN che ha un costo temporale inferiore.

Definizione 3.3. Sia d una qualunque linkage function, questa viene detta *riducibile* se per ogni terna di cluster A,B e C appartenenti ad un clustering eseguito tramite paradigma Greedy e tali che (A, B) siano RNN in un NN-Graph vale che: $d(A \cup B, C) \geq \min\{d(A, C); d(B, C)\}$

Proposizione 3.5. *Due cluster (C, D) che sono RNN in un qualunque momento formano una coppia che verrà unita da un algoritmo Greedy se la linkage-function è riducibile.*

Dimostrazione. Sia $C_n = (a_n, b_n)$ una coppia di cluster RNN in un NN-graph. Questi possono essere:

- La coppia di cluster più vicina tra tutte e quindi vengono uniti sia con questo tipo di algoritmo che da un algoritmo Greedy e quindi non c'è differenza tra i due
- Non sono la coppia di clusters più vicina in assoluto, pertanto esistono $C_1 = (a_1, b_1); C_2 = (a_2, b_2); \ldots; C_{n-1} = (a_{n-1}, b_{n-1})$ coppie di clusters con interdistanza inferiore rispetto a quella di (a_n, b_n) . Supponiamo anche che siano in ordine di distanza (non è restrittivo supporlo perché se così non fosse si può ottenere questa condizione a seguito di un riordinamento). Detto questo un algoritmo Greedy unirebbe in ordine $C_1, C_2, \ldots, C_{n-1}$. Dimostriamo ora che $C_n = (a_n, b_n)$ verrà necessariamente unito da un algoritmo Greedy. Se a_n, b_n sono RNN questo implica che $d(a_n, b_n) < d(a_n, K)$ e $d(b_n, K)\ \forall\ K$ diverso da b_n, a_n rispettivamente. Sia $i \in [1, n-1]$ e sia C_i un cluster i-esimo nella sequenza qui sopra $\implies$ $d(a_n, b_n) < d(a_n, a_i), d(a_n, b_i), d(b_n, a_i), d(b_n, b_i)$. Quando l'algoritmo Greedy unisce il cluster (a_i, b_i) formando C_i si ha che $d(C_i, K) \geq \min\{d(a_i, K); d(b_i, K)\} > d(a_n, b_n)$. Questo implica che (a_n, b_n) rimangono RNN ad ogni fusione i-esima per $i \in [1, n-1]$. A seguito di queste $n-1$ fusioni l'algoritmo Greedy andrà ad unire la coppia (a_n, b_n) che sarà diventata la coppia di cluster più vicina.

□

3.4.2 Applicabilità dell'algoritmo

Come si vede dalla prova della correttezza, affinchè l'algoritmo produca un clustering corretto deve valere la «reducibility property» che è condizione sufficiente a garantire una gerarchia di fusioni corretta. Deduciamo quindi che questo algoritmo si possa applicare a tutte le linkage-function riducibili e in particolare per:

1. Single-Linkage
2. Average-Linkage
3. Ward's method

Per la prima di queste linkage-function sono già stati presentati degli algoritmi per risolvere il problema di Hierarchical Clustering cosa non fatta per le altre due. E' importante sottolineare che l'algoritmo è universale, come conseguenza dell'astrazione del concetto di vicinanza, purchè valga la proprietà di riducibilità.

3.4.3 Analisi della complessità temporale e spaziale

Per poter analizzare nel complesso l'algoritmo viene in primis esposta una possibile implementazione. Si deve predisporre uno stack S per tenere traccia dei clusters/-punti che formano la NN-chain. Si deve inizializzare un 'insieme che conterrà ad ogni iterazione i clusters rimasti. Questo insieme può essere creato mediante un array di puntatori a liste non ordinate. Ogni cella dell'array contiene un puntatore alla lista non ordinata che contiene tutti gli elementi relativi a quel cluster. Inizialmente si hanno n insiemi (uno per punto). Inoltre per la costruzione del dendrogramma si devono inizializzare n nodi (uno per punto) che andranno a costituire le foglie del dendrogramma. Durante l'esecuzione dell'algoritmo verranno creati altri $n-1$ nodi che rappresenteranno le varie fusioni. Ogni nodo interno conterrà dei puntatori «parent», «left», «right» al nodo padre e ai suoi due figli. Eventualmente i campi del nodo possono essere arricchiti per contenere altre informazioni utili. Ad ogni iterazione viene cercato il NN di un cluster e questo richiede al più $n-1$ valutazioni. A seguito di questo o viene aggiunto un cluster o ne vengono tolti due da S. Ogni cluster viene aggiunto ad S una sola volta e quando viene tolto da S viene fuso con un altro cluster creandone uno nuovo. Le istruzioni «pop()» sullo stack sono quindi tante quante i cluster creati cioè $n-1$ mentre le istruzioni «push()» sono tante quante i cluster iniziali più quelli creati cioè $n+n-1=2n-1$. Ad ogni iterazione il numero massimo di operazioni eseguite sarà $n-1$ per la ricerca del NN. I costi legati alla costruzione del dendrogramma sono trascurabili rispetto alla ricerca del NN, infatti sono costanti. Questo porta ad una complessità temporale pari a $O(n^2)$. Il contributo alla complessità spaziale è $O(n)$ perché lo stack al massimo contiene n elementi , l'insieme per costruzione non può contenere più di $2n$ elementi che anche un upper-bound per il dendrogramma.

3.5 Tecniche basate su MST

In questa sezione vengono consigliati dei metodi, per Hierarchical Clustering Single-Linkage, basati su MST. Dal lavoro fatto da F.J.Rohlf [5] si riesce a trasformare un minimum spanning tree in una gerarchia di cluster in tempo $O(n)$. L'obiettivo diventa quindi trovare l'MST in tempi migliori rispetto all'algoritmo di Prim o Kruskal. Algoritmi veloci per la costruzione dell'MST si possono trovare in un lavoro fatto da Jon Louis Bentley e Jerome H. Friedman nel 1977 [6]. Vengono presentati algoritmi per trovare un MST di una collezione di punti in uno spazio $\mathbb{R}^k$. I vertici del grafo rappresentano i punti dello spazio e gli archi rappresentano le distanze tra

questi punti. Detto questo il grafo è denso e questo implica che $m = \frac{n(n-1)}{2}$. Gli algoritmi classici per la risoluzione di questo problema sono stati presentati da Prim e Kruskal e sono stati inseriti anche in questo scritto. Entrambi questi algoritmi possono essere applicati a qualunque tipo di grafo e proprio per questo, essendo generali, non riescono ad usare i vantaggi geometrici portati dal fatto che i vertici del grafo rappresentano punti di $\mathbb{R}^k$. Per esempio se una persona cercasse di trovare un MST tra le mille città più popolose al mondo non considererebbe mai un arco da Roma a NewYork perché si percepisce subito che esistono città più vicine a Roma. Dato che l'uomo può usare questo tipo di considerazioni per la costruzione dell'MST, che riducono la complessità dell'algoritmo, sarebbe ottimo riuscire a sfruttarle anche in un computer. Esiste un algoritmo che porta ad avere una complessità al caso pessimo pari a $O(n \log n)$ usando i diagrammi di Voronoi ma questo algoritmo è stato progettato solo per funzionare in uno spazio bidimensionale e non è stato esteso a spazi più generali. Vedremo ora degli algoritmi che risolvono il problema della ricerca di un MST per spazi di dimensione generica e per qualunque tipo di linkage function/metrica. L'algoritmo costituisce una variante di quello di Prim che sfrutta la ricerca veloce del «nearest neighbor» grazie alla geometria dello spazio. Le performance dell'algoritmo sono difficili da valutare analiticamente , ma da una simulazione Montecarlo si dimostra che il tempo computazionale è proporzionale ad $n \log n$ in tutti gli scenari di funzionamento sopra descritti. Questo algoritmo è molto più veloce dei tradizionali a parte per dataset di dimensione ridotta dove la differenza non è marcata.

3.5.1 Single-fragment algorithm

Prim diede molte definizioni utili a capire gli elementi che entrano in gioco nella costruzione di un MST e sulla base di queste enunciò due principi che portano alla costruzione del suddetto.

Definizione 3.4. Un nodo si dice *isolato* quando, ad un dato livello di costruzione fissato, non è ancora stato effetuata alcuna connessione.

Definizione 3.5. Un *frammento* è un sottoinsieme dei vertici connesso.

Definizione 3.6. La *distanza di un nodo da un frammento*, al quale non appartiene, è la minima tra tutte le distanze calcolabili tra questo nodo e ogni nodo che costituisce il frammento

Definizione 3.7. Il *nearest neighbor di un nodo a* è il nodo b a distanza minima da a.

Definizione 3.8. Il *nearest neighbor di un frammento* C è il nodo b a distanza minima da C. Dove il la distanza tra nodo e frammento è stata calcolata come descritto nella definizione 3.6

Alla luce di queste definizioni Prim enunciò due principi utilizzabilli per la costruzione di un MST. Secondo il primo principio si può procedere unendo ogni punto isolato al «nearest neighbor» oppure, questo è il secondo principio, afferma che si può creare un MST unendo ad un frammento il suo vicino più vicino.

Prim dimostrò che se vengono aggiunti $n-1$ archi ,in accordo con uno di questi principi, questi archi formeranno un MST. Sia Prim che Dijkstra affermano che un algoritmo che crea un frammento singolo può essere implementato velocemente su un calcolatore. Il «single-fragment algorithm» usa il primo principio per costruire un frammento e poi tramite $n-2$ iterazioni ,sfruttando il secondo principio, va a creare l'MST. Il costo computazionale per selezionare il «nearest neighbor» di un frammento può essere ridotto in due modi.

1. Il primo modo consiste nel salvare in ogni nodo del frammento, ad ogni iterazione, un riferimento al nodo isolato più vicino e la distanza da quest'ultimo
2. Il secondo invece consiste nel salvare su ogni nodo isolato un riferimento al nodo «nearest neighbor» interno al frammento e la sua distanza

N.B: Questi archi potenzialmente usati per la costruzione dell'MST vengono detti «link».

