Simulazione di Traffico su un Grafo Diretto

Fabbri Simone, Lamma Tommaso, Pasquini Micheal, Spiller 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. 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, quindi nella simulazione lasciamo all'utente la possibilità di scegliere il numero di corsie dei sensi unici.

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. In Fig. 1 è rappresentata una città con pochi sensi unici, per mostrare la struttura a griglia.

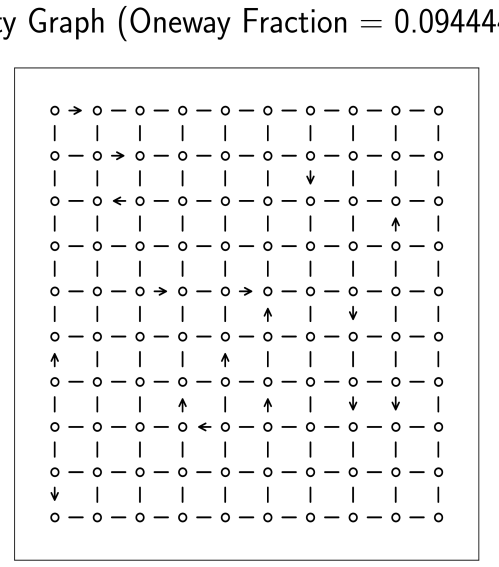

Figura 1: Città a griglia con circa l'1% di sensi unici.

Può avvenire che l'elevata frazione di sensi unici generi nodi inattraversabili, in Fig. 2 vediamo una strada con molti sensi unici dalla quale sono stati rimossi i nodi inattraversabili.

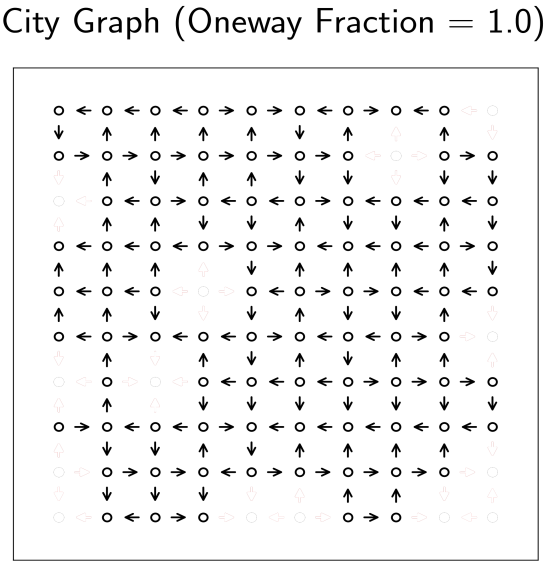

Figura 2: Città a griglia con soltanto sensi unici.

Notando che in quasi tutte le città con solo sensi unici appariva una struttura simile a quella in Fig. 1, sulla quale era possibile avere percorsi non banali, si è deciso di non mettere controlli per evitare di generare nodi inattraversabili, tale controllo può essere sicuramente aggiunto negli sviluppi futuri del programma.

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 vista 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 emerge da un controllo sul movimento delle automobili che impedisce alla singola automobile di entrare in una casella che contenga un numero di macchine pari al numero di corsie della strada.

Ogni automobile conta il numero di volte in cui passa da una casella a quella successiva e anche il numero di volte in cui è costretta a fermarsi a causa del traffico.

1.3 Parametri della Simulazione

Questo modello ha molti parametri liberi che lo rendono molto flessibile ma possono nascondere effetti che sono osservabili sono in alcuni range di tali parametri. I parametri sono i seguenti:

- **Numero di righe e colonne di nodi**: La città è rappresentata come una griglia rettangolare di nodi e l'utente può scegliere le dimensioni di questo rettangolo.

- **Numero di automobili**
- **Numero di corsie dei sensi unici**
- **Probabilità di generare un senso unico**: Questo parametro permette di controllare a priori la frazione di sensi unici nella città generata.
- **Statistiche delle strade**: Le lunghezze delle strade sono generate da una distribuzione gaussiana troncata definita dai seguenti parametri.

