**TRACCIA 2: BORUVKA**

L’obiettivo principale di questo lavoro è stato implementare l’algoritmo di Boruvka. Per la corretta realizzazione del codice è stato necessario ideare un metodo aggiuntivo, il metodo BuildRandomGraph, che ha permesso di creare dei grafi random su cui è stato più facile effettuare testing. Infine, dopo esser riusciti a progettare un algoritmo il più efficiente possibile, la nostra attenzione si è concentrata sul confrontare, tramite il modulo Cprofile, in termini di tempistiche ed efficienza, Boruvka con Kruscal e Prim, gli altri due algoritmi in grado di restituire un albero ricoprente di peso minimo. In particolare, la nostra attenzione si è incentrata principalmente sulle differenze di prestazione tra Boruvka e Kruskal, l’ultimo dei quali si è rilevato più efficiente dell’altro.

**INTRODUZIONE: L’algoritmo di Boruvka**

Boruvka, Kruscal e Prim hanno la funzionalità di determinare un albero ricoprente di costo minimo per un qualsiasi grafo non orientato, connesso e pesato sugli archi.

Boruvka nella sua implementazione sfrutta un approccio cosiddetto “greedy” il quale permette di includere o escludere opportunamente determinati archi dalla soluzione. Esso inizialmente è costituito da n alberi blu disgiunti, ognuno contenente un vertice. Ora, per trovare il minimo albero ricoprente, l’algoritmo si basa sulla seguente idea: “Per ogni albero blu T, scegli un arco di costo minimo incidente su T. Colora di blu tutti gli arche scelti“ e tale comando va iterato fino a quando non rimane un unico albero blu, il minimo albero ricoprente cercato. Tuttavia, bisogna precisare che un arco potrebbe essere selezionato più di una volta senza problemi, mentre è necessario che i costi degli archi del grafo, di cui vogliamo ottenere il minimo albero ricoprente, siano tutti distinti; qualora mancasse questa condizione fondamentale si può perturbare lievemente il costo degli archi in modo da garantirne l’unicità, senza compromettere la soluzione.

L’algoritmo di Boruvka può essere implementato in modo da richiedere tempo O(mlogn) nel caso peggiore, dove con m definiamo gli archi del grafo e con n il numero dei vertici; nel caso peggiore, m può tendere ad O(n^2).

**Progettazione e Design**

Come abbiamo descritto in precedenza, per implementare Boruvka siamo partiti dalla creazione della funzione BuildRandomGraph che restituisce un grafo randomico e, inoltre, l’utilizzo di questo metodo ci ha consentito di aggirare il problema della perturbazione degli archi. Per fare ciò abbiamo sfruttato un metodo della classe GraphHelper che già conoscevamo, BuildGraph,; seguendo tale struttura abbiamo generato la nostra funzione che, dati in input due interi, rappresentanti il numero dei nodi e il numero degli archi, ci ha permesso di realizzare un grafo arbitrario.

Prima di tutto abbiamo definito due liste, una per i nodi e una per i pesi degli archi; in particolare la prima avrà un numero di elementi compreso tra 0 e il numero dei nodi dato in input, mentre la seconda sarà composta da pesi compresi tra 1 e il numero degli archi +1, dato che il range è stato traslato di 1. Attraverso un ciclo for, tale procedimento verrà iterato esattamente n volte tante quanti sono gli archi prestabiliti dall’input. In questa prima fase è stato utilizzato il metodo “**itertools”** (di cui discuteremo più avanti) che ci ha permesso di creare tutte le possibili coppie tra nodi.

