Ottimizzazione della rete metropolitana cittadina tramite algoritmo genetico

Niccolò Barbieri; Francesco Pappone; Tommaso Pettinari

22 febbraio 2023

1 Introduzione

Lo scopo del nostro progetto è quello di realizzare un programma che, conoscendo il modo in cui la popolazione di una determinata area si muove nel territorio, realizzi una rete metropolitana che si adatti nel modo migliore possibile alle esigenze delle persone. Dopo aver studiato il problema, creando un modello fisico-matematico semplice ma di validità generale capace di descrivere la situazione, abbiamo deciso di risolverlo utilizzando la tecnica dell'algoritmo genetico, un procedimento euristico con cui è possibile risolvere problemi di ottimizzazione.

2 Algoritmo genetico

Ci occuperemo ora di spiegare in modo conciso in cosa consiste la tecnica di risoluzione da noi scelta. Come già detto, l'algoritmo genetico è un procedimento utile per tentare di risolvere problemi di ottimizzazione per i quali non è possibile ideare un algoritmo risolutivo di tipo lineare o polinomiale. L'attributo "genetico" è un chiaro riferimento ai processi di evoluzione biologica e al principio di selezione naturale a cui questo tipo di algoritmo si ispira. L'algoritmo, infatti, simula i processi che in natura portano alla cosiddetta "sopravvivenza del più adatto", cioè dell'individuo con *fitness* maggiore.

La prima operazione necessaria per l'algoritmo genetico è quella della codifica delle possibili soluzioni in un formato adatto. Generalmente si utilizza un vettore di lunghezza fissata formato solo da 1 o 0, che contiene tutte le informazioni sulla possibile soluzione al problema. Il vettore binario viene detto cromosoma, poiché effettivamente contiene il materiale genetico che caratterizza univocamente una soluzione.

Dopo aver stabilito un metodo di codifica (e decodifica) di una soluzione, si procede alla generazione casuale di una popolazione iniziale di possibili soluzioni, ciascuna individuata da un diverso cromosoma, e poi si calcola la fitness di ogni individuo della popolazione. A tale scopo occorre definire una funzione, detta funzione di reward, capace di quantificare la fitness di una possibile soluzione.

Nel nostro caso, ad esempio, la funzione di reward potrebbe discriminare una soluzione da un'altra rispetto a quante persone la linea metropolitana è capace di trasportare nell'unità di tempo (vedremo nelle sezioni successive quale è stata la nostra scelta per la funzione di reward).

Calcolata la funzione di reward per tutti gli individui della popolazione iniziale, alcuni elementi vengono selezionati (casualmente o secondo una logica precisa) per il processo di crossover: il loro materiale genetico viene mescolato, secondo la tecnica di crossover prescelta, per ottenere due nuovi individui a partire da quelli iniziali. In questo modo si produce una seconda generazione di individui, su cui il procedimento viene ripetuto nuovamente (calcolo della fitness, selezione dei genitori, crossover e generazione di nuovi individui). Dopo un certo numero di iterazioni, quando si ritiene di aver raggiunto un livello di fitness soddisfacente per la soluzione, si interrompe l'algoritmo, ottenendo una soluzione per il problema iniziale.

Le regole di selezione dei genitori e la tecnica di crossover utilizzata sono varie e dipendono sia dall'algoritmo che si vuole realizzare, sia dal tipo di problema di ottimizzazione. Ad esempio una regola di selezione può essere quella di scegliere soltanto un certo numero di individui con la fitness più elevata. Nel nostro caso, invece, si è deciso di selezionare in modo casuale i genitori, ma di portare avanti nelle generazioni successive una piccola percentuale di individui con fitness più elevata, in modo da non rischiare di perdere la soluzione migliore (viene detta tecnica dell'elitarismo). Per quanto riguarda il crossover, il nostro algoritmo genetico utilizza un crossover di tipo uniforme, ovvero ogni elemento del cromosoma di un genitore ha una certa probabilità di essere scambiato con l'elemento corrispettivo nel cromosoma dell'altro genitore.

Infine, un'ultima ma fondamentale tecnica utilizzata nell'algoritmo genetico è la mutazione, che consiste nella modifica casuale del cromosoma di un individuo della popolazione. La mutazione, che generalmente è di tipo puntiforme, ovvero viene modificato il valore di un singolo elemento del cromosoma, è utile per evitare la stagnazione dell'evoluzione in un "ottimo locale", ampliando lo spazio di ricerca delle soluzioni.

3 Modellizzazione

Nella seguente sezione esporremo le assunzioni, i modelli matematici e le approssimazioni che hanno portato allo sviluppo concettuale dell'idea di dinamica di una metropolitana su una città, attorno al quale abbiamo poi sviluppato il codice.

3.1 Definizioni e concetti fondamentali

Il problema della costruzione di una metropolitana su una mappa è strettamente legato a due meccanismi: uno rappresentativo della dinamica dei vagoni della metropolitana e uno della dinamica attesa nell'uso delle persone del servizio. Il

nostro approccio si è focalizzato, dunque, sulla modellizzazione di questi due aspetti.

E' stato fondamentale, nello sviluppo della nostra soluzione, notare la particolare natura del problema: quello del posizionamento di una metropolitana su di una mappa è un problema di natura discreta e geometrica, in particolare è importante che il modello sia in grado di esprimere il concetto di direzione, così da poter catturare l'intenzione di spostarsi da una parte all'altra della città.

Per prima cosa si definisce il dominio su cui si costruisce la metropolitana: la mappa della città è rappresentata da un quadrato $X = [0,1] \times [0,1] \subset \mathbb{R}^2$. Sulla mappa definiamo dunque una griglia bidimensionale in funzione di due numeri interi n ed m, così che $n \times m$ sia il numero totale di nodi e che il j-esimo nodo sia associato alle coordinate

$$\left(l_x \cdot (j \mod n) \; ; \; l_y \cdot \lfloor \frac{j}{m} \rfloor \right), \tag{1}$$

dove $l_x = \frac{1}{n}$ e $l_y = \frac{1}{m}$ rappresentano le distanze associate ai segmenti, rispettivamente orizzontali e verticali, che congiungono nodi adiacenti nella griglia.