  (i) Lunghezza massima

  (ii) Lunghezza minima

  (iii) Media delle lunghezze

  (iv) Deviazione standard delle lunghezze

Il parametro di controllo del traffico, nel codice indicato come *traffic index*, è definito come

$$t = \frac{\langle p \rangle + \langle s \rangle}{\langle p \rangle},$$

dove p è il numero di volte in cui l'automobile si è mossa alla casella successiva e s è il numero di volte in cui ha dovuto fermarsi a causa del traffico.

L'indice di traffico t rappresenta nel nostro modello il rapporto tra i tempi reali di percorrenza e quelli ideali, perciò ci aspettiamo che essso sia sempre maggiore o uguale a 1 e tenda a 1 diminuendo il numero di auto, come visibile in Fig. 3.

2 Struttura del Progetto

2.1 Classi

car.cpp La classe **Car** ha come attributi privati:

- **short int _steps**, ovvero il numero di volte in cui l'automobile si è spostata effettivamente nel reticolo.
- **short int _stops**, ovvero il numero di volte in cui l'automobile ha dovuto fermasi a causa del traffico.
- **short int _offset**, ovvero la posizione all'interno della strada.
- **bool _at_destination**, che diventa vera quando l'auto giunge a destinazione.
- **short int _delay**, ovvero un ritardo nella partenza, poiché dato che tutte le auto sono generate con offset nullo, per evitare sovrapposizioni devono partire una alla volta.

road.cpp La classe **Road** ha come attributi privati:

- **short int _car_number**, ovvero il numero di auto presenti nella strada.
- **short int _road_length**, ovvero la lunghezza della strada.
- **short int _width**, ovvero il numero di corsie della strada, uguale a 1 nelle strade che fanno parte di un doppio senso, e variabile per i sensi unici.
- **static ***, dove * si riferisce ai parametri statistici delle strade spiegati nella sezione precedente, messi come *static* per migliorare l'uso della memoria.

node.cpp La classe **Node** ha come unico attributo privato:

- **short int _index**, ovvero l'etichetta del nodo.

city.cpp La classe **City** ha come attributi privati:

- **Node** _path** e **short int** _distance**, che servono a determinare il percorso più efficiente tra due nodi.
- **void _floyd_warshall**, che dalla matrice di adiacenza determina i due attributi precedenti.
- **Node* _node_set**, ovvero l'insieme dei nodi della città.
- **Road** _adj_matrix**, ovvero la matrice di adiacenza del grafo diretto che rappresenta la città.
- **short int _n_rows** e **short int _n_coloumns**, che sono ripettivamente il numero di righe e colonne della griglia che è la città.
- **float _oneway_fraction**, ovvero un parametro che in generazione determina la probabilità di avere sensi unici, tale parametro può essere diverso dall'effettiva frazione, soprattutto per città piccole, ma nei grafici riportiamo sempre la frazione effettiva invece che questo parametro.

simulator.cpp La classe **Simulator** necessita di due *struct* aggiuntive:

1. **Car_Info**, che ha come attributi il percorso di un'auto, il nodo appena passato, e un puntatore a macchina per gestire i dati di traffico.
2. **Result**, che contiene le statistiche rilevanti della simulazione.

Inoltre ha come attributi privati:

- **std::vector⟨Car_Info⟩ _car_vector**, tale vettore è usato per associare ad ogni auto un percorso ed un nodo tramite l'indice all'interno del std::vector. Tale implementazione è necessaria in quanto a ogni iterazione viene chiamato un *sort* che permette di muovere prima le auto con offset maggiore, riducendo il numero di iterazioni necessarie.
- **Result _result**, ovvero i risulati della simulazione.
- **short int _cars_at_destination**, fa proseguire la simulazione finché non eguaglia il numero di automobili generate.

- **City _city**, ovvero la città su cui viene effettuata la simulazione.
- **short int _car_number**, ovvero il numero di auto generate.
- **void _mv_car(int car_index)**, che gestisce tutti i controlli del traffico e muove le automobili. Tale metodo prende la posizione della macchina in _car_vector, poiché senza di esso non sappiamo nulla della macchina.
- other useful inline functions.

js_interface.cpp La classe *js_interface.cpp* serve alla grafica, scritta in javascript per interagire con il codice della simulazione in C++.

numpy_parser.cpp La classe *numpy_parser.cpp* serve a creare file python con numpy.array leggibili dalla libreria di graficamento *matplotlib*, a partire da std::vector del C++.