Successivamente, dopo aver importato il modulo random, abbiamo eliminato un elemento dalla lista tramite pop scegliendo in modo casuale la sua posizione all’interno della lista stessa; tale passaggio è stato effettuato utilizzando randint che permette di individuare un valore compreso fra un minimo e un massimo, cioè per tutta la lunghezza della lista, determinando così la posizione nella lista dalla quale verrà eliminato l’elemento prima detto. Dopo ogni iterazione le due liste si aggiornano, perdendo passo dopo passo una coppia di vertici e un valore che coincide con il peso dell’arco; quando eliminiamo tali valori dalle liste ne conserviamo il valore cosicché si possa riutilizzare per aggiornare “ginfo”, una tupla di tuple contenente informazioni sul nostro grafo. In questo modo i pesi degli archi saranno sempre diversi; di conseguenza, così facendo, siamo riusciti a creare un grafo randomico superando il problema della perturbazione. Tuttavia, nello strutturare Boruvka, ci siamo accorti che BuildRandomGraph restituiva un grafo scelto casualmente e che non sempre era connesso, ma, essendo questa una condizione necessaria, attraverso la funzione controllo cleanGraph siamo riusciti ad apportare le giuste correzioni; tale funzione, infatti, ci ha permesso di aggiungere degli archi nell’eventualità in cui almeno un vertice fosse isolato, ossia avesse grado 0, generando in tal modo un grafo connesso e permettendo quindi il calcolo di MST per ogni algoritmo.

**ITERTOOLS**

Per rendere più efficiente in memoria la nostra struttura, ci siamo avvalsi di un nuovo metodo itertools che, attraverso la funzione combinations, la quale necessita di una lista e di un numero come input, restituisce una lista che ci ha permesso di creare una combinazione di valori di due elementi appartenenti a list\_of\_nodes, dando luogo quindi alle possibili coppie di vertici da mettere in una tupla.

Abbiamo deciso di adottare questo metodo perché è il migliore per capacità di consumo della memoria e perché, nel caso di insiemi di grandi elementi, ci semplificherebbe l’analisi se scegliessimo di fare un esempio con un numero di nodi ed archi elevato.

**Argparse**

Argparse è un modulo molto utile per creare un’interfaccia da linea di comando in tre semplici passi:

1. Importare Argparse
2. Creare un oggetto di tipo ArgumentParser
3. Aggiungere gli argomenti del programma;

tale modulo è fondamentale per eseguire tutte le prove che vogliamo per riga di comando. Nel creare il nostro progetto, dopo aver importato il modulo argparse, abbiamo creato un parser a cui abbiamo aggiunto, per mezzo di parser.addArgument, gli argomenti corrispondenti ai valori assegnati per riga di comando, cioè nodi ed archi, prestando attenzione al type che, in questo caso, coincide con un intero. Infine per richiamare e stampare gli argomenti è stato impiegato parse\_args.

**Cprofile**

Per eseguire i confronti, abbiamo utilizzato il modulo Cprofile che non è un altro che un’estensione proveniente dal C in grado di fornirci delle statiche riguardanti le varie funzioni che va ad analizzare. Attraverso questo modulo, possiamo usare la funzione profile.runctx() la quale, rispetto al classico run, ci permette di prendere come valori delle variabili locali o globali; così facendo il tutto va a dipendere dal contesto in cui è utilizzato. In questo caso, siccome passiamo un grafo ”G” alle varie funzioni, siamo costretti ad usare runctx.

**IMPLEMENTAZIONE DI BORUVKA**

L’idea che abbiamo utilizzato nel realizzare Boruvka può considerarsi la stessa che vi è dietro la progettazione dell’algoritmo di Prim,; infatti, così, come quest’ultimo aggiornava continuamente la sua frontiera fino ad arrivare al minimo albero ricoprente, Boruvka, allo stesso modo aggiorna, passo dopo passo, la lista degli archi di costo minore e, ogni qual volta rileva degli archi di costo minore che non sono già presenti nello stesso albero blu, li aggiunge alla soluzione fino ad ottenere un unico albero blu.

