Simulazione dell'evoluzione di un'epidemia sulla base del modello SIR

Silvia Vicentini

21 Gennaio 2024

Contents

1 Introduzione **2**

2 Cenni teorici **2**

3 Struttura del programma **3**
3.1 Classe Epidemic 4
3.2 Cartella Graph 5
3.2.1 Classe Curve 5
3.2.2 Classe Grid 6
3.2.3 Classe Key 7
3.2.4 Classe Text 7
3.2.5 Classe Draw 8
3.3 Classe Simulation 9

4 Strategia di test **11**

5 Compilazione tramite CMake **12**

6 Input, output e risultati attesi **13**

1 Introduzione

Il programma implementato ha lo scopo di simulare la diffusione di un'epidemia. Scritto in linguaggio C++ standard, il codice è salvato su *GitHub* visibile al seguente link: epidemic. Per la rappresentazione grafica dei dati si è fatto ricorso alla libreria esterna SFML, disponibile su piattaforma Ubuntu 22.04. Il codice è interamente formattato tramite tool *clang-format*, i test vengono eseguiti attraverso *Doctest* e la compilazione per mezzo di *CMake*. In quanto il programma è stato realizzato da un unico autore, non si è fatto ricorso dello strumento *Git* per la collaborazione.

2 Cenni teorici

Il programma sviluppato, composto da due parti, consente di simulare la diffusione di un'epidemia. Nella prima parte si applica il modello teorico SIR: si suddivide una popolazione chiusa di N abitanti nelle seguenti categorie:

- S, i *suscettibili*, coloro che possono essere contagiati
- I, gli *infetti*, coloro che possono trasmettere la malattia
- R, i *rimossi*, i guariti o morti

Dal momento che le morti per malattia sono incluse nella categoria R e le nascite o le morti per altre cause non vengono tenute in considerazione, vale la relazione

$$S + R + I = N$$

dove N, I, $R \in \mathbb{N}$. Sulla base di questo modello, ogni individuo, una volta guarito, diventa immune alla malattia, perciò chi appartiene alla categoria S può solo entrare a far parte del gruppo I, mentre gli individui in I possono solamente transitare all'insieme R. Lo spargimento dell'agente patogeno dipende da due fattori:

- β, il tasso di contagio, ovvero la probabilità di essere contagiati
- γ, il tasso di guarigione, cioè la probabilità di guarire

dove $\beta \in [0, 1]$ e $\gamma \in [0, 1]$. Si dice che l'epidemia è in espansione se $\frac{dI}{dt} > 0$. Dato che $\frac{dI}{dt} = (\beta\frac{S}{N} - \gamma) \cdot I$, allora $\frac{\beta}{\gamma} > \frac{N}{S}$. Considerato che all'inizio dell'epidemia $S \approx N$, la malattia si diffonde solo se è verificata la condizione che il rapporto $r_0 = \frac{\beta}{\gamma} > 1$ e cioè se

$$\beta > \gamma$$

La variazione dei tre parametri S, I ed R è definita dalle seguenti tre leggi discretizzate per intervalli di tempo $\Delta t = 1$ giorno:

$$\begin{aligned} S_i &= S_{i-1} - \beta\frac{S_{i-1}}{N}I_{i-1} \\ I_i &= I_{i-1} + \beta\frac{S_{i-1}}{N}I_{i-1} - \gamma I_{i-1} \\ R_i &= R_{i-1} + \gamma I_{i-1} \end{aligned} \tag{1}$$

Tale modello teorico apporta notevoli semplificazioni alla reale diffusione di un'epidemia, nonostante ciò, consente di trattare l'andamento generale del problema in maniera semplice e chiara.

Per rendere più realistica la diffusione dell'epidemia si è scelto di attivare una quarantena qualora il numero degli infetti raggiunga il 60% del numero totale N di cittadini. Questa viene protratta per la durata di 14 giorni. In tale caso la diffusione della malattia è definita dalle equazioni sottostanti

$$\begin{aligned} S_i &= S_{i-1} \\ I_i &= N - S_i - R_i \\ R_i &= R_{i-1} + \gamma I_{i-1} \end{aligned} \tag{2}$$

Infatti il numero di persone suscettibili non può variare, la probabilità di guarire resta costante e il numero di individui infetti varia mantenendo però costante il numero di individui totali nella popolazione. Il progetto si sviluppa a partire da queste equazioni.

