Calcolo MST: confronto tra Kruskal e Prim

Rebecca Ceccatelli

giugno 2022

1 Introduzione

L'intento di questa relazione è confrontare i due principali algoritmi di calcolo di alberi ricoprenti minimi di grafi non orientati connessi: l'algoritmo di Kruskal e l'algoritmo di Prim.

2 Nozioni teoriche fondamentali e descrizione del problema

Un albero ricoprente minimo (in inglese, *minimum spanning tree: MST*) di un grafo connesso con archi non orientati pesati è un albero che:

- Contiene tutti i vertici del grafo;
- Contiene un sottoinsieme di archi del grafo tali che la sommatoria dei pesi degli archi sia la minima tra tutte le sommatorie negli alberi ricoprenti che si potrebbero individuare.

Dato che il calcolo dell'MST si basa principalmente sulla determinazione del sottoinsieme di archi del grafo, i principali algoritmi di calcolo sono algoritmi iterativi che vanno a popolare l'insieme A degli archi dell'albero, fino a quando quest'ultimo non diventa ricoprente.
Si assume che prima di ogni iterazione A sia un sottoinsieme degli archi di un qualche MST.
In base a quale criterio, ad ogni iterazione, viene scelto un arco da aggiungere all'insieme A? Un arco *(u,v)* viene aggiunto all'insieme solo se è sicuro, i.e. se e solo se il nuovo insieme *A U (u,v)* è ancora un sottoinsieme di un qualche MST.

Il problema del calcolo dell'MST è riconducibile, dunque, al problema della determinazione di archi sicuri. Entrambi gli algoritmi proposti si servono di una struttura dati sottostante per determinare, ad ogni iterazione, un arco sicuro: l'algoritmo di Kruskal utilizza una *Union-Find*, mentre l'algoritmo di Prim utilizza una *coda con priorità minima*.

2.1 Algoritmo di Kruskal

La *Union-Find* è una struttura dati per insiemi disgiunti dinamici. Ogni insieme disgiunto è identificato in un "rappresentante di classe", e la struttura dà la possibilità di creare un nuovo insieme a partire da un elemento x specificato, unire tra loro due insiemi a cui appartengono due diversi elementi x ed y, restituire il rappresentante dell'insieme contenente l'elemento x specificato.
Attraverso queste tre semplici operazioni è possibile determinare le componenti connesse su un grafo non orientato e, successivamente, l'MST. L'algoritmo di Kruskal opera in questo modo, e la sua complessità è strettamente legata all'implementazione scelta per la struttura dati *Union-Find*. In *figura 1* è riportato lo pseudocodice dell'algoritmo.

```
MST-KRUSKAL(G, w)
    A ← ∅
    for ogni vertice v ∈ G.V
        MAKE-SET(v)
    ordina gli archi di G.E in senso non decrescente rispetto al peso w
    for ogni arco (u, v) ∈ G.E, preso in ordine di peso non decrescente
        if FIND-SET(u) ≠ FIND-SET(v)
            A ← A ∪ {(u, v)}
            UNION(u, v)
    return A
```

Figure 1: Pseudocodice dell'algoritmo di Kruskal.

```
MST-PRIM(G, w, r)
    Q ← ∅
    for ogni u ∈ G.V
        u.key ← ∞
        u.π ← NIL
        INSERT(Q, u)
    DECREASE-KEY(Q, r, 0) // r.key ← 0
    while Q ≠ ∅
        u ← EXTRACT-MIN(Q)
        for ogni v ∈ G.Adj[u]
            if v ∈ Q and w(u, v) < v.key
                v.π ← u
                DECREASE-KEY(Q, v, w(u, v)) // v.key ← w(u, v)
```

Figure 2: Pseudocodice dell'algoritmo di Prim.

2.2 Algoritmo di Prim

La coda con priorità minima è una coda in cui, ad ogni *pop*, viene estratta la chiave con valore minimo. Questa può essere implementata tramite *MinHeap*, un *Heap* in cui vale la seguente proprietà: in ogni nodo i, eccetto la radice, il valore del nodo padre è minore o uguale al valore del nodo stesso.
L'algoritmo di Prim utilizza una coda di priorità minima Q per tenere traccia dei vertici che l'albero non ha ancora raggiunto e per avere la garanzia di aggiungere ogni volta un arco sicuro all'insieme degli archi A. La complessità di questo algoritmo è $ElogV$, dove E è il numero di archi del grafo e V è il numero di vertici. In *figura 2* è riportato lo pseudocodice dell'algoritmo.

