Studio sperimentale di algoritmi MST

Arianna Tusi

Giugno 2021

1 Introduzione

1.1 Problema affrontato

Il lavoro svolto è incentrato sull'implementazione di algoritmi per determinare un minimo albero ricoprente (Minimum Spanning Tree). Tale problema consiste nel trovare un sottografo T di un grafo G=(V,E) non orientato, connesso e pesato sugli archi tale che:

- T sia un albero
- T contenga tutti i veritici di G
- abbia un costo minimo

1.2 Algoritmi considerati e obiettivi

Esistono vari algoritmi che si occupano, dato un grafo, di individuare un suo MST. Questo progetto è focalizzato sull'algoritmo di Prim e l'algoritmo MST a tempo lineare.
Questo studio ha avuto due fasi fondamentali:

- **Fase di implementazione**: in questo stadio sono state realizzate le implementazione dei due algoritmi (utilizzando Python 3.9 come linguaggio di programmazione e PyCharm come IDE).
- **Fase di test**: è stato svolto uno studio sperimentale su quanto creato nella fase precedente cosi da valutarne il comportamento dal punto di vista temporale, sia ad alto livello (numero di istruzioni dominanti) che a basso livello (tempo di CPU) ed infine è stata analizzata la qualità della soluzione fornita dagli algoritmi implementati.

2 Fase di implementazione

2.1 Algoritmo di Prim

L'algoritmo di Prim calcola sempre il minimo albero ricoprente per ogni grafo in input, per questo motivo viene considerato un algoritmo ottimo/esatto. É un algoritmo di visita, che partendo da un nodo iniziale (scelto arbitrariamente) esamina

tutti i nodi del grafo, ponendo in una coda con priorità H l'insieme dei nodi da visitare. Durante un'iterazione, viene estratto il nodo u con peso minimo da H; successivamente viene scelto l'arco incidente su u con peso minore ed infine tale arco viene posto all'interno dell'MST. L'algoritmo termina quando la coda con priorità risulta vuota.

Per realizzare quest'implementazione[1] è stata utilizzata come coda con priorità gli `Heap di Fibonacci` in quanto è considerata un'ottima implementazione di Heap visto che le operazioni più costose hanno una complessità $\Theta(\log(n))$.

In questo algoritmo sono state usate tre operazioni fondamentali dell'heap di Fibonacci:

- `ExtractMin()` $T_{am}(n) = \Theta(\log(n))$

  Questa operazione esegue due sottoperazioni: estrae (elimina) il nodo minimo e realizza la ristruttazione dell'albero. Per realizzare quest'ultima operazione si utilizza la procedura `Consolidate` che itera con i passi definiti di seguito fino a quando ogni radice nella lista delle radici ha grado diverso:

  - Trova due radici x e y nella lista delle radici con lo stesso grado, dove $key[x] \leq key[y]$
  - Collega y a x, rimuove y dalla lista delle radici e lo rende figlio di x. Questa operazione è realizzata dalla procedura `Heap-Link`.

- `Insert(nodo v)` $T_{am}(n) = \Theta(1)$

  In questo caso il nodo v diventa un albero a nodo singolo con chiave e valore, avente i campi *degree[v] = 0* e *mark = False*.

- `DecreaseKey(nodo v, chiave d)` $T_{am}(n) = \Theta(1)$

  Questa operazione modifica il valore della chiave del nodo v con il nuovo valore d. Se il valore risultante della chiave di v dovesse violare la proprietà di ordinamento degli heap, è necessario tagliare tutto il sottoalbero radicato in e dal suo genitore, aggiungendolo alla collezione di alberi e rendendo v una nuova radice, ripristinando così la proprietà di ordinamento.

É possibile osservarne l'implementazione nel file *modFibonacci.py*[2].

Tenendo presente che vengono eseguite n operazioni di `ExtractMin`, n operazioni di `Insert` e al massimo $(m-n)$ operazioni di `DecreaseKey`, utilizzando come coda con priorità l'Heap di Fibonacci, si ottiene un andamento teorico di $\Theta(m + nlogn)$ nel caso peggiore [3].