La seconda parte del programma simula l'epidemia attraverso un automa cellulare, ovvero una griglia dove ciascuna casella rappresenta un individuo della popolazione. Ciascuna casella può esprimere uno dei tre stati definiti dal modello SIR: a seconda del colore assunto può voler indicare individuo suscettibile, infetto, rimosso o spazio vuoto, cioè dove non sono presenti individui. La griglia evolve ad intervalli di tempo discreti. Ogni casella viene influenzata dallo stato delle caselle adiacenti che definiscono il suo passaggio a stati successivi nel rispetto delle leggi della probabilità. Un individuo nello stato S circondato da n individui nello stato I, può transitare allo stato I secondo la probabilità p data da

$$p = 1 - (1 - \beta)^n \tag{3}$$

Un individuo infetto può invece transitare allo stato R secondo la probabilità γ, dato che secondo il modello SIR la probabilità di guarigione non è influenzata dal numero di persone attorno a sé.

3 Struttura del programma

Il progetto è organizzato nei seguenti files e cartelle:

Cartella epidemic Contiene *epidemic.hpp*, *epidemic.cpp* ed *epidemic.test.cpp*, un header file, il rispettivo source file e il file che raccoglie i test. Questi servono per studiare come si modifica giorno per giorno nel corso dell'epidemia il numero di persone nei gruppi S, R e I.

Cartella graph È suddivisa a sua volta in sottocartelle, ciascuna contenente file di intestazione e relativo file sorgente. Nella cartella è in più presente un file di test denominato *graph.test.cpp*. Tutti i metodi ivi dichiarati e definiti hanno lo scopo di graficare l'andamento delle variabili S, I e R calcolate dalla classe *Epidemic* definita nella cartella precedente.

Cartella simulation Organizzata tra header, source e test file, contiene la parte di programma finalizzata alla simulazione dell'epidemia tramite automa cellulare.

main_graph.cp Richiama la funzionalità dei metodi che consentono di studiare l'andamento delle variabili S, I ed R e graficarle.

cellular_automaton.cpp Richiama la funzionalità dei metodi che consentono di realizzare l'automa cellulare l'altro.

Tutti i files sono racchiusi in uno stesso namespace di nome *ep*. I metodi che non modificano il valore ritornato sono definiti come *const*.

3.1 Classe Epidemic

La classe *Epidemic* è quella che determina, sulla base delle leggi (1) e (2) l'evoluzione dell'epidemia.

Nel file *epidemic.hpp* si dichiara inizialmente una struct nominata *Population*. Questa rappresenta uno dei possibili modi in cui la popolazione si può suddividere tra S, I ed R. Per garantire una maggior precisione nei risultati, si vogliono eseguire i calcoli su oggetti di tipo double, ma, dato che S, I ed R appartengono all'insieme dei numeri naturali, li si vuole restituire come int. Per fare ciò, la struttura è un template parametrizzato dal tipo *type*. Talvolta, in base alle esigenze, si ha dunque a che fare con una popolazione di tipo int, altre volte con una di tipo double.

I membri privati della classe sono le probabilità β e γ, lo stato iniziale della popolazione, definito da un oggetto di tipo Population<int>, il numero di giorni T per il quale si vuole prevedere l'andamento dell'epidemia, il numero totale di abitanti N, la percentuale di infetti rispetto ad N oltre la quale scatta la quarantena e la durata della quarantena. Infine vi è un vettore, *simulation_double_*, che immagazzina i risultati dei calcoli svolti su una popolazione composta da oggetti di tipo double.

Tra i metodi privati vi sono:

calculate Calcola lo stato successivo della popolazione sulla base di quello precedente applicando le leggi (1). I valori così ottenuti sono di tipo Population<double>.

lockdown Svolge un ruolo analogo alla funzione *calculate* nel caso in cui si effettui un lockdown. Calcola lo sviluppo dell'epidemia a partire dalle leggi (2).

round Esegue l'arrotondamento per eccesso o per difetto dei valori double in maniera tale da costruire un oggetto di tipo Population<int>.

keep_total_constant Verifica che l'arrotondamento eseguito tramite round conservi costante il numero totale di individui N. Nel caso in cui N venga diminuito, si aggiunge una unità a S, se viene aumentato si sottrae una unità a R, così da non modificare il valore di I e non alterare lo studio dell'epidemia.

