Edit distance

Diego Biagini

July 12, 2019

1 Introduzione

Il grafo è una struttura che ha numerose applicazioni sia nei campi dell'informatica sia in quelli dell'ottimizzazione.
Alcuni tipi di operazioni possibili su di esse sono le visite e la ricerca di cammini, o insiemi di cammini, particolari.
Gli esperimenti che saranno eseguiti serviranno a esaminare due tipi particolari di ricerche in un grafo: la ricerca delle componenti connesse e la ricerca dell'MST(minimum spanning tree), quest'ultima sarà effettuata attraverso l'uso dell'algoritmo di Kruskal.
Ciò che ci interesserà scoprire maggiormente è il comportamento degli algoritmi per queste ricerche all'aumentare dei nodi del grafo.
É necessario infatti che essi siano efficienti su grafi relativamente estesi date le applicazioni reali, un esempio per le componenti connesse è nell'analisi di immagini e quindi OCR, mentre la ricerca di un albero di connessione minimo è un classico problema che si presenta nelle telecomunicazioni.

2 Cenni teorici

2.1 Grafi

Un grafo può essere visto come un insieme di elementi in relazione tra loro.
Questa relazione è indicata dagli archi che li connettono.
Un grafo può essere individuato attraverso due insiemi:

- l'insieme dei vertici V
- l'insieme degli archi E ,ogni arco è indicato da una coppia di vertici

Un grafo si dice **non diretto** se E è composto da coppie di vertici ordinate.
Viene definito **cammino** di un grafo dal vertice u a u' una sequenza di vertici $< v_0, ..., v_k >$ tale che $v_0 = u, v_k = u'$ e $(v_i - 1, v_i)$ è un arco presente in E per $i = 2, ..., k - 1$. Se $v_0 = v_k$ e il cammino è formato da almeno un arco invece siamo in presenza di un **ciclo**

É possibile memorizzare i grafi in diversi modi, due opzioni possibili sono le seguenti:

- **Lista di adiacenza**, viene creata una lista per ogni nodo e queste liste contengono gli altri nodi a cui è collegato
- **Matrice di adiacenza**, si memorizza una matrice $A \in (|V| \times |V|)$, se esiste un arco tra il nodo i-esimo e j-esimo allora $a_{ij} = 1$, altrimenti $a_{ij} = 0$.

Ognuna di queste rappresentazioni ha certi vantaggi minori rispetto alle altre, per esempio con una lista viene usata in genere meno memoria ma cercare se un certo nodo è collegato ad un altro diventa più difficoltoso.
E' possibile visitare un grafo in numerosi modi, le due visite che hanno più usi sono la **BFS**(Breadth first search) e la **DFS**(Depth first search).

2.2 Strutture dati per insiemi disgiunti

Si occupano di gestire un qualsiasi numero di insiemi dinamici contenenti elementi di vario tipo.
Per identificare ciascun insieme viene scelto un rappresentante.
Le operazioni che devono essere garantite sono:

- **Make-Set(x)**, crea un insieme contenente un solo elemento,x

- **Union(x,y)**, fonde due insiemi in modo che il rappresentante di ogni elemento sia lo stesso
- **Find-Set(x)**, restituisce il rappresentante di x

Le implementazioni possibili per questa struttura dati sono:

- attraverso liste, ogni insieme è rappresentato da una lista concatenata e ogni elemento dell'insieme ha un puntatore al rappresentante; qui è possibile usare l'euristica dell'unione pesata
- attraverso una foresta di alberi, ogni insieme è identificato da un albero, siamo in grado di trovare il rappresentante risalendo l'albero, inoltre viene memorizzato il rango di ogni albero, ovvero la sua altezza; qui è possibile usare l'euristica dell'unione per rango, simile all'unione pesata, e la compressione dei cammini in modo che la scalata dell'albero sia più corta possibile.

Usando la rappresentazione con una foresta di alberi e implementando entrambe le sue euristiche siamo in grado di realizzare una struttura dati molto efficiente, infatti l'unione(che è l'operazione più costosa) tra due insiemi di dimensione m e n assume un costo del tipo $O(m\alpha(m,n))$ dove α è la funzione inversa di Ackermann, una funzione che cresce così lentamente da poter essere considerata costante nelle applicazioni reali.
Quindi abbiamo che il costo dell'unione è lineare rispetto a m.