[1] consultare il file README.md per i particolari sull'esecuzione

[2] l'implementazione di tale heap non è stata realizzata da me.

[3] è possibile definire la modalità di esecuzione impostando il parametro `-e`

Di seguito è possibile osservare lo pseudocodice seguito per l'implementazione.

Algorithm 1 Algoritmo di Prim

Per realizzare questo algoritmo è stato necessario generare un grafo connesso

```
1: for each (vertice v in G) do
2:     d(v) ← +∞
3: end for
4: T ← albero formato da un solo nodo s
5: CodaPriorità S
6: d(s) ← 0
7: S.insert(s,0)
8: while (not S.isEmpty()) do
9:     u ← S.deleteMin()
10:    for each (arco (u,v) in G) do
11:        if d(v) = +∞ then
12:            S.insert(v, w(u, v))
13:            d(v) ← w(u, v)
14:            rendi u padre di v in T
15:        else if w(u, v) < d(v) then
16:            S.decreaseKey(v, d(v) − w(u, v))
17:            d(v) ← w(u, v)
18:            rendi u nuovo padre di v in T
19:        end if
20:    end for
21: end while
22: return T
```

2.2 Expected linear time MST algorithm

L'Expected linear time MST algorithm (anche chiamato KKT) è un'algoritmo randomico per determinare la foresta minima ricoprente (Minimum Spanning Forest[4] A differenza della precedente implementazione, è necessario avere un grafo pesato, come input, in cui non ci siano vertici isolati e tutti i pesi degli archi siano unici. In queso modo si assicura che il Minimum Spanning Tree sia unico [2].

Questo algoritmo è ricorsivo ed è formato da due passi fondamentali: lo **step di Borůvka** grazie al quale si riduce il numero di vertici di un fattore minimo di due e la **fase di campionamento casuale** in cui vengono scartati degli archi così da ridurre la densità (rapporto tra il numero di archi e il numero di vertici).

Di seguito vengono riportati gli step seguiti durante l'implementazione[5]:

- **Step(0):** Se il grafo G è vuoto ritorna una foresta vuota. Altrimenti procedere con i passi successivi.
- **Step(1):** Applicare lo step di Borůvka due volte consecutivamente così da ridurre il numero di vertici almeno di un fattore di quattro.
- **Step(2):** Nel grafo contratto, scegliere un sottografo H selezionando ogni arco indipendentemente con probabilità $\frac{1}{2}$. Applicare ricorsivamente l'algoritmo ad H, ottenendo la foresta minima ricoprente F di H. Trovare tutti gli archi *F-heavy* ed eliminarli dal grafo contratto H.
- **Step(3):** Applicare l'algoritmo ricorsivamente al grafo rimanente cosi da ottenere la foresta ricoprente *F'*. Ritornare gli archi contratti nello Step(1) e gli archi di *F'*.

Lo **Step(1)** è stato implementato rappresentando il grafo in input come una collezione di archi ed è stata utilizzata la struttura dati `Union-Find` per tenere traccia delle componenti.

Lo **Step(2)** è stato realizzato utilizzando il modulo `random` così da generare un numero randomico tra 0 e 1 che rappresentasse una certa probabilità p. Inoltre, per individuare gli archi *F-heavy* è stata implementata la seguente definizione: Considerando F come una foresta di un grafo G, è possibile indicare con F(x,y) il percorso, se esiste, che connette x e y in F e con $w_f(x, y)$ il peso massimo di F(x,y). Se x e y non sono connessi in F allora $w_f(x, y) = \infty$. Un arco di F è detto *F-heavy* se $w(x, y) > w_f(x, y)$, altrimenti è detto *F-light*.