2.2 Considerazioni

La scelta di avere oggetti che essenzialmente sono solo uno o due interi (short) è dovuta ad alcuni problemi nella creazione della matrice di adiacenza. Infatti usare strutture più ingombranti causava errori di segmentazione durante la creazione della città, probabilmente dovuto al fatto che la memoria occupata dalla matrice di adiacenza eccedeva quella riservata al programma. La scelta di avere classi non interagenti che possono interagire solo tramite la classe *Simulator*, è dovuta a problemi di ricorsività emersi durante le prime versioni del programma.

3 Risultato delle Simulazioni

3.1 Car Increment Simulation

Le simulazioni mostrano che fissata la città, l'indice di traffico aumenta all'aumentare delle automobili e tende a 1 per il numero di automobili che tende a zero. Ovviamente tale simulazione non avrebbe senso, per ciò partiamo da almeno 1 macchina nella città, l'indice di traffico che emerge dalla simulazione con una sola macchina sarà 1 poiché non può esistere un rallentamento. In Fig. 3 è riportato l'esito di circa 70 simulazioni che incrementano di 1 il numero di automobili da 1 a 2000. Ciò è stato fatto su una città 5×5 senza sensi unici per verificare che effettivamente il traffico aumentasse all'aumentare delle auto.

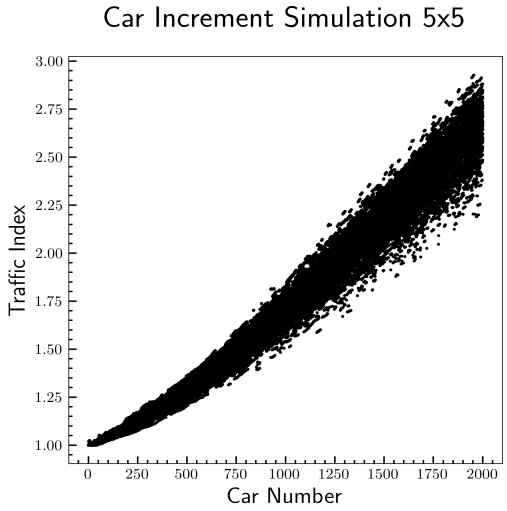

Figura 3: Car Increment Simulation su una città 5×5 senza sensi unici.

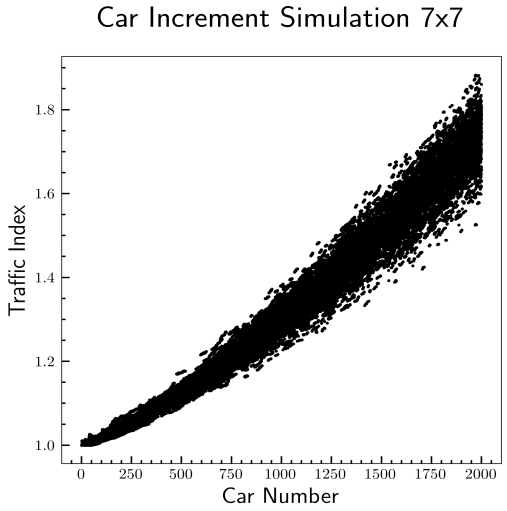

Figura 4: Car Increment Simulation su una città 7×7 senza sensi unici.

Per entrambe queste simulazioni si è usata la seguente statistica delle strade:

- media della lunghezza : 20
- deviazione standard della lunghezza : 10
- lunghezza massima : 30
- lunghezza minima : 10

Un' osservazione interessante che può essere fatta a partire da queste statistiche è che su città rispettivamente 5×5 e 7×7 generano una superficie stradale media data da $20 \times 5 \times 4 \times 2 = 800$ e $20 \times 7 \times 6 \times 2 = 1680$. Data questa considerazione, notiamo che quando il numero di auto è confrontabile con la superficie media nelle due simulazioni l'indice di traffico si trova sempre attorno a 1.5. Perciò possiamo pensare che senza ulteriori modifiche topologiche, l'indice di traffico su una città a griglia senza sensi unici dipenda esclusivamente dalla densità di automobili.

Il fatto che a densità uguale a 1 il programma non si blocchi è dovuto ad un controllo che fa partire le auto con diversi ritardi, quindi non tutte le auto indicate da car number saranno nella città contemporaneamente. Da questa analisi preliminare possiamo concludere che il traffico emerge dal nostro modello e dipende dalla densità di auto, come suggerito da tutti i modelli di traffico esistenti.