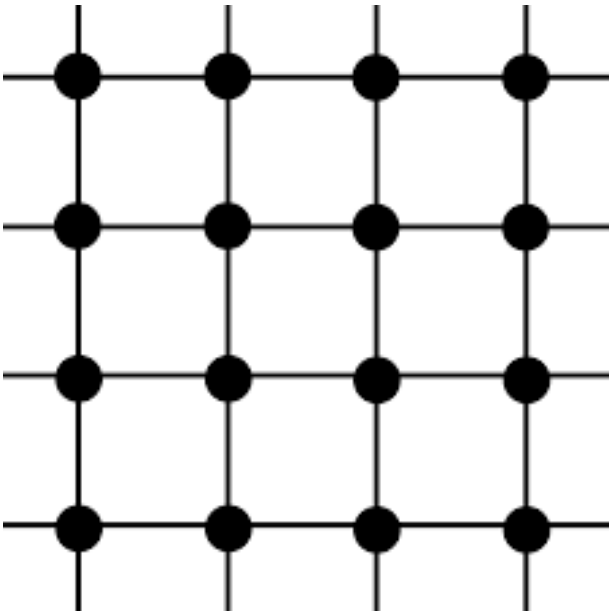

Figura 1: Rappresentazione di una possibile griglia 4×4, usata per rappresentare 16 possibili stazioni, una per ogni nodo

3.2 Dinamica della popolazione

Per modellizzare lo spostamento della popolazione nel territorio (o meglio, "l'intenzione di spostamento" visto che i dati che il programma riceverà in input

rappresenteranno medie temporali o previsioni future di come e dove la gente intende muoversi) abbiamo individuato due parametri determinanti: la quantità di persone intenzionate a spostarsi tra due nodi arbitrari e la quantità di persone presenti in ogni nodo. Oltre a ciò, abbiamo assunto che le persone che usufruiscono della metropolitana cerchino di raggiungere la propria destinazione con il percorso più breve possibile.

Al netto di queste considerazioni, abbiamo deciso di strutturare la dinamica nel modo seguente. Abbiamo definito un grafo "delle intenzioni", $\mathcal{G}_{\mathcal{I}} = (V_I, \mathcal{E}_I)$, con V_I insieme dei nodi (gli stessi della griglia bidimensionale del paragrafo precedente) e $\mathcal{E}_I$ insieme dei segmenti, completamente connesso, e due funzioni definite su di esso, $p : V_I \to \mathbb{N}$ e $s : \mathcal{E}_I \to [0, 1]$. Intuitivamente, possiamo assegnare a p e a s un vettore $\mathbf{p} \in \mathbb{R}^{n \times m}$ e una matrice $\mathbf{S} \in \mathcal{M}(\mathbb{R})_{(n \times m) \times (n \times m)}$, rispettivamente. La funzione p rappresenta la distribuzione di popolazione presente sui nodi, dunque p_j associa a ciascun nodo il numero di persone presenti su di esso. La funzione s rappresenta la probabilità che le persona decidano di muoversi da un nodo all'altro, dunque a s_{ij} sarà associata la probabilità che una persona sul nodo i voglia spostarsi al nodo j.

Come per tutti i grafi, a $\mathcal{G}_{\mathcal{I}}$ è associata una matrice di adiacenza $\mathbf{A}_I$, che noi definiamo per comodità come $(A_I)_{ij} = (p_i \cdot s_{ij})$. Tale matrice rappresenta il punto di partenza dell'algoritmo genetico, il quale cercherà di trovare la migliore struttura possibile della metropolitana in relazione ad $\mathbf{A}_I$.

3.3 Dinamica della metropolitana

Per rappresentare la metropolitana, in particolare la disposizione geometrica delle stazioni e dei binari, è stato costruito un modello a partire dalla griglia definita precedentemente sulla mappa. Sono stati definiti due grafi sovrapposti (i vertici di entrambi sono gli stessi di quelli della griglia). Il primo dei due, $\mathcal{G}_{\mathcal{B}}$, rappresenta con i suoi collegamenti i binari disponibili ed è vincolato ad avere al più gli stessi collegamenti presenti nella griglia. Il secondo grafo, $\mathcal{G}_{\mathcal{V}}$, permette al modello di tenere conto della possibilità che le persone scelgano di muoversi camminando da un nodo all'altro. Chiaramente, $\mathcal{G}_{\mathcal{V}}$ avrà una struttura "complementare" a $\mathcal{G}_{\mathcal{B}}$, ovvero presenterà dei collegamenti soltanto dove non sono previsti dei binari (dunque si assume che, potendo, una persona prenda sempre il treno per muoversi tra due nodi collegati direttamente, e che proceda a piedi solo se non può fare altrimenti).

E' possibile notare come $\mathcal{G}_{\mathcal{B}}$, a differenza di $\mathcal{G}_{\mathcal{I}}$, abbia una matrice di adiacenza vincolata dalla struttura di griglia, dunque generalmente sparsa e dotata di una struttura particolare, riportata in Fig. 2. In particolare, per una griglia di dimensioni $n \times m$, il numero di possibili collegamenti (non direzionati) è di $2nm - n - m$.

3.4 Funzione di Reward

Un elemento cruciale alla corretta definizione di un problema di ottimizzazione è la funzione di reward: per trovare la migliore delle possibili metropolitane è

