Shortest Path: confronto tra Dijkstra, BFS e d-Heap

Arriu Simone - 65170, Serra Sergio - 65165

09 Settembre 2020

Contenuti

1 Obiettivo

L'obiettivo del progetto è quello di risolvere un problema di shortest path utilizzando tre differenti algoritmi: algoritmo di Dijkstra, algoritmo di Dijkstra con binary heap e breath-first search (BFS). Il confronto prevede delle comparazioni tra grafi con numero di nodi n e archi m via via crescenti, in cui viene riportato il valore della funzione obiettivo e il tempo di esecuzione. Le complessità dei tre algoritmi sono rispettivamente $O(n^2)$, $O(m\ log\ n)$ e $O(n+m)$.

Per testare i tre algoritmi, è stato usato un pc con le seguenti caratteristiche:

- *Processore*: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
- *RAM:* 16GB
- *Architettura*: 64 bit

2 Modello matematico

Prima di procedere con l'implementazione degli algoritmi, è utile illustrare il modello matematico del *Shortest Path Problem*. Consideriamo una rete diretta $G = (N, A)$ dove:

- N è l'insieme dei nodi, con $|N| = n$
- A è l'insieme degli archi, con $|A| = m$
- c_{ij} è il costo associato ad ogni arco $(i, j) \in A$
- x_{ij} è una variabile intera che indica quante volte l'arco (i, j) è stato attraversato

Il modello matematico è il seguente:

$$Minimize \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1}$$

$$s.t. \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} n-1 & i = s \\ -1 & \forall i \in N - \{s\} \end{cases} \tag{2}$$

$$x_{ij} \geq 0,\ \forall (i,j) \in A \tag{3}$$

Consideriamo la versione del Shortest Path Problem in cui vogliamo identificare i percorsi minimi dal nodo s verso ogni qualsiasi altra destinazione i, con $i \in N - \{s\}$. Assumiamo inoltre che:

- Il costo c_{ij} associato ad ogni arco è intero
- Il grafo è orientato
- Il grafo non contiene cicli negativi
- il grafo contiene un percorso diretto dal nodo s verso ogni altro nodo nella rete

3 Implementazioni algoritmi

Vediamo ora in dettaglio ciascun algoritmo utilizzato, con annessa descrizione e pseudocodice.

3.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra è uno dei più utilizzati per risolvere problemi di Shortest Path e appartiene alla classe degli algoritmi *label setting*. L'obiettivo dell'algoritmo è quello di trovare i percorsi minimi partendo da un nodo sorgente s verso ogni altro nodo all'interno della rete.

L'algoritmo di Dijkstra mantiene una etichetta di distanza $d(i)$ per ogni nodo i, che ne identifica la distanza dal nodo sorgente s. Ad ogni iterazione, l'algoritmo suddivide l'insieme di nodi in due gruppi: nodi etichettati temporaneamente e permanentemente. I primi sono nodi le cui etichette devono essere ancora analizzate e per cui quindi la loro distanza non è, in quel momento, ancora dimostrata essere quella ottimale. Mentre i nodi con etichetta permanente hanno associata una distanza non ulteriormente migliorabile (soluzione ottima).

Inizialmente il nodo s viene etichettato con distanza 0 in maniera permanente, e ogni altro nodo i nella rete viene etichettato temporaneamente con ∞. L'algoritmo seleziona un nodo i con il valore di etichetta temporanea minima, questo nodo viene reso permanente e scansiona gli archi adiacenti ad esso andando a vedere $A(i)$ (lista di adiacenza degli archi del nodo i) così da aggiornare le etichette delle distanze dei nodi adiacenti. L'algoritmo termina quando tutti i nodi sono stati designati come permanenti.

L'algoritmo di Dijkstra mantiene un out-tree diretto T con il nodo sorgente come radice che attraversa i nodi con etichette di distanza finite. L'algoritmo è capace di mantenere quest'albero usando gli indici dei predecessori (se l'arco $(i, j) \in T$ allora $pred(j) = i$). L'algoritmo inoltre mantiene invariata la proprietà che ogni arco (i, j) dell'albero soddisfa la condizione $d(j) = d(i) + c_{ij}$ rispetto alle etichette delle distanze correnti.