Quando un nuovo nodo viene aggiunto al frammento (fino a quel momento costruito) questi link devono essere aggiornati. L'aggiornamento di questi link richiede un tempo diverso a seconda che si usi la prima implementazione o la seconda.

1. Con la prima alternativa, l'aggiornamento richiede di trovare il «nearest neighbor» del nuovo nodo appena aggiunto tra tutti i nodi isolati. Inotre è necessario ricalcolare il «nearest neighbor» per i nodi del frammento che avevano come NN il nodo appena aggiunto. Il costo computazionale per ogni aggiornamento è proporzionale a $m(n-k)$ con $m \in [1,k]$ e $k \in [1,n-1]$. Qui m è il numero di volte che dobbiamo calcolare un NN ad un aggiornamento mentre k è il numero di nodi appartenenti al frammento ad una data iterazione. Il costo

totale degli aggiornamenti è $\frac{n(n-1)}{2}$ con $m = 1$ in tutti gli aggiornamenti (caso migliore) oppure $\frac{n^2(n-1)}{6}$ con $m = K$ in tutti gli aggiornamenti (caso peggiore).

2. Con il secondo tipo di implementazione l'aggiornamento dei link richiede di calcolare la distanza tra tutti i nodi isolati rimanenti e il nuovo nodo appena inserito al frammento e vedere se questa diventa più piccola rispetto alla precedente. Il numero di calcoli è: $\sum_{i=1}^{n-1} n - i = \frac{n(n-1)}{2}$

A seguito di questa valutazione chiramente si ha che il secondo metodo è più vantaggioso. Infatti è anche quello usato da Prim nel suo algoritmo per la ricerca di MST. Tuttavia esistono casi dove il primo metodo risulta essere più vantaggioso. Per esempio se la ricerca dei NNs non richiede di visionare tutti i nodi isolati ma solo una parte e se gli m calcoli per il NN non sono richiesti ad ogni aggiornamento. Friedman,Bentley and Finkel in un altro lavoro hanno presentato una struttura dati, con relativi algoritmi associati, chiamata «k-d tree» che per la costruzione richiede un tempo pari a $O(n \log n)$ utile per ottimizzare la ricerca del NN. A seguito della costruzione il tempo necessario per cercare un NN è di $O(\log n)$. Il numero di ricerche del «nearest neighbor» (che è il valore m) può essere limitato mettendo tutti i nodi del frammento in una coda con priorità. Nella coda con priorità vengono inseriti i nodi appartenenti al frammento. La priorità di un nodo del frammento è inversamente proporzionale alla distanza che presenta dal «nearest neighbor» isolato. Quindi il nodo del frammento con il «nearest neighbor» isolato più vicino ha priorità più alta. In accordo con il secondo principio di Prim, il link con priorità più alta nella coda è quello da usare per aggiungere il prossimo nodo al frammento. Ad ogni iterazione verrà estreatto il nodo più vicino al frammento tramite il metodo removeMin() della coda. Il «nearest neighbor» del nodo che viene aggiunto al frammento, iterazione per iterazione, deve essere di volta in volta calcolato. Non è pero necssario, ad ogni iterazione, calcolare il NN di tutti i nodi del frammento che avevano come NN il nodo appena aggiunto. Questo perchè i nodi del del frammento che avevano come «nearest neighbor» isolato il nodo appena aggiunto avevano per definizione distanza minima da quest'ultimo. Questo implica che il calcolo del nuovo NN isolato porterà ad avere una distanza da quest'ultimo sicuramente maggiore o uguale alla precedente (in altre parole la distanza precedente consiste in un lower-bound per le distanze da NN futuri). Equivalentemente , in termini di priorità nella coda, la priorità corrente costituisce un upper-bound alla priorità reale (vedi definizioni 3.9,3.10 qui di seguito). In questo modo si riesce a limitare il valore m in quanto si risparmiano molte ricerche del NN isolato. Le ricerche del NN isolato vengono eseguite solo in caso di necessità ovvero nel momento in cui un nodo con

una certa priorità corrente risulta essere anche quello con priorità più alta all'interno della coda.

Definizione 3.9. La *priorità corrente* di un vertice coincide con la priorità che il vertice ha prima che un nuovo vertice venga aggiunto

Definizione 3.10. La *priorità reale* di un vertice coincide con la priorità che il vertice ha dopo che un nuovo vertice è stato aggiunto

Nella pagina seguente può essere visionato lo pseudocodice dell'algoritmo.

Algorithm 8 Single-fragment MST algorithm

Input
$\{x\}_{i=1}^{n}$ con $x_i \in \mathbb{R}^k$
Output: MST del grafo associato

Costruisci il K-d tree dell'insieme di punti
Scegli arbitrariamente un primo nodo
Trova il suo NN
Crea un link con il suo NN
Metti il nodo e la sua reale priorità in una coda con priorità
repeat
    repeat
        x = Q.removeMin()
        y = NN isolato di x
        Crea un link da x ad y
        Aggiorna la priorità di x (adesso è reale)
        Inserisci x nella coda con priorità
    until la priorità più alta non è reale
    x = Q.removeMin()
    y = nodo isolato linkato ad x
    Inserisci il link (X, Y) come arco dell'MST
    Trova il NN di y
    Crea un link con il NN di y
    Metti y e la sua reale priorità nella coda
until non vengono aggiunti $n - 1$ nodi al frammento

3.5.2 Analisi complessità temporale

Dato che sia la ricerca del NN che gli inserimenti nella coda con priorità possono essere fatti in tempo $O(\log n)$ questo algoritmo richiede, per ogni inserimento di un link nell'MST un tempo pari a $O(m_i \log n)$ con i indice dell'iterazione. Quindi la complessità totale risulta essere: $\sum_{i=1}^{n-1} m_i \log n = \log n \sum_{i=1}^{n-1} m_i = (n-1)\overline{m} \log n$ dove $\overline{m}$ è la media dei valori di m nelle varie iterazioni. Se $\overline{m}$ risulta essere minore o

molto minore di $\frac{3n}{2 \log n}$ allora la complessità risulta essere inferiore ad $O(n^2)$ e quindi si ha un algoritmo più performante rispetto a quello generale di Prim.

3.5.3 Analisi della complessità spaziale

I contributi alla comeplssità spaziale sono dovuti alla lista che contiene gli archi che formano l'MST che da un contributo pari a $O(n)$. Inoltre dobbiamo tenere in considerazione lo spazio richiesto dalla coda con priorità, ancora una volta pari a $O(n)$. E infine lo spazio richiesto dal k-d tree che è ancora $O(n)$ in quanto, per costruzione, si ha un punto dello spazio $\mathbb{R}^k$ per ogni nodo dell'albero e quindi i nodi dell'albero sono tanti quanti i punti che costituiscono il dataset. Detto questo sommando tutti i contributi si ha una complessità spaziale totale pari a $O(n)$

3.5.4 Multifragment algorithm

La velocità dell'algoritmo single-fragment dipende dal valore di $\overline{m}$ che dipende a sua volta dalla configurazione dei punti nello spazio. Inoltre dipende dal modo in cui il frammento cresce e quindi dall'ordine con il quale vengono aggiunti i link. Zahn ha affermato, in suo lavoro, che i frammenti tendono a crescere secondo un gradiente di densità dei punti. I principi di Prim per la costruzione di un MST assicurano che il frammento,una volta creato, tende a crescere verso le zone a più alta densità fino a raggiungere un massimo. Una volta arrivato lì tenderà ad aggiungere links per costruire l'MST andando verso zone a bassa densità. Il numero di calcoli per determinare NNs dipende fortemente dalla direzione di crescita del frammento, quindi cambia notevolmente se la direzione presa fa aumentare la densità o se la fa diminuire. Se il frammento sta andando verso zone più dense dello spazio allora ogni nodo aggiunto al frammento tende ad avere un NN più vicino rispetto ai nodi già nel frammento e quindi la sua priorità sarà tale da metterlo nelle posizioni più alte della coda con priorità. Detto questo dato che la priorità di un nodo appena aggiunto è sempre la priorità reale, se questa è anche una delle più alte ecco che il ciclo while interno esegue poche iterazioni in quanto questo nodo con priorità reale verrà estratto con un removeMin() in poche iterazioni. L'esatto contrario si ottiene quando il frammento cresce andando verso zone a bassa densità. Da queste osservazioni si deduce immediatamente che è preferibile crescere un frammento andando da zone a bassa densità verso zone a più alta densità. In molte distribuzioni di punti reali si vede chiaramente che esistono poche regioni altamente dense e molte invece poco dense. Pertanto partendo da un punto arbitrario supponendo che sia

in una regione poco densa (il che è poco probabile considerando che ci sono più punti nelle zone più dense) con poche iterazioni si giungerebbe alle zone dense e gran parte dell'algoritmo verrebbe eseguita andando da zone ad alta densità verso zone a bassa densità peggiorando la complessità temporale. Per ovviare a questo problema si utilizza una strategia basata sulla crescita di più frammenti. Si parte da un punto con la densità locale più bassa e si comincia a crescere un frammento che porterà a raggiungere una zona a massima densità. Una volta arrivati qui si ferma la crescita del frammento e si fa partire (dal punto isolato a più bassa densità locale rimasto) la crescita di un secondo frammento. Il secondo crescerà finché non si unirà ad un frammento esistente o finché non raggiungerà una zona con densità massima. Si itera questo processo fino a che c'è almeno un punto isolato. Alla fine ci si troverà o con un frammento singolo o con molti frammenti non connessi che essendo stati costruiti con l'algoritmo (single-fragment) sono sottoalberi dell'MST. Sia "i" il numero di frammenti non connessi presenti, servono "i-1" links per unirli e creare un MST. Questi possono essere determinati nel seguente modo: si prende il frammento più piccolo (cioè contenente il numero di nodi inferiore) e si continua la sua crescita. Il primo arco che si aggiungerà porterà alla fusione di due frammenti. Si scelgono i frammenti più piccoli per ridurre il tempo di unione. Si continua questa procedura (cercando di volta in volta il frammento più piccolo) fino a che non si ottiene un unico frammento che sarà un MST. L'implementazione di questo algoritmo richiede la stima della densità locale di un punto che può essere fatta ancora una volta tramite un k-d tree. Per bloccare la crescita di un frammento si può impostare un determinato valore m_0 tale che se l'inserimento di un link comporta più di m_0 ricerche di un NN allora si blocca la crescita del frammento e si inizia la crescita del successivo.