3.2 Oneway Increment Simulation

Per vedere se aumentare il numero di sensi unici è in alcuni casi conveniente, abbiamo fatto una simulazione aumentando la frazione di sensi unici fissate 1000 automobili utilizzando le stesse statistiche

stradali della sezione precedente. Tale simulazione è stata eseguita per diverse larghezze dei sensi unici, ovvero una, due e tre corsie. L'incremento del parametro di generazione _oneway_fraction è stato scelto essere 0.001. In Fig. 5 è mostrata la simulazione con sesi unici larghi una sola corsia che, causando una riduzione della superficie media, è chiaramente sconveniente.

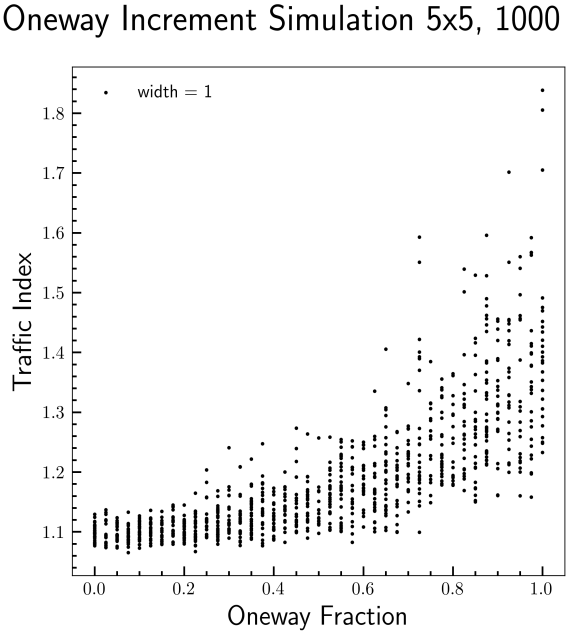

Figura 5: Oneway Increment Simulation su una città 5×5, con larghezza uguale a uno.

Da questo grafico possiamo notare che il traffico aumenterà sempre aumentando i sensi unici a meno della fluttuazione dovuta al posizionamento dei sensi unici nella città. Infatti si può notare che la sezione verticale della fascia nera del grafico si allarga andando verso destra. Ciò significa che in città con tanti sensi unici l'indice di traffico dipenderà molto dal modo in cui essi sono disposti e non solo dalla frazione.

In Fig. 6 e in Fig. 7 è mostrata la stessa simulazione con sensi unici rispettivamente a due e tre corsie.

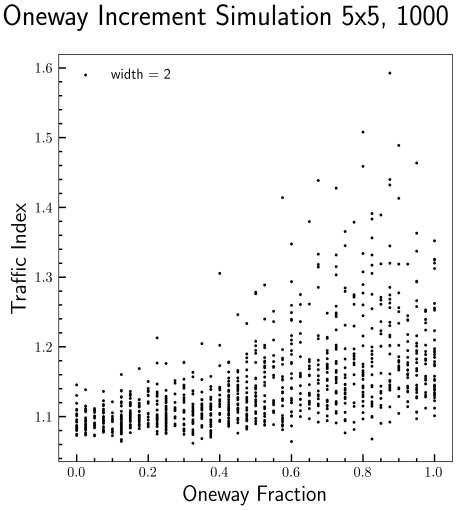

Figura 6: Oneway Increment Simulation su una città 5×5, con larghezza uguale a due.

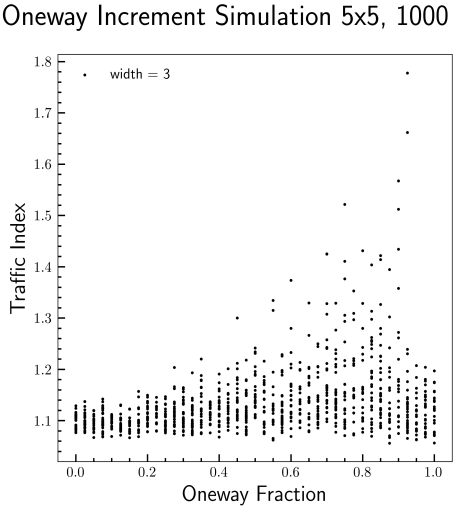

Figura 7: Oneway Increment Simulation su una città 5×5, con larghezza uguale a tre.