3 Esperimenti

3.1 Scopo degli esperimenti

Premessa: le considerazioni successive verranno rilette alla luce di questa osservazione: il parametro *probabilità di presenza di archi* utilizzato negli esperimenti corrisponde concettualmente ad un'indicazione del numero di archi mediamente presenti su un grafo.

Lo scopo degli esperimenti che verranno successivamente illustrati è sostenere le seguenti tesi:

- Se il grafo contiene pochi nodi ($V \approx 20$), indipendentemente dal numero di archi presenti, allora l'algoritmo di calcolo dell'MST più efficiente è quello di Kruskal.
- Se il grafo contiene molti nodi allora, fissata una probablità p, oltre una certa soglia limite del numero di nodi l'algoritmo di Prim diventa più efficiente di quello di Kruskal.
- La soglia limite del numero di nodi oltre cui l'algoritmo di Prim diventa più conveniente rispetto a quello di Kruskal dipende in maniera inversamente proporzionale dalla probabilità p di presenza di archi nel grafo. In particolare, la soglia limite raddoppia ogni volta che la probabilità p si dimezza.

le quali possono essere così riassunte:

- Per grafi piccoli (grafi la cui foresta è composta da pochi nodi ed è poco folta) l'algoritmo di calcolo dell'MST più conveniente è quello di Kruskal.
- Per grafi di dimensione medio-grande l'algoritmo di calcolo dell'MST più conveniente è quello di Prim.

3.2 Descrizione degli esperimenti e prestazioni attese

Oltre ad alcuni test che verificano il corretto funzionamento e l'affidabilità dei due algoritmi di calcolo dell'MST, i test più significativi a cui ci si affida in questa relazione sono due:

- Il primo test (Test1) vuole mettere a confronto, generando un grafico, la complessità temporale degli algoritmi di Kruskal e Prim, a cui viene dato in ingresso lo stesso grafo casuale. In questo test la probabilità p di presenza di archi è fissata e il numero di nodi V del grafo è crescente.
- Il secondo test (Test2) vuole mettere a confronto, generando un grafico, la complessità temporale degli algoritmi di Kruskal e Prim, a cui viene dato in ingresso lo stesso grafo casuale. In questo test il numero di nodi V del grafo è fissato e la probabilità p di presenza di archi è crescente.

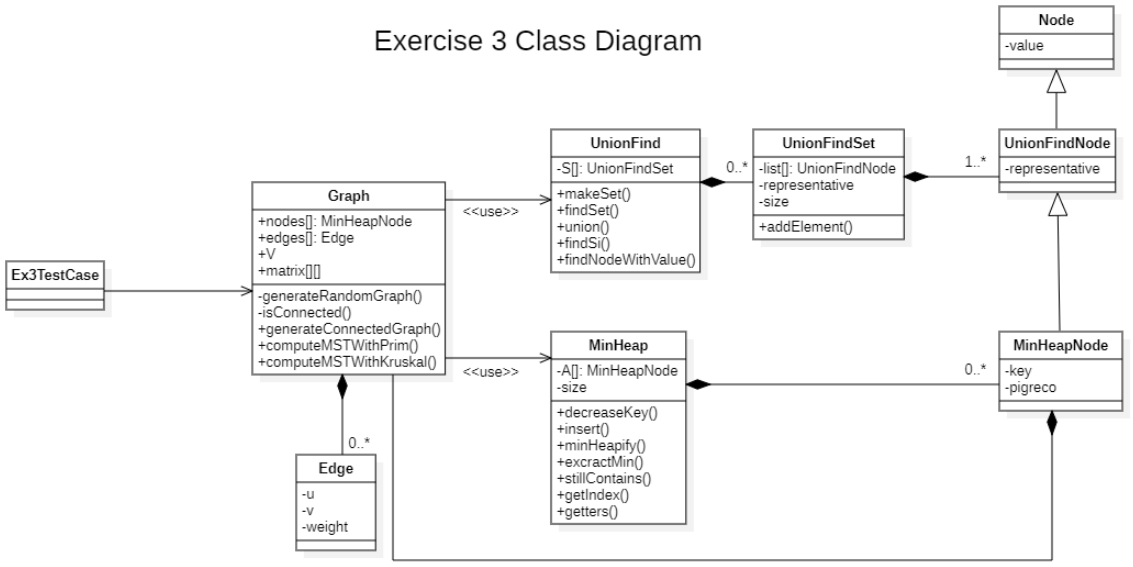
Nella sezione di *Presentazione e analisi dei risultati sperimentali*, variando opportunamente i parametri V e p, i risultati prevenienti dai due test vengono combinati al fine di verificare le prestazioni attese esposte nella sezione *3.1*.