2.3 Componenti connesse di un grafo

Dato un grafo $G = (V, E)$ è possibile partizionarlo in dei suoi sottoinsiemi, detti componenti connesse. Una componente connessa può essere vista come relazione di equivalenza tra nodi di un grafo, secondo la relazione "è raggiungibile da".
Quindi due nodi u e v appartengono alla stessa componente connessa se e solo se esiste un cammino tra loro.
Per trovare le componenti connesse in modo semplice si usa il seguente algoritmo, che fa uso di insiemi disgiunti:

1 Connected-Components(G)

```
1: for ogni vertice v ∈ G.V do
2:     MAKE-SET(v)
3: end for
4: for ogni arco (u, v) ∈ G.E do
5:     if FIND-SET(u) ≠ FIND-SET(v) then
6:         UNION(u, v)
7:     end if
8: end for
```

2.4 MST

Dato un grafo diretto pesato(ogni arco ha un peso $w(u,v)$) e connesso è possibile trovare un albero minimo di connessione.
Questo è un sottoinsieme aciclico $T \subseteq E$ che connette tutti i vertici e che ha peso totale minore, ovvero

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

deve essere minimo

Tutti gli algoritmi per trovare l'MST di un grafo sono costruiti nel modo seguente: Per trovare un

2 Generic-MST(G)

```
1: A ← ∅
2: while A non forma un albero di connessione do
3:    trova un arco (u, v) sicuro per A
4:    A ← A ∪ {(u, v)}
5: end while
```

arco sicuro è utile il seguente teorema:
Sia $G = (V, E)$ un grafo non orientato e connesso con un peso associato ad ogni arco. Sia A un sottoinsieme di E contenuto in un qualche MST per G. Sia $(S, V - S)$ un qualunque taglio che rispetta A. Allora se l'arco (u, v) è un arco leggero che attraversa $(S, V - S)$ è un arco sicuro per A.

Da questo è possibile arrivare a sviluppare l'algoritmo di Kruskal per la ricerca dell'albero di connessione minimo di un grafo: Il tempo di esecuzione è dipendente dall'implementazione degli insiemi

3 MST-Kruskal(G)

```
1: A ← ∅
2: for ogni vertice v ∈ G.V do
3:    MAKE-SET(v)
4: end for
5: Ordina gli archi in senso non decrescente rispetto al peso w
6: for ogni arco (u, v) ∈ G.E, presi in ordine di peso non decrescente do
7:    if FIND-SET(u) ≠ FIND-SET(v) then
8:       A ← A ∪ {(u, v)}
9:       UNION(u, v)
10:   end if
11: end for
12: Return A
```

disgiunti.
Abbiamo che il tempo per l'ordinamento è $O(Elg(E))$ e il numero di Union eseguite sarà $O(E)$.
Usando una foresta di insiemi disgiunti sappiamo che il costo di E Union è $O(E\alpha(E, V)$ che è un $O(Elg(E))$, quindi il costo totale dell'algoritmo di Kruskal è $O(Elg(E))$.

3 Esperimenti svolti

Gli esperimenti fatti si occuperanno di analizzare le prestazioni dell'algoritmo per trovare le componenti connesse di un grafo e l'algoritmo di Kruskal.
I grafi presi in considerazione saranno grafi pesati generati casualmente.
La dimensione dei grafi generati sarà sempre crescente ed inoltre saranno testati vari modi di collegare i loro nodi.
In particolare per ogni dimensione del grafo saranno generati 5 diversi grafi, in ognuno di essi la probabilità che un nodo sia collegato ad un altro sarà diversa.
La crescita del numero di nodi sarà esponenziale, ad ogni ciclo sarà raddoppiata fino ad arrivare ad un massimo di 2^{15}.
Per ogni grafo generato prima di tutto verranno ricercate le componenti connesse e sarà registrato il tempo di esecuzione.
Dopodichè se è stata trovata esattamente una componente connessa è possibile eseguire la ricerca dell'MST con l'algoritmo di Kruskal.