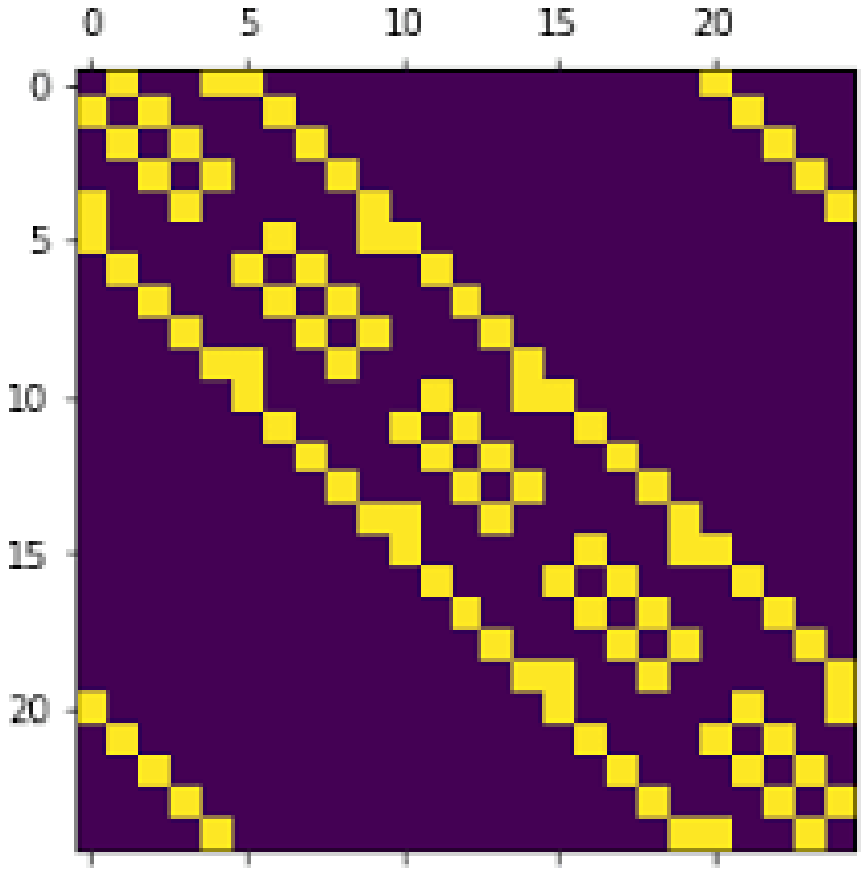

Figura 2: Struttura di una matrice di adiacenza associata ad una griglia bidimensionale. In giallo i valori non nulli.

fondamentale definire cosa renda buona una metropolitana. La nostra soluzione a questo problema si è avvalsa delle considerazioni sviluppate nelle precedenti sezioni.

Siano $\mathcal{G}_\mathcal{I}$, $\mathcal{G}_\mathcal{B}$ e $\mathcal{G}_\mathcal{V}$ rispettivamente il grafo delle intenzioni, il grafo dei binari e il suo grafo complementare di una mappa $\mathcal{X}$, con $\mathbf{A}_I$, $\mathbf{A}_B$ e $\mathbf{A}_V$ matrici di adiacenza associate. Definiamo $\mathbf{B} = (T\mathbf{A_V}+\mathbf{A_B})$, dove T è un numero reale che rappresenta il tempo di percorrenza di un tratto di strada a piedi, siano inoltre p e s le funzioni rappresentanti le popolazioni e le probabilità di spostamento, come definito nella sezione 3.2. Sia, infine, D una matrice in $\mathcal{M}(\mathbb{R})_{(n\times m)\times(n\times m)}$ le cui entrate rappresentano le distanze euclidee associate al percorso minimo tra l'i-esimo ed il j-esimo nodo (percorso vincolato sulla nuova matrice, $\mathbf{B}$), allora definiamo la funzione di Reward come

$$R(\mathbf{A},\mathbf{B},\mathbf{p},\mathbf{S}) = \sum_{i,j}\frac{S_{ij}P_i}{D(B)_{ij}} - k||A_B||_1 - k'\Theta(||A_B||_1 - M)\cdot(||A_B||_1 - M) \quad (2)$$

con $M \in \mathbb{N}$ numero massimo di binari atteso, Θ funzione di Heavyside e k,k' parametri reali. Il parametro k' è stato scelto molto grande in modo da creare una sorta di barriera di potenziale che confini il risultato ad un numero di binari minore del limite M. L'altro parametro, k, rappresenta il costo di ciascun binario, e permette un *tuning* della soluzione molto più accurato: aumentando o abbassando leggermente tale valore, è possibile scendere in maniera maggiore o minore sotto la soglia M di binari massimi. Stabilire un valore per k è più complicato dato che non esiste una dipendenza funzionale del numero di binari finali in funzione di n, m, k e M (k' è da considerare prossimo a infinito), tuttavia in prima approssimazione si può considerare che per ottenere un numero di binari prossimo a M, k deve essere circa $\frac{4000}{(n\cdot m)}$. Questa approssimazione è valida per griglie nell'intervallo da 4×4 a 8×8.

4 Il programma

Il programma si articola in diversi moduli. I principali sono quelli che si occupano della generazione dei dati e dell'esecuzione dell'algoritmo (scritti in Python) e quelli che si occupano della rappresentazione grafica dei risultati ottenuti (scritti in C++). Le restanti parti sono necessarie per gestire il programma nel suo complesso e per interfacciarsi con l'utente.

All'avvio, il programma entra in un menù che permette all'utente di scegliere se vuole avviare una nuova simulazione oppure se preferisce caricare direttamente il risultato di una simulazione già effettuata. Nel primo caso viene eseguito un programma in Python che genera dei dati casuali in base a dei parametri inseriti dall'utente e poi avvia l'algoritmo genetico per trovare la soluzione di metropolitana migliore. Nel secondo caso si apre un nuovo menù che permette di scegliere tra degli esempi di città la cui soluzione ottimale è già stata ottenuta (Fig. 3).

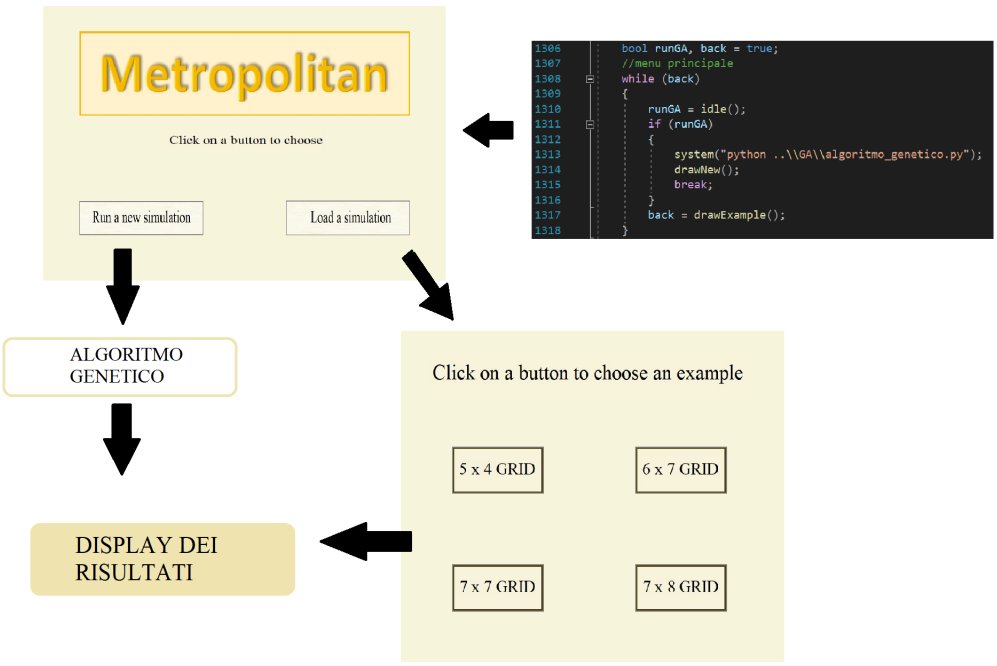

Figura 3: La funzione *idle* disegna il menù principale e, in base alla scelta dell'utente, restituisce un valore booleano *true* se è stata espressa la decisione di eseguire una nuova simulazione. In tal caso viene eseguito il programma in Python e poi vengono mostrati i risultati. Altrimenti, la funzione *drawExample* propone all'utente alcune simulazioni già effettuate.

