Relazione di Algoritmi Avanzati

Laboratorio 3 - Minimum Cut

Nicola Carlesso - matricola 1237782
Federico Brian - matricola 1243422

A.A. 2019/2020

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

Il presente documento descrive le scelte architetturali ed implementative del terzo elaborato di laboratorio del corso di Algoritmi Avanzati. Di seguito, verrà offerta una panoramica sul lavoro svolto dagli studenti Nicola Carlesso e Federico Brian, riguardante le prestazioni dell'algoritmo di Karger per il problema del *Minimum Cut*[1], rispetto a quattro parametri:

* Il tempo impiegato dalla procedura di `FullContraction`;
* Il tempo impiegato dall'algoritmo completo per ripetere la contrazione un numero sufficientemente alto di volte;
* Il *discovery time*, ossia il momento in cui l'algoritmo trova per la prima volta il taglio di costo minimo
* L'errore nella soluzione trovata rispetto al risultato ottimo.

Il problema del minimum cut è definito come segue.

1.1 Definizioni preliminari

Per definire il problema del mincut, abbiamo bisogno di dare prima delle definizioni topiche:

Def. 1.1. ***(Multiinsieme).*** *Collezione di oggetti con ripetizioni.*

* $S = \{\{v : v \in S\}\}$
* $\forall\ v \in S\ m(v) \in \mathbb{N} \setminus \{0\}$ *con* $m(v)$ = *molteplicità di* v*, cioè quante copie di* v *ci sono in* S*.*

Def. 1.2. ***(Multigrafo non orientato).*** *multigrafo* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *tale che:*

* $\mathcal{V} \subseteq \mathbb{N}, \mathcal{V}$ *finito;*
* $\mathcal{E}$ *è un multiinsieme a elementi del tipo* $\{u, v\} : u \neq v$ *(no self loops).*

Remark: un grafo semplice grafo $G = (V, E)$ è anche un multigrafo. Il viceversa non è vero.

Def. 1.3. ***(Cammino).*** *In un multigrafo, un cammino è una sequenza di nodi dove* $\forall\ u, v \in \mathcal{V}$ $\Rightarrow \exists\ (u, v) \in \mathcal{E}$.

Cioè esiste almeno un lato che connette ogni coppia di nodi in $\mathcal{V}$.

Def. 1.4. ***(Connettività).*** *Un multigrafo è connesso se* $\forall\ u, v \in \mathcal{V} \Rightarrow \exists$ *un cammino che li connette.*

Def. 1.5. ***(Taglio).*** *Dato un multigrafo* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *connesso, un taglio* $\mathcal{C} \subseteq \mathcal{E}$ *è un multiinsieme di lati tale che* $\mathcal{G}' = (\mathcal{V}, \mathcal{E} - \mathcal{C})$ *non è connesso.*

Equivalentemente, si può dire che $\mathcal{G}'$ ha almeno due componenti connesse.

[1] d'ora in poi mincut

1.2 Problema del Minimum Cut

"Dato un multigrafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ non orientato, un taglio $\mathcal{C} \subseteq \mathcal{E}$ è il multiinsieme di lati di cardinalità minima tale che $\mathcal{G}' = (\mathcal{V}, \mathcal{E} \setminus \mathcal{C})$ non è connesso."

2 Scelta del linguaggio di programmazione

Per lo svolgimento di questo *assignment*, come per il precedente, è stato scelto Java nella sua versione 8 come linguaggio di programmazione. La scelta è derivata, principalmente, da due fattori:

* è stato sia studiato durante il percorso di laurea triennale, sia approfondito autonomamente da entrambi;
* in Java, è possibile utilizzare riferimenti ad oggetti piuttosto che oggetti stessi. Questo ha permesso un'implementazione degli algoritmi che si potrebbe definire "accademica", perché coerente con la complessità dichiarata e semanticamente vicina allo pseudocodice visto a lezione. Sono infatti evitate complessità aggiuntive derivanti da inavvertite copie profonde.

