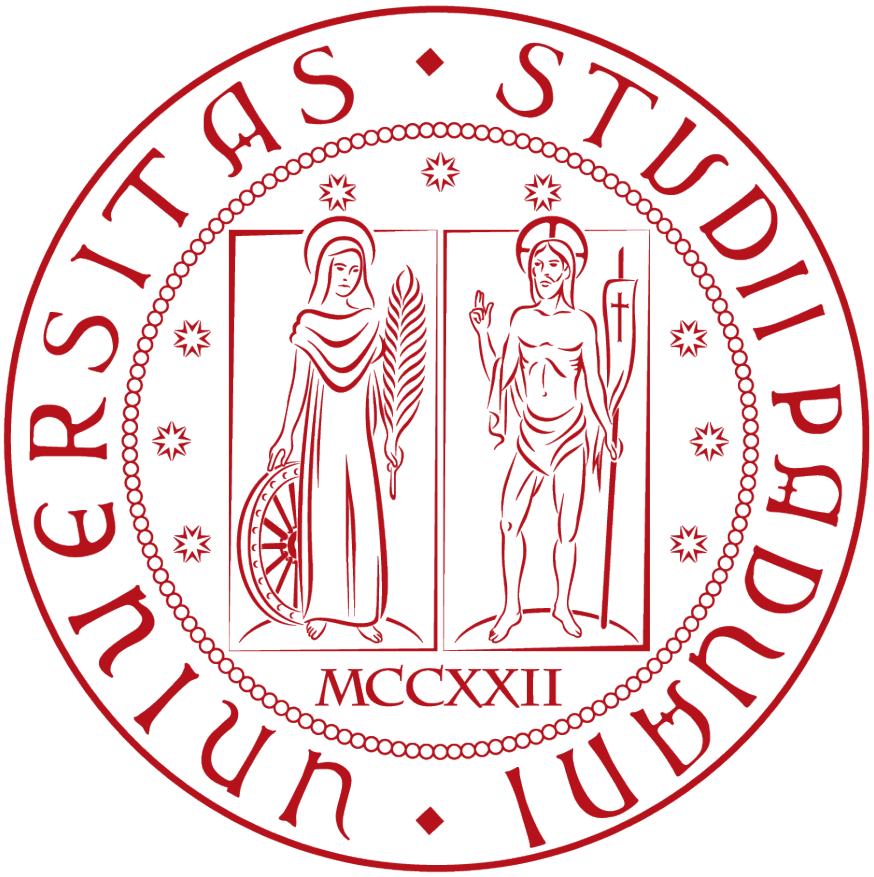

Laboratorio 1 - MST

Redazione	Alessandro Sgreva - 2029003
	Nicola Salvadore - 2026882
Corso	Algoritmi avanzati
Anno accademico	2020-2021
Destinato a	Prof. Davide Bresolin
	Prof. Michele Scquizzato

Descrizione

Relazione sul confronto sperimentale degli algoritmi di Prim, Kruskal Naive e Kruskal con Union-find, per la risoluzione del problema di Minimum Spanning Tree (MST).

Sommario

1 Introduzione

In questa sezione verranno descritti i tre algoritmi implementati. Tali algoritmi sono:

- **algoritmo di Prim** implementato con Fibonacci Heap;
- **algoritmo di Kruskal** nella sua implementazione "naive";
- **algoritmo di Kruslal** implementato con Union-Find.

1.1 Algoritmo di Prim

Algoritmo per la determinazione del Minimum Spanning Tree a partire da un grafo non orientato e con pesi non negativi. Viene definito come:

- *algoritmo greedy*: perchè valuta di volta in volta le soluzioni localmente migliori, senza mettere in discussione le scelte precedenti;
- *algoritmo esatto*: perchè fornisce una soluzione precisa per ogni istanza del problema, senza effettuare arrotondamenti, approssimazioni o imprecisioni di altra natura;
- *algoritmo ottimo*: perchè presenta la soluzione migliore o una delle soluzioni migliori.

Viene generalmente fonita in input una *lista di adiacenza*, ovvero un array in cui ogni posizione corrisponde ad un vertice, il quale punta ad una generica lista concatenata contenente tutti i vertici adiacenti al vertice considerato. Si assume inoltre che ogni vertice possieda i campi:

- `key[v]`: valore associato al vertice;
- π`[v]`: puntatore al padre del vertice nel Minimum Spanning Tree.