Tra i metodi pubblici sono presenti:

Epidemic Il costruttore della classe. Questo richiede come argomenti i due parametri β e γ, lo stato iniziale della popolazione ed il numero di giorni T per cui si vuole studiare l'epidemia. Nel corpo del costruttore sono contenuti i throws, che impongono le condizioni di esistenza dei sei parametri che vengono passati al costruttore. Se le condizioni imposte non sono rispettate, viene lanciato in output un messaggio di errore. Oltre a ciò, il costruttore converte lo stato iniziale della popolazione da Population<int> a Population<double> e costruisce il vettore *simulation_double_* applicando le funzioni *calculate* o *lockdown* a seconda che si sia in lockdown o in stato di libera circolazione.

evolution Costruisce il vettore contenente oggetti di tipo Population<int>, arrotondando tramite *round* e *keep_total_constant* il vettore *simulation_double_*.

Metodi getter Sono sei in totale e restituiscono i parametri dell'epidemia, il numero totale di individui N e il vettore *simulation_double_*, quest'ultimo utile per eseguire i test.

print_results Crea un file intitolato *results.txt* utilizzando la libreria <fstream> della standard library. Tramite la sintassi std::ofstream e l'operatore <<, stampa sul file i valori raccolti nel vettore costruito in *evolution*, indicando per ogni stato della popolazione il giorno a partire dall'inizio dell'epidemia in cui si ottiene tale risultato.

3.2 Cartella Graph

La cartella *Graph* contiene tutte le classi utili a realizzare un grafico che mostri l'andamento nel tempo delle tre variabili S, I ed R (vedi Fig. 1). Per fare ciò ci si è serviti della libreria grafica SFML.

3.2.1 Classe Curve

La classe *Curve* realizza quegli oggetti necessari per graficare le tre curve che rappresentano l'andamento delle tre variabili S, I ed R.

Il primo membro privato della classe è un oggetto di tipo *Epidemic* che rappresenta l'epidemia che si vuole graficare. Da questo si ricavano i parametri N e T e *population_state_*, cioè il vettore che registra sotto forma di Population<int> l'andamento dell'epidemia calcolato precedentemente dalla funzione *evolve* della classe *Epidemic*. Si definiscono poi la finestra su cui si vogliono disegnare le curve e un elenco di costanti di tipo float o int che rappresentano dei parametri geometrici.

I metodi privati della classe sono:

nearest_multiple Dato che si è scelto di scomporre l'asse x in otto parti e l'asse y in sei, questa funzione consente di determinare il multiplo di otto o di sei più vicino rispettivamente a T o a N, ma che sia maggiore di questi. Così facendo, si garantisce che le curve ottenute siano contenute all'interno degli assi e che ne rispettino la scala.

generate_curve Utile a realizzare una curva ottenuta a partire dalla successione di segmenti che hanno come estremi i punti di coordinate (T_{i-1}, x_{i-1}) e (T_i, x_i) dove x può indicare uno tra i tre valori S, I, R, $i \in [1, T]$ e T_i esprime il giorno corrispondente. Tra i parametri della funzione vi è un oggetto di tipo *sf::Color*, che serve per consentire di diversificare il colore della curva a seconda del significato che assume x.

generate_dots Consente di realizzare un punto al centro di ciascun segmento di cui è composta la curva generata da *generate_curve*. Il colore del punto coincide col colore della curva a cui appartiene.

S_values, I_values e R_values Estraggono i valori di S, I ed R da *population_state_*. Sono quindi dei vettori di interi che mostrano come evolve nel tempo ciascuno dei tre raggrippamenti in considerazione.

Tra i metodi pubblici compaiono:

Curve Il costruttore che richiede come argomenti un oggetto di tipo *Epidemic* e la finestra su cui disegnare il grafico.

S_curve, I_curve, R_curve, S_dots, I_dots, R_dots Sei funzioni che disegnano le curve e i puntini per graficare le tre variabili descrittive della popolazione. La curva che rappresenta l'andamento delle persone suscettibili è realizzata in blu, quella che rappresenta l'andamento degli infetti in rosso e quella che descrive i rimossi in verde.

3.2.2 Classe Grid

Questa classe serve per generare quegli oggetti necessari alla realizzazione di una griglia che consenta di poter interpretare i valori espressi dalle tre curve.

I membri privati della classe sono innanzitutto la finestra su cui disegnare la griglia, e una serie di costanti di tipo float o int di significato geometrico.

I metodi privati sono:

create_line Serve a disegnare un segmento note le coordinate dei punti estremi del segmento ed il suo colore.

create_lines Realizza un insieme di linee parallele di colore azzurro che possono essere verticali od orizzontali e che attraversano tutta la cornice che racchiude il grafico.

I metodi pubblici sono:

Grid Il costruttore, richiede come argomento la finestra su cui disegnare la griglia e inizializza i vari parametri geometrici del sistema.

bottom_side, top_side, left_side e right_side Necessari a rappresentare i quattro lati di colore nero che incorniciano il grafico.

vertical_lines e horizontal_lines Realizzano il fascio di rette rispettivamente verticali e orizzontali di colore azzurro che attraversano il grafico grazie al metodo *create_lines* definito precedentemente.