Per questo sarà registrato il tempo di esecuzione e il peso totale dell'MST.
Inoltre sotto una certa dimensione(numero nodi ¡= 4000 per motivi di tempo di esecuzione) sarà calcolata l'altezza dell'MST.
Questo problema richiede una visita BFS per ogni nodo ed è quindi particolarmente lenta.

4 Documentazione del codice

`Node:`
`value`
`father`
`rank`
`sons`

Struttura dati che identifica un nodo di un albero,ma anche elemento che farà parte di un insieme disgiunto.
Contiene il valore, un riferimento al padre, la lista di nodi figli(usata per elencare tutti i nodi) e il rango(usato nell'unione per rango).

`make_set(value)`
`union(set1,set2)`
`find_set(node)`
`link(x,y)`

Operazioni possibili sugli insiemi disgiunti implementati con foreste di alberi disgiunti.

`set_to_list(set)`

Passato un insieme come parametro restituisce una lista contenente tutti gli elementi di quell'insieme.

`create_graph(n,p)`
`create_weighted_graph(n,p,min,max)`

Dato il numero di nodi e la probabilità di esistenza di un arco generano la matrice di adiacenza di un grafo e la restituiscono.
Per creare un grafo pesato è inoltre necessario fornire il minimo e massimo valore che possono assumere gli archi.

`create_graph_from_arcs(n, arcs)`

Dato il numero di nodi e una lista degli archi costruisce la matrice di adiacenza del grafo relativo e la restituisce.

`adjacent_nodes(graph,node)`

Dato un grafo e un nodo restituisce una lista contenente tutti i nodi adiacenti a quello fornito

`get_arcs_list(graph)`

Dato un grafo restituisce una lista di tuple, ogni tupla indica un arco del grafo.

`order_arcs(graph)`

Dato un grafo restituisce la lista dei suoi archi ordinata secondo i loro pesi.

`connected_components(graph)`

Dato un grafo restituisce una lista di liste, ogni lista contiene dei nodi e indica una componente connessa del grafo.

`kruskal_algorithm(graph)`

Dato un grafo esegue l'algoritmo di Kruskal e restituisce la lista degli archi che formano il suo albero di connessione minimo.

`BFSNode:`
`d`
`p`
`color`

Struttura dati che serve a tenere traccia delle informazioni necessarie di ogni nodo per poter eseguire la visita BFS su un grafo.

`BFS(graph, start)`

Dato un grafo e un nodo esegue la visita BFS sul grafo partendo dal vertice passato.
Restituisce una lista di BFSNode che corrisponde allo stato del grafo dopo la visita.

`BFS_max(graph, start)`

Esegue la visita BFS sul grafo partendo dal nodo passato e restituisce il valore massimo di BFSNode.d alla fine della ricerca, ovvero la distanza da start al nodo.