I passi dell'algoritmo sono i seguenti:

1. inizialmente si pongono tutti i campi `key[v]` a $+\infty$ e tutti i campi π`[v]` a `NIL`;
2. si prende il vertice fornito in input come radice dell'albero e si pone la sua chiave a 0;
3. si inseriscono tutti i vertici rimasti in un MinHeap (nell'implementazione qui proposta) e li si estrae in ordine crescente;
4. si scorre quindi la lista delle adiacenze del vertice estratto `u`, considerando solo i vertici `v` ancora all'interno del MinHeap precedente;
5. per ognuno di essi tale che la sua distanza dal vertice `u` sia la minore tra tutti quelli considerati, si pone π`[v]` uguale a `u` (inserendo di fatto `v` nel Minimum Spanning Tree;
6. si conclude il ciclo aggiornando il campo `key[v]` con il valore della distanza tra `u` e `v`.

La complessità dell'implementazione dell'algoritmo con Fibonacci Heap è: $O(E * log(V))$. Dove E è il numero di archi e V è il numero di vertici.

1.2 Algoritmo di Kruskal

Algoritmo per la determinazione del Minimum Spanning Tree a partire da un grafo non orientato e con pesi non negativi. Viene definito come:

- *algoritmo ottimo*: perchè presenta la soluzione migliore o una delle soluzioni migliori;
- veloce tanto quanto l'algoritmo di Prim, se implementato propriamente.

L'idea di base dell'algoritmo è quella di gestire una foresta di alberi disgiunti tra loro in modo tale da selezionare solo gli archi di peso minimo, che collegano tra loro questi alberi. Tutti gli alberi della foresta devono essere aciclici. Dato quindi $G = (V, E)$ un grafo connesso, non orientato e pesato (con V ed E definite come sopra), si consideri:

- X come sottinsieme degli archi E;
- X è inizialmente vuoto;
- X è sottoinsieme di qualche Minimum Spanning Tree;
- si considera $G_x = (V, X)$ un grafo contenente tutti i vertici di G;
- inizialmente G_x non contiene archi, la foresta è quindi formata da $|V|$ alberi disgiunti.

Durante l'esecuzione, l'algoritmo ricerca un arco sicuro da aggiungere alla foresta. Ricerca, fra tutti gli archi che collegano due alberi qualsiasi nella foresta, un arco (u, v) di peso minimo. Viene definita quindi come *invariante* del ciclo:

- finchè G_x non sarà composta da un solo albero, ogni albero contenuto in G_x sarà aciclico e G_x sarà un sottoinsieme di qualche Minimum Spanning Tree.

Una generica implementazione dell'algoritmo è così descritta:

1. viene creata una foresta di grafi;
2. vengono ordinati tutti gli archi del grafo in ordine crescente;
3. vengono valutati gli archi ordinati, uno per volta, per essere inseriti (se opportuno) nella foresta;
4. un arco viene aggiunto alla foresta se: è sicuro, collega due alberi disgiunti e non genera cicli (sempre vero se verificata la seconda condizione).

La complessità dell'algoritmo dipende naturalmente dall'implementazione:

- **implementazione Naive**: ha complessità $O(V * E)$;
- **implementazione con Union-Find**: ha complessità $O(E * log(V))$.

2 Scelte implementative

In questa sezione verranno discusse le principali scelte implementative e le relative motivazioni.

2.1 Linguaggio di programmazione

Con riferimento all'esercitazione svolta in classe è stato deciso di svolgere l'attività di Laboratorio utilizzando il linguaggio Python.

2.2 Strutture dati per archi, vertici e grafi

Di seguito viene descritto come sono modellate la classi *Arc (arco)*, *Vertex (vertice)* e *Graph (grafo)*.
In aggiunta alle strutture, è presente anche un metodo `graph_generator`, utilizzato per generare un grafo a partire da un file `.txt` fornito in input.

2.2.1 Arc

Possiede quattro campi dati:

- **vert1**: che rappresenta il primo vertice dell'arco;
- **vert2**: che rappresenta il secondo vertice dell'arco;
- **weight**: che rappresenta il peso associato all'arco;
- **label**: che rappresenta l'etichetta assegnata all'arco durante l'esecuzione degli algoritmi.

