Simulazione di Traffico su un Grafo Diretto

Fabbri Simone, Lamma Tommaso, Pasquini Micheal, Gianluca

May 2021

Indice

1 Introduzione

1.1 Scopo del progetto

Il progetto è finalizzato alla simulazione del traffico su una città modellizzata come un grafo diretto, ovvero con doppi sensi e sensi unici. La domanda che ci siamo posti all' inizio della programmazione era se in un qualche modo i sensi unici potessero essere vantaggiosi in alcune condizioni rispetto ai doppi sensi a parità di capienza, perché semplicemente sostituire due corsie alternate con una sola corsia diminuirebbe la superficie della città e a parità di numero di automobili aumenterebbe la densità e porterebbe sicuramente ad uno svantaggio.

1.2 Modello

La città è rappresentata come un grafo a griglia, e tramite una probabilità impostata dall'utente è possibile controllare la frazione di sensi unici. All'inizio della simulazione la matrice di adiacenza è creata e tale probabilità è la probabilità che in fase di creazione un senso alternato diventi senso unico. Le strade sono modellizzate come dei contenitori di automobili che sanno soltanto il numero di automobili che contengono, però dato che ogni automobile conosce il suo offset sulla strada questo modello è analogo ad una strada visibile come un array in ogni casella del quale è possibile mettere al più un numero di macchine pari al numero di corsie della strada.

In questa simulazione il traffico non è simulato tramite modelli fluidodinamici ma emerge da un controllo sul movimento delle automobili che impedisce di avere più automobili affiancate che corsie, il che è abbastanza realistico.

1.3 Modalità e parametri della simulazione

L'applicazione permette di fare due tipi di simulazione, una grafica e una "batch" (non grafica); in quella batch si può scegliere tra due simulazioni, "car increment" e bla bla bla

Dopo aver scelto questi parametri, l'utente vedrà un grafo dove potrà selezionare quali sono i nodi di partenza e di arrivo, dove si genereranno e cercheranno di arrivare le macchine. Sono disponibili alcuni preset, ma l'utente può selezionare i nodi a piacere.

Infine, quando si avvierà effettivamente la simulazione, l'utente vedrà colorarsi gli archi - le strade - a seconda della loro occupazione:

- **Strada blu**: la strada è vuota o contiene una macchina
- **Strada verde**: la strada è occupata al massimo al 50% della capacità
- **Strada arancione**: l'occupazione della strada è tra il 50% e il 75% della capacità
- **Strada rossa**: la strada è occupata per più del 75%.

La scelta di non rappresentare direttamente le strade vuote è stata determinata dal meccanismo con cui le strade sono colorate: infatti, il server restituisce al "client", la parte di applicazione responsabile della visualizzazione grafica, le strade che hanno almeno una macchina; il client controlla poi il rapporto tra l'occupazione della strada e la dimensione della stessa, e a seconda del valore colora gli archi-strade. Questo permette al client di ricevere solo le strade che hanno *almeno* una macchina, e non **tutte** le strade; infatti, la cardinalità delle strade che sono occupate è decisamente inferiore alla cardinalità dell'insieme di tutti archi del grafo, e questo permette un aumento delle prestazioni - in quanto bisogna controllare meno archi - al costo di una perdita minima di precisione, in quanto le strade blu possono essere vuote o contenere una sola macchina.

Le strade blu quindi rappresentano sostanzialmente quelle strade che sono state occupate dalle macchine, per permette all'utente di visualizzare i percorsi compiuti da queste.

2 Struttura del progetto (classi)

js_interface.cpp bla bla

3 Risultato simulazioni e analisi dati

4 Aspetti Informatici

4.1 Algoritmo di Floyd-Warshall

Dato in inpout un grafo $G = (V, E)$ e una funzione $w : E(G) \to R$ che assegna pesi agli archi del grafo (anche negativi) , e $V(G) = 1, 2, 3, ..., n$ Dall'esecuzione dell'algoritmo, otteniamo in output la matrice quadrata $n \times n$ contenente i costi dei cammini minimi per ogni coppia di vertici del grafo: $D^{(n)}$ con $d^{(n)}_{(i,j)}$ pari al costo di un cammino minimo tra i e j. Abbiamo bisogno poi, per la ricostruzione del cammino, di una matrice di appoggio $P_{i,j,}$ contenente il nodo predecessore per la ricostruzione del cammino. Utilizza la programmazione dinamica (ossia per ogni passo, il programma prende una decisione).

Pseudocodice dell'algoritmo:

Inizializzazione:
Pongo a 0 di tutti i valori di D^n

```
for i = 1 to n:
    for j = 1 to n:
        dist[i][j] = weight(i,j)
```

con **weight** funzione che riporta il peso dell'arco tra il nodo i e il nodo j, o ∞ se l'arco non esiste.

Nello pseudocodice, la matrice D^n prende il nome di dist[i][j]
Calcolo dei cammini minimi tra le coppie:

```
floydWarshall(int [0..n, 0..n] dist) {
        for h = 1 to n:
        for i = 1 to n:
                j = 1 to n :
                if (dist[i][j] > dist[i][h] + dist[h][j])
                        dist[i][j] := dist[i][h] + dist[h][j]
```

Al termine abbiamo il peso del cammino tra ogni coppia di nodi, ma vogliamo sapere quali nodi assieme formano i cammini.
Ricostruzione del cammino:
Usiamo un'altra struttura dati quindi, nella quale memorizziamo il predecesore di j che li collega. La matrice in questione è pred[i][j].

```
int [0..n, 0..n] dist;
int [0..n, 0..n] pred;
for i = 1 to n
    for j = 1 to n
        dist[i][j] = Weight(i,j)
        pred[i][j] = i

floyd-warshall
for h =1 to n
    for i = 1 to n
        for j = 1 to n
            if (dist[i][j] > dist[i][h] + dist[h][j]) then
                dist[i][j] := dist[i][h] + dist[h][j];
                pred[i][j] := pred[h][j];
            endif
        endfor
    endfor
endfor
```

La complessità, sia nel caso migliore temporalmente che nel caso peggiore, è $O(|V|^3)$. La scelta è ricaduta su Floyd-Warshall in quanto questo in una sola esecuzione calcola i cammini minimi prendendo come sorgenti ogni nodo del grafo, a differenza di Dijkstra e Bellman-Ford che lo calcolano da una singola sorgente, ad ogni esecuzione dell'algoritmo. Non avendo cicli negativi all'interno del grafo, l'algoritmo non fallisce mai.