4.1 Generazione dei dati ed esecuzione dell'algoritmo genetico

Il nucleo principale del programma è quello che si occupa dell'esecuzione dell'algoritmo genetico. Questa parte è stata scritta in Python per poter sfruttare la libreria *geneticalgorithm*, che svolge in modo automatico tutte le procedure esposte nella sezione 2 (generazione, selezione, crossover, mutazione e calcolo della funzione di reward).

Poiché non disponiamo di dati reali riguardanti il traffico cittadino, prima di procedere con l'esecuzione dell'algoritmo è necessario creare dei dati verosimili da poter utilizzare. È necessario cioè fornire un valore alle funzioni p e s introdotte nel paragrafo 3.2 per ogni punto del dominio (Fig. 4). Questa operazione viene svolta dopo aver chiesto all'utente di fornire il numero di righe e di colonne in cui suddividere la griglia bidimensionale $n \times m$.

```
MaxPeople=1000

#costruzione matrice spostamenti
TravelPeople=np.random.randint(MaxPeople,size=(n*m))
TravelMatrix=np.zeros(shape=(n*m,n*m,))
TravelMatrix = np.random.rand(n*m, n*m)
for i in range(k):
    for j in range(k):
        if(i==j):
            TravelMatrix[i,j]=0

#normalizzazione
for i in range(k):
    somma=TravelMatrix[i].sum()
    for j in range(k):
        TravelMatrix[i,j]=TravelMatrix[i,j]/somma
```

Figura 4: L'oggetto *TravelPeople*, che corrisponde al vettore **p** (ovvero il numero di viaggiatori per ogni nodo), viene riempito con $n \times m$ numeri interi random, ciascuno compreso nell'intervallo tra 1 e 1000. L'oggetto *TravelMatrix*, che invece rappresenta la matrice **S**, viene inizializzato con numeri random compresi nell'intervallo $[0, 1]$. I valori sulla diagonale, in seguito, vengono azzerati e gli elementi di ciascuna riga vengono normalizzati (l'intero k è il prodotto $n \cdot m$).

Prima dell'esecuzione dell'algoritmo genetico sono necessari due ulteriori step:

1. codificare in formato vettoriale binario l'informazione sul numero e sulla posizione dei binari (ovvero ciò su cui l'algoritmo genetico deve agire);
2. definire la funzione di reward.

L'informazione sui binari è contenuta nel grafo $\mathcal{G}_B$ il quale, come già detto nella sezione 3.3, ha una struttura vincolata dalla sottostante griglia bidimensionale su cui è definito, e perciò può avere un numero massimo di $2nm - n - m$ collegamenti. Questa sarà dunque anche la dimensione del vettore binario su cui l'algoritmo genetico lavorerà.

Per quanto riguarda la funzione di reward, è innanzitutto necessario decodificare il vettore binario, convertendolo in una matrice di adiacenza, così da poter associare ogni binario ad un segmento della griglia (Fig. 5). La funzione

```
#convertitore vettore-matrice
def trasf(righe, colonne, vettore):
    dim=righe*colonne
    matrice= np.zeros(shape=(dim,dim))
    j=0
    for i in range(dim):
        if((not(i))+((i+1)%colonne)):
            matrice[i,i+1]=vettore[j]
            matrice[i+1,i]=vettore[j]
            j=j+1
        if (i+colonne<dim):
            matrice[i,i+colonne]=vettore[j]
            matrice[i+colonne,i]=vettore[j]
            j=j+1
    return matrice
```

Figura 5: La funzione riceve come argomenti il numero di righe n, il numero di colonne m e il vettore che trasporta l'informazione codificata riguardo i binari. La funzione riempie soltanto specifici elementi dell'oggetto *matrice* con i valori presenti nel vettore, secondo la struttura di una matrice di adiacenza associata ad una griglia bidimensionale (cfr. Fig. 2, § 3.2).

di reward opera sulla matrice di adiacenza appena ottenuta, denominata *State-Matrix*, secondo quanto esposto nel paragrafo 3.4 (Fig. 6). Prima si ottiene la matrice $\mathbf{B} = (T\mathbf{A_V} + \mathbf{A_B})$, che viene quindi moltiplicata per una matrice (anch'essa in $\mathcal{M}(\mathbb{R})_{(n\times m)\times(n\times m)}$) contenente le distanze euclidee tra ogni coppia di nodi. $\mathbf{B}$ viene utilizzata per calcolare il valore dell'equazione (2).

Definita la funzione di reward si procede dunque all'esecuzione dell'algoritmo genetico. I parametri dell'algoritmo sono stati variati diverse volte durante le simulazioni, in modo da riuscire a trovare quelli ottimali per far convergere alla soluzione. Generalmente è stata utilizzata una popolazione di 100 individui con 1000 iterazioni, per risolvere matrici più piccole, mentre la dimensione della popolazione è stata incrementata (anche fino a 1000) per matrici più grandi. La frazione di popolazione con fitness più elevata che viene conservata tra una generazione e l'altra è stata posta al 5%, mentre la percentuale di individui che subiscono il crossover è del 20%. Il crossover effettuato è di tipo uniforme, con

```python
def RewardFunction1(StateVector):
    Reward=0
    CountBinari=0
    StateMatrix=trasf(n, m, StateVector)

    for i in range(len(StateVector)):
        if (StateVector[i] == 1):
            CountBinari+=1

    #modifico le velocità di percorrenza
    StateMatrix[StateMatrix == 0] = 1000
    np.fill_diagonal(StateMatrix, 0)

    #applico la metrica alla state matrix
    StateMatrix = distances*StateMatrix

    StateGraph=nx.DiGraph(StateMatrix)

    for i in range(k):
        for j in range((k)):
            if(i!=j):
                D = nx.algorithms.shortest_paths.weighted.dijkstra_path_length(StateGraph,i,j)
                Reward+=(TravelPeople[i]*TravelMatrix[i,j])/D
    Reward -= RailCost* CountBinari
    if(CountBinari>MaxBinari):
        Reward-=50000*(CountBinari-MaxBinari)
    return -Reward
```