5 Risultati sperimentali

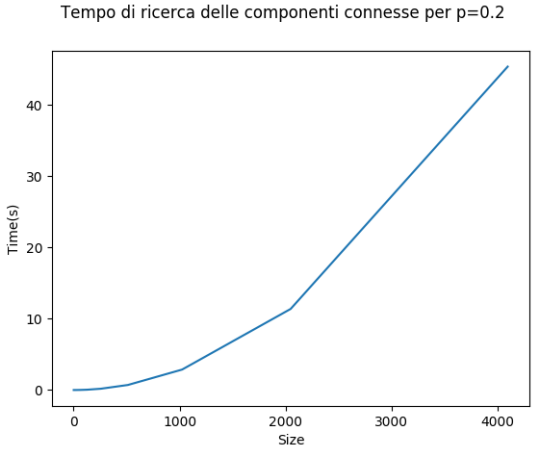
5.1 Componenti connesse

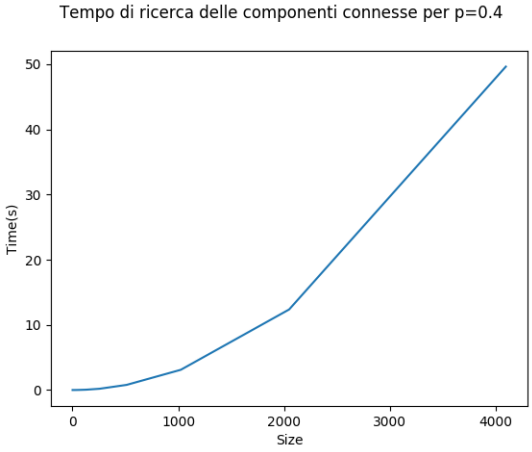
Numero di componenti connesse

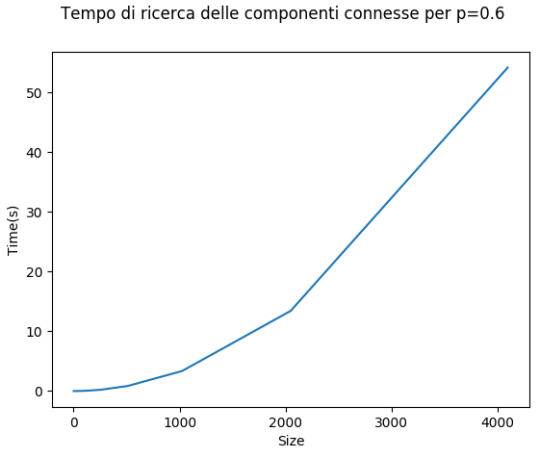
```
Size=1
p=0.2 CCs:1     p=0.4 CCs:1     p=0.6 CCs:1     p=0.8 CCs:1     p=1  CCs:1
Size=2
p=0.2 CCs:2     p=0.4 CCs:2     p=0.6 CCs:1     p=0.8 CCs:1     p=1  CCs:1
Size=4
p=0.2 CCs:3     p=0.4 CCs:3     p=0.6 CCs:1     p=0.8 CCs:1     p=1  CCs:1
Size=8
p=0.2 CCs:6     p=0.4 CCs:2     p=0.6 CCs:1     p=0.8 CCs:1     p=1  CCs:1
Size=16
p=0.2 CCs:1     p=0.4 CCs:1     p=0.6 CCs:1     p=0.8 CCs:1     p=1  CCs:1
Size=32
p=0.2 CCs:1     p=0.4 CCs:1     p=0.6 CCs:1     p=0.8 CCs:1     p=1  CCs:1
```

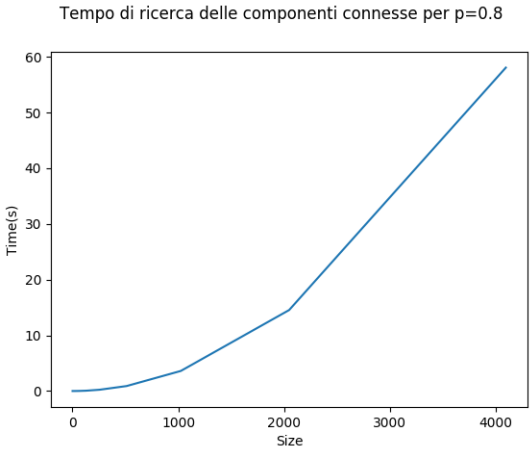
```
Size=64
p=0.2 CCs:1      p=0.4 CCs:1      p=0.6 CCs:1      p=0.8 CCs:1      p=1  CCs:1
Size=128
p=0.2 CCs:1      p=0.4 CCs:1      p=0.6 CCs:1      p=0.8 CCs:1      p=1  CCs:1
Size=256
p=0.2 CCs:1      p=0.4 CCs:1      p=0.6 CCs:1      p=0.8 CCs:1      p=1  CCs:1
Size=512
p=0.2 CCs:1      p=0.4 CCs:1      p=0.6 CCs:1      p=0.8 CCs:1      p=1  CCs:1
Size=1024
p=0.2 CCs:1      p=0.4 CCs:1      p=0.6 CCs:1      p=0.8 CCs:1      p=1  CCs:1
Size=2048
p=0.2 CCs:1      p=0.4 CCs:1      p=0.6 CCs:1      p=0.8 CCs:1      p=1  CCs:1
Size=4096
p=0.2 CCs:1      p=0.4 CCs:1      p=0.6 CCs:1      p=0.8 CCs:1      p=1  CCs:1
```