3.2.3 Classe Key

In questa classe si vogliono creare gli oggetti per realizzare una legenda al grafico. Ciò include tre segmenti con al centro un puntino di colori blu, rosso o verde, affiancati dall'indicazione di ciò che questo simbolo rappresenta, cioè rispettivamente i suscettibili, gli infetti ed i rimossi.

I membri privati della classe sono la finestra grafica ed i parametri geometrici del problema, nonché l'oggetto *font_*, che definisce il carattere con cui realizzare le scritte. Questo oggetto viene poi caricato dal file *Arialn.ttf* nel corpo del costruttore della classe.

I metodi privati della classe sono **create_line**, **create_dot** e **create_text**, che creano un segmento, un punto e una riga di testo, li posizionano sulla finestra grafica e attribuiscono ad essi un colore.

I metodi pubblici sono **key_lines**, **key_dots** e **key_description**, necessari per creare i segmenti ed i puntini che riprendono le tre curve del grafico e la descrizione a parole di quale parametro tra S, I ed R rappresentano. Infine viene definito il costruttore **Key**, che prende come unico argomento la finestra grafica. Nel suo corpo viene caricato l'oggetto *font_* col carattere Arial Narrow definito nel file *Arialn.txt* allegato nel progetto. Nel caso durante l'esecuzione si notasse che non compaiono scritte sul grafico, si consiglia di sostituire la riga 10 del file *graph/key/key.cpp*:

```
    font_.loadFromFile("../../Arialn.ttf");
```

con il path completo per raggiungere il file che si vuole caricare, nel mio caso:

```
  font_.loadFromFile("/home/silvia/pf_labs/progetto/graph/
     Arialn.ttf");
```

3.2.4 Classe Text

Il ruolo di questa classe è creare quegli oggetti necessari per inserire nel grafico le varie scritte: i numeri lungo l'asse delle ascisse e delle ordinate, i titoli degli assi e del grafico stesso.

I membri privati sono un oggetto di tipo *Epidemic*, la finestra grafica, il font con cui realizzare le scritte e diversi parametri geometrici. Dall'oggetto di tipo *Epidemic* si estraggono i valori T, numero di giorni per cui simulare l'epidemia, ed N, numero totale di individui nella popolazione.

Tra i metodi privati compaiono:

create_text È una funzione a cui si passano come parametri iniziali il testo che si vuole scrivere, le coordinate del punto della finestra grafica dove si vuole posizionare tale scritta e le dimensioni del font. La funzione consente di creare un oggetto di tipo

sf::Text, di posizionarlo nella finestra, di definire le sue dimensioni e di associargli il colore nero.

generate_numbers Dato il valore massimo che l'asse può rappresentare (variabile da epidemia ad epidemia, espresso da T per le ascisse e da N per le ordinate), stabilisce i sette o i cinque valori intermedi rispettivamente tra 0 e il valore massimo (vedi Fig. 1). Tutti questi valori, che coincidono con i valori assunti dai segmenti che compongono la griglia, vengono convertiti da int in stringhe. Vengono poi posizionati sulla finestra e scritti in nero.

I metodi pubblici sono:

Text Il costruttore che ha come argomenti un oggetto di tipo *Epidemic* e la finestra su cui disegnare le scritte. Inizializza alcune delle variabili geometriche ed il font secondo lo stile del carattere Arial Narrow. Nel caso il font non venisse caricato correttamente, si consiglia di seguire le indicazioni precedentemente fornite. Le costanti *max_x_val_* e *max_y_val_*, rispettivamente il massimo valore sull'asse x e sull'asse y, vengono calcolati dalla funzione *nearest_multiple* di cui sotto.

nearest_multiple Stessa funzione che compare nella classe *Curve*, consente di rappresentare le curve in maniera tale che siano interamente contenute all'interno della cornice del grafico nel rispetto della scala. Per verificare la correttezza di questa funzione è stato aggiunto nella cartella *graph* un file di test chiamato *graph.test.cpp*. Ci si è interrogati su come evitare la ripetizione, dato però che questo è l'unico metodo in comune tra le due classi che hanno ruoli molto diversi, si è concluso che la scelta più efficiente è quella di non sfruttare l'ereditarietà virtuale.

set_Title, x_axis_name, y_axis_name, x_numbers, y_numbers Creano il titolo del grafico, i titoli degli assi ed i numeri agli assi sfruttando i metodi precedentemente definiti.