Descrizione di uno step di Borůvka: considerando un grafo G, per ogni vertice viene selezionato l'arco di peso minimo incidente al vertice. Una volta

[4]La differenza tra MST e MSF è semplice perchè in un grafo disconnesso è impossibile trovare il minimo albero ricoprente (MST) in quanto ci sono diverse componenti connesse, l'unica cosa che si può fare in questo caso è determinare il minimo albero ricoprente per ogni componente connessa e ciò viene definito proprio come un foresta minima ricoprente (MSF).

[5]è possibile consultare il file README.md per i particolari sull'esecuzione

individuati questi archi si procede contraendo tutti gli archi selezionati, sostituendo con un singolo vertice la componente connessa indotta dagli archi selezionati. Infine vengono eliminati tutti i nodi isolati, tutti i cicli appartenenti ad una stessa componente connessa e tutti gli archi, tranne quello di peso minimo tra due componenti connesse.

É possibile osservare lo pseudocodice utilizzato per implementare lo step di Borůvka[1].

Algorithm 2 Step di Borůvka

```
1: Inizializzare una foresta F in modo che sia un insieme di alberi formati da un solo
   vertice, uno per ogni vertice del grafo.
2: while F ha più di una componente do
3:     for each (arco (u,v) di G) do
4:         if u e v appartengono a componenti differenti then
5:             if (u,v) è l'arco di peso minore della componente u then
6:                 Impostare l'arco (u,v) come l'arco di peso minore per la componente u
7:             end if
8:             if (u,v) è l'arco di peso minore della componente v then
9:                 Impostare l'arco (u,v) come l'arco di peso minore per la componente v
10:            end if
11:        end if
12:    end for
13:    for each componente il cui arco di peso minore è diverso da "None" do
14:        Aggiungi l'arco di peso minore a F
15:    end for
16: end while
```

Output F è la foresta minima ricoprente

Da un'analisi probabilistica è possibile osservare che l'algoritmo impiega un tempo lineare $\Theta(m)$, ad eccezione del caso peggiore in cui si ha $\Theta(min\{n^2, mlogn\})$[6].

3 Verifica correttezza implementazione

Prima di iniziare la fase successiva è stato realizzato uno script (Test.py) per verificare se tutte le implementazioni create fossero corrette. Per eseguirlo è necessario inserire nella riga 9, all'interno del metodo `open()` il path del file in cui è stato salvato il grafo da studiare (il grafo deve essere salvato in formato adjlist, per far si che lo script lo riesca a leggere).

Al termine dell'esecuzione verrà mostrato a video il peso relativo al minimo albero ricoprente ottenuto sia dall'algoritmo di Prim (preceduta dalla stringa "Peso mst Prim:") sia dall'algoritmo MST a tempo lineare (preceduta dalla stringa "Peso algoritmo MST:"). É possibile ottenere anche la lista degli archi che compongono l'albero, in questo caso l'algoritmo MST stamperà a video gli archi (nodo,nodo,peso) mentre l'algoritmo di Prim riprodurrà l'elenco degli archi in formato (nodo,weight).

[6] è possibile definire la modalità di esecuzione impostando il parametro `-e`

4 Fase di test

La metrica di performance considerata è il **tempo**, mentre gli indicatori di performance associati ad essa sono il **numero di operazioni dominanti**, misurate incrementando un contatore, e il **tempo di CPU** (livello concreto).

Dato che entrambi gli algoritmi studiati sono basati sul calcolo di un minimo albero ricoprente, le operazioni considerate dominanti sono l'inserimento e la modifica di ogni arco appartentente all'MST.

4.1 Parametri e Fattori

La misurazione del tempo impiegato dall'algoritmo può essere influenzata da numerosi parametri. Tutti gli esperimenti sono stati condotti con la seguente configurazione:

- **Hardware**:
  - CPU: Intel Core i5 dual-core a 2,7GHz
  - RAM: 8 GB 1867 MHz
- **Sistema Operativo**: mac OS Mojave Versione 10.14.6
- **Interprete Python**: versione 3.9

Per quanto riguarda la scelta dei fattori (parametri controllabili) sono state effettuate le seguenti scelte:

- **Taglia dei nodi (n)**: per poter analizzare un trend nell'andamento del tempo d'esecuzione sono stati considerati grafi con diverse dimensioni.
- **Probabilità (p)**: per poter osservare le differenze di comportamento dei due algoritmi sono stati generati grafi randomici con diverse probabilità (di conseguenza, con un differente numero di archi).