Alla fine, T è detto albero del cammino minimo. Vediamo ora l'algoritmo di Dijkstra.

```
1  algorithm Dijkstra;
2  begin
3  S = ∅; S̄ = N;
4  d(i) = ∞,  ∀i ∈ N;
5  d(s) = 0 and pred(s) = 0;
6  while |S| < n do
7            begin
8            select i ∈ S̄ : d(i) = min{d(j) : j ∈ S̄};
9            S = S ⋃ {i}; S̄ = S̄ \ {i};
10           foreach (i, j) ∈ A(i) do
11           if d(j) > d(i) + c_ij then d(j) = d(i) + c_ij and pred(j) = i;
12           end;
13 end;
```

Nell'algoritmo facciamo riferimento all'operazione (riga 8) di selezione dell'etichetta della distanza provvisoria minima come un'operazione di *node selection*. Mentre facciamo riferimento all'operazione di controllare se le etichette correnti per i nodi i e j soddisfano la condizione $d(j) > d(i) + c_{ij}$ e poi impostare $d(j) = d(i) + c_{ij}$ come un'operazione di *distance update*.

Il codice utilizzato si trova al seguente indirizzo [**1**].

Il tempo computazionale dell'algoritmo di Dijkstra dipende da due operazioni:

1. *Node selection*: l'algoritmo esegue questa operazione n volte e ogni operazione richiede una scansione di ogni nodo etichettato temporaneamente. Pertanto, il tempo per l'operazione di *node selection* è $n + (n-1) + (n-2) + ... + 1 = O(n^2)$.

2. *Distance updates*: l'algoritmo esegue questa operazione $|A(i)|$ volte per un nodo i. In generale, l'algoritmo esegue questa operazione $\sum_{i \in N} |A(i)| = m$ volte. Dato che ogni operazione di *distance update* richiede un tempo pari a $O(1)$, l'algoritmo richiede un tempo totale pari a $O(m)$ per aggiornare tutte le etichette delle distanze.

Dunque l'algoritmo di Dijkstra risolve il problema del cammino minimo in $O(n^2)$.

3.2 Algoritmo di Dijkstra con binary heap

Un *heap* (*priority queue*) è una struttura dati che ci permette di eseguire operazioni su una collezione H di *oggetti*, ognuno dei quali ha associato un numero reale chiamato *chiave*. Se implementassimo l'algoritmo di Dijkstra usando un heap, H sarebbe la collezione di nodi con etichette delle distanze temporanee finite e la chiave di un nodo sarebbe la sua etichetta di distanza.

Più in dettaglio, utilizzeremo un *binary-heap* con ordinamento *min-heap*, in cui le chiavi di ciascun nodo sono sempre minori o uguali di quelle dei figli, e la chiave dal valore minimo appartiene alla radice. Le operazioni previste dall'heap sono le seguenti:

- *create-heap(H)*: crea un heap vuoto
- *find-min(i,H)*: trova e restituisce l'oggetto i con la chiave più piccola
- *insert(i,H)*: inserisce un nuovo oggetto i con una chiave predefinita
- *decrease-key(value,i,H)*: decrementa la chiave di un oggetto i dal suo valore corrente a *value*, il quale dev'essere più piccolo della chiave che sta sostituendo
- *delete-min(i,H)*: elimina un oggetto i con la chiave più piccola

Un binary heap richiede un tempo pari a $O(log\ n)$ per eseguire le operazioni di insert, decrease-key e delete-min e richiede un tempo di $O(1)$ per le altre operazioni. Conseguentemente, la versione dell'algoritmo di Dijkstra che usa il binary heap viene eseguito in un tempo $O(m\ log\ n)$.