4.2 Electron

Electron è un framework sviluppato da github, la cui prima versione risale al 2013. E quindi oramai un prodotto maturo, e sufficientemente affidabile. A riprova di ciò, molte note applicazioni usate tutti i giorni da milioni di utenti

sono create con questa tecnologia (Microsoft Teams e Atom sono alcuni tra gli esempi più noti [**?**]).

Electron serve per creare applicazioni per desktop basate su tecnologie web; più nel concreto, consiste di un browser chromium - che permette la visualizzazione dell'app - e dell'ambiente server Node.js - dove viene scritta la logica dell'app - , il che consente di creare un vero e proprio "sito web" che può girare su qualsiasi ambiente desktop (Linux, MacOS e Windows) per il quale l'applicazione sia stata compilata [**?**].

In questo progetto, è stato scelto per la facilità con cui è possibile realizzare la grafica in HTML5, javascript e css, la portabilità e la possibilità di usare codice C++ all'interno del server Node.js. Infatti, nonostante Node.js sia basato su javascript, è possibile creare dei "moduli" di codice nativo in C++, che vengono poi compilati e usati dal server.

La struttura del progetto su Electron è quindi la seguente:

- **codice del server**: il file `main.js` e i file nella cartella `javascript` contengono codice per la gestione del server;
- **codice del simulatore**: il cuore del progetto è contenuto nella cartella `cpp`, contiene i sorgenti in C++ che sono il "cuore" del progetto;
- **grafica**: i file relativi alla grafica, che sono in formato html, css e javascript sono contenuti nella cartella `html`.

4.2.1 Threading

L'applicazione ha due thread, gestiti dal server; questo perché se si eseguisse la simulazione sul thread principale, che è responsabile anche della gestione della grafica, l'applicazione diventerebbe *unresponsive*. La soluzione è stata quindi quella di creare un nuovo thread dove eseguire il codice in C++, per permettere all'utente di continuare ad interagire con l'applicazione.

4.2.2 Librerie javascript per grafica

Per la grafica sono state usate due librerie fondamentali, una per la rappresentazione dei grafi-città e una generica per la rappresentazione di box, bottoni e altri elementi grafici.

Materialize Per semplificare la gestione della grafica nell'interfaccia, abbiamo optato per l'utilizzo di **materialize** [**?**], che è un framework CSS basato su Material Design, lo standard grafico messo a punto da Google per lo sviluppo di interfacce user friendly e coerenti. Questo consente di avere oggetti esteticamente appaganti e responsive, con un sistema di impaginazione a righe/colonne automatico, che aiuta nell'allestimento delle varie pagine.

sigma Sigma.js è una libreria opensource, che serve per creare grafi [**?**]. La versione da noi usata è la v1.2.0, la quale non è l'ultima disponibile, ma la più stabile. Questa libreria permette di visualizzare grafi nel formato GEXF [**?**], funzionalità che non è stata usata nel progetto, ma che è stata tenuta in considerazione in quanto potenzialmente molto utile in sviluppi futuri. Per ulteriori informazioni si veda la sezione apposita sugli sviluppi futuri 5.

5 Sviluppi futuri

5.1 Analisi città esistenti

È possibile, oltre che interessante, effettuare simulazioni sul traffico su città esistenti:

step 1. ottenere una rappresentazione a grafico della città; si può fare con geojson e il progetto openstreetmap, oppure https://www.cityjson.org/datasets/, oppure [cerca su google roba a caso] oppure [progetto che non mi ricordo]

step 1.1: conversione geojson -¿ gexf http://ignacioarnaldo.github.io/OpenStreetMap2Graph/ sorgenti progetto: https://github.com/ignacioarnaldo/OpenStreetMap2Graph

step 2. creare la matrice di adiacenza compatibile col nostro programma (molto facile farlo da gexf)

step 3. visualizzare il grafico: gexf

GeoJSON è un formato per la memorizzazione di geometrie spaziali, nel quale abbiamo gli attributi descritti attraverso la JavaScript object notation. Possiamo rappresentare punti, spezzate, geometrie e collezioni che le contengono.

Esempio:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [11.1215698,46.0677293]
      },
      "properties": {
        "name": "Fontana dell'Aquila"
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
```

```
            [11.1214686,46.0677385],[11.121466,46.0677511],[11.1213806,46.0681
            [11.1213548,46.0682642],[11.1213115,46.0684385],[11.1212897,46.068
            [11.1212678,46.0686443]
          ]
        },
        "properties": {
          "lanes": 1,
          "name": "Via Rodolfo Belenzani"
        }
      }
    ]
}
```

Si rivela quindi un buon formato per rappresentare il grafo di una città tramite archi e nodi. Inoltre grazie alla sua flessibilità è possibile trovare online città reali, costruite in questo formato. geojson, formato strano usato da libreria javascript

6 Conclusioni