3.5.5 Osservazioni e analisi dell'algoritmo

Viene omesso lo pseudocodice per non appesantire la trattazione. Osserviamo però che la complessità temporale e spaziale di questo algoritmo è $O(n \log n)$ e $O(n)$ rispettivamente, ponendo il valore m_0 pari ad una costante non dipendente da n.

3.5.6 Conclusioni

I precedenti due algoritmi per la ricerca di un MST sono stati esposti perchè l'MST prodotto verrà trasformato nello Hierarchical Clustering. Questo risultato può essere ottenuto, per esempio, eseguendo primo uno di questi due algoritmi e poi

l'algoritmo di Kruskal come presentato nel secondo capitolo. Va però osservato che nell'analisi dei due algoritmi (single-fragment e multifragment algorithm) nella complessità temporale interviene il fattore $\overline{m}$ che dipende dalla dispozione dei punti nello spazio. Detto questo è importante sottolineare come questi algoritmi portino miglioramenti su input che rispettano ipotesi ben specifiche (vedere paragarafo 4 dell'articolo [6]) quindi non migliorano l'analisi della complessità temporale al caso pessimo. In altre parole riescono a migliorare l'esecuzione prettamente al lato pratico tramite gli accorgimenti esposti nella trattazzione sopra.

4 Agglomerative Hierarchical Clustering subquadratico per spazi ad alta dimensionalità

In questo capitolo verrà presentato un algoritmo per risolvere il problema computazionale dello Hierarchical Clustering con linkage-function Ward's method. A seguito dell'introduzione verranno forniti dei cenni sulle tecniche di Locality Sensitive Hashing. Poi verrà presentata una struttura dati per la ricerca di un ANN per punti e verrà enunciato il risultato ottenuto. Dopo aver descritto il Ward's method con le relative definizioni e proprietà si descriverà una struttura dati per la ricerca di un ANN per cluster , che costituirà l'elemento fondamentale dell'algoritmo che si intende presentare. Per concludere, oltre a presentare l'algoritmo con le relative dimostrazioni per la complessità temporale e l'approssimazione ottenuta, verranno riportati dei risultati sperimentali per giustificare la bontà dell'algoritmo sviluppato anche da un punto di vista pratico.

4.1 Introduzione

In questo recente lavoro si considerano come metriche per lo HC le seguenti: single-linkage, average linkage, Ward's method. Viene preso in esame il problema di fare lo HC con queste linkage function in spazi metrici ad alta dimensionalità, esempio $\mathbb{R}^d$ con d grande. Non esistono algoritmi performanti per risolvere il problema dello HC con questi tipi di linkage function in spazi metrici ad alta dimensionalità, in quanto si richiede implicitamente di risolvere il problema "closest pair". Questo problema consiste nel trovare la coppia di punti più vicina da un insieme di n punti dati ($n \in \mathbb{N}$). Come si è visto nello AHC (Agglomerative Hierarchical Clustering) in

input viene data anche la linkage-function, che definisce una misura di dissimilarità tra punti e cluster. Da questa misura di dissimilarità un HC può essere definito come un algoritmo, che mette inizialmente ogni punto in un cluster e ripetutamente crea nuovi cluster, unendo,ad ogni iterazione la coppia di cluster/punti più vicina secondo la linkage function data come input, ecco che ad ogni iterazione viene quindi chiesto di risolvere il problema "closest pair". Questo problema può essere risolto in tempo $O(n^2)$ con il metodo naive o in tempo $O(n \log n)$ con approccio divide and counquer. Osserviamo che il collo di bottiglia degli algoritmi basati su paradigma Greedy per la risoluzione dello HC, sta nella risoluzione del sottoproblema "closest pair". Un esempio di algoritmo che aggira questo problema è l'algoritmo basato sulla NN-chain già presentato. Se vale la reducibility property si garantisce la stessa gerarchia di cluster prodotta da un algoritmo basato su paradigma Greedy, ma la si può ottenere fondendo ad ogni iterazione coppie di punti RNN. Fino ad ora abbiamo visto i seguenti algoritmi:

Nome	Complessità temporale	Complessità spaziale
Naive alg.	$\Theta(n^3)$	$\Theta(n^2)$
Prim + Kruskal	$O(n^2)$	$\Theta(n)$
NN-chain	$O(n^2)$	$O(n)$
Single-fragment alg.	$(n-1)\bar{m}\log n$	$O(n)$

Nota bene: Il significato di $\bar{m}$ è stato spiegato nel capitolo precedente. Si ricorda che il valore $\bar{m}$ è il valore medio del numero di ricerce di un NN.

Si osservi che il primo algoritmo esegue correttamente lo HC per tutte le linkage-function, ma ha lo svantaggio di essere poco performante. Il secondo invece è corretto con complessità temporale del lower-bound teorico, ma è stato presentato solo per Single-Linkage (anche se in letteratura è noto un algoritmo simile anche per Ward's method e Average-Linkage). Il terzo invece, oltre ad avere complessità temporale e spaziale come da lower bound, funziona per tutte le linkage-function riducibili. L'ultimo algoritmo funziona solo per Single-Linkage e a volte c'è la possibilità di scendere sotto il lower bound teorico, ma con ipotesi sui dati in input. Quindi non è un algoritmo che migliora le complessità al caso pessimo. La domanda che si sono fatti gli autori di [7] è la seguente: quanto performanti potrebbero essere gli algoritmi per risolvere lo HC, con le sopra indicate linkage-function, se si permettessero delle approssimazioni nella risoluzione del problema? Questi algoritmi fondono di volta in volta i cluster con dissimilarità minima (secondo la linkage function specificata).

Di fatto si sono chiesti se si può ottenere o meno un miglioramento significativo nel tempo di esecuzione dell'algoritmo se a quest'ultimo è concesso di fare la cosidetta γ-approximate closest cluster. Ovvero evitare di fondere, ad ogni iterazione, coppie di cluster a minima distanza e accettando di poter fondere coppie di cluster che hanno distanza non superiore a γ volte la distanza minima, per quella specifica iterazione. Si è dimostrato che da questo tipo di approssimazione si possono ottenere considerevoli miglioramenti. Inoltre sono stati portati risultati pratici che hanno permesso di far vedere come i risultati ottenuti siano ancora significativi (nonostante l'approssimazione) con il vantaggio però di aver ridotto la complessità temporale. Si noti che la differenza tra le varie linkage-function sta nel significato che viene attribuito alla similarità tra due punti o clusters. Questo, influenza la scelta dei due cluster da unire iterazione per iterazione. E' noto, nel settore dello Hierarchical Clustering, che algoritmi che implementano lo HC Single-Linkage sono più veloci e di più facile implementazione rispetto a quelli che implementano il Ward's method o l'Average Linkage, tuttavia i risultati prodotti da queste due ultime linkage function sono molto più significativi rispetto a quelli prodotti dalla single-linkage. Tutti questi algoritmi possono essere implementati in tempo quadratico o quasi. Lo scopo del lavoro esposto al NeurIPS 2019 è quello di riuscire a trovare algoritmi implementabili in tempo sub-quadratico tramite la seguente approssimazione. Supponiamo che la coppia di cluster da unire ad una specifica iterazione sia a distanza l, diamo all'algoritmo la possibilità di unire qualunque coppia di cluster con distanza $l^{'}$ appartenente all'intervallo $[l, \gamma l]$ con γ costante, $\gamma > 1$. Quando l'approssimazione è concessa, allora, il tempo richiesto per la risoluzione del "closest pair" si riduce e viene meno anche il lower bound teorico $\Omega(n^2)$. Anche in alta dimensionalità, tecniche di locality sensitive hashing, possono trovare il γ-approximate nearest neighbors (ANN) con una distanza L1 con un tempo pari a $n^{O(\frac{1}{\gamma})}$ per chiamata.

Osservazione 1: Una chiamata permette di prendere un punto tra gli n dati e trovare il suo NN. Con le locality sensitive hashing si ottiene il risultato approssimato in tempo $n^{O(\frac{1}{\gamma})}$. Osserviamo che $\gamma > 1$ (altrimenti non stiamo risolvendo il problema in maniera approssimata) e pertanto $\frac{1}{\gamma}$ sarà una quantità minore di uno. Quindi, la singola richiesta di trovare il NN di un punto dato, la si ottiene in tempo inferiore rispetto alla linear search. Come si vede dalla espressione matematica della complessità temporale, il "tempo" di ricerca è tanto inferiore quanto più grande è l'approssimazione fatta, che quantitativamente andiamo a rappresentare con il valore di γ. Il problema della "closest pair", richiede però di trovare la coppia di punti più vicina tra tutte le possibili coppie formabili tra i punti dati. Questo lo si

ottiene effettuando una γ-approximate nearest neighbor per ogni altro punto e poi prendendo la minima tra tutte le distanze ottenute con queste chiamate. Dato che le singole chiamate restituiscono il NN di un punto dato approssimato, così facendo si ottiene la risoluzione del problema "closest pair" in maniera approssimata. La complessità temporale richiesta è pari $nn^{O(\frac{1}{\gamma})} = n^{1+O(\frac{1}{\gamma})}$. Notiamo che in generale il problema della "closest pair" verrebbe risolto in tempo $\Omega(n^2)$.