In Fig. 6 a differenza del grafico precedente, notiamo che per un'alta frazione di sensi unici si hanno comunque città con un indice di traffico relativamente basso. Quindi possiamo dire che esiste un modo per disporre molti sensi unici a molte corsie per ridurre l'indice di traffico. Tuttavia tale fatto potrebbe anche essere dovuto all'aumento della superficie media della città all'aumentare dei sensi unici, che perciò andrebbe a ridurre la densità di automobili riducendo l'indice di traffico. Tale effetto è maggiormente accentuato in Fig. 7, mettendo sensi unici a tre corsie.

In questi tre grafici l'effetto di "quantizzazione" della frazione di sensi unici è dovuto alle piccole dimensioni della città. Infatti il numero ridotto di strade, questa volta intese senza direzione, fa in modo che le frazioni di sensi unici non possano variare continuamente.

Un'altra analisi che è stata fatta, approssimando linearmente gli esiti delle car increment simulation a diverse frazioni di sensi unici, consiste nel vedere come la pendenza di tali rette varia al variare dei sensi unici, il grafico è riportato in Fig. 9 mentre in Fig. 8 è riportata una giustificazione grafica dell'approssimazione lineare. La pendenza di tali rette rappresenta la risposta della città all'aumentare della densità di automobili.

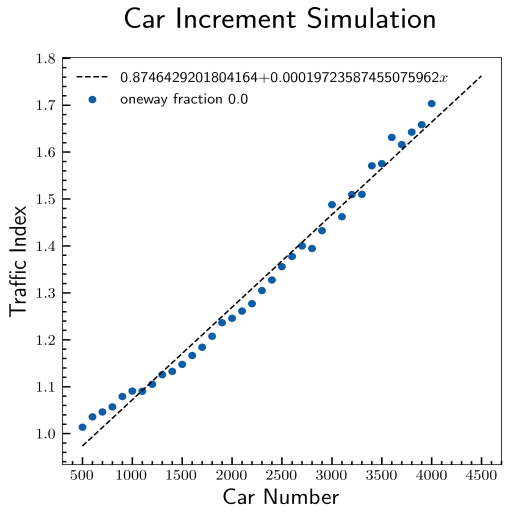

Figura 8: Car Increment Simulation su una città 10×10.

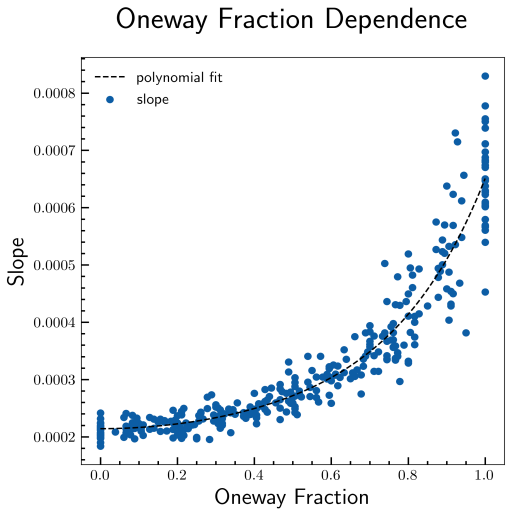

Figura 9: Variazione della pendenza della retta in funzione della frazione di sensi unici.

Si è scelto un range lontano da $car\_number = 1$ dove l'indice di traffico non può seguire un andamento lineare per rispettare il limite ideale. Sulla stessa città 10×10 si sono eseguite varie car increment simulations variando i sensi unici, l'esito è riportato nella figura seguente[1]. Da tale grafico è evidente che le città con un numero minore di sensi unici rispondono meglio all'aumentare della densità di automobili, come ci dovremmo aspettare da un modello di traffico.

[1]Si è scelto un fit polinomiale per semplicità.

Un'ultima analisi è quella relativa ai tempi di esecuzione della oneway increment simulation su una città 10×10, il cui esito segue in Fig. 10.