I suoi metodi sono:

- **weight()**: utilizzato per ottenere l'accesso al peso dell'arco;
- **opposite(s)**: utilizzato per ottenere il vertice opposto a quello fornito (s).

2.2.2 Vertex

Possiede cinque campi dati:

- **key**: che rappresenta il valore associato al vertice (`default = 'inf'`);
- **parent**: che rappresenta il vertice "genitore";
- **id**: che rappresenta l'identificativo associato al vertice;
- **flag**: che indica se il vertice è già stato estratto dallo spanning tree;
- **size**: che indica il numero di vertici presenti nel sotto-albero di cui è radice (utilizzato nell'Union-find).

Il suo unico metodo è:

- **is_root()**: funzione di utility per l'Union-find, che definisce se un vertice è una root.

2.2.3 Graph

Possiede cinque campi dati:

- **arch_list**: che rappresenta la lista degli archi contenuti nel grafo;
- **vert_list**: che rappresenta la lista dei vertici contenuti nel grafo;
- **n_vertex**: che indica il numero totale di vertici attualmente nel grafo;
- **n_arches**: che indica il numero totale di archi attualmente nel grafo;
- **adj**: array che rappresenta le liste di adiacenza per tutti i vertici del grafo. Ovvero una lista di tutti i vertici ad essi adiacenti con relativo peso.

I suoi metodi sono:

- **add(v1, v2, w)**: utilizzato per aggiungere un arco al grafo, aggiornando le liste di adiacenza e la lista degli archi;
- **pop()**: utilizzato per rimuovere l'ultimo arco nella lista degli archi, aggiornando liste di adiacenza e vertici relativi;
- **graph_total_weight**: utilizzato per ottenere la somma dei pesi di tutti gli archi del grafo.

Sono presenti inoltre altri metodi relativi alla struttura Union-find, che verranno descritti in seguito, nella sezione dedicata all'implementazione di tale struttura.

2.3 Implementazione del Fibonacci Heap

Fibonacci Heap é una particolare implementazione dell'heap, che mantiene una lista di radici ognuna delle quali deve avere rango diverso. Questa struttura ha un comportamento lazy. L'inserimento di un valore avviene in tempo $O(1)$, perché questo viene inserito direttamente nella lista di grado 0. Il consolidamento della struttura dati, che si occupa di mantenere la proprietà della struttura (cioè che un nodo può avere rango al massimo $log(V)$), é avviato solamente quando viene invocato il metodo `extract_min`. La proprietà citata, come il fatto che due radici non possono avere lo stesso rango, non sono quindi esattamente invarianti, perché é mantenuta solamente durante l'esecuzione di un metodo.
L'implementazione della struttura Fibonacci Heap è stata sviluppata con riferimento alla fonte qui riportata: https://github.com/ivan-ristovic/fheap/blob/master/fheap.py.
Tale implementazione è stata esaminata, corretta e ri-elaborata per le esigenze del progetto di Laboratorio.

2.3.1 Node

Questa classe descrive i nodi dell'heap. Contiene i seguenti attributi:

- **value**: rappresenta il valore del nodo sulla quale sono eseguiti tutti i confronti. Sono supportati solo valori numerici;
- **id**: riferimento (tramite numero intero all'indice della tabella nell'algoritmo di Prim) al vertice del grafo;

- **parent/child/left/right**: puntatori ai nodi nell'heap, rispettivamente il padre, il figlio "preferito", e ai suoi fratelli a destra e a sinistra. La lista dei fratelli é circolare, cioè non esiste una testa;
- **deg**: é il grado del nodo, cioè il numero di sottoalberi a cui é collegato;
- **mark**: é un valore booleano utilizzato per marcare i nodi in alcuni metodi.

2.3.2 FibonacciHeap

Questa classe rappresenta la struttura Fibonacci Heap. Contiene i seguenti campi dati principali:

- **root_list**: è un puntatore che mantiene un riferimento al lista delle radici;
- **min_node**: è un riferimento al nodo con `value` minore nell'heap (che si trova nella lista delle radici);
- **total_num_elements**: è il numero di elementi totali di nodi nell'heap.