Nell’ideazione di Boruvka abbiamo seguito delle linee guida generali:

1. In input l’algoritmo deve sempre prendere un grafo pesato, connesso e orientato; affinché fosse possibile ciò abbiamo creato, come già detto, il metodo BuildRandomGraph in grado di restituirci un grafo randomico a cui, inoltre, è stato applicato cleanGraph in modo da renderlo sempre connesso.
2. Per dar vita al codice è stato necessario, in primis, impostare una foresta di alberi blu corrispondente inizialmente a tutti i nodi del nostro grafo: avevamo, dunque, esattamente n vertici e n alberi blu.
3. Si è dovuto poi definire una lista che tenesse conto di tutti gli archi di costo minore per ogni albero blu incontrato; tale lista ci è risultata utile perché ci forniva gli archi più “economici” che, poi, hanno permesso la fusione dei nostri alberi blu.
4. Si è realizzata una classe Find in cui abbiamo implementato dei metodi, che sfruttano tecniche di PathCompression e union by rank, necessari per le fusioni degli alberi blu e per la realizzazione dei cosiddetti super-vertici, alberi blu unici. Dopodiché si è proceduto con l’introduzione di un metodo find utile per verificare se due vertici sono parte dello stesso albero blu e, quindi, per individuare gli eventuali archi da escludere nella soluzione.
5. Impostare MST vuoto
6. Finché non si arriva ad un unico albero blu, va trovato l’arco di peso minore che connette tale albero blu ad un qualsiasi altro e, a mano a mano, è necessario aggiornare la lista degli archi dell’MST; iterando tale passaggio si ottiene un unico albero blu 🡪 MST.

**ANALISI PRESTAZIONI**

Le performance dei tre algoritmi sono state valutate tramite l’utilizzo del modulo Cprofile il quale restituisce, oltre al tempo totale impiegato per calcolare il minimo albero ricoprente, i tempi delle varie funzioni che sono state richiamate per portare a termine l’esecuzione dei differenti algoritmi.

Quindi il primo valore restituito, come detto, è il tempo totale d’esecuzione, da cui si nota che l’algoritmo di Kruscal risulta più efficiente di Boruvka, nonostante il parallelismo tempistico fra i due, dovuto principalmente all’implementazione più lineare del primo, ottenuta tramite strutture dati quali Union-Find che risultano più proficue. Al fine di esaminare al meglio queste differenze si possono distinguere più casi su cui far lavorare i 3 algoritmi.

1. Una prima situazione è quella in cui abbiamo confrontato le prestazioni degli algoritmi attraverso la creazione di un grafoRandom con 10 archi e 10 nodi nella quale non si è riscontrato un differente tempo totale per la lo svolgimento degli algoritmi; si è notato così che in grafi con un basso numero di archi e nodi non si vengono a evidenziare tempistiche differenti . Tuttavia è possibile osservare come il numero di chiamate alle varie funzioni, necessarie per l’effettuazione dell’algoritmo, sia decisamente a favore di Kruscal che ne impiega un numero quasi pari alla metà di quelle necessarie per gli altri due.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | NCALLS | TOTTIME | PERCALL | CUMCALL | PERCALL | | FILENAME | |
| KRUSKAL | 1 | 0.000 | 0.000 | 0.000 | | 0.000 | | <string>:1(<module>) | |
| PRIM | 1  1  1  54  1  1  93  1  55 | 0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 | 0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 | 0.000  0.000  0.000  0.000  0.001  0.001  0.000  0.001  0.000 | | 0.000  0.000  0.000  0.000  0.001  0.001  0.000  0.001  0.000 | | Sorting.py:166(quicksort)  Mst.py: 64(Kruscal)  {built-in method .exec}  {method ‘app.’ ‘list’obj.}  <string>:1(<module>)  Mst.py: 87(Prim)  {built-in method built.len}  {built-in method built.exec}  {method ‘app.’ ‘list’obj.} | |
| BORUVKA | 1  2  1  144/108  1 | 0.000  0.000  0.000  0.000  0.000 | 0.000  0.000  0.000  0.000  0.000 | 0.001  0.000  0.001  0.000  0.001 | | 0.001  0.000  0.001  0.000  0.001 | | <string>:1(<module>)  Sorting.py:166(quicksort)  Mst.py: 64(Kruscal)  Mst.py: 29(find)  {built-in method built.exec} | |

Allora, in una situazione del genere, Kruskal esegue 339 chiamate di funzioni impiegando un tempo pari a 0.000; Prim, invece, ne effettua 554 in 0.001 secondi, lo stesso tempo in cui Boruvka svolge 622 chiamate di funzioni.