3.3 Sviluppo del codice

Al fine di realizzare gli esperimenti precedentemente descritti 'e stato sviluppato del codice in Python per implementare le strutture dati interessate e per generare automaticamente un vasto campione di dati da utilizzare per tracciare i grafici tramite il pacchetto matplotlib. La piattaforma di test ha operato su un processore Intel a 2 GHz, su sistema operativo Windows.

4 Documentazione del codice

4.1 Diagramma di classe



4.2 Classi e Unittest

- Classi:
  - **Node**: rappresenta un generico nodo, custodisce l'attributo *value*.
  - **UnionFindNode**: rappresenta un nodo all'interno di un set di una struttura dati *UnionFind*, quindi un elemento di un insieme. Eredita da *Node* e in aggiunta custodisce l'attributo *representative*. Implementa il seguente metodo:
    - *setRepresentative(repr)::void*: aggiorna il valore di *representative* a *repr*.
  - **UnionFindSet**: rappresenta un set all'interno di una struttura dati *UnionFind*, quindi è un insieme. Custodisce gli attributi *list* (lista di *UnionFindNode*), *representative* e *size*. Implementa il seguente metodo:
    - *addElement(element)::void*: permette di aggiungere un elemento di tipo *UnionFindNode* al set.
  - **UnionFind**: implementa una struttura dati che gestisce una lista di insiemi, ognuno identificato dal suo rappresentante. Implementa i seguenti metodi:
    - *makeSet(x)::void*: aggiunge alla lista un nuovo insieme contenente l'elemento x.

    - *findSet(x)::UnionFindNode*: ritorna il rappresentante dell'insieme *Si* all'interno della lista che contiene l'elemento *x*.
    - *union(x,y)::void*: fonde in un unico insieme gli insiemi all'interno della lista che contengono rispettivamente gli elementi *x* e *y*.
    - *findSi(x)::Set*: helper method che ritorna il set all'interno della lista che contiene l'elemento *x*.
    - *findNodeWithValue(value)::UnionFindNode*: helper method che ritorna il nodo il cui campo *value* corrisponde al parametro *value*.
- **MinHeapNode**: rappresenta un nodo all'interno della struttura dati *MinHeap*. Eredita da *UnionFindNode* perchè i suoi attributi sono necessari per controllare successivamente la connettività di un grafo, e in aggiunta custodisce gli attributi *key* e *pigreco*. Di conseguenza rappresenta anche un nodo all'interno di un grafo, perchè è la classe che include tutti gli altri tipi di nodo e che quindi permette di sfruttare sia le potenzialità di un *MinHeap* sia quelle di una *UnionFind*.
- **MinHeap**: viene utilizzato per implementare una coda con priorità minima. Custodisce gli attributo *A* (lista di *MinHeapNode* e *size*. Implementa i seguenti metodi:
    - *getParent(keyIndex)::int*: ritorna l'indice del nodo padre del nodo con indice *keyIndex*.
    - *getLeftChild(keyIndex)::int*: ritorna l'indice del nodo figlio siistro del nodo con indice *keyIndex*.
    - *getRightChild(keyIndex):int*: ritorna l'indice del nodo figlio destro del nodo con indice *keyIndex*.
    - *getMinimum()::MinHeapNode*: ritorna il nodo con chiave *key* minima.
    - *decreaseKey(i,newKeyValue)::void*: aggiorna il valore della chiave in posizione *A[i]* al valore *newKeyValue*, minore del precedente, e sistema il nodo nella posizione corretta all'interno della coda. In pratica, aumenta la sua priorità.
    - *insert(node)::void*: inserisce il nuovo nodo *node* nella posizione corretta all'interno della coda.
    - *minHeapify(i)::void*: ristabilisce la proprietà di *MinHeap* sul sottoalbero con radice in *A[i]*.
    - *extractMin()::MinHeapNode*: ritorna e rimuove dalla coda il nodo con chiave *key* minima, riasserendo contestualmente la proprietà di *MinHeap* nell'albero.
    - *stillContains(nodeValue)::Bool,MinHeapNode*: helper method usato nell'algoritmo di Prim, controlla se nella coda è ancora presente un nodo con valore *nodeValue*.
    - *getIndex(nodeValue)::int*: helper method usato nell'algoritmo di Prim, ritorna l'indice del nodo nella coda il cui valore corrisponde a *nodeValue*.