Figura 6: La funzione di reward riceve come input il vettore codificato, *StateVector*, e provvede a trasformarlo in una matrice di adiacenza, *StateMatrix* (ovvero $\mathbf{A_B}$). La funzione calcola, quindi, la matrice $\mathbf{B} = (T\mathbf{A_V} + \mathbf{A_B})$, sostituendo agli zeri il valore $T = 1000$ ($\mathbf{A_V}$ e $\mathbf{A_B}$ sono infatti complementari). La diagonale deve comunque contenere sempre valori nulli. L'oggetto *distances* è una matrice contenente le distanze euclidee tra ciascuna coppia di punti della griglia. In questo modo, tramite l'utilizzo della libreria *NetworkX* (nx nel codice), considerati due nodi i e j, è possibile calcolare la lunghezza D del percorso più breve tra di essi lungo il grafo pesato corrispondente alla matrice $\mathbf{B}$ (si usa a tale scopo l'algoritmo di Dijkstra). Si noti, infine, come il risultato della funzione di reward sia restituito con il segno meno, in modo che una fitness maggiore della soluzione produca un *Reward* minore.

una probabilità del 50% per ciascun elemento del vettore binario. Infine, la probabilità di mutazione è stata cambiata spesso tra una simulazone e l'altra, assumendo valori tra il 5% e il 30%.

4.2 Rappresentazione grafica

Per la rappresentazione grafica dei risultati ottenuti si è scelto di scrivere un programma in C++ che utilizzasse GLFW, una libreria di OpenGL. Illustrare il funzionamento di GLFW esula dagli scopi di questa relazione, perciò si eviterà di andare nello specifico riguardo le procedure di rendering. Basti sapere che GLFW permette di utilizzare OpenGL moderno, in cui la cosiddetta pipeline grafica (tutto il processo di realizzazione di un oggetto, dalla definizione dei punti alla colorazione e rappresentazione sullo schermo) è programmabile attraverso dei piccoli programmi eseguiti dalla GPU detti shader, scritti in un loro specifico linguaggio, GLSL, molto simile al C. GLi shader ricevono dal programma informazioni sulle caratteristiche dei punti che dovranno essere disegnati, le rielaborano (modificandole eventualmente) e, infine, producono l'immagine da mostrare sullo schermo.

La parte di codice che si occupa della rappresentazione grafica si divide a sua volta in diverse sezioni:

- inizializzazione dell'ambiente grafico, compilazione e linking degli shader;
- creazione dei buffer di dati da inviare agli shader, contenenti le informazioni su coordinate, colore e texture dei diversi oggetti da disegnare;
- ciclo di rendering, che attiva gli shader e disegna gli oggetti.

Evitando di soffermarci sulle operazioni più tecniche, analizziamo ora come sono stati creati i buffer di dati e come sono state disegnate (in modo semplificato e schematico) la città e la metropolitana.

La città è stata rappresentata con una griglia bidimensionale, con n righe e m colonne (Fig. 7). Ad ogni intersezione è stato disegnato un nodo, rappresentato da un cerchio colorato in modo differente in base al numero di persone che popolano quel nodo. Il colore viene assegnato considerando la popolazione di ciascun nodo relativamente agli altri nodi: quello con popolazione minima è verde, quello con popolazione massima è rosso, alla media dei valori viene assegnato il colore giallo, gli altri assumono sfumature intermedie.

Il codice che si occupa della creazione del buffer di dati con coordinate e colore dei nodi è visibile in Fig. 8. Prima si ricavano le posizioni dei centri di ciascun nodo utilizzando l'espressione delle coordinate dei nodi (1), poi si costruisce il cerchio tutto intorno.

Le intenzioni di spostamento delle persone, cioè la matrice $\mathbf{S}$, sono state rappresentate disegnando i collegamenti tra i vari nodi e colorandoli in base alla frazione di popolazione che è intenzionata a percorrere quella tratta (con lo stesso criterio utilizzato per colorare i nodi). I collegamenti sono visibili solo per un nodo alla volta (per non appesantire troppo la rappresentazione, essendo il

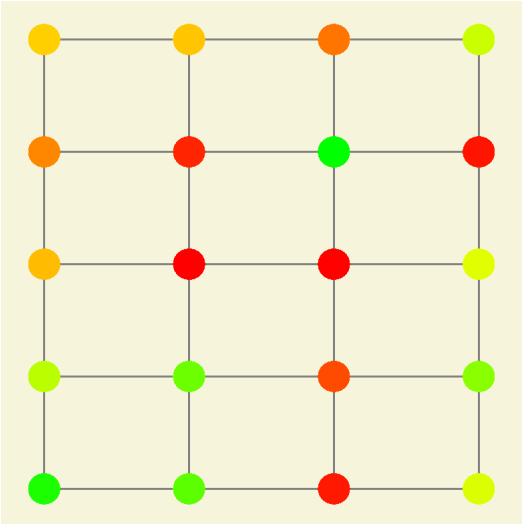

Figura 7: Rappresentazione schematica di una città 5×4. I nodi assumono le varie tonalità di colore in funzione della popolazione di ciascun nodo.

grafo $\mathcal{G}_{\mathcal{I}}$ completamente connesso) e compaiono quando l'utente clicca sul nodo che vuole esaminare (Fig. 9).

Per quanto riguarda i binari e le stazioni della metropolitana, non è stato necessario creare un nuovo buffer di dati con le coordinate dei punti da disegnare, ma è stato sufficiente riprendere quello creato per i nodi, dato che le stazioni si trovano in corrispondenza di essi. Tuttavia, occorre fornire un modo per leggere i dati già esistenti in modo diverso. È stato, quindi, creato e inviato agli shader un vector di indici, la cui funzione è quella di comunicare quali dei nodi contengono una stazione e tra quali stazioni sia necessario disegnare una linea per indicare la presenza di un binario (Fig. 10). Anche i binari, così come il grafo delle intenzioni, non sono permanentemente visibili sulla mappa, ma vengono attivati e disattivati a scelta dell'utente. In questo caso la pressione del tasto barra spaziatrice mostra o nasconde la metropolitana (Fig. 11).