4.2 Cenni su Locality Sensitive Hashing (LSH)

Le locality sensitive hashing sono tecniche algoritmiche per mappare un insieme di elementi, che appartengono ad un universo molto ampio, in un'insieme di "bucket", con l'intento di far sì che oggetti simili finiscano nello stesso «bucket» con alta probabilità. Siccome gli oggetti simili finiscono nello stesso «bucket» con alta probabilità, questa tecnica può essere usata per effettuare il clustering o la ricerca del vicino più vicino. Questo tipo di hashing differisce dalla classica tecnica di hashing in quanto cerca di massimizzare (e non minimizzare) le collisioni. Oppure, alternativamente, questa tecnica può essere vista come un modo per ridurre la dimensionalità di dati che appartengono ad uno spazio metrico ad alta dimensionalità. Si ottiene una versione a bassa dimensionalità mantenendo o preservando le distanze relative tra oggetti. Per ulteriori chiarimenti si rimanda al seguente libro [8].

4.3 Approximate Nearest Neighbor search per punti

Per cercare il NN (Nearest Neighbor) di un punto dato in maniera approssimata si può usare una struttura dati basata su tecniche di LSH (non è l'unico modo di farlo) operante nel seguente modo. Dato $\gamma > 1$ un parametro e dato un qualunque $P \subset \mathbb{R}^d$ con $d \in O(\log n)$, la struttura dati (che indichiamo con D) permette le seguenti operazioni:

- Inserimento di un punto $x \in P$ in tempo $O(n^{f(\gamma)})$
- Rimozione di un punto $x \in P$ in tempo $O(n^{f(\gamma)})$
- dato $x \in P$ deve restituire un punto x' tale che $d(x, x') \leq \gamma d(x, NN(x))$ (con d distanza L_2) in tempo $O(n^{f(\gamma)})$

La $f()$ usata nella soprastante descrizione è una generica funzione che ha come argomento γ che è il parametro di approssimazione. Nel caso di distanza L_2 si riesce ad ottenere $f(\gamma) \in O(\frac{1}{\gamma^2})$.

4.4 Risultato ottenuto

Nell' articolo sopra riportato vengono proposti due algoritmi con complessità subquadratica: uno per Ward's method e uno per Average-Linkage. Tuttavia nel seguito verrà discusso solo il primo di questi due. In particolare sono riusciti ad ottenere il seguente teorema che verrà dimostrato nel seguito.

Teorema 4.1. *Dato un insieme di n punti in* $\mathbb{R}^d$ *e una struttura dati* $\sqrt{\gamma}$*-Approximate Nearest Neighbor che supporta inserzione, rimozione e chiamata (come sopra descritta) in tempo T allora esiste uno HC con Ward's method con un'approssimazione pari a* $\gamma(1+\epsilon)$ *ottenibile in tempo* $O(nT\epsilon^{-1}\log(\Delta n)\log n)$

Osservazione: Il ruolo delle variabili ϵ e Δ verrà chiarito nel seguito. Inoltre, successivamente, verrà anche spiegato come si ottiene l'approssimazione citata.

4.5 Descrizione del Ward's method

Prima di descrivere il Ward's method vengono date delle definizione e proposizioni utili a chiarire quest'ultimo.

Definizione 4.2. Sia $P \subset \mathbb{R}^d$ si definisce *aspect ratio* la quantità Δ come segue:

$$\Delta = max_{u,v \in P} dist(u,v) \tag{4.1}$$

Osservazione: Si noti che l'aspect ratio per come è definito esprime la distanza tra i punti più distanti presenti in P. A volte la distanza che viene considerata non è la distanza tra punti classica, ma bensì una distanza riscalata della distanza minima dei punti in P. Ovvero, dati $x_1, ..., x_n \in P$ sia $\delta = min_{u,v \in P} dist(u,v)$ la distanza riscalata $dist_{new}(x_i, x_j) = \frac{dist(x_i,x_j)}{\delta}$. Così facendo, dati $x_i, x_j \in P$, $dist_{new}(x_i, x_j)$ esprime la distanza tra i due punti x_i, x_j in multipli della distanza minima.

Definizione 4.3. Dato C_i un cluster, il suo *centroide* è definito come il suo punto medio in $\mathbb{R}^d$ e lo indichiamo con $\mu(C_i)$, formalmente:

$$\mu(C_i) = \frac{1}{|C_i|} \sum_{x \in C_i} x \tag{4.2}$$

Definizione 4.4. Dato C_i un cluster, si definisce *Error Sum of Square* la quantità

$$ESS(C) = \sum_{x \in C} (x - \mu(C))^T (x - \mu(C)) \tag{4.3}$$

Osservazione: Ricordando che il prodotto scalare di un vettore trasposto con se stesso coincide per definizione con la sua norma al quadrato, la precedente sommatoria esprime la somma delle distanze euclidee al quadrato, tra i punti appartenenti al cluster C e il centroide calcolato come definito nella definizione qui sopra. In particolare questa sommatoria esprime il grado di «dispersione» dei punti attorno al centroide.

Definizione 4.5. Dato un clustering $C = \{C_1, ..., C_l\}$, si definisce la *somma dei quadrati degli errori di un clustering* come segue:

$$ESS(C) = \sum_{c \in C} ESS(c) \tag{4.4}$$

dove c è un qualunque cluster appartenente al clustering C.

Definizione 4.6. Si definisce *variazione della somma dei quadrati* (a seguito di una fusione di due cluster) la seguente quantità:

$$\Delta ESS(C_i, C_j) = ESS(C_i \cup C_j) - ESS(C_i) - ESS(C_j) \tag{4.5}$$

Lemma 4.7. *Dati due cluster C_i, C_j e dati i due centroidi $\mu(C_i), \mu(C_j)$ il centroide del cluster $C_i \cup C_j$ sta nella retta passante per $\mu(C_i), \mu(C_j)$ ad una distanza $\frac{|C_j|}{|C_i \cup C_j|} \|\mu(C_i) - \mu(C_j)\|$ da $\mu(C_i)$*

Dimostrazione. Per dimostrare il lemma dobbiamo dimostrare due cose:

1. Il centroide del cluster unione sta nella retta passante per i due centroidi dei due cluster originali

2. Il centroide dell'unione si trova alla distanza specificata dal centroide $\mu(C_i)$

Per perseguire questo scopo, prima di tutto, andremo a calcolare le coordinate del centroide del cluster unione, dopodichè si dimostrerà che quest'ultimo appartiene alla retta passante per i due centroidi dei cluster originali e infine si dimostrerà l'affermazione fatta sulla distanza tra quest'ultimo e il cluster $\mu(C_i)$.

1. Posizione del centroide del cluster unione:

$$\begin{aligned}\mu(C_i \cup C_j) = \frac{1}{|C_i \cup C_j|} \sum_{x \in C_i \cup C_j} x = \frac{1}{|C_i \cup C_j|}(\sum_{x \in C_i} x + \sum_{x \in C_j} x) = \\ \frac{1}{|C_i \cup C_j|}(\mu(C_i)|C_i| + \mu(C_j)|C_j|) = \frac{\mu(C_i)|C_i| + \mu(C_j)|C_j|}{|C_i \cup C_j|}\end{aligned} \tag{4.6}$$

2. Dimostrazione di appartenenza alla retta: l'equazione della retta passante per i due centroidi dei due cluster originali è:

$$r : \mu(C_i) + \alpha(\mu(C_j) - \mu(C_i)) = (1 - \alpha)\mu(C_i) + \alpha\mu(C_j), \alpha \in \mathbb{R} \tag{4.7}$$

   Il punto $\frac{\mu(C_i)|C_i| + \mu(C_j)|C_j|}{|C_i \cup C_j|}$ sta nella retta se e solo se esiste $\alpha \in \mathbb{R}$ tale che:

$$(1 - \alpha)\mu(C_i) + \alpha\mu(C_j) = \frac{\mu(C_i)|C_i| + \mu(C_j)|C_j|}{|C_i \cup C_j|} \tag{4.8}$$

   Si verifica facilmente che per $\alpha = \frac{|C_j|}{|C_i \cup C_j|}$ l'uguaglianza risulta essere vera, il che dimostra l'asserto.

3. Dimostrazione dell'affermazione fatta sulla distanza: indichiamo con d la distanza di $\mu(C_i)$ dal punto $\frac{\mu(C_i)|C_i| + \mu(C_j)|C_j|}{|C_i \cup C_j|}$

$$d = \| \frac{|C_j|}{|C_i \cup C_j|}(\mu(C_j) - \mu(C_i)) \| = \frac{|C_j|}{|C_i \cup C_j|} \|(\mu(C_j) - \mu(C_i))\| \tag{4.9}$$

□

Il lemma appena dimostrato sarà d'ausilio nella dimostrazione della proposizione seguente.

Proposizione 4.1. *La variazione della somma dei quadrati sopra definita è espri-*

mibile con la seguente formula:

$$\Delta ESS(C_i, C_j) = \frac{|C_i||C_j|}{|C_i| + |C_j|}\|\mu(C_i) - \mu(C_j)\|^2 \tag{4.10}$$

Dimostrazione. Per dimostrare la seguente formula, si consideri la seguente quantità:

$$ESS(C_i \cup C_j) = \sum_{x \in C_i \cup C_j} \|x - \mu(C_i \cup C_j)\|^2 = \sum_{x \in C_i} \|x - \mu(C_i \cup C_j)\|^2 + \sum_{x \in C_j} \|x - \mu(C_i \cup C_j)\|^2 \tag{4.11}$$

Ora, espandiamo la prima delle due sommatorie (nell'altra si potranno eseguire le stesse operazioni fatte in quest'ultima per giungere a risultati analoghi) e poi andremo a sommare il contributo come dichiarato dalla formula sopra, per ottenere, dopo qualche spostamento di termini, la tesi di questa proposizione.

