Implementazione dell'algoritmo di Borůvka

STEFANO PICA

GIUSEPPE LASCO

FRANCESCO GELSOMINO

Introduzione all'algoritmo

- Avendo un grafo, l'algoritmo di Borůvka permette di ricavarne il minimum spanning tree (minimo albero ricoprente). L'algoritmo parte considerando ogni nodo del grafo come un albero. Per ogni albero seleziona l'arco di costo minimo uscente da esso che lo collega agli alberi rimanenti. Il processo termina quando ne rimane solo uno, il minimo albero ricoprente.
  Il tempo richiesto da questo algoritmo è O(mlogn) con n numero di nodi ed m numero di archi.
- Tale algoritmo può essere implementato senza una struttura dati particolare.
  Noi abbiamo pensato di usare la struttura dati Union-Find che ci permette di tenere traccia dell'appartenenza dei nodi ai vari alberi, attraverso le Find sui nodi, e di fondere tali alberi durante l'esecuzione, attraverso le Union.

Progettazione

- Per consentire l'implementazione dell'algoritmo di Borůvka abbiamo creato un modulo, chiamato MyGraphHelper, che importa graphHelper. In questo modulo abbiamo definito alcuni metodi statici tra cui listEdge che crea una lista di archi su cui lavora l'algoritmo e edgeSentinel che crea un arco sentinella con peso ad infinito con il quale vengono fatti i confronti durante l'esecuzione di Borůvka.
- Abbiamo modificato il metodo buildGraph rispetto alla classe padre in modo tale che sia in grado di creare archi con pesi random, sia perturbati che non. Inoltre abbiamo inserito il metodo fastBuildGraph che permette di creare grafi più velocemente ma con minore accuratezza. I due metodi si avvalgono di cleanGraph per restituire grafi connessi e senza archi ridondanti.

Progettazione

- Per permettere di controllare la compatibilità dei parametri da passare al buildGraph, quindi costruire il grafo, abbiamo creato un modulo mstParser che prende come parametri:
  - Il numero di nodi
  - Il numero di archi
  - Flag che permette di scegliere se creare archi con peso perturbato
  - Flag che consente di creare coppie di archi con lo stesso peso

Progettazione

- Infine abbiamo creato la funzione Borůvka. Questa funzione si avvale della struttura union-find (quick-union con path compression), che ci permette di tenere traccia dell'appartenenza dei nodi ai vari alberi (attraverso le Find sui nodi) e fondere tali alberi durante l'esecuzione (attraverso le Union). Abbiamo creato una lista di archi sentinella lunga quanto il numero di nodi del grafo; questa viene aggiornata durante l'esecuzione confrontando gli archi presenti in essa con quelli appartenenti al grafo e scegliendo sempre quello di costo minimo.
- Al termine dei confronti tale lista conterrà l'insieme degli archi di costo minimo uscenti da ogni nodo. Facendo una scansione di quest'ultima si fondono gli alberi collegati dagli archi appena analizzati finché non si avrà un unico albero. La lista così ottenuta conterrà gli archi del minimum spanning tree.
- Abbiamo fatto in modo che tale funzione gestisca anche archi con lo stesso peso avvalendosi della posizione, quindi dell'indice, all'interno della lista di archi.

Implementazione

Il codice di Borůvka è strutturato in diverse fasi

- Nella prima fase inizializziamo:
- l'arco 'sentinella'
- la lista di archi sfruttando il metodo listEdge descritto in precedenza
- la variabile mstWeight: mantiene aggiornato il peso totale dell'albero
- la variabile numTree: inizialmente corrisponde al numero dei nodi, ovvero gli alberi iniziali, successivamente si ridurrà logaritmicamente nel corso dell'esecuzione in modo da terminare il programma nel momento in cui il valore sarà uguale a 1, ovvero avremo l'albero finale
- la lista di archi temporanei (tempMstEdge) contenente archi sentinella che saranno aggiornati e rimpiazzati dagli archi di costo minimo uscenti da ogni nodo nel corso dell'esecuzione.
- una lista vuota mstEdge che raccoglie gli archi del minimo albero ricoprente

A questo punto eseguiamo n makeSet (dove n è il numero di nodi).

Implementazione

- Fase di ricerca del minimo albero ricoprente.
  Questa fase è controllata da un ciclo while che utilizza la variabile numTree come condizione di uscita. Scorrendo la lista di archi listEdge analizziamo il singolo arco, quindi effettuiamo una find sulla testa e sulla coda dell'arco e controlliamo se questi due nodi fanno parte dello stesso albero o meno. Nelle linee di codice 136-139 si effettua una find-root sul nodo testa e nodo coda dell'arco restituendoci un oggetto di tipo 'nodo'. Facendo una find su tale oggetto, risaliamo all'elemento che possiamo utilizzare come indice nella lista degli archi temporanei.
  Nel caso in cui fanno parte dello stesso albero, gli archi non vengono considerati e non vi è alcuna alterazione delle variabili. Nel caso in cui non fanno parte dello stesso albero si controlla se nelle posizioni relative alla testa e alla coda, nella lista degli archi temporanei, vi è un arco sentinella oppure un arco con peso maggiore rispetto a quello analizzato. In tal caso si sostituisce l'arco in quelle determinate posizioni con l'arco in analisi, altrimenti si prosegue. Da notare che in questa fase l'algoritmo gestisce anche archi con peso uguale avvalendosi dell'indice in listEdge.
  In seguito scorriamo la lista degli archi temporanei ed eseguiamo una union nel caso in cui la testa e la coda sono differenti, ovvero appartengono a due alberi distinti, in questo modo fondiamo i due alberi e non consideriamo tutti gli archi interni ad essi aggiornando la lista finale degli archi e il peso complessivo dell'albero.
  Questo procedimento viene ripetuto fino al raggiungimento di un unico albero, il minimo albero ricoprente.
  La funzione restituisce il peso totale dell'albero e la lista di archi che ne fanno parte.