4.3 Funzioni e moduli complementari

4.3.1 Gestione dei dati

I dati generati dal programma in Python (la distribuzione di popolazione mobile sulla città, le intenzioni di spostamento e, soprattutto, il risultato dell'algoritmo genetico, con posizione e numero di binari) devono essere disponibili per il programma che si occupa della grafica. A tale scopo, dopo la conclusione dell'algoritmo genetico, il programma in Python crea un file di testo in cui scrive i valori n, m, il vettore $\mathbf{p}$, la matrice $\mathbf{S}$ e la matrice di adiacenza $\mathbf{A}_B$. Questi dati vengono resi disponibili al programma C++ attraverso le istanze di due classi,

```
329     //coordinate dei nodi
330     for (int i = 0; i < nPoints; i++)
331     {
332         nodes.push_back(((double)(i % c) * 0.9) / ((double)c) * 2 - ((double)c - 1) * 0.9 / (double)c);
333         nodes.push_back((double)((int)(i / c) % r) * 0.9 / ((double)r) * (-2) + ((double)r - 1) * 0.9 / (double)r);
334         nodes.push_back(0);
335     }
336
337     //creazione dei punti e dei colori
338     double radius = 0.05;
339     double angle = 0;
340     vector<double> points;
341     if (nodes.size() == 3 * static_cast<unsigned long long>(nPoints))
342     {
343         for (size_t j = 0; j < nodes.size(); j += 3)
344         {
345             angle = 0;
346             vector<double> color = colors(j / 3, window);
347             points.push_back(nodes[j]);
348             points.push_back(nodes[j + 1]);
349             points.push_back(nodes[j + 2]);
350             points.push_back(color[0]);
351             points.push_back(color[1]);
352             points.push_back(0.0);
353             for (int i = 0; i < STEPS; i++)
354             {
355                 double x = radius * cos(angle) + nodes[j];
356                 double y = radius * sin(angle) + nodes[j + 1];
357
358                 points.push_back(x);
359                 points.push_back(y);
360                 points.push_back(0.);
361                 points.push_back(color[0]);
362                 points.push_back(color[1]);
363                 points.push_back(0.0);
364
365                 angle += 2 * M_PI / (double)STEPS;
366             }
367         }
368     }
```

Figura 8: Le coordinate (tridimensionali) dei centri dei nodi vengono immagazzinate dentro il vector di double *nodes*, utilizzando la formula per le coordinate (1) leggermente modificata, dato che la finestra grafica viene mappata in [-1,1] su entrambi gli assi (non in [0,1]). È quindi necessario moltiplicare per 2 e successivamente traslare il valore delle coordinate ottenuto (righe 332-333). Gli interi r, c e *nPoints* contengono rispettivamente il valore del numero di righe, del numero di colonne e il prodotto tra di essi. Il vector *points* contiene invece coordinate dei centri e dei punti delle circonferenze e i relativi colori. Si noti come il colore viene calcolato dalla funzione *colors* che restituisce le componenti rossa e verde (il blu è fisso a 0). Il calcolo delle coordinate dei punti delle circonferenze (righe 353-366) è necessario in quanto OpenGL permette di disegnare immediatamente soltanto figure semplici (il punto, il triangolo, la linea). Se si vuole disegnare un cerchio, è necessario, perciò, costruirlo con numerosi triangoli (360 nel nostro caso) con il vertice nel centro. Le basi costituiscono la circonferenza: più l'angolo al vertice è piccolo, migliore è l'effetto che si ottiene.

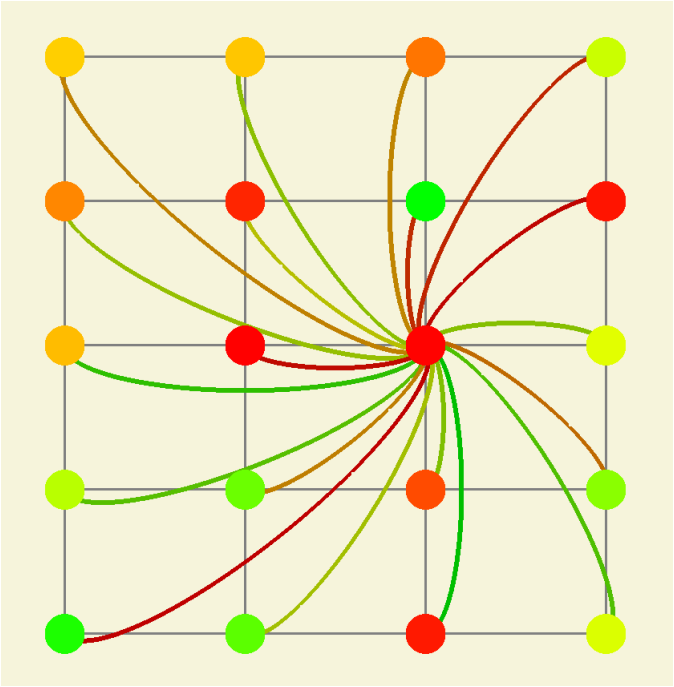

Figura 9: Intenzioni di spostamento per il nodo in terza riga e terza colonna (posizione (2,2)). I collegamenti sono realizzati con metà perimetro di un ellisse.

```
508     for (auto i = 0; i < elements * elements; i++)
509     {
510         if (binari[i])
511         {
512             if (stations.empty())
513                 stations.push_back(i / elements);
514             else if(stations.back() != (i / elements))
515                 stations.push_back(i / elements);
516             indices.push_back(i / elements);
517             indices.push_back(i % elements);
518         }
519     }
520
521     for (auto i : stations)
522         indices.push_back(i);
```

Figura 10: Il vector di bool *binari* ha dimensioni $(n \cdot m)^2 = (elements)^2$ ed è il corrispettivo, in forma di vettore, della matrice $\mathbf{A}_B$. A partire da *binari*, vengono riempiti due nuovi vector: *stations* e *indices*. Nel primo vengono inseriti, una sola volta ciascuno, gli indici dei nodi che contengono una stazione (righe 512-515, il vector viene controllato per accertarsi che non ci siano doppioni, e quindi l'indice del nodo viene inserito). Il secondo contiene le coppie di nodi tra cui deve essere disegnato un binario (righe 516 e 517, si inserisce prima il nodo di partenza e poi il nodo di arrivo). Il vector *indices*, infine, viene riempito anche con gli indici delle stazioni (riga 522), così da avere tutte le informazioni in un unico oggetto.