Tenendo presente il lemma sopra esposto:

$$\begin{aligned}
&\sum_{x \in C_i} \|x - \mu(C_i \cup C_j)\|^2 = \sum_{x \in C_i} \|x - (\mu(C_i) - \frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j))\|^2 = \\
&\sum_{x \in C_i} \|(x - \mu(C_i)) + \frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j)\|^2 = \\
&\sum_{x \in C_i} \|(x - \mu(C_i))\|^2 + 2\sum_{x \in C_i}(x - \mu(C_i))\frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j)) + \\
&\sum_{x \in C_i} \|\frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j))\|^2 = \\
&ESS(C_i) + 2(\sum_{x \in C_i}(x - \mu(C_i)))\frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j)) + \frac{|C_i||C_j|^2}{(|C_i| + |C_j|)^2}\|(\mu(C_i) - \mu(C_j))\|^2
\end{aligned} \tag{4.12}$$

Si osservi che:

$$\begin{aligned}
&2(\sum_{x \in C_i}(x - \mu(C_i)))\frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j)) = \\
&2(\sum_{x \in C_i} x - \sum_{x \in C_i} \mu(C_i))\frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j)) = \\
&2(|C_i|\mu(C_i) - \mu(C_i)|C_i|)\frac{|C_j|}{|C_j| + |C_i|}(\mu(C_i) - \mu(C_j)) = 0
\end{aligned} \tag{4.13}$$

Concludendo si ottiene che:

$$\sum_{x \in C_i} \|x - \mu(C_i \cup C_j)\|^2 = ESS(C_i) + \frac{|C_i||C_j|^2}{(|C_i| + |C_j|)^2} \|(\mu(C_i) - \mu(C_j))\|^2 \tag{4.14}$$

Similmente, svolgendo l'altra sommatoria della formula 4.11 si ottiene che:

$$\sum_{x \in C_j} \|x - \mu(C_i \cup C_j)\|^2 = ESS(C_j) + \frac{|C_j||C_i|^2}{(|C_i| + |C_j|)^2} \|(\mu(C_i) - \mu(C_j))\|^2 \tag{4.15}$$

Ora sommando 4.14 con 4.15 come previsto da 4.11 si ottiene:

$$\begin{aligned} ESS(C_i \cup C_j) = \sum_{x \in C_i \cup C_j} \|x - \mu(C_i \cup C_j)\|^2 = ESS(C_i) + ESS(C_j) + \\ \frac{|C_j||C_i|^2 + |C_i||C_j|^2}{(|C_i| + |C_j|)^2} \|(\mu(C_i) - \mu(C_j))\|^2 \end{aligned} \tag{4.16}$$

Quindi:

$$ESS(C_i \cup C_j) - ESS(C_i) - ESS(C_j) = \frac{|C_j||C_i|}{|C_i| + |C_j|} \|(\mu(C_i) - \mu(C_j))\|^2 \tag{4.17}$$

□

Disponendo ora di tutti i risultati necessari descriviamo l'esecuzione del classico algoritmo per Ward's method. Quest'ultimo crea una gerarchia di cluster (rappresentabili come al solito da un dendrogramma) dove ogni livello rappresenta un clustering dei punti e dove i cluster al livello l sono sottoinsiemi dei cluster al livello $l + 1$. Come al solito si parte a considerare i singoli punti come cluster e questi costituiscono il livello zero del dendrogramma. Poi, dato il clustering di un livello $l - esimo$, si determina il clustering del livello $l + 1 - esimo$ unendo i due cluster del livello $l - esimo$ che producono (con la loro fusione) il minor aumento della $ESS(C)$, con C che indica un clustering (non un cluster). Per togliere ogni ambiguità, si tenga a mente che per «clustering» si intende l'insieme dei cluster formati ad un dato livello $l - esimo$. Sia $C = \{c_1, ..., c_l\}$ il clustering presente ad un dato livello l, a questo punto l'algoritmo sceglierà due cluster $C_i, C_j \in C$ da fondere in unico cluster che diventerà $C_i \cup C_j$. Analizzando la $ESS(C)$ del clustering al livello l e $l + 1$ si ha

che:

$$ESS(C_l) = \sum_{k=1}^{l} ESS(c_k) = \sum_{k=1,k\neq i,j}^{l} ESS(c_k) + ESS(c_i) + ESS(c_j) < \sum_{k=1,k\neq i,j}^{l} ESS(c_k) + ESS(c_i \cup c_j) = ESS(C_{l+1}) \tag{4.18}$$

Nota:
I pedici l,$l+1$ indicano il livello del clustering preso in considerazione.
Osservazione:
La precedente disuguaglianza discende dal fatto che:

$$\Delta ESS(c_i, c_j) = ESS(c_i \cup c_j) - ESS(c_i) - ESS(c_j) = \frac{|c_i||c_j|}{|c_i| + |c_j|}\|\mu(c_i) - \mu(c_j)\|^2 > 0 \tag{4.19}$$

quindi: $ESS(c_i \cup c_j) > ESS(c_i) + ESS(c_j)$. Come detto in precedenza vogliamo che l'aumento della ESS del clustering sia minimo, ovvero, indicando i due cluster da unire alla prossima iterazione con c_i, c_j:

$$\begin{aligned} c_i, c_j &= \arg \min_{c_k,c_h\in\{c_1,\ldots,c_l\},c_k\neq c_h} ESS(C_{l+1}) - ESS(C_l) = \\ &\arg \min_{c_k,c_h\in\{c_1,\ldots,c_l\},c_k\neq c_h} ESS(c_k \cup c_h) - ESS(c_k) - ESS(c_h) = \\ &\arg \min_{c_k,c_h\in\{c_1,\ldots,c_l\},c_k\neq c_h} \Delta ESS(c_k, c_h) \end{aligned} \tag{4.20}$$

che formalmente esprime la seguente affermazione: minimizzare la $ESS(C_{l+1}) - ESS(C_l)$ equivale a trovare la coppia di cluster che minimizza la quantità definita nella definizione 4.6. Quindi, supponendo di salvare cardinalità dei cluster e relativi centroidi, si tratta di calcolare tutte le suddette quantità e prenderne la minima, il che conduce come già anticipato al problema della «closest pair». Per ottenere una complessità temporale migliore rispetto agli algoritmi già studiati si risolverà il problema nel modo seguente. All'inizio del capitolo si è descritta una possibile implementazione di una struttura dati per la ricerca di un ANN di un punto dato. Ora vogliamo far sì che, queste strutture dati, possano calcolare l' ANN di un cluster. Grazie alla formula 4.10 la distanza tra due cluster può essere calcolata efficientemente con $O(1)$ operazioni e il calcolo di una distanza tra due punti (i due centroidi). Detto questo, possiamo adattare la struttura dati per la ricerca di un ANN per punti, in una struttura dati per la ricerca di un ANN per cluster. Questo verrà spiegato nel paragrafo successivo.

4.6 Approximate nearest neighbor di un cluster

L'algoritmo che viene descritto nell'articolo, per trovare il nearest neighbor di un cluster, si basa su una struttura dati in cui la distanza tra due cluster C_i, C_j è la quantità $\Delta ESS(C_i, C_j) = ESS(C_i \cup C_j) - ESS(C_i) - ESS(C_j)$. Questa struttura dati si costituisce di più strutture dati per punti (come quelle descritte in precedenza). In particolare dato un parametro $\epsilon > 0$ la nostra NNDS (Nearest Neighbor Data Structure) per cluster, che indichiamo con $D(\gamma, \epsilon)$, consiste in strutture dati per punti con parametro d'errore $\sqrt{\gamma}$. Esiste una struttura dati per punti, che indichiamo con P^l, per ogni $l \in \{(1+\epsilon)^i | i \in [1, ..., \lceil \log_{1+\epsilon} n \rceil], i \in \mathbb{N}\}$.

Proposizione 4.2. *Il numero di strutture dati per punti appartiene ad* $O(\epsilon^{-1} \log n)$.

Dimostrazione. Dato che $l \in \{(1+\epsilon)^i | i \in [1, ..., \lceil \log_{1+\epsilon} n \rceil], i \in \mathbb{N}\} \Rightarrow \#NNDS = \lceil \log_{1+\epsilon} n \rceil < 1 + \log_{1+\epsilon} n \implies \#NNDS \in O(\log_{1+\epsilon} n)$. Per dimostrare l'asserto dobbiamo far vedere che $\log_{1+\epsilon} n \in O(\frac{\log n}{\epsilon})$. Dalle proprietà dei logaritmi e dallo sviluppo in serie di Taylor di $\ln(1+x)$ abbiamo che:

1. $\log_{1+\epsilon} n = \frac{\ln n}{\ln(1+\epsilon)}$
2. Per $x > -1$ vale che: $\frac{x}{1+x} \leq \ln(1+x) \leq x$

Applicando il punto due a $\ln(1+\epsilon)$ si ottiene che per $\epsilon \in (0,1)$, $\ln(1+\epsilon) \in \Theta(\epsilon)$. Ne consegue che: $\log_{1+\epsilon} n = \frac{\ln n}{\ln(1+\epsilon)} = \Theta(\frac{\ln(n)}{\epsilon}) = \Theta(\frac{\log(n)}{\epsilon})$ □

Nella struttura dati per cluster $D(\gamma, \epsilon)$ si prevede la possibilità di eseguire le seguenti due operazioni:

- **Insertion(C)**:l'inserimento di un cluster viene eseguito inserendo il suo centroide $\mu(C)$ nella struttura dati per punti P^i con i tale che:

$$(1+\epsilon)^{i-1} \leq |C| < (1+\epsilon)^i$$

- **Query(C)**: $\forall l \in \{(1+\epsilon)^i | i \in [0, ..., \lceil \log_{1+\epsilon} n \rceil], i \in \mathbb{N}\}$ viene eseguita una nearest neighbor approssimata per $\mu(C)$ in P^l, indichiamo il risultato ottenuto con $NN_l(C)$. La routine ritorna l'$NN_l(C)$ che minimizza la quantità $\Delta ESS_{C,NN_l(C)}$

Di seguito vengono fatte delle osservazioni che dovrebbero chiarire la composizione e lo scopo della struttura dati per cluster $D(\gamma, \epsilon)$.