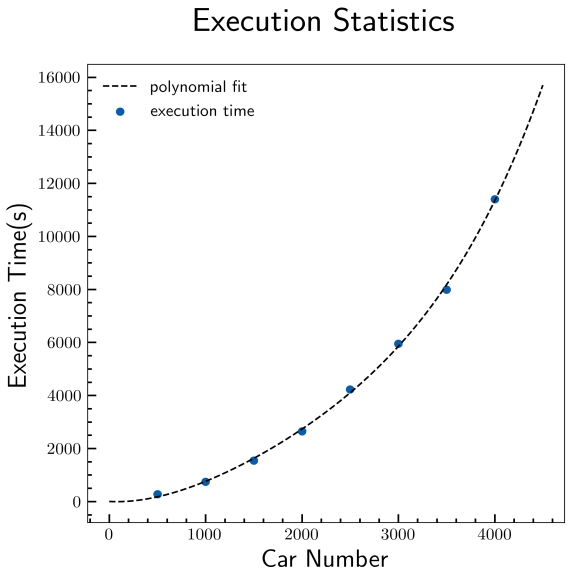

Figura 10: Tempo di esecuzione di oneway increment simulation al variare del numero di auto generate.

Tale analisi dà un' idea di come il numero di iterazioni necessarie a portare le auto a destinazione non cresca linearmente come sarebbe in una città ideale, e tale effetto è dovuto all'emergere del traffico.[2]

3.3 Considerazioni

Da tali risultati non possiamo dire che in alcuni casi convenga mettere sensi unici a più corsie piuttosto che sensi alternati, tuttavia i risultati suggeriscono l'emergenza del traffico all'aumentare della densità, che è necessario per stabilire la correttezza del nostro modello. Inoltre è verificato il limite ideale, ovvero che per un numero di auto tendente a zero l'indice di traffico tende a 1, e questo ci dà un'ulteriore dato a favore del corretto funzionamento del modello. Gli effetti ricercati possono essere stati nascosti da varie caratteristiche del nostro modello, tali caratteristiche sono discusse nella conclusione.

[2]In realtà è anche dovuto al *sort* chiamato ad ogni iterazione.

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 usare una matrice di appoggio $P_{i,j,}$ contenente il nodo predecessore per la ricostruzione del cammino. Questo algoritmo 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 [1]).

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 [2].

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** [3], 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 [4]. 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 [5], 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.

4.2.3 Modalità grafica

La modalità grafica permette di costruire una simulazione scegliendo tra diversi parametri; questi sono:

- **Dimensione** del grafo (numero righe, numero colonne)
- **Lunghezza** delle strade (minima e massima)
- **Percentuale di sensi unici**, tra 0 (nessun senso unico) e 1 (solo sensi unici)
- **Numero di corsie** per strada
- σ**, media** gaussiana

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.

5 Sviluppi Futuri

5.1 Analisi città esistenti

È possibile, oltre che interessante, effettuare simulazioni su città esistenti. Questo si può ottenere attraverso un insieme di standard e tecnologie già esistenti; diamo di seguito un esempio di implementazione:

1. la prima cosa da fare per analizzare città esistenti, è procurarsi il modello di una città già esistente. Questo è molto facile, grazie a svariati progetti che si sono occupati di questo problema, come ad esempio openstreetmap e [6]. Questi progetti permettono inoltre di scaricare le città in un formato standard chiamato GeoJSON, cosa estremamente vantaggiosa in quanto questo può essere facilmente processato e analizzato da diverse librerie gratuite opensource; questo formato è inoltre descritto nell'RFC7946 ([7]) e viene approfondito più avanti.

2. A questo punto, occorre che la città sia caricata come modello sia per la grafica - e quindi che sia possibile disegnare dal programma la città - sia nella parte logica in C++; siccome in questa parte le città sono rappresentate come matrici di adiacenza, occorre quindi convertite una città descritta da GeoJSON in una matrice di adiacenza. Il primo problema è risolto direttamente dal formato GEXF, un formato supportato da sigmajs (si veda la sezione 4.2.2). Siccome questo formato è supportato direttamente da sigmajs, basta convertire il file GeoJSON per ottenere un modo diretto di disegnare la città: i dati convertiti possono essere rappresentati direttamente come un grafo da sigmajs; per fare la conversione, sono disponibili progetti come [8].

3. Infine, il problema di convertire GeoJSON in matrice di adiacenza può essere risolto molto più facilmente convertendo la città in formato gexf in matrice di adiacenza, essendo questa rappresentazione molto più vicina appunto ad una matrice di adiacenza rispetto a GeoJSON.