  - **Edge**: rappresenta un arco all'interno di un grafo. Custodisce gli attributi *u, v* e *weight*.
  - **Graph**: rappresenta un grafo. Custodisce gli attributi *V, nodes* (lista di *MinHeapNode*), *edges* (lista di *Edge*), *matrix*. Implementa i seguenti metodi:
    * *generateRandomGraph(V,probability)::Graph*: metodo statico che genera un grafo casuale di V nodi (non necessariamente connesso), cambiando il peso degli archi ad un valore casuale secondo la probabilità *probability*.
    * *isConnected()::Bool*: restituisce *True* se il grafo è connesso, *False* altrimenti. L'implementazione sfrutta la struttura dati *UnionFind*.
    * *generateConnectedGraph(V,probability)::Graph*: metodo statico che genera un grafo casuale di V nodi sicuramente connesso, utilizzando i precedenti metodi *generateRandomGraph()* e *isConnected()*.
    * *computeMSTWithKruskal()*: calcola il Minimum Spanning Tree del grafo utilizzando l'algoritmo di Kruskal, basandosi quindi sulla struttura dati *UnionFind*. Ritorna la lista A degli archi che fanno parte dell'SMT e il peso totale dell'MST.
    * *computeMSTithPRim(rootIndex)*: calcola il Minimum Spanning Tree del grafo utilizzando l'algoritmo di Prim a partire dal nodo in posizione *rootIndex*, basandosi quindi sulla struttura dati *MinHeap*. Ritorna la lista A degli archi che fanno parte dell'MST e il peso totale dell'MST.
    * *setMatrix(matrix)::void*: helper method che serve per effettuare tests, data la matrice *matrix* che descrive il grafo.

- Unittest, file *Tests*: contiene i tests necessari per verificare il corretto funzionamento delle classi precedenti e per svolgere gli esperimenti.
  - *testKruskal()*: verifica il corretto funzionamento dell'algoritmo di Kruskal nel generare un MST su vari grafi dati.
  - *testPrim()*: verifica il corretto funzionamento dell'algoritmo di Prim nel generare un MST su vari grafi dati.
  - *testSameMSTWeight()*: verifica che il peso di due MST, generati a partire dallo stesso grafo ma rispettivamente tramite l'algoritmo di Kruskal e di Prim, sia lo stesso.
  - *testComplexityIncreasingV()*: genera un grafico che mette a confronto la complessità temporale degli algoritmi di calcolo di un MST di Kruskal e di Prim, a partire dallo stesso grafo. Si suppone che la probabilità *probability* di presenza di un arco sia data, e che invece il numero di nodi V sia variabile.

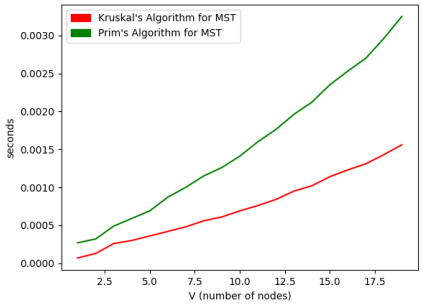
- *testComplexityIncreasingProbability()*: genera un grafico che mette a confronto la complessità temporale degli algoritmidi calcolo di un MST di Kruskal e di Prim, a partire dallo stesso grafo. Si suppone che il numero di nodi V sia dato, e che invece la probabilità *probability* di presenza di un arco sia variabile.
- *computeMovingAverage(array)::[]*: helper method che calcola la media mobile dei dati nel vettore *array*.