Osservazione 1): Si noti che le strutture dati P^l, al variare di l, creano degli intervalli contigui. Il primo di questi ha estremo sinistro in uno e l'ultimo di questi ha l'estremo destro che è strettamente maggiore di n, che è il numero di punti sui quali si esegue il clustering. D'altronde lo scopo delle strutture dati per punti P^l è quello di partizionare l'insieme di tutte le possibili cardinalità dei cluster che si possono formare. Quindi, quanto affermato nelle righe sopra è ragionevole in quanto, è contemplato un numero di elementi appartenenti ad un cluster pari ad uno (situazione iniziale in cui ogni punto è un cluster) e un numero pari ad n (situazione finale in cui tutti i punti appartengono ad un singolo cluster).

Osservazione 2): Si noti che la scelta fatta sui valori di l, permette di coprire tutto l'intervallo della retta reale $[1, n]$, con un numero di strutture dati logaritmico in n. La soluzione più banale che poteva venire a mente per partizionare le possibili cardinalità, avrebbe previsto una struttura dati $\forall l \in [1, n]$ con $l \in \mathbb{N}$, il che avrebbe portato ad un numero di strutture dati lineare in n.

Osservazione 3): Si noti che l'ampiezza degli intervalli cresce esponenzialmente al variare di l, infatti, definendo a l'ampiezza di un generico intervallo si ha che:

$$a = (1+\epsilon)^l - (1+\epsilon)^{l-1} = (1+\epsilon)^l(1 - \frac{1}{1+\epsilon}) = (1+\epsilon)^l(\frac{\epsilon}{1+\epsilon})$$

inoltre è facile convicersi del fatto che esistono valori di l tali per cui l'ampiezza dell'intervallo risulta maggiore di uno e pertanto può darsi il caso in cui una struttura dati P^l contenga cluster di cardinalità diverse e consecutive. Per esempio, potrebbe darsi il caso che per qualche l, P^l contenga cluster di cardinalità trenta e trentuno. Questo è un solo esempio numerico che ha lo scopo di chiarire l'affermazione sopra, non va inteso come una proposizione o un fatto, perchè se questo caso si verifica, ed è di interesse esserne certi, allora questo va dimostrato rigorosamente.

Osservazione 4): Si osservi inoltre che esistono degli l che danno luogo ad intervalli della retta reale che non contengono numeri naturali. Tali l daranno luogo a delle strutture che verrano quindi inizializzate ma non usate, in quanto la cardinalità di un insieme è un numero naturale.

Di seguito viene riportato e dimostrato il seguente lemma che specifica la complessità temporale delle due operazioni e una proprietà di $D(\gamma, \epsilon)$.

Lemma 4.8. *Per ogni $\epsilon > 0$ la struttura dati $D(\gamma, \epsilon)$ ha le seguenti proprietà:*

1. *La complessità temporale di «Insertion(C)» è: $O(n^{f(\sqrt{\gamma})}\epsilon^{-1}\log n)$*
2. *La complessità temporale di «Query(C)» è: $O(n^{f(\sqrt{\gamma})}\epsilon^{-1}\log n)$*
3. *La routine «Query(C)» ritorna un cluster $C^{'}$ tale che: $ESS(C \cup C^{'}) - ESS(C) - ESS(C^{'}) \leq (1+\epsilon)\gamma \min_{B \in D(\gamma,\epsilon)} ESS(C \cup B) - ESS(C) - ESS(B)$*

Dimostrazione. Dimostriamo le affermazioni nell'ordine:

1. Per quanto riguarda la complessità temporale di «Insertion(C)» è stato dimostrato che le strutture dati P^l non sono più di $\epsilon^{-1}\log n$,quindi, dato che nella peggiore delle ipotesi si controllano tutte le strutture dati una alla volta e quella giusta è l'ultima, eseguiamo al più $(\epsilon^{-1}\log n - 1)$ operazioni e un successivo inserimento. Stimiamo dall'alto questa quantità con $\epsilon^{-1}\log n$ inserimenti il che conduce a dimostrare che non vengono eseguite più di $n^{f(\sqrt{\gamma})}\epsilon^{-1}\log n$ operazioni, ricordando che un inserimento ha costo $n^{f(\sqrt{\gamma})}$. Questo permette di giungere alla tesi.

2. Per quanto concerne la complessità temporale del metodo «Query(C)», viene eseguita una ANN per ogni struttura dati P^l la quale restituisce a sua volta un valore. Ogni ricerca ANN costa $n^{f(\sqrt{\gamma})}$. Ogni valore restituito viene poi inserito nella formula per il calcolo della quantità $\Delta ESS()$ con uno dei due argomenti pari a C e poi viene presa la minima di tutte queste valutazioni. Per fare ciò, sono necessarie altre $\epsilon^{-1}\log n$ operazioni con complessità temporale costante. Quindi sommando i vari ccontributi si ottiene che la compessità totale del metodo è $O(n^{f(\sqrt{\gamma})}\epsilon^{-1}\log n)$ ancora una volta.

3. Sia $P^{l^{'}}$ una qualunque delle strutture dati che costituiscono $D(\gamma, \epsilon)$, sia C^* un qualunque cluster presente in $P^{l^{'}}$, per ogni altro cluster $C^{'}$ in $P^{l^{'}}$ si ha che

$$|C^{'}| < (1+\epsilon)^l \leq (1+\epsilon)|C^*|$$

   La prima delle due disuguaglianze è vera per costruzione di $P^{l^{'}}$, la seconda invece è vera, in quanto, la costruzione di $P^{l^{'}}$ garantisce che $|C^*| \geq (1+\epsilon)^{l-1}$. Dimostriamo ora la terza affermazione facendo uso del fatto appena esposto. Sia C_0 il cluster che minimizza (in maniera esatta e non approssimata) la quantità $\Delta ESS(C, C_i)$. Sia j l'indice della struttura dati P^j alla quale appartiene. Sia $C^{'}$ il cluster ritornato dall'invocazione del metodo $ANN(\mu(C))$

invocata dal metodo «Query(C)».

Consideriamo la seguente quantità: $\Delta ESS(C, C^{'}) = \frac{|C||C^{'}|}{|C| + |C^{'}|}\|\mu(C) - \mu(C^{'})\|$

$\leq \frac{|C||C^{'}|}{|C| + |C^{'}|}\|\mu(C) - \mu(C_0)\|\gamma$

Questo è vero per il funzionamento della struttura dati per punti P^j.

Inoltre per il fatto sopra esposto vale che: $|C^{'}| \leq (1+\epsilon)|C_0|$

Sfruttando questa disuguaglianza otteniamo la seguente derivazione:

$\Delta ESS(C, C^{'}) \leq \frac{|C||C^{'}|}{|C| + |C^{'}|}\|\mu(C) - \mu(C_0)\|\gamma = \frac{|C|}{1 + \frac{|C|}{|C^{'}|}}\|\mu(C) - \mu(C_0)\|\gamma$

$\leq \frac{|C|}{1 + \frac{|C|}{(1+\epsilon)|C_0|}}\|\mu(C) - \mu(C_0)\|\gamma = \frac{(1+\epsilon)|C||C_0|}{(1+\epsilon)|C_0| + |C|}\|\mu(C) - \mu(C_0)\|\gamma$

$\leq \frac{(1+\epsilon)|C||C_0|}{|C_0| + |C|}\|\mu(C) - \mu(C_0)\|\gamma =$

$(1+\epsilon)\gamma \min_{B \in D(\gamma,\epsilon)} ESS(C \cup B) - ESS(C) - ESS(B)$

Per comprendere le disuguaglianze ricordare che $\epsilon > 0$. Adesso possono accadere due cose: la quantità $\Delta ESS(C, C^{'})$ è la minima tra tutte le calcolate e pertanto viene restituito $C^{'}$ che rispetta la tesi. Oppure la suddetta quantità non è la minima, pertanto non verrà restituito il cluster $C^{'}$ ma un cluster $C^{''}$ tale che $\Delta ESS(C, C^{''}) \leq \Delta ESS(C, C^{'}) \leq (1+\epsilon)\gamma \min_{B \in D(\gamma,\epsilon)} ESS(C \cup B) - ESS(C) - ESS(B)$ il che conferma la tesi ancora una volta.

□

A seguito di questo risultato è bene precisare da quale funzione e in che modo viene cercato il NN di un cluster in maniera approssimata. La routine che implementa tale approssimazione è la routine «Query(C)» che per un dato cluster C cerca l'ANN(C) (Approximate Nearest Neighbor). Di fatto quando invochiamo «Query(C)» stiamo minimizzando (in maniera approssimata) la quantità $\Delta ESS(C, C_i)$, vista l'implementazione possiamo affermare che l'approssimazione è data dalle strutture dati P^l che eseguono una γ-approximate nearest neighbor per il centroide $\mu(C)$ all'interno di tutte le strutture dati che costituiscono la $D(\gamma, \epsilon)$. Tuttavia (come si vede dal terzo punto del lemma precedente) anche la scelta di ϵ influenza il grado di approssimazione. Il parametro γ influenza il grado di approssimazione della ricerca del NN di un punto, mentre l'approssimazione nella ricerca del NN di un cluster

è influenzata dalla scelta di ϵ e γ. Quindi, scegliendo γ il più possibile vicino ad uno, si riduce l'approssimazione sulla ricerca del NN per un cluster e per un punto, mentre, scegliere ϵ il più possibile vicino a zero, permette di migliorare solo l'approssimazione della ricerca del NN per un cluster. La scelta di questi parametri va ovviamente fatta tenendo in considerazione quello che si vuole ottenere. Non è attualmente chiaro come questi parametri influenzino il clustering ottenuto rispetto a quello che si avrebbe eseguendo l'algoritmo esatto. Tuttavia verranno presentati dei risultati sperimentali che evidenzieranno la bontà del clustering nonostante le approssimazioni.