3.2.5 Classe Draw

Questa classe serve per disegnare sulla finestra grafica tutti gli oggetti precedentemente definiti utili alla realizzazione del grafico.

I membri privati della classe sono un oggetto di tipo *Epidemic*, la finestra grafica e quattro oggetti di tipo *Curve*, *Grid*, *Key* e *Text*. Non vi sono metodi privati, i metodi pubblici includono solo il costruttore e la funzione **draw_graph**. Il costruttore richiede come argomenti la finestra grafica e un oggetto di tipo *Epidemic*, sfruttando questi due oggetti si possono inizializzare i restanti quattro membri privati.

Nel metodo *draw_graph* si disegnano le curve, la griglia, la legenda e le scritte. Si aggiunge inoltre la possibilità di salvare l'immagine stampata sulla finestra grafica premendo il comando "s" e di chiudere la finestra premendo il pulsante "close". Il programma restituisce in output la conferma del corretto salvataggio del grafico.

3.3 Classe Simulation

In questa sezione del progetto si vuole realizzare l'automa cellulare (vedi Fig. 2).

Primo tra i membri privati della classe vi è un oggetto di tipo *Epidemic*, da cui vengono estratti tutti i parametri che lo descrivono. Questi valori servono per poter configurare lo stato iniziale della griglia. Altri membri privati sono la finestra grafica su cui disegnare il modello e gli interi *parts_* e *N_cells_*. Per inizializzare *parts_* si usa il metodo fornito dalla standard library, *std::ceil*, che restituisce il più piccolo intero maggiore o uguale al valore fornito come argomento. Queste due costanti consentono di stabilire in quante caselle suddividere la griglia quadrata per contenere tutti gli N cittadini. Compaiono poi quattro variabili di tipo int che esprimono il numero di giorni trascorsi, di suscettibili, di infetti e di rimossi e vengono aggiornate ad ogni iterazione. Includendo la libreria della standard library <random>, si dichiarano un oggetto di classe *std::random_device* che genera il seme utile per inizializzare altri generatori di numeri casuali, ed un oggetto di tipo *std::default_random_engine* per generare numeri casuali da distribuzioni predefinite. Infine si dichiarano due costanti di natura geometrica ed il font utilizzato per realizzare le scritte.

I metodi privati includono:

key_cells e key_texts Creano gli oggetti, celle e scritte, che costituiscono la legenda dell'automa cellulare.

set_text Crea un oggetto di tipo *sf::Text* che serve per indicare il numero dei giorni trascorsi ed il numero dei componenti di S, I ed R. La funzione definisce inoltre il colore, la posizione, la grandezza ed il font della scritta.

display_grid_values Si dichiara un vettore *grid* di dimensione pari al numero di caselle dell'automa. Tramite l'algoritmo *std::iota* si riempie un secondo vettore di interi, chiamato *CellNumber*, con tutti i valori compresi tra 0 e *N_cells_*. Li si mischia in maniera casuale. Si riempie *grid* coi valori 0, 1, 2 e 3, che stanno a rappresentare rispettivamente lo stato S, I, R o le celle vuote. Ciascuno di questi quattro valori compare nel vettore tante volte quanto indicato dal numero di individui espresso dal relativo insieme di appartenenza. Nell'inserire i valori in *grid* non si rispetta l'ordine crescente, bensì quello definito da *CellNumber*. In questo modo la configurazione iniziale della griglia risulta contenere S, I ed R individui sparsi casualmente tra le caselle a disposizione.

draw_grid La funzione ha lo scopo di realizzare attraverso oggetti di tipo *sf::RectangleShape* la griglia dell'automa cellulare. Si associa ad ogni casella della griglia un valore appartenente a *grid*. Dunque si attribuisce alle caselle con valore 0 il colore blu, che rappresenta il gruppo dei suscettibili, alle caselle con valore 1 il colore rosso, correlato agli infetti, ai quadrati di valore 2, che indicano rimossi, si attribuisce il colore verde, infine le caselle vuote, di valore 3, sono in bianco.

calculate Consente di far evolvere ciascuna casella allo stato successivo: fa transitare le celle rappresentanti un soggetto nello stato S in I, mentre converte le caselle nello

stato I in R. Il passaggio da S a I avviene in base ad una certa probabilità definita dalla legge (3), mentre il passaggio da I a R avviene secondo la probabilità γ. Ottenuti questi due valori probabilistici, si estrae casualmente da una distribuzione uniforme di oggetti di tipo double compresa tra 0 e 1 un valore casuale, se questo è minore del valore che esprime la probabilità si fa evolvere la cella.

counter Scorrendo tutto il vettore *grid*, il metodo conta il numero di caselle che possiedono lo stesso valore tra 0, 1, 2 o 3.