4.2 Livelli

I diversi valori assegnati ad ogni fattore sono i seguenti:

- **n**: le taglie dei nodi considerati sono 100, 200, 400, 800, 1600
- **m**: le probabilità considerate sono 5%, 10%, 20%, 40% ,ovvero m = (0.05, 0.1, 0.2, 0.4)

4.3 Generazione degli input

Le istanze di input (grafi) da utilizzare per i test sono state generate tramite lo script input "genInput.py", in accordo con i livelli sopra menzionati. In questo modo sono stati generati 20 grafi connessi, tutti con taglie diverse e con i pesi degli archi distinti [7]. Inoltre tali istanze sono state salvate, in formato adjlist

[7] come richiesto dall'algoritmo KKT [2]

(elenchi di adiacenza su più righe) nella cartella "Input".
Ogni file contiene una diversa istanza di input, in cui nella prima riga si trova il nodo sorgente seguito dal grado d del nodo e nelle successive d-righe ci sono i nodi di destinazione con i relativi pesi (ciò viene ripetuto per tutti i nodi del grafo).

5 Esecuzione dei test

5.1 Test sul numero di istruzioni dominanti

I primi test sono stati finalizzati al conteggio del numero di istruzioni dominanti eseguite dai due algoritmi. É stato possibile realizzare questo tipo di test inserendo un contatore all'interno del codice e incrementandolo all'occorrenza (a differenza dei successivi, questo test è indipendente dal calcolatore che esegue l'algoritmo).

Per eseguire questa fase sono stati considerati tutti i livelli di n e per ognuno di essi sono stati analizzati tutti i livelli di p, ottenendo 5×4 design points per ognuna delle implementazioni.

Tali risultati sono stati salvati in un file csv (nella cartella InstructionCount/gnp, con il formato nomeAlgoritmo_tipoGrafo_percentuale.csv).
Successivamente sono stati plottati, tramite lo script Multiplot.py[8], i risultati ottenuti con i relativi andamenti teorici.

Di seguito sono stati riportati i grafici relativi agli andamenti ottenuti per le due implementazioni realizzate con tale test.

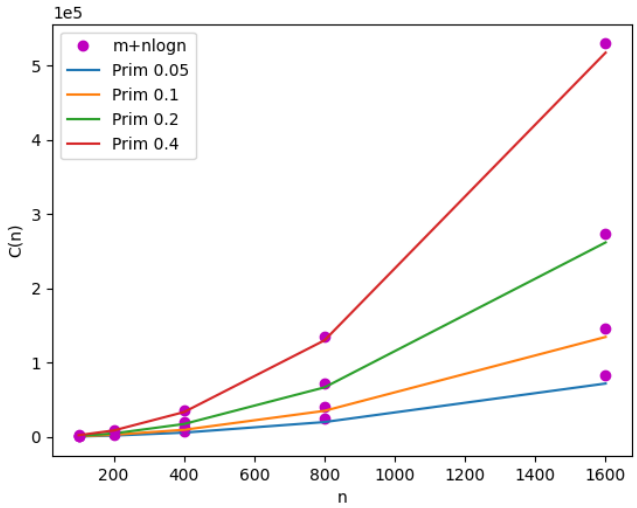
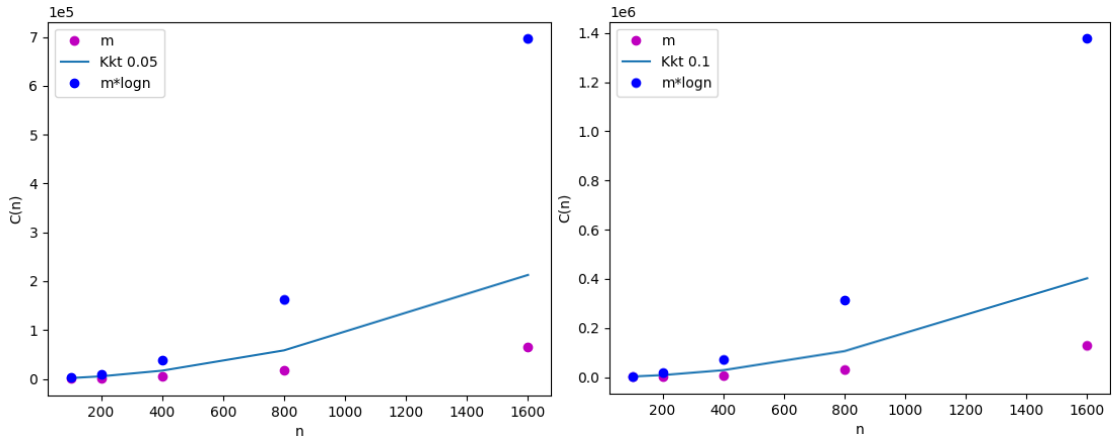