I suoi metodi principali (escludendo quindi quelli a loro supporto) sono:

- **insert(node)**: funzione che inserisce un nuovo nodo nella lista di radici dell'heap e aumenta il counter di elementi all'interno dello stesso;
- **extract_minimum()**: funzione che Estrae il nodo con chiave minore e consolida l'heap. Per fare questo inserisce i figli del `min_node` nella lista delle radici e rimuove il nodo eliminando i suoi riferimenti. Infine viene chiamato il metodo `consolidate` che verifica che nessuna radice abbia lo stesso grado, nel caso contrario unisce i due alberi inserendo quello con key maggiore come figlio dell'altro. In questo metodo si garantisce, inoltre, che la lunghezza della lista delle radici sia al massimo logaritmica rispetto al numero di elementi nell'heap;
- **decrease_key(node, value)**: funzione che assegna un nuovo valore `value` a un nodo dell'heap , mantenendo l'invariante dello stesso per il quale ogni nodo padre é minore dei nodi figli. Il nodo viene subito aggiornato, viene effettuato il `cut` del nodo, inserendolo nella lista delle radici e marcato false. Viene poi eseguito un `cascading_cut` sul padre del nodo precedente cioè: se il padre di questo non é una radice viene marcato (`mark = True`), nel caso contrario viene tagliato e si esegue un `cascading_cut` su suo padre, fino ad arrivare alla lista delle radici o a un nodo con padre con `mark = False`. Tornando al nodo aggiornato, viene controllato se la sua nuova value é minore del `min_value` attuale e nel caso quest'ultimo viene aggiornato. Il tempo di esecuzione di questo metodo é ammortizzato a costante dato che dipende dal numero di sottoalberi del nodo aggiornato.

2.4 Implementazione del Merge sort

La realizzazione dell'algoritmo di Kruskal ha richiesto l'implementazione di un algoritmo di Merge Sort in complessità $O(n * log(n))$, per l'ordinamento degli archi dei grafi in base al loro peso. Tale implementazione è avvenuta in maniera semplice e standard.

2.5 Implementazione del DFS

La realizzazione dell'algoritmo di Kruskal, in versione Naive, ha richiesto l'implementazione di un algoritmo DFS, per il controllo sulla ciclicità del grafo. Dato il numero limitato di possibili esecuzioni ricorsive, il gruppo ha optato per l'implementazione di una sua versione iterativa, con l'impiego di liste di adiacenza e con complessità $O(E + V)$.
Allo scopo di ridurre per quanto possibile il tempo di esecuzione dell'algoritmo DFS, e quindi il suo impatto sull'algoritmo di Kruskal Naive, è stato deciso di interrompere la sua esecuzione non appena venga identificato un ciclo.
Inoltre, quest'implementazione dell'algoritmo DFS non effettua un analisi di tutte le componenti connesse del grafo, in quanto (come verrà trattato in seguito) ai fini dell'algoritmo di Kruskal Naive questo non è necessario.

2.6 Implementazione dell'Union-find

La Union-find è una struttura dati per la gestione di insiemi disgiunti di oggetti. Le operazioni da essa supportate sono:

- **initialize**: funzione che dato un array di oggetti, crea una struttura dati Union-find, con ogni oggetto presente in tale array;
- **find**: funzione che dato un oggetto x, ritorna il nome del set che contiene x;
- **union**: funzione che dati due oggetti x, y unisce i set che contengono x e y nel singolo set (solo se i due oggetti non sono già sullo stesso set).

Le rispettive complessità sono:

- **initialize**: $O(n)$;
- **find**: $O(log(n))$;
- **union**: $O(log(n))$.

Dove n è il numero di oggetti nella struttura dati.
Nel progetto la struttura dati è stata implementata tramite l'introduzione di un attributo `parent`, inizializzato con l'id del vertice stesso, all'interno della classe *Vertex*. Le tre operazioni sono state implementate come metodi della classe *Graph*. A supporto di tali operazioni è stato introdotto nella classe *Vertex* un campo aggiuntivo `size`, che indica il numero di vertici presenti nel sotto-albero di cui tale vertice è radice, ed un metodo `is_root` che definisce se un vertice è una root.

3 Risultati ottenuti

In questa sezione verranno descritti i risultati ottenuti, corredati da grafi esplicativi e risposte alle domande fornite in consegna.