Fase di test

- Premessa:
  - Facendo un'analisi del nostro codice abbiamo concluso che esso rispetta, in linea di massima, il tempo di esecuzione proprio dell'algoritmo, ovvero O(mlogn), poiché il ciclo while esterno termina dopo logn volte e per ogni ciclo vengono scansionati tutti gli archi.
  - Nella restituzione del peso complessivo del mst, su grafi con archi perturbati, i tre algoritmi danno risultati che variano di un'approssimazione poco significativa nonostante prendano i medesimi archi. Abbiamo commentato le linee di codice che permettono la stampa delle liste di archi ordinate restituite da Kruskal e Borůvka per dimostrare che restituiscono gli stessi archi anche se la somma dei pesi risulta diversa.

Fase di test

- Riportiamo di seguito i risultati di alcuni test

```
Tue Feb 28 01:09:24 2017    /Users/stefanopica/provaProfile1000-1000

         2399441 function calls (2396457 primitive calls) in 1.099 seconds

   Ordered by: cumulative time
   List reduced from 420 to 9 due to restriction <'mst_Boruvka'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    1.099    1.099 mst_Boruvka.py:4(<module>)
        1    0.001    0.001    1.078    1.078 mst_Boruvka.py:211(main)
        1    0.016    0.016    0.876    0.876 mst_Boruvka.py:55(prim)
        1    0.032    0.032    0.079    0.079 mst_Boruvka.py:113(boruvka)
        1    0.003    0.003    0.048    0.048 mst_Boruvka.py:32(kruskal)
        1    0.001    0.001    0.003    0.003 mst_Boruvka.py:251(<listcomp>)
        1    0.001    0.001    0.002    0.002 mst_Boruvka.py:230(<listcomp>)
        1    0.000    0.000    0.000    0.000 mst_Boruvka.py:241(<listcomp>)
        1    0.000    0.000    0.000    0.000 mst_Boruvka.py:25(MST)
```

cProfile con 1000 nodi 1000 archi

Fase di test

```
Tue Feb 28 01:11:40 2017    /Users/stefanopica/provaProfile10000-10000

         203911604 function calls (203879970 primitive calls) in 76.945 seconds

   Ordered by: cumulative time
   List reduced from 420 to 9 due to restriction <'mst_Boruvka'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.012    0.012   76.945   76.945 mst_Boruvka.py:4(<module>)
        1    0.016    0.016   76.909   76.909 mst_Boruvka.py:211(main)
        1    0.194    0.194   74.268   74.268 mst_Boruvka.py:55(prim)
        1    0.460    0.460    1.244    1.244 mst_Boruvka.py:113(boruvka)
        1    0.037    0.037    0.600    0.600 mst_Boruvka.py:32(kruskal)
        1    0.007    0.007    0.026    0.026 mst_Boruvka.py:251(<listcomp>)
        1    0.006    0.006    0.023    0.023 mst_Boruvka.py:230(<listcomp>)
        1    0.003    0.003    0.003    0.003 mst_Boruvka.py:241(<listcomp>)
        1    0.000    0.000    0.000    0.000 mst_Boruvka.py:25(MST)
```

cProfile con 10000 nodi 10000 archi

Fase di test

```
Tue Feb 28 01:15:01 2017    /Users/stefanopica/provaProfile10000-100000

         228457902 function calls (228246498 primitive calls) in 107.874 seconds

   Ordered by: cumulative time
   List reduced from 420 to 9 due to restriction <'mst_Boruvka'>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.019    0.019  107.874  107.874 mst_Boruvka.py:4(<module>)
        1    0.185    0.185  107.836  107.836 mst_Boruvka.py:211(main)
        1    3.230    3.230   90.277   90.277 mst_Boruvka.py:55(prim)
        1    2.781    2.781    7.121    7.121 mst_Boruvka.py:113(boruvka)
        1    0.155    0.155    6.078    6.078 mst_Boruvka.py:32(kruskal)
        1    0.007    0.007    0.029    0.029 mst_Boruvka.py:230(<listcomp>)
        1    0.007    0.007    0.026    0.026 mst_Boruvka.py:251(<listcomp>)
        1    0.003    0.003    0.003    0.003 mst_Boruvka.py:241(<listcomp>)
        1    0.000    0.000    0.000    0.000 mst_Boruvka.py:25(MST)
```

cprofile con 10000 nodi 100000 archi

Conclusione

- Come possiamo notare dai vari test, il nostro codice risulta più lento di Kruskal, probabilmente per la sua maggiore complessità, per l'esecuzione di un numero maggiore di controlli e per la doppia scansione della lista di archi, ma comunque molto più veloce di Prim.
- Abbiamo inserito, inoltre, l'import della classe Edge in graphHelper e abbiamo aggiunto il metodo __str__ in CmpEdge in modo da stampare l'oggetto arco.

Istruzioni per l'uso

- Il programma parte da linea di comando chiamando il modulo mst_Boruvka.py con i parametri (nodi, archi, flag per perturbare i pesi, e flag per produrre archi con peso uguale);
- il parser controlla i parametri che il buildGraph (o il fastBuildGraph) prendono per creare il grafo che infine viene utilizzato dagli algoritmi di Kruskal, Prim e Borůvka.

  Naturalmente i codici funzionano anche in maniera indipendente.
- Tutti i file contenuti nell'archivio dovranno essere posti nella cartella mst tranne CmpEdge che deve essere posto in mst/tree.