4.7 L'algoritmo

Definizione 4.9. Si definisce *valore di fusione di due cluster* la quantità: $ESS(A \cup B) - ESS(A) - ESS(B)$

L'algoritmo inizia considerando, come al solito, ogni punto come un singolo cluster, inizializzando anche la struttura dati descritta nella sezione precedente, che rappresenta il cuore dell'algoritmo. Dopodichè, l'algoritmo crea una lista I con un insieme di valori che partizionano l'insieme di tutti i possibili valori di fusione (si veda definizione sopra).

Claim: Dati n punti aventi la coppia a distanza minima con distanza pari ad 1 e la coppia a distanza massima con distanza pari a Δ, si ha che la cardinalità della partizione del numero totale di tutti i possibili valori di fusione (usando un campionamento geometrico) è $O(\log(n\Delta))$.

Dimostrazione. Per dimostrare il claim, prima di tutto, individuiamo un upper bound per la quntità $\Delta ESS(C_i, C_j)$. $\Delta ESS(C_i, C_j) = ESS(C_i \cup C_j) - ESS(C_i) - ESS(C_j) \leq ESS(C_i \cup C_j) = \sum_{x \in C_i \cup C_j}(x - \mu(C_i \cup C_j)^T(x - \mu(C_i \cup C_j)) \leq n\Delta^2$. Per campionare geometricamente un intervallo con estremo destro in $n\Delta^2$ si necessita di un numero di punti pari a $\#punti = \lceil \log_2(n\Delta^2) \rceil \leq 1 + \log_2(n\Delta^2) \leq 2\log_2(n\Delta^2) = 2(\log_2(n) + 2\log_2(\Delta)) \leq 2(2\log_2(n) + 2\log_2(\Delta)) = 4\log_2(n\Delta) \implies \#punti \in O(\log(n\Delta))$ □

Osservazione: Partizionare le distanze di fusione in maniera logaritmica permette di mantenere limitato il numero di iterazioni che vengono svolte dal ciclo «for each» (si veda l'algoritmo qui di seguito).

Ad ogni iterazione è presente un dato clustering (ricordare che cosa si intende con il termine clustering) e l'algoritmo decide quali sono i due prossimi cluster da unire. Tutti i cluster che costituiscono un clustering ad una data iterazione vengono chiamati, con ovvio significato del termine, «unmerged clusters».

Algorithm 9 Ward's method algorithm

Input
$\{x\}_{i=1}^{n}$ con $x_i \in \mathbb{R}^d$

Output: AHC dei punti dati in input con Ward's method

1: Sia L la lista contenente tutti gli «unmerged clusters» (inizialmente contiene tutti i punti)
2: **for each** $\forall \nu \in I$ **do**
3: ToMerge $\leftarrow$ L
4: **while** ToMerge non è vuoto **do**
5: Prendi un cluster C da ToMerge e rimuovilo
6: NN(C) $\leftarrow$ ApproximateNearestNeighbor(C)
7: **if** $ESS(C \cup NN(C)) - ESS(C) - ESS(NN(C)) \leq \nu$ **then**
8: Sia $C^{'} \leftarrow C \cup NN(C)$
9: Rimuovi $NN(C)$ da ToMerge e aggiungi $C^{'}$ a ToMerge
10: Rimuovi $C, NN(C)$ da L e aggiungi $C^{'}$ a L

Osservazione: Notare che alla riga nove dell'algoritmo, il centroide di $\mu(C^{'})$ segue immediatamente dalla formula chiusa del lemma 4.7, perchè si dispone di: $\mu(NN(C)), \mu(C), |C|, |NN(C)|$.

Proposizione 4.3. *Sia $T(n)$ la complessità temporale dell'algoritmo si ha che:* $T(n) \in O(n^{1+f(\sqrt{\gamma})}\epsilon^{-1}\log(n\Delta)\log n)$

Dimostrazione. Il ciclo «for each» viene eseguito un numero di volte pari a $O(\log(n\Delta))$. Ad ogni iterazione vengono eseeguite al più n query e inserimenti nella struttura dati $D(\gamma, \epsilon)$, dato che ognuna di queste ha costo $O(n^{f(\sqrt{\gamma})}\epsilon^{-1}\log n)$ si ha che il costo di una singola iterazione è $O(n^{1+f(\sqrt{\gamma})}\epsilon^{-1}\log n)$. Come già detto vengono eseguite al più $O(\log(n\Delta))$ iterazioni, il che porta a concludere che $T(n) \in O(n^{1+f(\sqrt{\gamma})}\epsilon^{-1}\log(n\Delta)\log n)$ □

Notare che nella complessità temporale dell'algoritmo compare il termine $n^{f(\sqrt{\gamma})}$, il che sottolinea la dipendenza delle performance dell'algoritmo dalle performance della struttura dati che permette di effettuare la ricerca del ANN. Quindi, futuri algoritmi più performanti per la ricerca del ANN di un punto dato porteranno implicitamente miglioramenti a questo algoritmo.

Un'ulteriore osservazione va fatta per chiarire l'utilizzo della lista I. In particolare, quest'ultima, viene introdotta per ridurre il numero di chiamate effettuate alla struttura dati $D(\gamma, \epsilon)$ che altrimenti sarebbe dell'ordine $O(n^2)$. Allo stesso tempo, però, questa porta implicitamente con sè un ulteriore elemento di approssimazione. Infatti, così facendo, non si segue necessariamente l'ordine di merge tra cluster che prevede, ad ogni iterazione, di fare fondere i due cluster più vicini (in senso approssimato o esatto).

NOTA: Arrivati a questo punto si noti che la dimostrazione del teorema 4.1 è stata raggiunta.

Osservazione: Un'ulteriore osservazione da fare riguarda i clustering ottenuti alla fine di ogni iterazione del ciclo «for each». In particolare si potrebbe pensare che alla fine di ogni iterazione, data una qualunque coppia di cluster C, C', valga che $\Delta ESS(C, C') \geq \nu$ con il valore ν relativo alla iterazione considerata. Questo sarebbe vero nel momento in cui la ricerca del nearest neighbor fosse una ricerca esatta e non approssimata. Infatti, supponiamo che l'algoritmo sia giunto ad una data iterazione i-esima del ciclo «for each» e indichiamo il valore di fusione con ν_i (indica appunto il valore ν relativo alla iterazione i-esima) inoltre, supponiamo per assurdo che alla fine della iterazione i-esima esistano due unmerged cluster C, C' tali che $\Delta ESS(C, C') \leq \nu$. Dato che stiamo ipotizzando la ricerca del nearest neighbor esatta, questo implica immediatamente un assurdo in quanto durante l'iterazione i-esima viene sicuramente invocato il metodo «Query(C)» che restituirà C' se C, C' sono a distanza minima, oppure un C'' se C, C' non sono a distanza minima pertanto alla fine della iterazione i-esima la coppia C, C' non può essere una coppia di unmerged cluster. Tuttavia, nell'algoritmo sopra descritto, la ricerca del nearest neighbor non è esatta ma bensì approssimata. In particolare per il lemma 4.8 si sa che il metodo «Query(C)» restituisce un cluster C' tale che $\Delta ESS(C, C') \in [\min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B); (1+\epsilon)\gamma \min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B)]$ il che appunto fa capire che può restituire il cluster a minima distanza ma anche no. Detto questo potrebbe darsi il caso in cui, durante l'iterazione i-esima, esistono due cluster C, C' tali che $\Delta ESS(C, C') \leq \nu$ ma il metodo «Query(C)» restituisce un cluster C'' tale per cui $\Delta ESS(C, C'') \geq \nu$ il che porta l'algoritmo a scartare dalla lista «ToMerge» il cluster C, che non potrà quindi più essere fuso in quell'iterazione. Verrà poi successivamente fatto lo stesso procedimento con il cluster C' e si può giungere alle stesse conclusioni con analoghi ragionamenti. Quindi alla fine dell'iterazione i-esima è possibile che esistano due cluster (unmerged cluster) C, C' tali che $\Delta ESS(C, C') \leq \nu$.

Tuttavia è comunque possibile dare delle garanzie sulla minima distanza di fusione tra una qualunque coppia di cluster alla fine di una data iterazione. Per riuscire a trovare questo limite inferiore, fondamentalmente, basta chiedersi quale è il caso in cui, sicuramente, una coppia di cluster viene unita ad una data iterazione. Siccome ,come già detto, il metodo «Query(C)» ritorna una coppia di cluster tali che $\Delta ESS(C, C') \in [\min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B); (1+\epsilon)\gamma \min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B)]$ per far sì che il cluster C venga sicuramente unito con un altro cluster è sufficiente che esista un altro cluster (che ancora una volta indichiamo con C') tale che $\Delta ESS(C, C') \leq \frac{\nu}{(1+\epsilon)\gamma}$, così facendo a seguito dell'invocazione del metodo «Query(C)» il cluster C verrà fuso con C' nella iterazione i-esima perchè l'estremo destro del intervallo risulterà essere minore o al più uguale a ν. Da quanto detto possiamo enunciare la seguente invariante.

Invariante:
Alla fine di una data iterazione i-esima del ciclo «for each», relativa ad un valore di fusione ν_i, per ogni coppia di unmerged cluster (C, C'), si ha che $\Delta ESS(C, C') \geq \frac{\nu}{(1+\epsilon)\gamma}$

Si riporta di seguito la dimostrazione per essere completi e formali, ma la dimostrazione segue banalmente dai ragionamenti sopra esposti.

Dimostrazione. Supponiamo per assurdo che alla fine di una iterazione i-esima esistano due cluster C, C' tali che $\Delta ESS(C, C') < \frac{\nu}{(1+\epsilon)\gamma}$ questo implica che la quantità $\min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B) \leq \frac{\nu}{(1+\epsilon)\gamma}$. Quindi, quando verrà invocato il metodo «Query(C)», verrà ritornato un cluster C'' (che potrebbe coincidere anche con C') tale che $\Delta ESS(C, C'') \in [\min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B); (1+\epsilon)\gamma \min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B)] \subset [\min_{B \in D(\gamma,\epsilon)} \Delta ESS(C, B); \nu]$ il che permette di concludere che il cluster C verrà fuso necessariamente con un altro cluster durante l'iterazione i-esima, giungendo ad un assurdo. □

Questo risultato ci permette di dire che l'algoritmo, ad ogni iterazione, garantisce di unire una coppia di cluster che sta al più a $\gamma(1+\epsilon)$ volte la distanza tra i due cluster che un algoritmo esatto avrebbe unito come affermato dal teorema 4.1. Tuttavia nessuna garanzia viene data rispetto al dendrogramma finale.