3.1 Domanda 1 - Risultati

I risultati relativi ai tempi di esecuzione sono stati ottenuti utilizzando la funzione `time.perf_counter_ns()`, che fornisce la misurazione del tempo in nanosecondi. Il garbage collector é stato disabilitato durante l'esecuzione degli algoritmi.
Per realizzare il grafico del tempo di esecuzione effettivo in rapporto alla complessità, sono stati misurati i tempi di esecuzione per i quattro grafi forniti ad ogni dimensione di input, per poi calcolare un loro valore medio da inserire nel grafico.
In aggiunta, per i grafi da meno di 1000 vertici sono stati misurati i tempi di 100 esecuzioni dei singoli algoritmi e calcolata una media, così da rendere più affidabili i risultati.
É stato aggiunto un grafico di riferimento assieme a quello dei tempi, realizzato calcolando la complessità attesa moltiplicata a una costante. La costante é stata calcolata dalla media tra tutti i valori, ottenuti dal seguente calcolo:

$$TempoPerInput(x)/ComplessitaPerInput(x)$$

Ad esempio per l'algoritmo di Prim si é calcolato per ogni misurazione su grafi con V numero di vertici e E numero di archi (medi) il valore $E*log(V)$. Qualora il grafico dei tempi effettivi risulti entro i limiti della complessità asintotica di riferimento (linea sempre inferiore al riferimento), questi verrà ritenuto accettabile

3.1.1 Tabella risultati

I risultati grezzi ottenuti, senza nessuna rielaborazione sono quindi i seguenti:

Tabella 3.1: Tabella dei risultati

Numero di vertici	Numero di archi	Prim	Kruskal Naive	Kruskal	Somma pesi MST
10	9	$3.80e+06$	$2.90e+06$	$2.19e+06$	29316
10	11	$4.26e+06$	$2.76e+06$	$2.31e+06$	16940
10	13	$4.89e+06$	$3.37e+06$	$2.52e+06$	-44448
10	10	$4.37e+06$	$3.37e+06$	$2.30e+06$	25217
20	24	$1.03e+07$	$8.60e+06$	$5.21e+06$	-32021

Tabella 3.1: (continua)

Numero di vertici	Numero di archi	Prim	Kruskal Naive	Kruskal	Somma pesi MST
20	24	$1.01e+07$	$9.38e+06$	$5.34e+06$	25130
20	28	$1.11e+07$	$1.06e+07$	$5.85e+06$	-41693
20	26	$1.06e+07$	$8.91e+06$	$1.18e+07$	-37205
40	56	$2.52e+07$	$2.10e+07$	$1.14e+07$	-114203
40	50	$2.24e+07$	$2.45e+07$	$1.15e+07$	-31929
40	50	$2.26e+07$	$2.33e+07$	$1.12e+07$	-79570
40	52	$2.46e+07$	$2.27e+07$	$1.13e+07$	-79741
80	108	$5.49e+07$	$8.47e+07$	$2.48e+07$	-139926
80	99	$5.60e+07$	$9.03e+07$	$2.38e+07$	-198094
80	104	$5.57e+07$	$8.12e+07$	$2.44e+07$	-110571
80	114	$5.74e+07$	$1.13e+08$	$2.69e+07$	-233320
100	136	$7.01e+07$	$1.43e+08$	$3.25e+07$	-141960
100	129	$7.27e+07$	$1.12e+08$	$3.10e+07$	-271743
100	137	$7.12e+07$	$9.28e+07$	$3.16e+07$	-288906
100	132	$7.51e+07$	$1.11e+08$	$3.18e+07$	-229506
200	267	$1.53e+08$	$4.10e+08$	$6.82e+07$	-510185

Tabella 3.1: (continua)

Numero di vertici	Numero di archi	Prim	Kruskal Naive	Kruskal	Somma pesi MST
200	269	$1.55e+08$	$3.79e+08$	$6.74e+07$	-515136
200	269	$1.48e+08$	$3.79e+08$	$6.73e+07$	-444357
200	267	$1.55e+08$	$4.82e+08$	$6.84e+07$	-393278
400	540	$3.35e+08$	$1.53e+09$	$1.48e+08$	-1119906
400	518	$3.16e+08$	$1.34e+09$	$1.45e+08$	-788168
400	538	$3.28e+08$	$1.40e+09$	$1.46e+08$	-895704
400	526	$3.21e+08$	$1.51e+09$	$1.44e+08$	-733645
800	1063	$6.79e+08$	$5.82e+09$	$3.07e+08$	-1541291
800	1058	$6.86e+08$	$5.62e+09$	$3.09e+08$	-1578294
800	1076	$6.92e+08$	$5.41e+09$	$3.12e+08$	-1664316
800	1049	$6.86e+08$	$5.79e+09$	$3.13e+08$	-1652119
1000	1300	$8.70e+06$	$8.44e+07$	$4.47e+06$	-2089013
1000	1313	$8.95e+06$	$9.53e+07$	$4.62e+06$	-1934208
1000	1328	$8.89e+06$	$9.43e+07$	$4.60e+06$	-2229428
1000	1344	$8.83e+06$	$9.36e+07$	$4.56e+06$	-2356163
2000	2699	$1.91e+07$	$3.85e+08$	$9.66e+06$	-4811598