I metodi pubblici sono:

Simulation Il costruttore, che ha come argomenti la finestra grafica e l'oggetto di tipo *Epidemic*. Nel corpo del costruttore si inizializza l'oggetto *font_* secondo il carattere Arial Narrow. Come consigliato precedentemente, si suggerisce di sostituire la riga del codice dove si carica il font col path corrispondente al file *Arialn.ttf* sul proprio dispositivo nel caso in cui si osservasse che le scritte non vengono caricate correttamente sulla finasta grafica.

calculate_neighborSum Calcola quante delle otto caselle circostanti una cella specifica sono di tipo I.

probability Calcola la probabilità del passaggio da S a I secondo la legge (3).

wrapValue Tratta la griglia come se questa non avesse confini, ma fosse disposta sulla superficie di una sfera. Secondo questo criterio le celle che si trovano sui bordi sentono dell'influenza delle celle che si trovano nella parte opposta della griglia.

get_parts_ Restituisce il valore di *parts_*, utile per i test.

cellular_automaton Apre la finestra e ivi disegna la griglia nella configurazione iniziale. Questa viene aggiornata ad intervalli discreti di 1 secondo. A lato della griglia si disegnano anche la legenda e le scritte che tengono il conto della variazione degli individui suscettibili, infetti e rimossi. La finestra si chiude dopo che è trascorso un numero di intervalli discreti pari al numero di giorni per cui si vuole studiare l'epidemia. Si può chiudere la finestra prima di questo tempo premendo il pulsante close.

I metodi *display_grid_values*, *calculate*, *calculate_neighborSum* e *cellular_automaton* non sono dichiarati come const dal momento che modificano lo stato interno dell'oggetto chiamante di classe *Simulation*.

4 Strategia di test

All'interno del progetto sono stati implementati tre files di unit test tramite l'utilizzo di Doctest: *epidemic.test.cpp* per verificare che lo sviluppo dell'epidemia venga studiato nel corretto rispetto delle leggi (1) e (2), ed i files *graph.test.cpp* e *simulation.test.cpp*.

Per testare la classe *Epidemic* si è scelto innanzitutto di verificare i throw, cioè che i parametri del problema vengono accettati dal programma solamente qualora questi appartengano ai relativi campi di esistenza. Più che testare ogni singolo metodo, per verificare il corretto funzionamento della classe è sembrato più opportuno controllare che l'output finale dato dai vettori contenenti l'evoluzione della pandemia in double o in int fosse corretto. Affinché questo avvenga, devono funzionare correttamente tutte le funzioni che consentono di costruire tali vettori e devono essere eseguite durante l'iterazione corretta. Per verificare la funzione *calculate*, si è quindi scelto di verificare la correttezza del vettore *simulation_double_* per tre diversi casi di epidemie nelle quali non si attivano mai quarantene. Lo stesso si è svolto per la funzione *lockdown*, ma studiando epidemie durante le quali si attivano lockdown. Si è poi voluto studiare la correttezza del vettore in casi più generici andando ad analizzare almeno il 34esimo elemento del vettore. Passando poi al vettore di oggetti di tipo Population<int>, si analizzano i casi in cui la popolazione viene sottostimata o sovrastimata dalla funzione *round* e ci si accerta che venga attuata la giusta correzione dalla funzione *keep_total_constant*. Si verifica il risultato finale che viene stampato in output espresso dalla funzione *evolution*. Tutti i valori utilizzati per i test sono stati calcolati tramite il foglio di calcolo Excel. Infine, per testare la funzione *print_results*, si verifica direttamente che questa stampi i valori attesi sul file *results.txt*.

I files contenuti nella cartella Graph hanno come unico scopo quello di creare oggetti appartenenti alla libreria SFML, che poi vengono stampati sulla finestra grafica. Nessuno di questi esegue quindi calcoli. Per questo non si è ritenuto necessario scrivere un file di test per ciascuno dei files sorgenti della cartella. È stato implementato un unico file di test, *graph.test.cpp*, per verificare il corretto funzionamento del metodo *nearest_multiple*, così da assicurarsi che la scala usata in ciascuno dei due assi del grafico sia corretta. Si verifica il corretto funzionamento di *draw_graph*, stampando il grafico su schermo.