3 Scelte implementative

Come specificato nel precedente paragrafo, nell'implementazione dei tre algoritmi si è cercato di creare meno oggetti possibile usando quasi esclusivamente riferimenti. Questo ha permesso non solo un risparmio in termini di memoria ma anche di prestazioni: nell'implementazione abbiamo infatti cercato di creare strutture dati che risparmiassero quanta più memoria possibile, dato che l'algoritmo *HeldKarp*, oltre ad essere computazionalmente oneroso, richiede anche l'utilizzo di molta memoria. Difatti, nell'eseguire l'esecuzione del `main`, è necessario richiedere l'utilizzo di più memoria RAM per il corretto funzionamento dell'algoritmo attraverso il flag `-Xmx6500m`.

3.1 Modello

Le componenti del modello, vale a dire le classi presenti all'interno del `package` chiamato `lab2.model`, comprendono tutte le strutture dati utilizzate nella risoluzione dei tre problemi assegnati.

* `AdjacentMatrix`: matrice di adiacenza usata per rappresentare i grafi, che si presenta come una normale matrice $n \times n$ simmetrica rispetto alla sua diagonale nulla[2]. Gli algoritmi `HeldKarp`, `CheapestInsertion` e `TriangleTSP`, difatti, la trattano come tale. In verità è stata implementata come matrice triangolare inferiore, contenendo `n - 1` array di lunghezza crescente: questa scelta è stata fatta per risparmiare memoria, evitando di costruire una matrice quadrata. Tale classe presenta i metodi standard `get(u,v)` e `set(u,v)`, per ottenere ed impostare il peso del lato (u,v), ed un motodo `getMinAdjacentVertexWeightIndex(v)` per ottenere il lato con peso minore che ha per estremo il vertice v;
* `Graph`: contiene esclusivamente una matrice di adiacenza, dato che tutte le informazioni necessarie dei nodi (i quali iniziano ad essere contati da 0) e dei lati possono essere ottenute analizzando la matrice di adiacenza;
* `Node`: classe non utilizzata direttamente da *Graph*, ma usata per costruire il MST (Minimum Spanning Tree) per l'algoritmo di 2-approssimazione richiesto. Essa dunque contiene solo l'*ID*, un riferimento al nodo padre e una lista di riferimenti ai nodi figli;

[2] poiché non esistono cappi

* `Edge`: come la classe *Node*, anche *Edge* viene utilizzata solo per costruire l'MST attraverso l'algoritmo *Kruskal* implementato nello scorso assignment.
* `DisjointSet`: questa struttura dati gestisce partizioni di oggetti, rappresentati con un numero intero che li identifica. Ogni oggetto può stare in una sola delle partizioni degli insiemi disgiunti presenti. La struttura dati è utilizzata all'interno dell'algoritmo `Kruskal`.
* `TSP`: contiene tutti gli algoritmi richiesti dagli assignment, più le funzioni ausiliarie per il corretto funzionamento di questi ultimi, in particolar modo è importante menzionare:
  - `copyWithoutV`: funzione *utility* di `HeldKarpCore` che serve a ricreare l'insieme $S \setminus \{v\}$, necessaria all'algoritmo di Held e Karp per proseguire con le chiamate ricorsive;
  - `getResults`: funzione che ritorna $(d[0, V], T)$, cioè il valore del ciclo hamiltoniano di costo minimo disponibile dopo T minuti di computazione;
  - `preorder`: metodo utilizzato per l'algoritmo di 2-approssimazione *TrangularTSP*, per ottenere una lista pre-ordinata dei nodi del MST ottenuto dal metodo `Kruskal`.

  TSP contiene inoltre tre campi dati, utilizzati dall'algoritmo `HeldKarp`:

  - `d`: struttura dati che rappresenta $(d[v, S], \texttt{cost})$, cioè che mappa la coppia <v, S> con il corrispondente peso del cammino minimo che parte da 0 e termina in v, visitando tutti i nodi in S. Per memorizzare il valore di $d[v, S]$, quindi, si inserisce una nuova chiave <v, S> in `d` e la si mappa con il rispettivo costo minimo calcolato. Per recuperare un valore, invece, si accede alla struttura dati con la chiave <v, S>, chiedendo quale valore sia stato ivi memorizzato. Nell'implementazione, quindi, è stato scelto di rappresentarla con il tipo `HashMap<Pair<Integer, ArrayList<Integer>>, Integer>`;
  - `pi`: struttura dati dello stesso tipo e del tutto simile a `d`, che mappa la medesima chiave con un valore intero che rappresenta il nodo predecessore di v;
  - `w`: variabile segnaposto della matrice di adiacenza del grafo su cui calcolare `HeldKarp`, incapsulata nella classe `TSP` al solo scopo di offrire maggior leggibilità e coerenza con lo pseudocodice visto in classe.