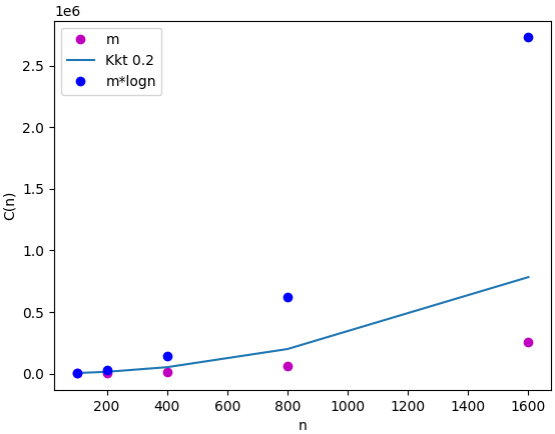
Figura 1: Andamento ottenuto per l'algortimo di Prim

[8]consultare README.md per i particolari sull'esecuzione

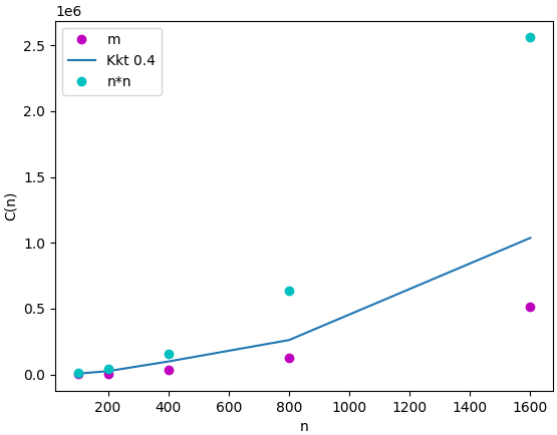


(a) KKT per grafo con $p = 0.05$

(b) KKT per grafo con $p = 0.1$



(c) KKT per grafo con $p = 0.2$



(d) KKT per grafo con $p = 0.4$

Figura 2: Andamenti ottenuti per l'algortimo KKT

Per l'algoritmo KKT è stato necessario graficare il comporamento dei vari grafi per ogni livello di probabilità perchè l'andamento nel caso peggiore dipende dal numero di archi che compongono il grafo. Infatti, è possibile osservare dalle figure 2a, 2b, 2c che per i grafi con probabilità $0.05, 0.1$ e 0.2 il minimo tra n^2 ed $mlogn$ è $mlogn$. Risultato differente si ottiene per il grafo con probabilità pari a 0.4 dove risulta essere minore n^2 rispetto a $mlogn$ (come mostrato in figura 2d).