Nella classe Simulation compaiono metodi di natura molto diversa, alcuni servono per creare oggetti dalla libreria SFML da disegnare sulla finestra, tali da non necessitare di test. Altri consentono di attribuire a delle variabili dei valori casuali e dunque in questo caso non si possono eseguire test data la natura randomica del problema. Si possono svolgere test solo sulle funzioni che eseguono calcoli, quali *wrapValue*, *probability* e *calculate_neighborSum*. Si verifica inoltre la correttezza del valore assegnato a *parts_*. Si esamina il metodo *cellular_automaton* direttamente stampando l'automa su finestra grafica.

5 Compilazione tramite CMake

Per la compilazione del programma ci si è serviti di CMake, uno strumento open source progettato per semplificare il processo di compilazione. Questo consente di scrivere tutte le opzioni di configurazione un'unica volta in un file intitolato *CMakeLists.txt*. Si richiede l'utilizzo del linguaggio C++17 standard e l'abilitazione ai warnings definiti dal comando di compilazione -Wall -Wextra. Si abilitano i controlli di tipo adress sanitizer e undefined-behaviour sanitizer, strumenti di rilevamento degli errori di memoria durante l'esecuzione. Si richiede il pacchetto SFML versione 2.5 e la componente "graphics". Si definiscono due eseguibili, relativi ai due files coi quali può interagire l'utente, nominati *epidemic_graph* e *cellular_automaton*, e si elenca l'insieme dei file sorgente che devono essere compilati per generare l'eseguibile. A questi due eseguibili si allega la libreria SFML necessaria per l'interfaccia grafica. Si passa poi ad aggiungere la lista dei file sorgente per la creazione dei tre eseguibili da aggiungere ai test. Questi sono *epidemic.t*, *graph.t* e *simulation.t*, da eseguire uno alla volta. A questi ultimi due è stata a sua volta allegata la libreria SFML. L'ultimo comando del file abilita la formattazione automatica tramite *clang-format* di tutti i file sorgente che compongono il progetto.

Durante la scrittura del programma si è costruita l'area di compilazione del progetto tramite CMake in modalità Debug per mettere in evidenza eventuali errori grazie al seguente comando:

```
$ cmake -B build -S . -DCMAKE_BUILD_TYPE=Debug
```

Questo consente di creare una directory di build chiamata *build* che serve come area di configurazione. Una volta concluso il progetto si è passati a costruire l'area di compilazione tramite:

```
$ cmake -B build -S . -DCMAKE_BUILD_TYPE=Release
```

che, a differenza del comando precedente, permette di compilare in modalità Release, così da incrementare le prestazioni a runtime. Si consiglia di usare il secondo comando se si vuole semplicemente eseguire il programma, mentre se si vogliono attuare modifiche allo stesso è raccomandato ricorrere al primo.

Per compilare si esegue su terminale il comando:

```
$ cmake --build build
```

Per eseguire la parte di programma che studia l'epidemia sulla base delle leggi (1) e (2) e graficare i risultati ottenuti si esegue il comando:

```
$ ./build/epidemic_graph
```

Per eseguire la parte di programma che studia l'epidemia tramite automa cellulare si esegue il comando:

```
$ ./build/cellular_automaton
```

Infine per eseguire i test si eseguono i seguenti comandi:

```
$ ./build/epidemic.t
$ ./build/graph.t
$ ./build/simulation.t
```

6 Input, output e risultati attesi

I due files **main_graph.cpp** e **cellular_automaton.cpp** sono gli unici due files che richiamano la funzionalità delle classi definite nel programma.

Entrambi i files utilizzano la libreria <iostream> della standard library, inclusa in epidemic.hpp, per mandare messaggi in output all'utente e ricevere valori in input. Avviati entrambi gli eseguibili, compare su terminale in output la richiesta di inserimento dei parametri necessari per inizializzare l'oggetto di tipo Epidemic, cioè la pandemia che si vuole studiare. Per chiarire le loro condizioni di esistenza, queste vengono esplicitamente indicate in output. Fatto ciò, il file *main_graph.cpp* esegue il metodo *print_results* della classe Epidemic così da salvare su *results.txt* i valori dei parametri S, I ed R in evoluzione per l'arco di tempo richiesto. Nel caso in cui non si riuscisse ad aprire tale file, si stampa in output un messaggio di errore. Ecco un esempio del risultato ottenuto per una popolazione inizialmente spartita nelle categorie $S_0 = 997$, $I_0 = 10$, $R_0 = 0$, dove $\beta = 0.8$, $\gamma = 0.4$ e $T = 36$

```
Report of each of the stored states of population:
Day         S           I           R
----------------------------------------
0           997         10          0
1           989         14          4
2           978         19          10
3           963         27          17
4           943         36          28
5           916         49          42
[...]
33          177         1           829
34          177         1           829
35          176         1           830
36          177         0           830
```

Si invoca poi la funzione *draw_graph* della classe Draw, che disegna sulla finestra grafica *window*, precedentemente definita nel file, il grafico relativo ai dati raccolti. Su terminale viene chiesto di premere il pulsante "s" per salvare il grafico. In tal caso, questa viene salvata su file intitolato *screenshot.jpeg*. Si riporta in Fig. 1 il grafico dell'epidemia studiata nell'esempio precedente.