1. Il secondo caso di studio è quello in cui il grafo random presenta 100 archi e 100 nodi ; a questo punto si cominciano ad evincere le prime differenze a livelli tempistici, soprattutto per quanto riguarda Prim e Boruvka, mentre vengono avvalorate le precendenti considerazioni su Kruskal che si conferma essere ancora il più efficace. Quindi con 100 archi e 100 nodi Prim risulta essere il meno efficiente mentre Boruvka riesce a tenere il passo di Kruscal anche se con difficoltà.

In questo secondo caso, Kruskal esegue 3147 chiamate di funzioni impiegando un tempo pari a 0.004; Prim, invece, ne effettua 24206 in 0.014 secondi, mentre Boruvka svolge 13799 chiamate di funzioni in 0.012 secondi.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Ncalls | tottime | percall | cumtime | percall | filename |
| Kruskal  Prim  Boruvka | 1  1  1  1  787  1  1  1  6650  2943  1  3  1  1  893  2770/1638 | 0.000  0.000  0.000  0.000  0.000  0.000  0.001  0.000  0.000  0.000  0.000  0.000  0.001  0.000  0.000  0.001 | 0.000  0.000  0.000  0.000  0.000  0.000  0.001  0.000  0.000  0.000  0.000  0.000  0.001  0.000  0.000  0.000 | 0.004  0.002  0.004  0.004  0.000  0.014  0.014  0.014  0.000  0.000  0.012  0.007  0.012  0.012  0.000  0.001 | 0.004  0.002  0.004  0.004  0.000  0.014  0.014  0.014  0.000  0.000  0.012  0.002  0.012  0.012  0.000  0.000 | <string>:1(<module>)  Sorting.py:166(quicksort)  Mst.py: 64(Kruskal)  {built-in method .exec}  {method ‘app.’ ‘list’obj.}  <string>:1(<module>)  Mst.py: 87(Prim)  {built-in method .exec}  {built-in method .len}  {method ‘app.’ ‘list’obj.}  <string>:1(<module>)  Sorting.py:166(quicksort)  Mst.py: 144(Boruvka)  {built-in method .exec}  {method ‘app.’ ‘list’obj.}  mst.py:29(find) |

1. La terza possibilità da esaminare è quella in cui il grafo presenta 1000 archi e 1000 nodi; arrivati a questo punto si noterà come Prim aumenti drasticamente il suo tempo di esecuzione che andrà sfociare all’incirca in un secondo, mentre Boruvka riuscirà mantenersi su tempi dell’ ordine dei decimi di secondo; Kruscal resterà su tempi dell’ordine dei centesimi di secondo. Per quanto riguarda il numero di funzioni richiamate dai 3 algoritmi, Kruskal si conferma, anche in questo caso, essere il migliore, con in media un numero di “function calls” pari a cinquantamila, mentre Prim raggiunge un numero di quasi 2 milioni e Boruvka circa un centinaia di migliaia.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Ncalls | tottime | percall | cumtime | percall | filename |
| Kruskal  Prim  Boruvka | 1  1  1  1  8154  1  1  1  782033  274530  1  5  1  1  12994  60244/24640 | 0.000  0.000  0.003  0.000  0.001  0.000  0.016  0.000  0.049  0.024  0.000  0.000  0.019  0.000  0.001  0.016 | 0.000  0.000  0.003  0.000  0.000  0.000  0.016  0.000  0.000  0.000  0.000  0.000  0.019  0.000  0.000  0.000 | 0.044  0.025  0.044  0.044  0.001  0.856  0.856  0.857  0.049  0.024  0.198  0.123  0.198  0.199  0.001  0.016 | 0.044  0.025  0.044  0.044  0.000  0.856  0.856  0.857  0.000  0.000  0.198  0.025  0.198  0.0199  0.000  0.000 | <string>:1(<module>)  Sorting.py:166(quicksort)  Mst.py: 64(Kruskal)  {built-in method .exec}  {method ‘app.’ ‘list’obj.}  <string>:1(<module>)  Mst.py: 87(Prim)  {built-in method .exec}  {built-in method .len}  {method ‘app.’ ‘list’obj.}  <string>:1(<module>)  Sorting.py:166(quicksort)  Mst.py: 144(Boruvka)  {built-in method .exec}  {method ‘app.’ ‘list’obj.}  Mst.py:29(find) | |