3.2 Algoritmi

Il package `lab2.algorithm` contiene un'unica classe, `TSP`, che permette di risolvere il problema TSP utilizzando gli algoritmi `HeldKarp`, `CheapestInsertion` e `TrangularTSP`.

* `HeldKarp`: funzione adibita all'inizializzazione e alla chiamata dell'algoritmo di Held e Karp. Si occupa di preparare l'ambiente necessario alla funzione `HeldKarpCore` e di catturare eventuali eccezioni lanciate dalla JVM[3] dovute alla mancanza di memoria (`OutOfMemoryException`);
* `HeldKarpCore`: funzione principale che implementa l'algoritmo di Held e Karp, del tutto coerente con lo pseudocodice visto a lezione. L'unica differenza è, naturalmente, la gestione degli *interrupt*: se una certa istanza di TSP dovesse metterci più del tempo a disposizione T[4], l'algoritmo interrompe immediatamente ogni ulteriore visita del grafo lasciando comunque l'opportunità di memorizzare i risultati finora calcolati. Questo comporta, a volte, un ritardo sui tempi di computazione consentiti: ciò non è visto dagli studenti come un problema poiché, di fatto, la computazione viene interrotta quindi non si è in presenza di alcun *cheat*. L'algoritmo ha complessità $\mathcal{O}(n^2 \cdot 2^n)$

[3] Java Virtual Machine

[4] come avviene nella maggior parte dei casi

* `CheapestInsertion`: l'algoritmo fa uso del metodo `getMinAdjacentVertexWeightIndex` per trovare il lato col peso minore che ha come estremo il nodo 0. Viene creato un array di nodi che rappresenta il cammino per TSP ed un array coi nodi ancora non visitati. L'algoritmo dunque esegue un ciclo fino a quando il cammino non raggiunge lunghezza $n + 1$ e trova per ogni nodo non visitato (e per ogni lato presente nel cammino trovato fino a quel momento) il minore `minCost`, indicando che il nodo k non visitato deve essere inserito nel cammino e in quale posizione. L'algoritmo ha complessità $\mathcal{O}(n^3)$;

* `TriangleTSP`: l'algoritmo, attraverso l'utilizzo di `Kruskal`, ottiene prima il MST sotto forma di `Node` (un nodo che possiede la lista di puntatori ai nodi figli), dopodiché ottiene la lista pre-oridinata dei nodi dell'albero ottenuto. L'algoritmo ha complessità $\mathcal{O}(m \log n + n)$.

3.3 Main

Il package `lab2.main` contiene la classe `Main`, responsabile dell'esecuzione degli algoritmi.
All'interno vi sono tre funzioni:

* la funzione `Main` che fa partire il calcolo oppure il test del costo della soluzione per TSP secondo l'algoritmo desiderato;

* la funzione `compute` che si occupa di calcolare il costo della soluzione per TSP di ogni grafo presente nel dataset, di salvarlo in un file di testo e di proseguire al test dello stesso.
  Per utilizzare/testare i tre diversi algoritmi, inserire all'interno della funzione `compute` una tra le seguenti stringhe:

  – `HeldKarp` per l'algoritmo di Held e Karp;
  – `Heuristic` per l'algoritmo di 2-approssimazione che sfrutta l'euristica strutturale della disuguaglianza triangolare;
  – `2Approx` l'algoritmo che attraverso il calcolo del MST calcola una soluzione per TSP con 2-approssimazione.

* la funzione `test`, che esegue alcuni, banali, test sugli algoritmi utilizzati come *Kruskal*.

* le funzioni `printHeapInfo` e `formatSize` che sono adibite, rispettivamente, a fornire informazioni e rappresentare l'utilizzo di memoria degli algoritmi.

3.4 Test

Il package `lab2.test` contiene due classi:

* `TestTSP` il cui scopo è di testare la bontà delle soluzioni ritornate con i tre algoritmi sviluppati e di calcolarne l'eventuale errore relativo;

* `TestKruskal` che è servita per testare il funzionamento dell'algoritmo di *Kruskal*.