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],
          [11.1213806,46.0681452],[11.1213548,46.0682642],
          [11.1213115,46.0684385],[11.1212897,46.0685261],
          [11.1212678,46.0686443]
        ]
      },
      "properties": {
        "lanes": 1,
        "name": "Via Rodolfo Belenzani"
      }
    }
  ]
}
```


5.2 Topological Data Analysis

Una recente branca della matematica, chiamata *topological data analysis*, permette di confrontare dati di traffico da un punto di vista topologico. È infatti possibile a partire dalla funzione numero di macchine medio, definita sulle strade(ovvero il grafo), costruire un diagramma di persistenza o codice a barre che rappresenta fedelmente il dato. La distanza tra due codici a barre, più semplice da calcolare della distanza nello spazio delle funzioni definite sul grafo, permette di stimare la distanza effettiva tra i dati di traffico. Tali diagrammi di persistenza o codici a barre sono costruiti vedendo come variano le dimensioni dei gruppi di omologia degli insiemi di livello della funzione numero di auto. Nel caso di un grafo gli unici gruppi di omologia non banali sono il gruppo delle componenti connesse e quello dei cicli indipendenti. In futuro si potrebbe pensare di implementare un codice che produca questi diagrammi di persistenza e vedere come i sensi unci interagiscono con la topologia dei dati di traffico. Per un esempio di applicazione della TDA al traffico si veda [10].

5.3 Semafori e Rotonde

Al momento agli incroci il traffico generato è dovuto soltanto alla congestione e non a rotonde o semafori. In futuro si potrebbero aggiungere dei semafori o delle rotonde, per rendere le città più realistiche, ma tale aggiunta sicuramente andrebbe a coprire ulteriormente gli effetti dei sensi unici. Una cosa interessante potrebbe essere in alternativa costruire una città con soli semafori, e al variare dei sensi unici, tramite un algoritmo di geometric deep learning imparare la funzione che a ogni nodo associa un duty cycle(o due per le due direzioni) in modo da ridurre l'indice di traffico. In seguito si potrebbero confrontare le

diverse funzioni di duty cycle al variare del numero di sensi unici in modo statistico o ricorrendo alla TDA. La formalizzazione del geometric deep learning è molto recente (Maggio 2021), per avere un' idea di come funziona vedere [9].

6 Conclusioni

Nella simulazione che incrementa i sensi unici a tre corsie ci aspettavamo un minimo dell'indice di traffico per una frazione di sensi unici diversa da 0.0. Ciò non è emerso dalle simulazioni e abbiamo concluso che ci sono tre possibilità[3]:

- Effettivamente non è vantaggioso sostituire a sensi alternati sensi unici, anche se a tre corsie.
- I parametri della simulazione sono troppi e stiamo simulando fuori dal range in cui questo effetto è visibile.
- È molto più rilevante il modo di disporre i sensi unici che la loro frazione.

La terza di queste possibilità è giustificata dal fatto che, come si nota in Fig. 9, la dispersione dell'indice di traffico cresce all'aumentare dei sensi unici. Quindi data un'elevata frazione di sensi unici il modo di disporli potrebbe essere così influente da coprire gli effetti della nostra simulazione, dato che nella nostra simulazione non è possibile controllare la disposizione di tali sensi unici. Se il problema è questo, una possibile soluzione futura potrebbe essere data dalla *topological data analysis*.

Riferimenti bibliografici

[1] `https://www.electronjs.org/apps`.

[2] `https://www.electronjs.org/docs/tutorial/introduction`.

[3] `https://materializecss.com/`.

[4] `http://sigmajs.org/`.

[5] `https://gephi.org/gexf/format/`.

[6] `https://www.cityjson.org/datasets/`.

[7] `https://datatracker.ietf.org/doc/html/rfc7946`.

[8] `https://github.com/ignacioarnaldo/OpenStreetMap2Graph`.

[9] Michael M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. 2021. arXiv: `2104.13478 [cs.LG]`.

[10] Yu Wu et al. "Congestion barcodes: Exploring the topology of urban congestion using persistent homology". In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. 2017, pp. 1–6. DOI: `10.1109/ITSC.2017.8317777`.

[3] Esclusi eventuali bug nel codice.