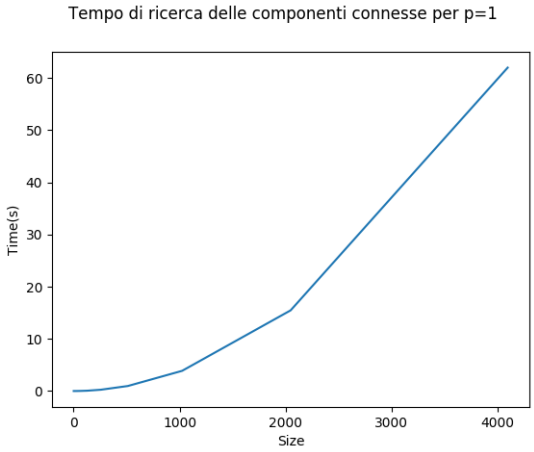
Tempo di esecuzione





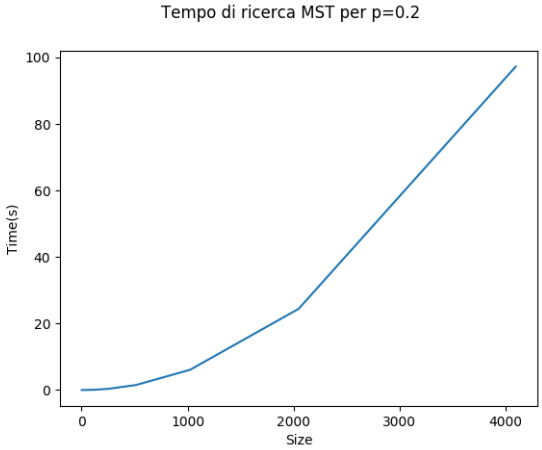


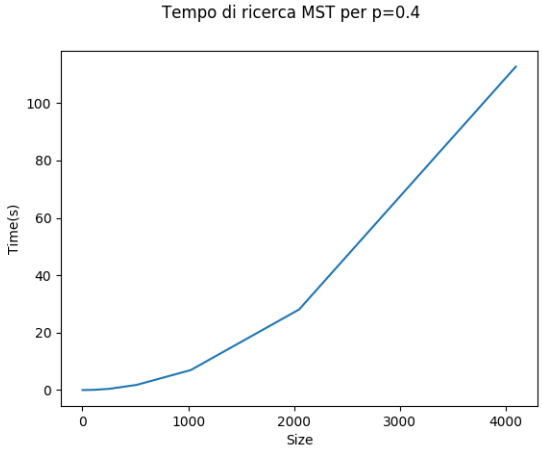


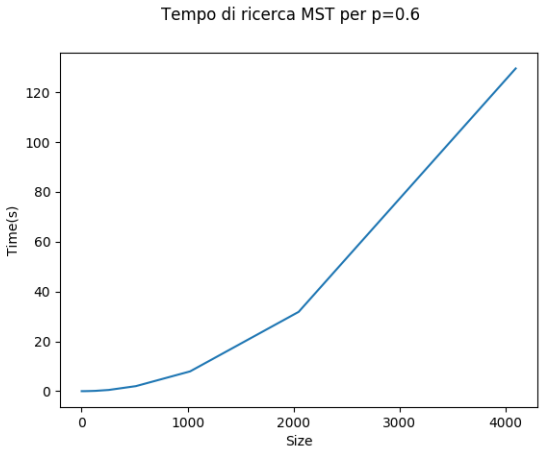


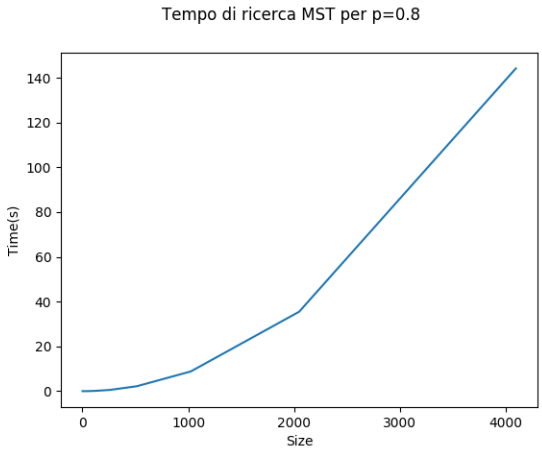
5.2 MST

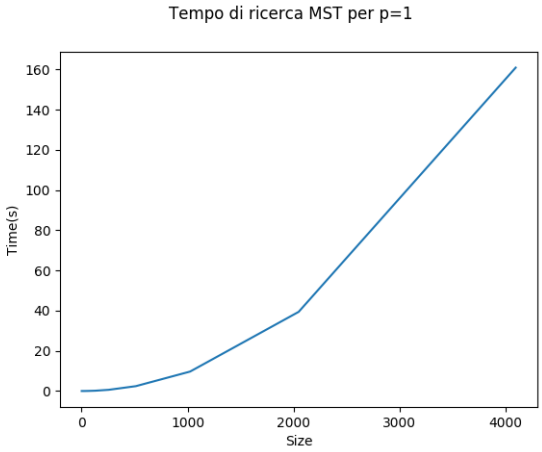
Tempo di esecuzione