Per poter eseguire esperimenti su algoritmi di calcolo dell'MST, si deve prima essere in grado di generare grafi su cui questi algoritmi siano applicabili. Per questo motivo una parte del codice sviluppato è dedicata alla generazione di grafi connessi, non orientati e pesati casuali, dato il numero di nodi V e la probabilità p di presenza di archi. Si generano grafi casuali non necessariamente connessi finchè uno di questi non supera il test di connettività, e allora quest'ultimo viene dato in ingresso agli algoritmi di calcolo dell'MST.

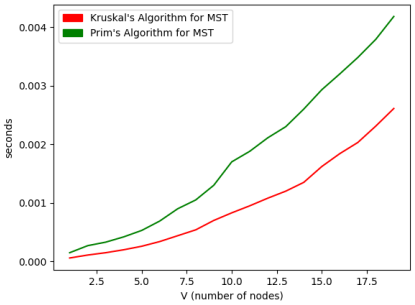
5 Presentazione e analisi dei risultati sperimentali

- I grafici nelle figure 3a, 3b, 3c evidenziano che, se il numero di nodi del grafo è basso ($V \approx 20$), allora l'algoritmo di Kruskal è più veloce dell'algoritmo di Prim nella determinazione dell'MST. Questo è vero indipendentemente dal numero di archi presenti, come testimoniano le figure precedenti ed è evidente nel grafico in figura 4.
  Un'osservazione interessante: al crescere della probabilità p, la forbice tra la complessità temporale dei due algoritmi si riduce sempre di più.

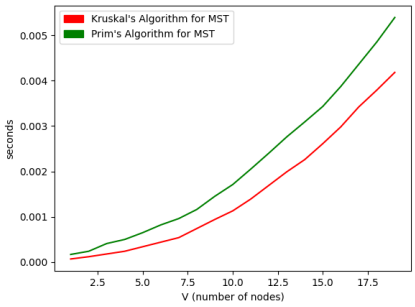
- I grafici nelle figure 6a, 6b, 5 evidenziano che, se è presente un numero sufficientemente grande di archi e il grafo contiene molti nodi, allora l'algoritmo di calcolo dell'MST più efficiente è quello di Prim. Si osserva, infatti, che, superata una certa soglia $limV$ del numero di nodi V, l'algoritmo di Kruskal viene battuto in efficienza dall'algoritmo di Prim.



(a) $maxV = 20$, $p = 10\%$.



(b) $maxV = 20$, $p = 50\%$.



(c) $maxV = 20$, $p = 100\%$.

Figure 3: Paragone tra la complessità temporale degli algoritmi di Kruskal e Prim, con probabilità p fissata e numero di nodi V crescente.

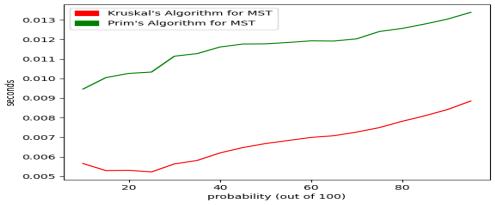


Figure 4: Paragone tra la complessità temporale degli algoritmi di Kruskal e Prim, con numero di nodi V fissato ($V = 20$) e probabilità p crescente.

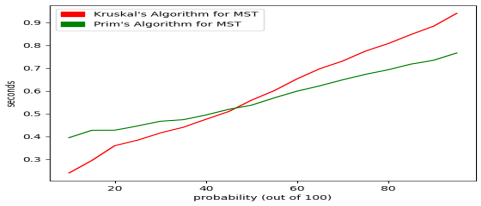
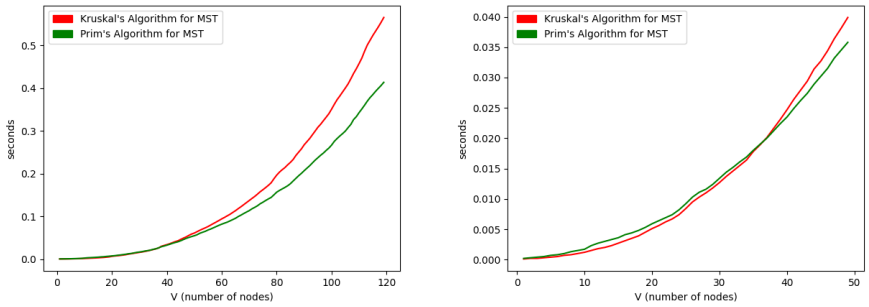


Figure 5: Paragone tra la complessità temporale degli algoritmi di Kruskal e Prim, con numero di nodi V fissato ($V = 120$) e probabilità p crescente.