Tabella 3.1: (continua)

Numero di vertici	Numero di archi	Prim	Kruskal Naive	Kruskal	Somma pesi MST
2000	2654	$1.84e+07$	$3.74e+08$	$9.52e+06$	-4739387
2000	2652	$1.86e+07$	$3.64e+08$	$9.48e+06$	-4717250
2000	2677	$1.85e+07$	$3.80e+08$	$9.83e+06$	-4537267
4000	5360	$3.88e+07$	$1.52e+09$	$2.09e+07$	-8722212
4000	5315	$3.84e+07$	$1.54e+09$	$2.27e+07$	-9314968
4000	5340	$3.90e+07$	$1.42e+09$	$2.21e+07$	-9845767
4000	5368	$3.89e+07$	$1.58e+09$	$2.14e+07$	-8681447
8000	10705	$8.27e+07$	$6.17e+09$	$4.48e+07$	-17844628
8000	10670	$8.21e+07$	$5.95e+09$	$4.80e+07$	-18798446
8000	10662	$8.11e+07$	$6.47e+09$	$4.80e+07$	-18741474
8000	10757	$8.24e+07$	$7.32e+09$	$4.61e+07$	-18178610
10000	13301	$1.03e+08$	$1.11e+10$	$6.16e+07$	-22079522
10000	13340	$1.06e+08$	$1.00e+10$	$7.11e+07$	-22338561
10000	13287	$1.04e+08$	$9.48e+09$	$6.23e+07$	-22581384
10000	13311	$1.04e+08$	$9.81e+09$	$6.60e+07$	-22606313
20000	26667	$2.18e+08$	$4.42e+10$	$1.46e+08$	-45962292

Tabella 3.1: (continua)

Numero di vertici	Numero di archi	Prim	Kruskal Naive	Kruskal	Somma pesi MST
20000	26826	$2.21e+08$	$4.55e+10$	$1.47e+08$	-45195405
20000	26673	$2.24e+08$	$4.46e+10$	$1.43e+08$	-47854708
20000	26670	$2.25e+08$	$4.73e+10$	$1.25e+08$	-46418161
40000	53415	$4.77e+08$	$2.71e+11$	$3.14e+08$	-92003321
40000	53446	$4.90e+08$	$2.70e+11$	$2.99e+08$	-94397064
40000	53242	$4.95e+08$	$2.80e+11$	$3.04e+08$	-88771991
40000	53319	$4.82e+08$	$2.81e+11$	$3.03e+08$	-93017025
80000	106914	$9.90e+08$	$1.35e+12$	$6.78e+08$	-186834082
80000	106633	$1.00e+09$	$1.26e+12$	$6.70e+08$	-185997521
80000	106586	$1.04e+09$	$1.06e+12$	$6.70e+08$	-182065015
80000	106554	$1.02e+09$	$1.01e+12$	$6.66e+08$	-180793224
100000	133395	$1.29e+09$	$1.75e+12$	$8.16e+08$	-230698391
100000	133214	$1.29e+09$	$1.76e+12$	$8.36e+08$	-230168572
100000	133524	$1.31e+09$	$1.77e+12$	$8.38e+08$	-231393935
100000	133463	$1.30e+09$	$1.74e+12$	$8.46e+08$	-231011693

Com'è visibile nella tabella, i pesi dei MST risultanti dai tre differenti algoritmi sono coincidenti. Questo, combinato con le pre e le post condizioni, ci ha confermato la correttezza dei tre algoritmi.

3.1.2 Risultato di Prim

Figura 3.1.1: Grafico dei tempi misurati per l'algoritmo di Prim.