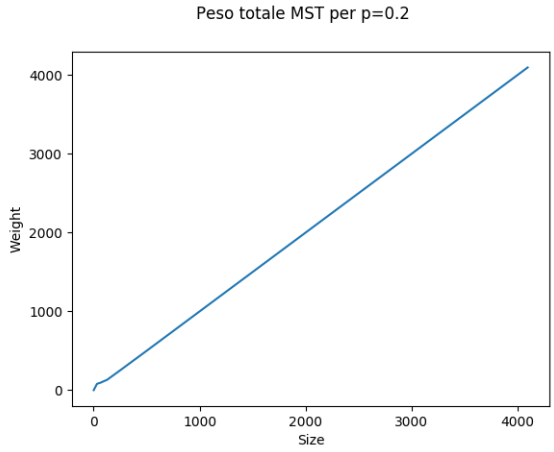


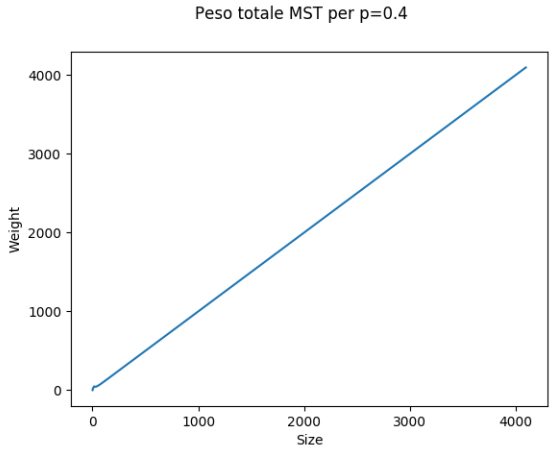


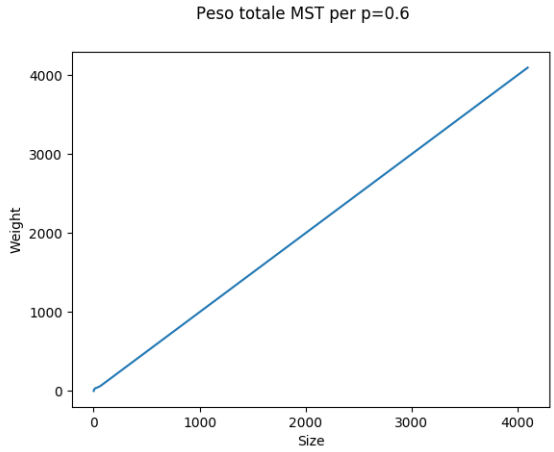


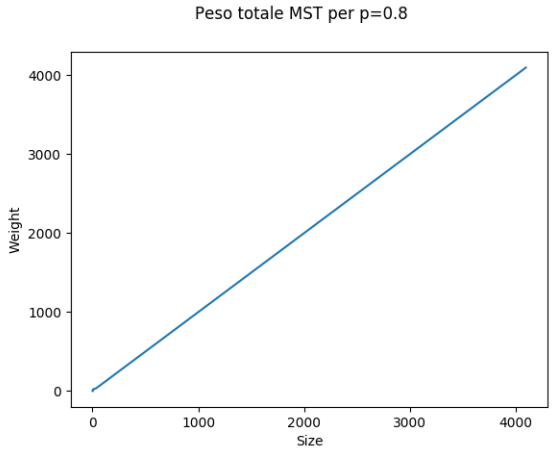

Tempo di ricerca MST per p=1
Time(s)
Size
0
20
40
60
80
100
120
140
160
0
1000
2000
3000
4000

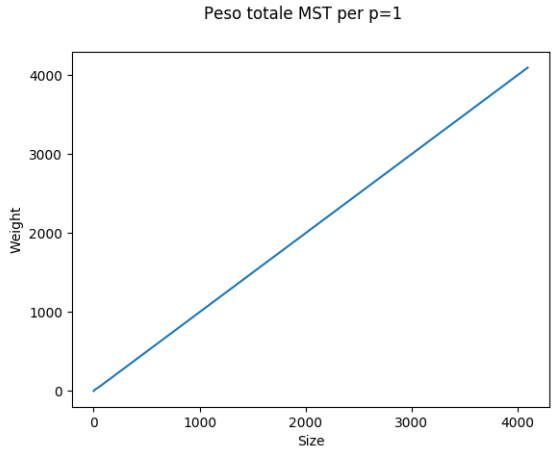
Peso MST