Considerazioni finali: Da questo test ho potuto osservare che in entrambi i casi il numero di confronti effettuati rispecchia l'andamento teorico.
In particolare, per l'algoritmo di Prim il risultato ottenuto da tale test segue fedelmente l'andamento nel caso peggiore, implementando la coda con priorità con l'Heap di Fibonacci $\Theta(min\{n^2, mlogn\})$, come raffigurato nella figura 1.
Per l'algoritmo KKT, l'andamento di $C(n)$ è compreso tra $\Theta(min\{n^2, mlogn\})$ (caso peggiore) e $\Theta(m)$ (caso migliore).

5.2 Test sul tempo di esecuzione

In questa fase di studio ci si è concentrati sull'analisi del tempo, che è a più basso livello rispetto al test precedente (alto livello), infatti sono stati eseguiti dei test per misurare il tempo di esecuzione (tempo di CPU) degli algoritmi implementati.

Nel test riguardante il conteggio delle operazioni dominanti si ha una **precisione massima** con una **mancanza di accuratezza**; in questo test, invece si ha **accuratezza** ma **mancanza di precisione** in quanto eseguendo più volte gli stessi test si possono ottenere dei risultati diversi, dovuti al fatto che il tempo di CPU è influenzato da molti aspetti (come ad esempio il numero di accessi in memoria).

I livelli considerati in questo test sono i medesimi della fase precedente, motivo per il quale si ha lo stesso numero di design points (20) per ogni algoritmo implementato.

Per la misurazione del tempo di CPU è stata utilizzata la funzione `process_time()` del modulo `time`, calcolando la differenza tra il tempo subito prima dell'esecuzione e il tempo subito dopo la fine dell'esecuzione, escludendo la fase in cui viene letto il grafo da file.

Tuttavia il tempo di esecuzione, come detto precedentemente, può fornire risultati diversi per uno stesso grafo in input, per tale motivo ogni grafo è stato eseguito tre volte e poi è stata fatta una media del tempo di esecuzione ottenuto.

Questo test è stato eseguito su entrambi gli algoritmi e i risultati ottenuti per ogni tipo di probabilità sono stati salvati in un file csv (nella cartella ExecutionTime/gnp, con il formato nomeAlgoritmo_tipoGrafo_percentuale.csv).
Infine, questi risultati sono stati graficati in modo da rendere più facile il confronto, dove l'asse delle ascisse rappresenta la taglia dei nodi mentre l'asse delle ordinate rappresenta il tempo di esecuzione.