In quest’ultima situazione, Kruskal esegue 60018 chiamate di funzioni impiegando un tempo pari a 0.044; Prim, invece, ne effettua 2181768 in 0.857 secondi, mentre Boruvka svolge 292826 chiamate di funzioni in 0.199 secondi.

Nelle tabelle sono stati inseriti, per ogni algoritmo, solamente quelle funzioni che restituivano valori (tempi, chiamate di funzioni, ecc..) rilevanti ai fini della prestazione dell’algoritmo.

**Conclusioni e osservazioni finali**

Nei vari testing si è notato che i tempi maggiori scaturiscono dall’esecuzione di metodi built-in, dal richiamare la funzione all’interno del file mst.py oppure da metodi(sortEdges) importati da altre classi (GraphHelper) e da funzioni – quicksort – utilizzate, ad esempio, per la realizzazione di questi metodi. In particolare, si può sottolineare come Boruvka e Kruskal rispettino i tempi di esecuzione con cui sono stati ideati – O(mlogn) – a meno di una costante poco influente ai fini dell’efficienza totale.

Un’altra osservazione da fare è che tutti i test effettuati, relativi alle varie tempistiche, sono stati eseguiti su algoritmi (Kruscal e Boruvka) che utilizzavano una variante dell’algoritmo sortEdges, necessario per ottenere una lista ordinata di archi. Infatti, per i confronti, abbiamo utilizzato un “SortEdges” che riordinasse gli archi attraverso quickSort invece di mergeSort poiché abbiamo ritenuto il primo più lineare ed efficiente e oltretutto meno dispendioso dal punto di vista della memoria.

Purtroppo, si è evidenziato anche che, per un numero eccessivo di nodi e di archi, solitamente maggiori ai 10000, il codice dà un errore di memoria causato, probabilmente, dal metodo itertools usato nella realizzazione del grafo random, il quale salva in memoria tutte le possibili permutazioni di nodi e archi, così da generare un messaggio di errore. Presumibilmente, sarebbe stato più produttivo tenere in considerazione solo le combinazioni correnti di nodi e archi in modo da evitare l’errore.

Infine, si è notato che sotto determinate ipotesi, il comportamento di Boruvka e Prim può deviare dalla normalità. In particolare, se il numero dei nodi è pari a quello degli archi e varia tra 12 e 100, oppure, se, fissato il numero di archi, il numero dei nodi risulta essere molto inferiore rispetto a questo, Prim può alternarsi a Boruvka divenendo più efficiente di quest’ultimo; in entrambi i casi, esiste un vincolo dopo il quale Boruvka diventa definitivamente migliore di Prim, ovvero il valore di 100 archi, che rappresenta, quindi, il cosiddetto “collo di bottiglia” per la maggior efficienza di Boruvka.

Marco Lanciotti - 0220422

Andrea Lombardo - 0217894

Leonardo Monnati -0228618