Peso totale MST per p=0.2
Weight
Size
0
1000
2000
3000
4000
0
1000
2000
3000
4000

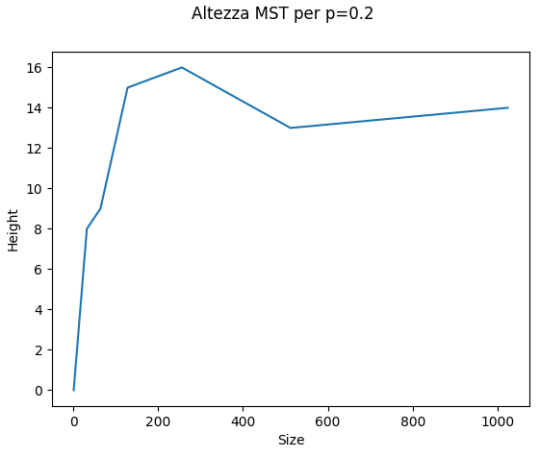

Peso totale MST per p=0.4
Weight
Size
0
1000
2000
3000
4000
0
1000
2000
3000
4000

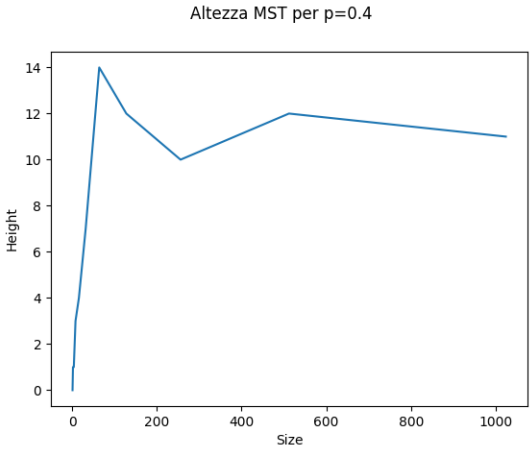

Peso totale MST per p=0.6
Weight
Size
0
1000
2000
3000
4000
0
1000
2000
3000
4000