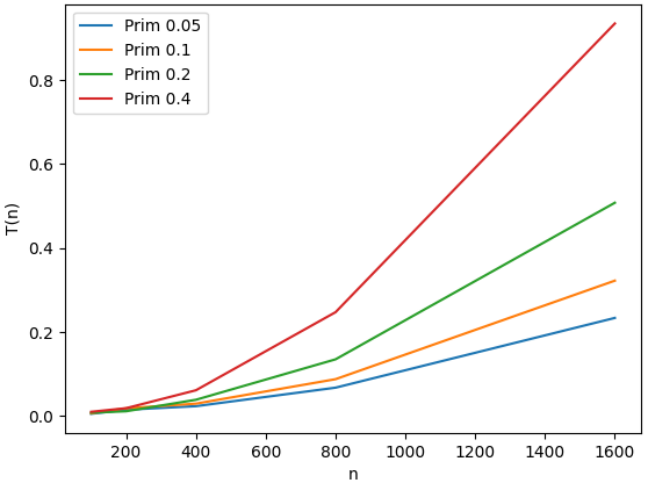


Figura 3: Andamento temporale ottenuto per l'algoritmo di Prim

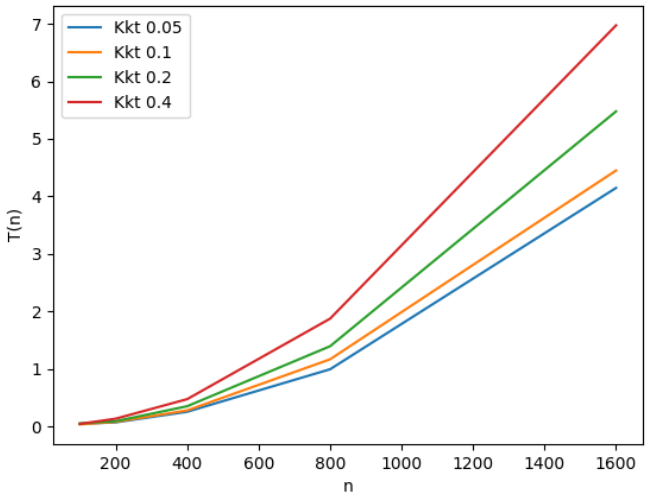


Figura 4: Andamento temporale ottenuto per l'algoritmo KKT

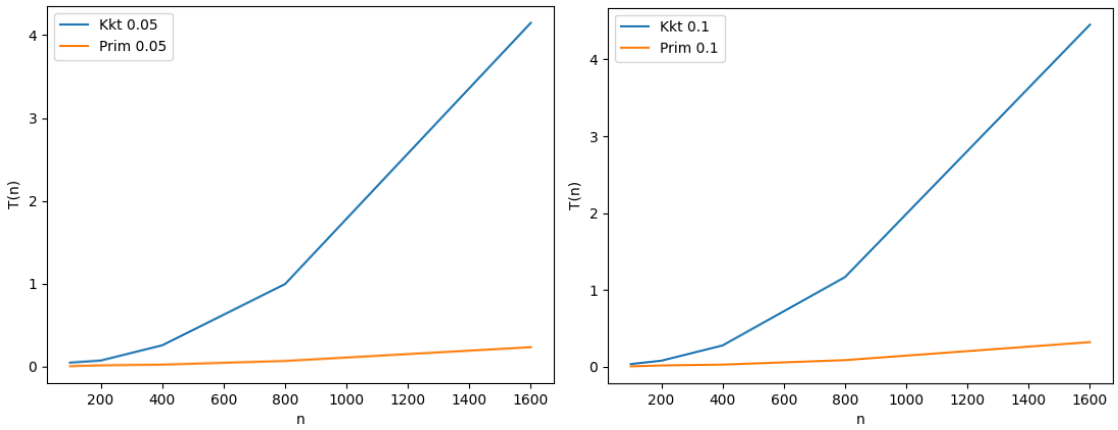
Considerazioni finali: Confrontando sia le figure 3, 1, sia le figure 4, 2 ho potuto osservare che l'andamento relativo al tempo di CPU per entrambi gli algoritmi rispecchia l'andamento del numero di operazioni dominanti eseguite.

5.3 Test sulla qualità della soluzione

La metrica più studiata dopo l'analisi del tempo di esecuzione per gli algoritmi è la qualità della soluzione. In questo progetto è stato eseguito questo tipo di test perchè avendo studiato algoritmi MST sia di tipo randomico (algoritmo MST a tempo lineare) sia di tipo deterministico (algoritmo di Prim) e ottenendo lo stesso risultato, è stato ritenuto necessario definire quale tra i due algoritmi studiati fornisse il risultato migliore.

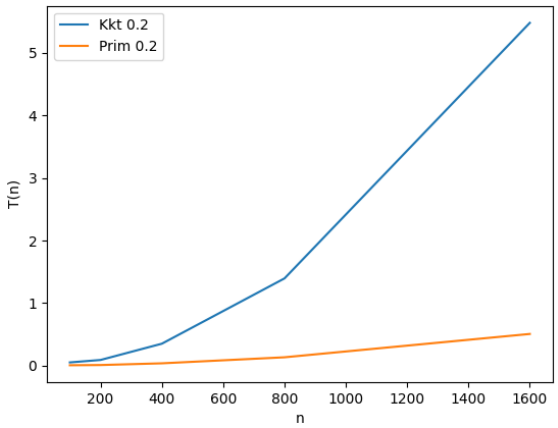
L'algoritmo MST di tipo randomico può fornire un mst approssimato (al contrario dell'algoritmo di Prim che calcola un mst ottimo), fornendo un costo che non risulta essere minimo. Tuttavia, per come ho implementato questo algoritmo si avrà sempre un MST minimo come risultato. Quindi, la metrica di interesse principale per questo test è stato il confronto del tempo di esecuzione ottenuto eseguendo le due implementazioni.