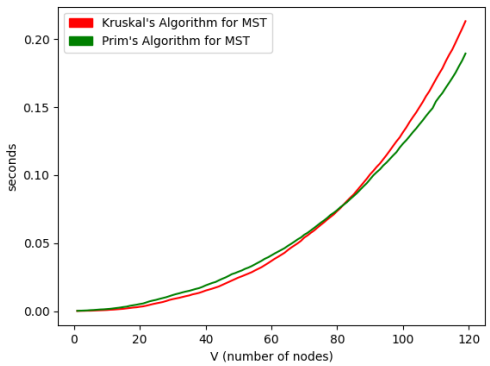


(a) $maxV = 120$, $p = 100\%$, la soglia limite è $limV \approx 38$.

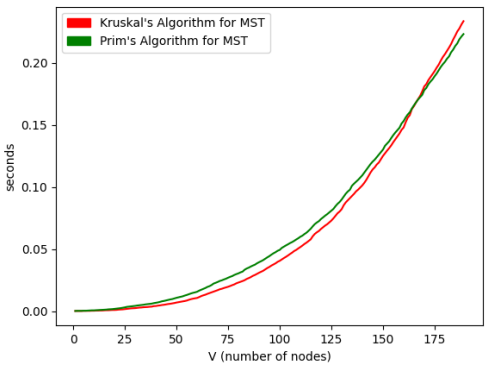
(b) Zoom della figura a fianco, per individuare meglio il punto di soglia.

Figure 6: Paragone tra la complessità temporale degli algoritmi di Kruskal e Prim, con probabilità p fissata e numero di nodi V crescente.

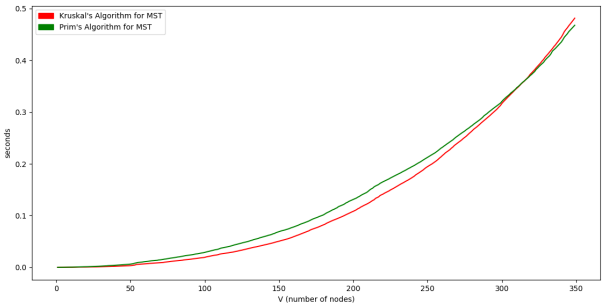
- La soglia $limV$ del numero di nodi oltre cui l'algoritmo di Prim è più efficiente dell'algoritmo di Kruskal è inversamente proporzionale alla probabilità p di presenza di archi: è tanto più alta quanto più bassa è la probabilità p fissata.
  In particolare, la soglia raddoppia ogni volta che la probabilità si dimezza.
  Nei grafici questo è evidente:
  - In figura 6a, dove la probabilità è fissata a $p = 100\%$, la soglia è $limV \approx 38$;
  - In figura 7a, dove la probabilità è fissata a $p = 50\%$, la soglia è $limV \approx 80$;
  - in figura 7b, dove la probabilità è fissata a $p = 25\%$, la soglia è $limV \approx 160$;
  - in figura 7c, dove la probabilità è fissata a $p = 12,\%$, la soglia è $limV \approx 320$.



(a) $p = 50\%$, $maxV = 120$, $limV \approx 80$.



(b) $p = 25\%$, $maxV = 190$, $limV \approx 160$.



(c) $p = 12{,}5\%$, $maxV = 350$, $limV \approx 320$.

Figure 7: Paragone tra la complessità temporale degli algoriti di Kruskal e Prim, con probabilità p fissata e numero di nodi V crescente.

6 Conclusioni

Gli esperimenti confermano la tesi: per grafi piccoli (la cui foresta è composta da pochi nodi ed è poco folta) l'algoritmo di calcolo dell'MST più efficiente è quello di Kruskal, per grafi di dimensione medio-grande l'algoritmo più conveniente è quello di Prim.