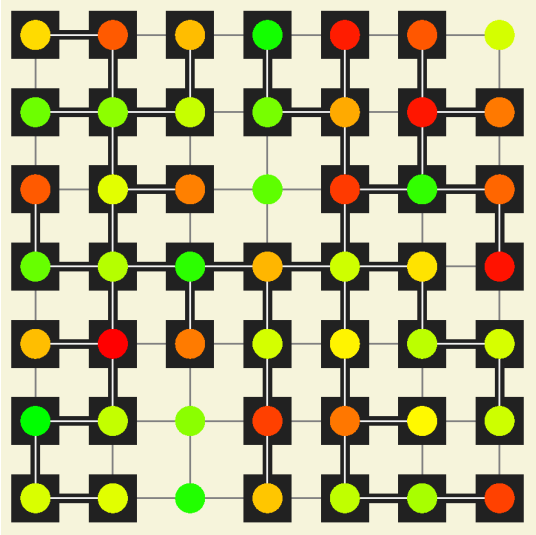

Figura 11: Esempio di linea metropolitana per una griglia 7×7. Le stazioni sono rappresentate con un quadrato nero, i binari con una riga spessa, bianca e nera.

la classe *Dati* e la classe *Global*, definite in file separati e inclusi nel programma principale. La classe *Dati* si occupa di leggere il file prodotto e memorizzarne i valori contenuti nei propri membri. La classe *Global* contiene un'istanza di *Dati* come suo membro, ma la sua funzione non si limita a questo. *Global* si occupa di gestire anche tutti gli altri dati necessari alla corretta esecuzione del programma, ovvero le coordinate dei nodi nella mappa (che devono essere visibili da quasi ogni funzione del programma) e le flag che indicano al programma se i collegamenti di uno specifico nodo o i binari della metropolitana devono essere visibili o meno.

4.3.2 Gestione degli input

L'utente interagisce con il programma in due modi: cliccando su determinati punti dello schermo (bottoni, pulsanti o i nodi della mappa) oppure premendo la barra spaziatrice per mostrare o nascondere la linea metropolitana. Il click del mouse viene recepito dal programma tramite l'invocazione di una funzione della libreria GLFW che restituisce le coordinate del puntatore al momento del click. Queste coordinate vengono confrontate con le coordinate degli elementi interattivi presenti nella finestra aperta al momento e, se il puntatore era in una posizione di interesse, attivano o disattivano le flag relative all'elemento cliccato.

Per quanto riguarda, invece, l'input da tastiera, questo viene gestito da una funzione di callback, caratteristica della libreria GLFW (Fig. 12). La funzione si attiva ad ogni ciclo di rendering, valuta lo stato dei tasti (ad ogni tasto è associata una flag interna alla libreria) e in base al tasto premuto esegue una specifica azione. Nel nostro caso è stato necessario fornire un'azione da eseguire solo per il caso della barra spaziatrice (in realtà anche per il tasto escape, ma solo per comodità, non per esigenza).

```
1331  static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
1332  {
1333      if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
1334          glfwSetWindowShouldClose(window, GLFW_TRUE);
1335      if (key == GLFW_KEY_SPACE && action == GLFW_RELEASE)
1336      {
1337          bool status = static_cast<Global*>(glfwGetWindowUserPointer(window))->getRailwayStatus();
1338          status = !status;
1339          static_cast<Global*>(glfwGetWindowUserPointer(window))->setRailwayStatus(status);
1340      }
1341  }
```

Figura 12: La funzione di callback prevede una possibile azione nel caso di pressione del tasto escape o della barra spaziatrice. Nel primo caso, chiude la finestra, nel secondo cambia il valore della flag relativa alla visibilità dei binari della metropolitana. Lo stato di visibilità è custodito da un puntatore ad un oggetto della classe *Global*. Per rendere visibile questo oggetto nelle diverse funzioni, la libreria GLFW mette a disposizione la funzione *glfwSetWindowUserPointer* (e la corrispettiva Get) che associa ad una specifica finestra un puntatore definito nel programma.

4.3.3 Building del programma

Per automatizzare le procedure di building del codice, si è ritenuto opportuno utilizzare CMake, in quanto rende possibile il contemporaneo building della libreria GLFW. In questo modo, unendo il codice sorgente della libreria al progetto, il programma funziona anche su dispositivi che non hanno già GLFW. Inoltre CMake permette di estendere facilmente il programma ad altri sistemi operativi.

5 Progetti Futuri

Il programma da noi sviluppato risulta efficace su matrici sparse di piccole dimensioni. Un progetto che partisse dall'implementazione di un algoritmo genetico ad hoc ottimizzato per problemi come il nostro, permetterebbe forse di poter risolvere casi più generali rispetto a una semplice griglia bidimensionale, con distribuzioni di popolazione più realistiche. In particolare, la libreria *geneticalgorithm* risulta carente dal punto di vista dell'operazione di crossover, permettendo di eseguire solo un crossover di tipo uniforme. Altre tecniche più efficienti di crossover potrebbero ridurre il tempo di calcolo, permettendo di aumentare la complessità del modello.

Un'ulteriore possibilità di sviluppo è la funzione di reward, che può essere interpretata, come visto in precedenza, come una stima del numero di persone in moto nella metropolitana. Ovviamente questa stima può essere affinata essendo stata fatta tramite considerazioni coerenti ma approssimative.

Ulteriori direzioni di ottimizzazione sono inoltre possibili: il problema di cui abbiamo trattato, infatti, è un classico problema di *Edge Prediction* nell'ambito del Machine Learning sui grafi, ovvero di previsione dell'esistenza o meno di un collegamento tra due nodi del grafo. In particolare, gli ultimi anni hanno visto uno sviluppo repentino di tecnologie di Neural Networks supervisionati e non, denominati *Graph Neural Networks* capaci, tra le altre cose, di risolvere problemi di Edge Prediction. Chiaramente il nostro problema andrebbe però affrontato nell'ambito del reinforcement learning, dato che la struttura del problema stesso non si presta facilmente ad un approccio supervisionato.

Un esempio di architettura utilizzabile potrebbe essere quella di *Graph Convolutional Network (GCN)* (Fig. 13),la quale, se propriamente riprodotta, ha il vantaggio di essere computazionalmente leggera (eredita la sparsità nei parametri dalle reti convoluzionali classiche) e sufficientemente sensibile alla geometria del grafo per ottimizzare correttamente la metropolitana. Chiaramente, in questo modo l'approccio seguito per trovare la soluzione sarebbe totalmente diverso rispetto all'algoritmo genetico.