Vediamo di seguito lo pseudocodice dell'algoritmo.

```
1  algorithm heap–Dijkstra
2  begin
3      create–heap(H);
4      d(j) = ∞, ∀j ∈ N;
5      d(s) = 0 and pred(s) = 0;
6      insert(s, H);
7      while H ≠ ∅ do
8      begin
9          find–min(i, H);
10         delete–min(i, H);
11         foreach (i, j) ∈ A(i) do
12         begin
13             value = d(i) + c_ij;
14             if d(j) > value then
15                 if(d(j) = ∞) then
16                     d(j) = value;
17                     pred(j) = i;
18                     insert(j, H);
19                 else
20                     d(j) = value;
21                     pred(j) = i;
22                     decrease–key(value, i, H);
23         end;
24     end;
25 end;
```

Il codice utilizzato si trova al seguente indirizzo [**2**].

3.3 BFS

A differenza di Dijkstra, se tutti gli archi hanno costo unitario, il problema si riduce a quello delle distanze minime, facilmente risolubile con una visita in ampiezza. Il BFS può essere usato come algoritmo di ricerca per identificare i cammini minimi da un nodo sorgente s verso gli altri nodi del grafo. Si parte dalla sorgente e si esplorano tutti i nodi allo stesso livello di profondità prima di passare a quelli al livello successivo. Vediamo ora lo pseudocodice dell'algoritmo di ricerca generico.

```
1  algorithm search
2  begin
3          unmark all nodes in N;
4          mark node s;
5          pred(s) = 0;
6          next = 1;
7          order(s) = next;
8          LIST = {s}
9          while LIST ≠ ∅ do
10         begin
11                 select a node i in LIST;
12                 if node i is incident to an admissible arc (i, j)
13                 begin
14                         mark node j;
15                         pred(j) = i;
16                         next: = next + 1;
17                         order(j): = next;
18                         LIST = LIST ⋃ {j};
19                 end
20                 else LIST = LIST \ {i};
21         end;
22 end;
```

Un algoritmo di ricerca parte dal nodo sorgente e identifica un numero crescente di nodi che sono raggiungibili dalla sorgente. In ogni punto intermedio della sua esecuzione, l'algoritmo di ricerca designa tutti i nodi della rete come in uno dei due stati: *marked* o *unmarked*. I nodi marcati sono noti per essere raggiungibili dalla sorgente, mentre lo stato dei nodi non marcati deve ancora essere determinato. Dal punto di vista algoritmico, viene usata una variabile LIST rappresentante l'insieme dei nodi marcati che l'algoritmo deve ancora esaminare, nel senso che alcuni archi ammissibili potrebbero uscire da essi.

L'algoritmo termina quando la rete non contiene più archi ammissibili. Se manteniamo l'insieme LIST come una coda, andremo sempre a selezionare nodi dalla parte superiore di LIST e andremo ad aggiungerli nella parte inferiore. In questo caso l'algoritmo di ricerca seleziona i nodi marcati secondo un ordine $FIFO$ (first-in, first-out). Questa versione di ricerca è chiamata appunto *breadth-first search* e l'albero di ricerca risultante è chiamato *breadth-first search tree*.

Il tempo di esecuzione totale di questo algoritmo è $O(n+m)$ dove n è il numero di nodi del grafo e m è il numero degli archi.

Il codice utilizzato si trova al seguente indirizzo [**3**].

4 Test

Ricapitolando, il progetto si basa sul confronto degli algoritmi di Dijkstra (forward), Dijsktra con binary heap e BFS, in termini di nodi e archi crescenti.

Per quanto riguarda la generazione dei grafi abbiamo utilizzato un generatore scritto in C [**4**], mentre gli algoritmi sono stati presi online (ogni sezione dell'algoritmo ha il suo link associato) controllando che fossero delle versioni corrette anche in termini di complessità dell'algoritmo.

Una volta generato il grafo in un file di testo, abbiamo implementato la parte di codice per convertire i dati ottenuti dal generatore in dati di input per i vari algoritmi. Di seguito i risultati ottenuti.