Come visibile dal grafico, i tempi misurati risultano sempre inferiori alla linea di riferimento per il caso peggiore, calcolata utilizzando complessità relativa all'implementazione di Prim con Fibonacci Heap: $O(E * log(V))$. La coincidenza dell'andamento delle due linee ci dà ulteriore conferma della corenza dell'implementazione nei confronti della complessità asintotica teorica prevista.

Nota: in un test precedente di tale implementazione era stata inserita una complessità $O((V+E) * log(V))$, con erroneo riferimento ad un'implementazione con heap generico (binario). Tale errore di percorso ci ha dato la possibilità di osservare direttamente la differenza di complessità tra implementazione con Fibonacci Heap e Heap binario all'interno del grafico, la cui linea di riferimento possedeva un andamento decisamente diverso rispetto a quella di misurazione.

3.1.3 Risultato di Kruskal Naive

Figura 3.1.2: Grafico dei tempi misurati per l'algoritmo di Kruskal Naive.

Come visibile dal grafico, i tempi misurati risultano sempre inferiori alla linea di riferimento per il caso peggiore, calcolata utilizzando complessità relativa all'implementazione Naive di Kruskal: $O(E * V)$. Come per l'algoritmo precedente, la coincidenza dell'andamento delle due linee ci dà ulteriore conferma della corenza dell'implementazione nei confronti della complessità asintotica teorica prevista.

3.1.4 Risultato di Kruskal con Union-find

Figura 3.1.3: Grafico dei tempi misurati per l'algoritmo di Kruskal con Union-find.

Come visibile dal grafico, i tempi misurati risultano sempre inferiori alla linea di riferimento per il caso peggiore, calcolata utilizzando complessità relativa all'implementazione con Union-find di Kruskal: $O(E * log(V))$. Come per gli algoritmi precedenti, la coincidenza dell'andamento delle due linee ci dà ulteriore conferma della corenza dell'implementazione nei confronti della complessità asintotica teorica prevista.

3.2 Domanda 2 - Comparazione algoritmi

Figura 3.2.1: Grafico di comparazione tra i tempi di esecuzione dei tre algoritmi.

Come è possibile notare, la differenza tra i tempi di esecuzione dei vari algoritmi per input di grande dimensione è molto ampia. Ad esempio per l'esecuzione di Kruskal Naive su grafi di input massimo (100000 vertici) il tempo richiesto è di più di un'ora per grafo, mentre per gli algoritmi di Prim e Kruskal con Union-find i risultati spaziano sempre nell'intervallo dai 2 ai 4 secondi per grafo. Questo è naturalmente legato alla diversa complessità asintotica tra il primo algoritmo ($O(V * E)$) ed i seguenti ($O(E * log(V))$), il che rende persino difficile notare la differenza di tempi misurati per il secondo e terzo algoritmo.

Per rendere più chiara questa differenza, di seguito viene presentato un grafico di comparazione dei due algoritmi più rapidi:

Figura 3.2.2: Grafico di comparazione tra i tempi di esecuzione di Prim e Kruskal con Union-find.

Pur essendo caratterizzati dalla stessa complessità asintotica, è possibile notare come l'algoritmo di Kruskal con Union-find sia chiaramente il più rapido. Considerando in particolare anche il basso peso in memoria di tale struttura, in confronto al Fibonacci Heap impiegato da Prim, è senza dubbio possibile definire tale algoritmo come il più efficiente dei tre esaminati.

4 Scelte tecniche

In questa sezione verranno descritte le scelte tecniche e le ottimizzazioni introdotte dal gruppo all'interno dell'elaborato.

4.1 Liste di adiacenza

Allo scopo di garantire ridurre il tempo computazionale dei tre algoritmi, la struttura *Graph* è stata dotata di liste di adiacenza per tutti i vertici. All'interno di tali liste sono state inserite delle tuple di valori contenenti l'indice del vertice adiacente e il peso dell'arco corrispondente: `(id, peso)`.

Nello specifico l'implementazione è stata realizzata tramite strutture dati `set()`. Queste garantiscono non solo l'unicità dei valori contenuti, ma anche una funzione di rimozione in tempo $O(1)$, particolamente utile in Kruskal Naive dove gli archi che generano un ciclo devono venire scartati e rimossi dalla lista.