6 Risultati e conclusioni

I risultati ottenuti dall'ottimizzazione della distribuzione dei binari rispetto alle intenzioni di spostamento risultano più che soddisfacenti per quanto riguarda

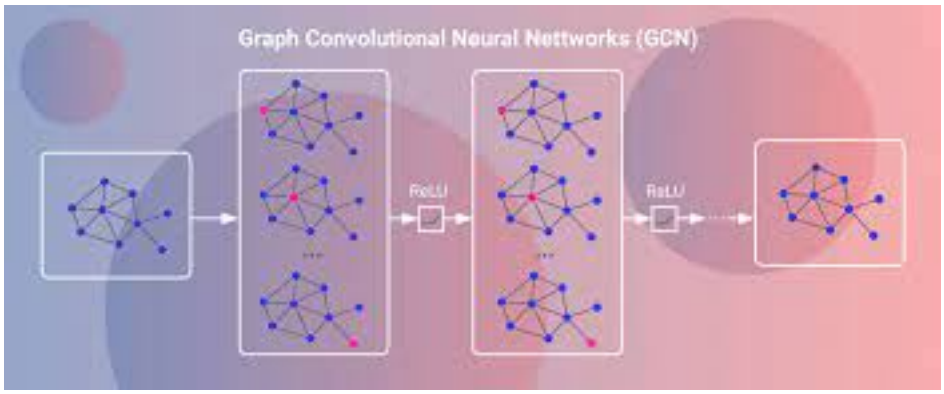

Figura 13: L'architettura a *Graph Convolutional Netwrok* (GCN) è una delle più utilizzate per la costruzione di reti neurali su dati in forma di grafi o network, e si basa sull'analogia con le reti convoluzionali classiche definite su spazi euclidei, generalizzando ai grafi l'operazione di convoluzione

città di dimensioni fino a 7×8 (Fig. 14). L'algoritmo genetico converge ad un numero di binari inferiore al limite massimo con un tempo di calcolo medio prossimo alle 24 ore (in realtà per matrici più piccole, come una 5×4, il programma impiega poche ore, mentre per matrici più grandi, come una 7×7, arriva anche fino a 30/36 ore).

Il programma tuttavia risulta poco efficace e non utilizzabile per matrici più grandi. Oltre al tempo di calcolo, che cresce in modo esponenziale all'aumentare del numero dei nodi, nel caso di matrici grandi l'algoritmo genetico non riesce a produrre come risultato un grafo connesso, lasciando sempre qualche piccola tratta ferroviaria scollegata rispetto al corpo centrale dei binari. Una possibile soluzione a tale problema potrebbe essere quella di modificare la funzione di reward, così da premiare risultati con grafi connessi. Ciò tuttavia aumenterebbe ancora di più il tempo di esecuzione dell'algoritmo genetico, già alto per matrici grandi. Se ne conclude che l'algoritmo genetico risulta uno strumento utile ma limitato per la risoluzione del nostro problema. L'utilizzo di network neurali potrebbe risultare globalmente più efficace, seppur più complesso da realizzare.

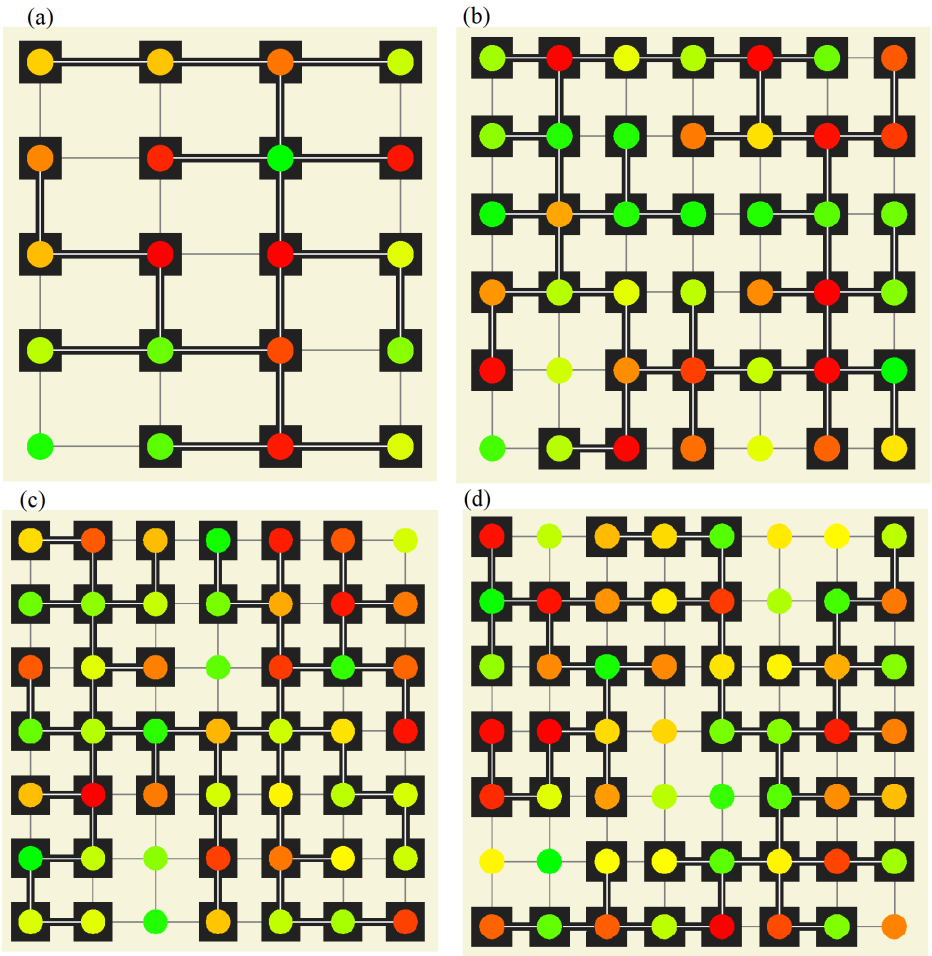

Figura 14: In figura si possono vedere i risultati ottenuti per città di diverse dimensioni. Nel caso di una griglia 5×4, (a), impostando un numero massimo di 25 binari, si è ottenuta una soluzione di 18 binari. Nel caso (b) di una griglia 6×7, si sono ottenuti 39 binari, rispetto ad un numero massimo consentito di 40. Per città 7×7 o 7×8, (c) e (d) rispettivamente, con un limite massimo di 45 binari sono state ottenute due soluzioni entrambe con 44 binari. Si può notare inoltre come la maggioranza dei nodi più popolosi sia direttamente collegato alla rete metropolitana.