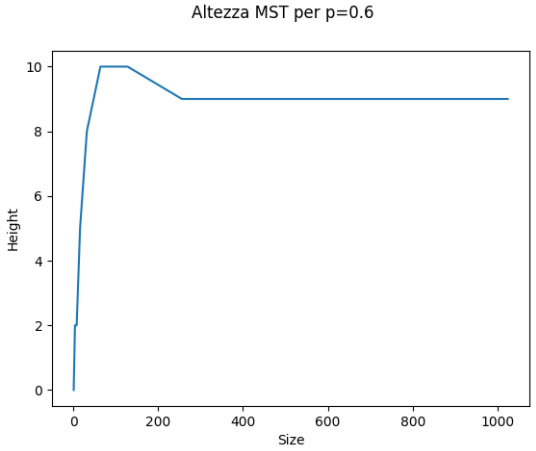

Peso totale MST per p=0.8
Weight
Size
0
1000
2000
3000
4000
0
1000
2000
3000
4000

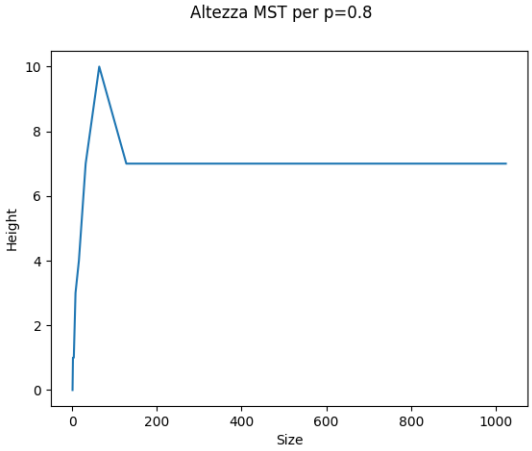

Peso totale MST per p=1
Weight
Size
0
1000
2000
3000
4000

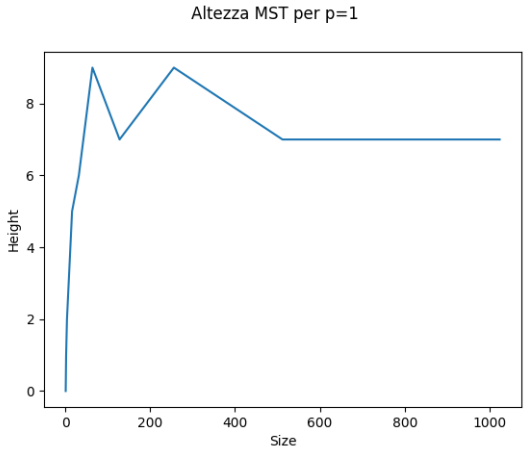
Altezza MST


Altezza MST per p=0.2
Height
Size
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000


Altezza MST per p=0.4
Height
Size
0
2
4
6
8
10
12
14
0
200
400
600
800
1000


Altezza MST per p=0.6
Height
Size
0
2
4
6
8
10
0
200
400
600
800
1000


Altezza MST per p=0.8
Height
Size
0
2
4
6
8
10
0
200
400
600
800
1000



6 Analisi e conclusioni

Dagli esperimenti emerge che molto spesso nel grafo generato si ha una sola componente connessa. Questo è particolarmente vero quando il numero di vertici del grafo è superiore a 10-20, infatti anche se la probabilità di avere un collegamento tra due nodi è abbastanza bassa in qualche modo sarà generato un cammino che collega qualsiasi coppia di nodi, più o meno lungo.
I tempi di esecuzione di questo algoritmo sono quelli attesi ovvero poco più che lineare rispetto al numero di archi, ovvero alla dimensione e alla probabilità di collegamento dei nodi.
Un comportamento molto simile lo ha anche il tempo di ricerca dell'MST, anche se le sue costanti sono più alte.
Il peso totale dell'MST è lineare rispetto al numero di nodi,questo ha però un comportamento particolare al variare della probabilità di presenza di archi.
Infatti da qualche centinaio di nodi in su e a parità del loro numero, nonostante due grafi siano più o meno popolati di archi(p diversi) il peso dei due MST è molto vicino, se non identico.
Infine è possibile osservare come l'altezza dell'MST inizialmente aumenta con l'aumentare del numero di nodi, questo comportamento però si attenua su grafi di dimensioni maggiori dove l'altezza si inizia a stabilizzare attorno a 10.