Per realizzare questo test sono stati plottati i dati relativi al running time dei due algoritmi per tutte le probabilità studiate (salvando le immagini in formato png nella cartella Images/Multiplot/QualitySolution con il formato QualitySolution_probabilità).
Nella pagina successiva è possibile esaminare quanto ottenuto.

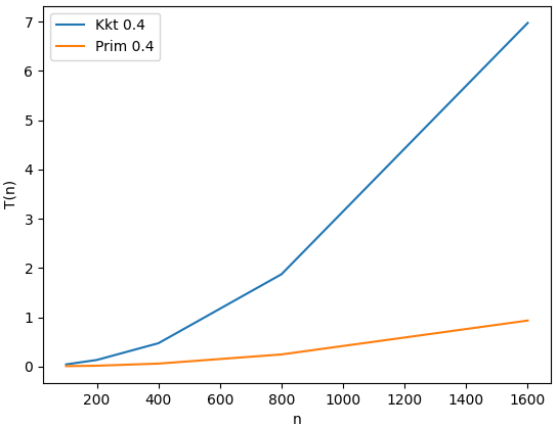


(a) Andamento grafo con $p = 0.05$

(b) Andamento grafo con $p = 0.1$



(c) Andamento grafo con $p = 0.2$



(d) Andamento grafo con $p = 0.4$

Figura 5: Andamenti per verificare la qualità della soluzione

Considerazioni finali:
Confrontando il tempo di esecuzione ottenuto dalle Figure 5a, 5b, 5c, 5d ho notato che l'algoritmo KKT impiega un tempo sempre maggiore rispetto all'algoritmo di Prim, per grafi che possiedono la stessa dimensione. Inoltre, maggiore è quest'ultima, maggiore sarà il tempo impiegato dall'algoritmo randomico per calcolare l'mst. Viceversa, l'algoritmo di Prim non risente molto dell'aumento della grandezza del grafo.
In definitiva è possibile affermare che l'algoritmo di Prim impiega un tempo minore per determinare l'albero minimo ricoprente rispetto all'algoritmo MST a tempo lineare, fornendo una qualità migliore della soluzione.

6 Conclusioni

Riassumendo, il lavoro che ho svolto è stato incentrato sullo studio del funzionamento dell'algoritmo di Prim e dell'algoritmo MST a tempo lineare e le rispettive complessità teoriche.
In un primo momento, sono stati implementati i due algoritmi in Python e solo dopo averne verificato la correttezza si è passati alla fase di sperimentazione. In particolar modo in questa fase sono state definite le metriche di performance, i relativi indicatori di performance, i parametri, i fattori e i corrispondenti livelli su cui sono stati eseguiti i vari test.
Dopo aver generato gli opportuni inputs, ho eseguito il test riguardante il numero di operazioni dominanti sulle implementazioni realizzate, verificando che tali andamenti rispettassero l'andamento teorico. In seguito, ho analizzato il tempo che tali algoritmi impiegavano nella loro esecuzione. Infine ho determinato quale algoritmo fornisse una soluzione migliore, eseguendo il test sulla qualità della soluzione.

Da tale studio ho potuto constatare che i due algoritmi rispettano le ipotesi fatte nella prima fase di questo progetto, in particolar modo con l'ultimo test eseguito è stato dimostrato come l'algoritmo di Prim, che è un algoritmo esatto, fornisca sempre una soluzione migliore rispetto all'algoritmo MST con tempo lineare atteso per tutti i grafi studiati.

Riferimenti bibliografici

[1] https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

[2] KARGER, David R.; KLEIN, Philip N.; TARJAN, Robert E. A randomized linear-time algorithm to find minimum spanning trees. Journal of the ACM (JACM), 1995, 42.2: 321-328.