4.8 Risultati sperimentali

Gli esperimenti fatti, riguardano l'algortimo approssimato per Ward's method. Quest'ultimo è stato implementato usando come linguaggio di programmazione C++11 su una CPU con clock a 2.5 Ghz e con otto core su sistema operativo Linux. Nell'implementazione dell'algoritmo si è fatto uso della libreria «FLANN» (Fast Library for Approximate Nearest Neighbor) e un algoritmo di loro implementazione per LSH. Quindi sono stati sviluppati due algoritmi, la cui differenza, sta nel metodo della ricerca del ANN.

1. Il primo algoritmo fa uso della libreria FLANN per la ricerca dell'ANN e si basa sull'utilizzo di strutture dati KD-tree. La procedura presente all'interno della libreria presenta due parametri che controllano l'esecuione dell'algoritmo. Il primo di questi, che è indicato con T, stabilisce il numero di alberi creati, il secondo,invece, tiene conto del numero di foglie visitate prima di fermare l'esecuzione dell'algoritmo e dare una risposta. Quest'ultimo è indicato con L.

2. Il secondo algoritmo utilizza, invece, la tecnica LSH per la ricerca del ANN e anche questo possiede due parametri che controllano l'esecuzione. Il primo parametro, H, controlla il numero di hash function usate e il secondo, r, tiene conto del collision rate.

Gli algoritmi sono stati poi comparati con un algoritmo per Ward's method presente all'interno della libreria «sci-kit learn». Quest'ultima è una libreria di python scritta anche essa in C++. Il parametro principale per controllare l'approssimazione è chiaramente ϵ che influenza anche il numero di strutture dati per punti che andranno a costituire la struttura dati per cluster che precedentemente è stata indicata con $D(\gamma, \epsilon)$. Per vedere i risultati ottenuti per questi algoritmi su dataset reali si rimanda a [7]. Per misurare le performance dei loro algoritmi rispetto a quello esatto della libreria «sci-kit learn», gli autori, si sono concentrati sulla versione Ward-FLANN con un set di parametri che peremette di ottenere il clustering più simile a quello generato dall'algoritmo esatto. I parametri sono i seguenti: $\epsilon = 8$, numero di alberi creati $T = 2$, numero di foglie visitate $L = 10$. I dataset sui quali eseguire gli algoritmi, sono stati creati usando il metodo «blobs» presente all'interno della libreria «sci-kit learn». I dataset sono stati generati in spazi euclidei con dimensioni $d = \{10, 20\}$, con un numero di punti che varia tra 10000 e 20000. Si è osservato poi, che i tempi di esecuzione, per Ward-FLANN e Ward-LSH sono molto simili tra di loro. Nulla è stato detto riguardo ai dendrogrammi prodotti con i due metodi. Nella pagina successiva vengono riportati due grafici a prova di quanto appena affer-

mato. Uno dei problemi di questo recente lavoro, sta nel fatto che manca un modo di misurare la differenza tra due dendrogrammi prodotti da due algoritmi diversi. Pertanto è difficile valutare le differenze del dendrogramma ottenuto mediante questo algoritmo ed un qualunque altro. Nella pagina seguente vengono riportate due immagini che mostrano l'andamento temporale di questo algoritmo rispetto a quello esatto. Le immagini sono state prese da [7].

Nota bene: Nei grafici della pagina seguente App-Ward rappresenta il tempo di esecuzione di Ward-FLANN mentre Ward rappresenta i tempi di esecuzione dell'algoritmo esatto.

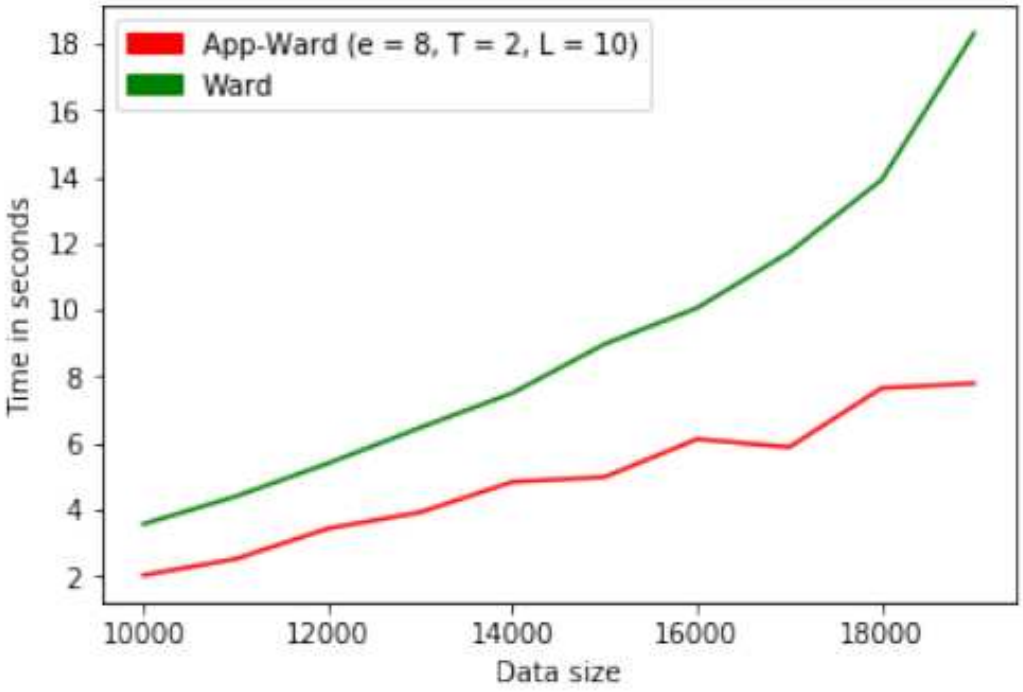

Figura 4.1: $\mathbb{R}^{10}$

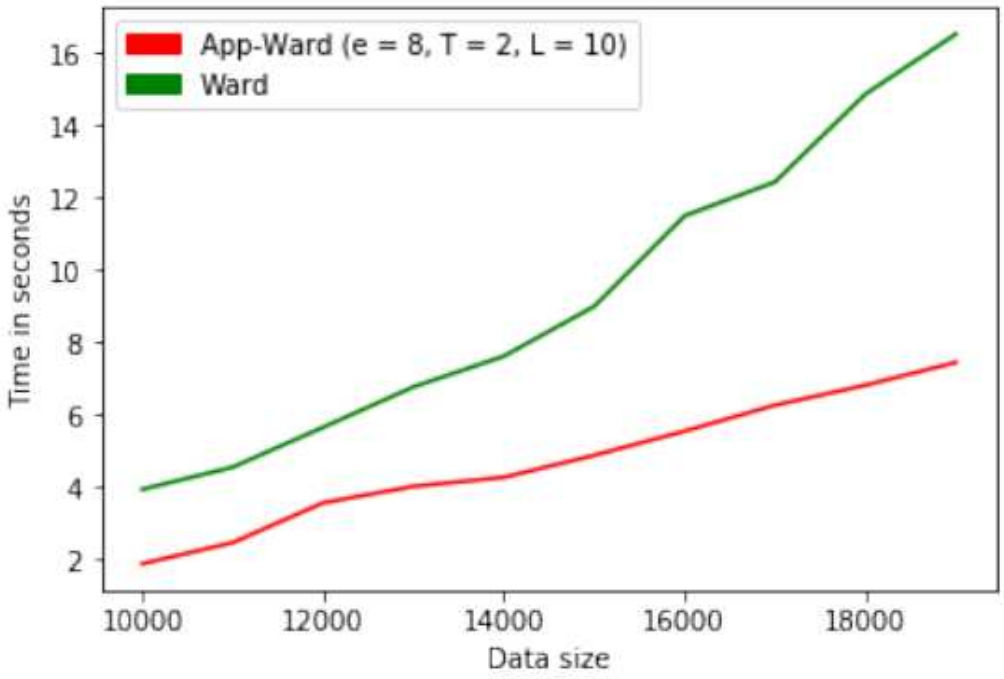

Figura 4.2: $\mathbb{R}^{20}$

5 Bibliografia

[1] M. F. R. R. Hennig C., Meila M., *Handbook of Cluster Analysis*. CRC Press, Boca Ranton,FL,USA,2015.

[2] E. B., *Cluster analysis*. Chichester, West Sussex, U.K: Wiley, 2011.

[3] M. H. G. Michael T. Goodrich, Roberto Tamassia, *Data Structures and Algorithms in Java*, 6th ed. Wiley, 2014.

[4] F. Murtagh, "A survey of recent advances in hierarchical clustering algorithms," *The Computer Journal*, vol. 26, no. 4, p. 6, November 1983.

[5] F. J. Rohlf, "Hierarchical clustering using the minimum spanning tree," *The computer journal*, vol. 16, pp. 93–95, 1973.

[6] J. L. B. e Jerome H. Friedman, "Fast algorithms for constructing minimal spanning trees in coordinate spaces," *IEEE Transactions on Computers*, vol. C-27, pp. 97–105, March 1977.

[7] H. H. Amir Abboud, Vincent Cohen-Addad, "Subquadratic high-dimensional hierarchical clustering," *NIPS'19: Proceedings of the 33rd International Conference on Neural Information Processing Systems*, no. 1039, p. 11580–11590, 2019.

[8] U. J. Rajaraman A., *Mining of Massive Datasets*. Cambridge University Press, 2010.