Nodi	Archi	Dijkstra F.O.	Tempo (s)	Binary Heap F.O.	Tempo (s)	BFS (s)
10	30	141	0.037	141	0.0266	0.053
40	400	315	0.05	315	0.0440	0.059
100	1800	446	0.148	446	0.104	0.060
200	2400	1230	0.409	1230	0.131	0.065
1000	9000	11125	2.075	11125	0.313	0.161
2000	14000	27676	7.506	27676	0.507	0.244
10000	120000	133832	19.425	133832	1.715	0.992
20000	400000	170415	74.941	170415	3.290	2.176

Dal punto di vista della complessità computazionale, l'implementazione tramite binary heap si conferma essere la migliore, mentre considerando un grafo i cui archi hanno un peso unitario, il risultato migliore in termini di tempo di esecuzione è stato il BFS.

Da una prima analisi, abbiamo notato come gli algoritmi che sono stati utilizzati riscontrino delle problematiche nel momento in cui il grafo considerato ha più archi con stessi nodi per testa e coda ma con peso differente (es. due archi $(2, 6)$ con peso 2 e 9 rispettivamente). In realtà questo problema viene gestito la prima volta in cui viene incontrato nel percorso, ma dalla seconda volta in poi non viene più gestito. Pertanto inizialmente le funzioni obiettivo avevano dei valori differenti se pur non eccessivamente distanti tra di loro.

Per ovviare a questo problema, abbiamo rimosso gli archi paralleli andando a scorrere la lista degli archi e rimuovendo quelli con stesso nodo come testa e coda ma con peso diverso, lasciando solamente l'arco con peso minore. Rieseguendo i test, come si può notare dalla tabella, le funzioni obiettivo dei due algoritmi sono uguali. Si può affermare dunque la correttezza dell'implementazione eseguita.

In conclusione, le complessità computazionali di ciascun algoritmo descritte in precedenza si riconfermano dai risultati dei test, in quanto aumentando il numero di nodi e archi del grafo, il risultato peggiore è ottenuto dall'implementazione di Dijkstra classica, ovvero l'algoritmo con complessità più elevata.

4.1 Confronto tra Dijkstra e BFS

Come detto in precedenza, se il grafo ha peso unitario per tutti i suoi archi, Dijkstra restituisce lo stesso percorso minimo (da un nodo sorgente verso tutti gli altri) uguale a quello restituito tramite l'applicazione del BFS. Di seguito i confronti dei percorsi e delle relative distanze minime ottenute su un grafo con 10 nodi e 30 archi.

Sorgente - Destinazione	Percorso	Distanza
1-2	1-9-6-7-2	3
1-3	1-10-3	2
1-4	1-9-6-8-4	4
1-5	1-9-6-7-5	3
1-6	1-9-6	2
1-7	1-9-6-7	2
1-8	1-9-6-8	3
1-9	1-9	1
1-10	1-10	1

Table 1: Distanza minima ottenuta tramite Dijkstra (sia versione originale che con binary heap) con peso unitario per ogni arco

BFS
1-9-10-6-3-7-8-2-5-4

Table 2: Albero ottenuto tramite BFS

Come si può notare dalle tabelle, il BFS agisce in base alla distanza degli altri nodi da quello sorgente. Infatti, inizialmente va dal nodo 1 al nodo 9 la cui distanza è 1, dall'1 al 10 sempre con distanza 1, per poi visitare i nodi successivi che avranno via via distanza sempre maggiore rispetto alla sorgente. Infatti l'ultimo nodo nel BFS, ovvero 4, è il nodo che dista maggiormente dalla sorgente, come riportato anche dalla tabella di Dijkstra.

Bibliografia

[1] *Dijkstra's shortest path algorithm*, https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[2] *Breadth First Search or BFS for a Graph*, https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[3] *Dijkstra's algorithm using binary heap*, https://www.techiedelight.com/single-source-shortest-paths-dijkstras-algorithm/

[4] *Generatore Grafi C*, https://www.spiral.net/software/apsp.html