4.2 Salto degli archi di self-loop

Per ridurre il tempo computazionale degli algoritmi di Kruskal e garantire valori entro i limiti stabiliti dalla sua complessità, è stato introdotto un controllo preventivo sugli archi da inserire. Infatti, qualora l'arco da inserire sia di evidente self-loop `Arc(Vertex(x), Vertex(x))`, l'aggiunta di quest'ultimo ed il controllo del grafo tramite DFS verranno saltati interamente, essendo infatti impossibile che un MST contenga archi di tale tipologia. Comportando quindi nel complesso un risparmo di tempo considerevole.

4.3 Analisi delle componenti connesse - DFS

Nonostante la maggior parte delle implementazioni dell'algoritmo DFS su interi grafi preveda un'analisi di tutte le componenti connesse del grafo, per verificarne l'eventuale ciclicità, nell'implementazione dell'algoritmo di Kruskal è stato deciso di svolgere tale analisi solo sulla componente connessa relativa all'ultimo arco aggiunto al grafo.

In un contesto standard, infatti, l'algoritmo di DFS riceve in input un intero grafo sconosciuto, di cui non si conoscono le caratteristiche delle varie componenti connesse, che potrebbero essere cicliche o non esserlo. Nel caso dell'implementazione dell'algoritmo di Kruskal, invece, si effettuano aggiunte successive di archi a partire da un grafo vuoto, dopo ognuna delle quali viene effettuato il controllo di ciclicità. Prima di ogni aggiunta possiamo quindi dichiarare di conoscere le caratteristiche delle varie componenti connesse attualmente presenti nel grafico, in quanto già testate dalle iterazioni precedenti.

Nello specifico, dopo l'aggiunta di un arco al grafo è possibile solo uno dei tre seguenti scenari:

- **l'arco non viene aggiunto ad alcuna componente connessa esistente**: tale aggiunta non modifica quindi le componenti connesse pre-esistenti, che quindi possiamo già garantire come non cicliche. È quindi sufficiente verificare l'eventuale ciclicità della nuova componente connessa appena generata;
- **l'arco viene aggiunto ad una componente connessa esistente**: la componente connessa alla quale l'arco è stato aggiunto potrebbe essere diventata ciclica, ed è quindi opportuno verificare tale eventualità. Non è comunque necessario verificare la ciclicità delle altre componenti del grafo, in quanto non impattate da tale aggiunta;

- **l'arco connette due componenti connesse pre-esistenti**: la connessione da parte dell'arco di due componenti connesse trasforma queste ultime in un'unica componente connessa, facendoci di fatto ritornare ad una situazione non dissimile da quella precedentemente citata. È quindi sufficiente, allo stesso modo, verificare la ciclicità solo della componente connessa appena formata.

Nel complesso il risparmio sulla quantità di esecuzioni dell'algoritmo DFS si è rivelato essere un elemento importante per ridurre il tempo operazionale dell'algoritmo di Kruskal Naive.

5 Valutazione finale

Dopo aver attentamente esaminato le misurazioni ed i grafici ottenuti dall'esecuzione dei tre algoritmi, possiamo certamente concludere che l'algoritmo di Kruskal con Union-find è il più efficiente di quelli testati.

Come già menzionato, il gruppo ha effettuato un approfondito lavoro di ottimizzazione non solo dei tre algoritmi principali, ma anche delle strutture dati e degli algoritmi ausiliari impiegati.

Nello specifico, la struttura dati sicuramente più impegnativa da realizzare è stata sicuramente il Fibonacci Heap, per poter ottenere un algoritmo di Prim con complessità $O(E * log(V))$. Varie guide di implementazione ricercate sul Web, infatti, riportavano descrizioni di implementazione o snippet di codice generalmente corretti, ma con bug funzionali piuttosto gravi al momento dell'utilizzo (es. loop infiniti sull'iterazione della root list). È stato quindi lavoro del gruppo correggere tali bug e inconsistenze di complessità (es. $O(n)$ nell'operazione di `extract_min` invece di $O(log(n))$).

Per quanto riguarda l'algoritmo di Kruskal Naive, senza dubbio il tempo di esecuzione per gli input più grandi è stato un ostacolo problematico a test multipli ed esaustivi dell'algoritmo su tutto il dataset. Ad incidere su tale valore è sicuramente anche il linguaggio di programmazione Python.

L'implementazione della struttura Union-find per l'algoritmo di Kruskal è stato l'impegno meno problematico dell'intero progetto, richiedendo semplicemente l'introduzione di alcuni metodi alle strutture già esistenti e ridotte modifiche strutturali.