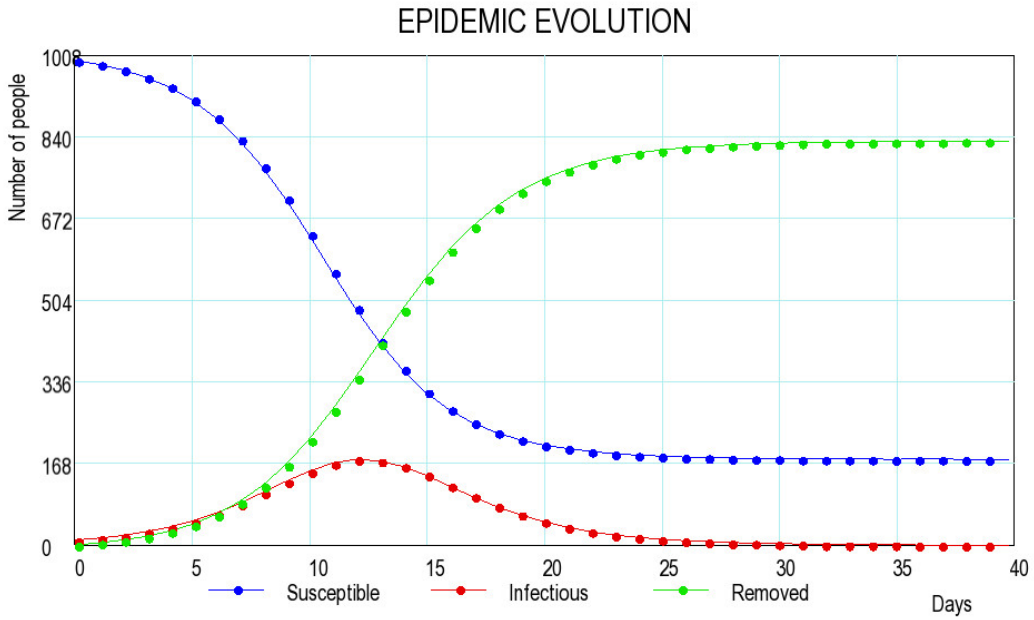

Figure 1: Grafico relativo all'epidemia con parametri iniziali $\beta = 0.8$, $\gamma = 0.4$, $S_0 = 997$, $I_0 = 10$, $R_0 = 0$, $T = 40$

Si osserva, a partire dai dati ottenuti, che il programma funziona nella maniera richiesta e cioè che trascorso un determinato periodo di tempo, variabile a seconda dell'epidemia in analisi, il numero di infetti arriva ad annullarsi. Questo vuol dire, così come atteso, che l'epidemia arriva ad estinguersi. Si sono confrontati i grafici ottenuti con i valori calcolati e si osserva che i primi rispettano correttamente l'andamento dei secondi. Per avere comunque maggiore certezza dei risultati ottenuti, si sono studiate le stesse epidemie con foglio di calcolo Excel e si è osservato che i grafici così ottenuti coincidono.

Il file *cellular_automaton.cpp* non stampa in output alcun valore numerico, ma consente, una volta inseriti i parametri iniziali e definita la finestra *window*, di avviare l'automa cellulare tramite il metodo *cellular_automaton* della classe *Draw*. L'automa si chiude automaticamente una volta trascorsi il numero di giorni richiesti. Se si vuole chiudere la finestra grafica prima che sia trascorso il numero di iterazioni indicato, si può premere il pulsante "close". In Fig. 2 si riporta un esempio di automa cellulare in una delle infinite configurazioni in cui si può presentare.

Poiché l'automa cellulare rappresenta in maniera casuale la diffusione dell'epidemia, seppure seguendo determinate regole, non si può stabilire con esattezza come questo si comporterà. Tuttavia si osserva che a partire da una casella infetta rossa, l'epidemia si diffonde a macchia verso le caselle circostanti, senza però occuparle tutte, così come ci si

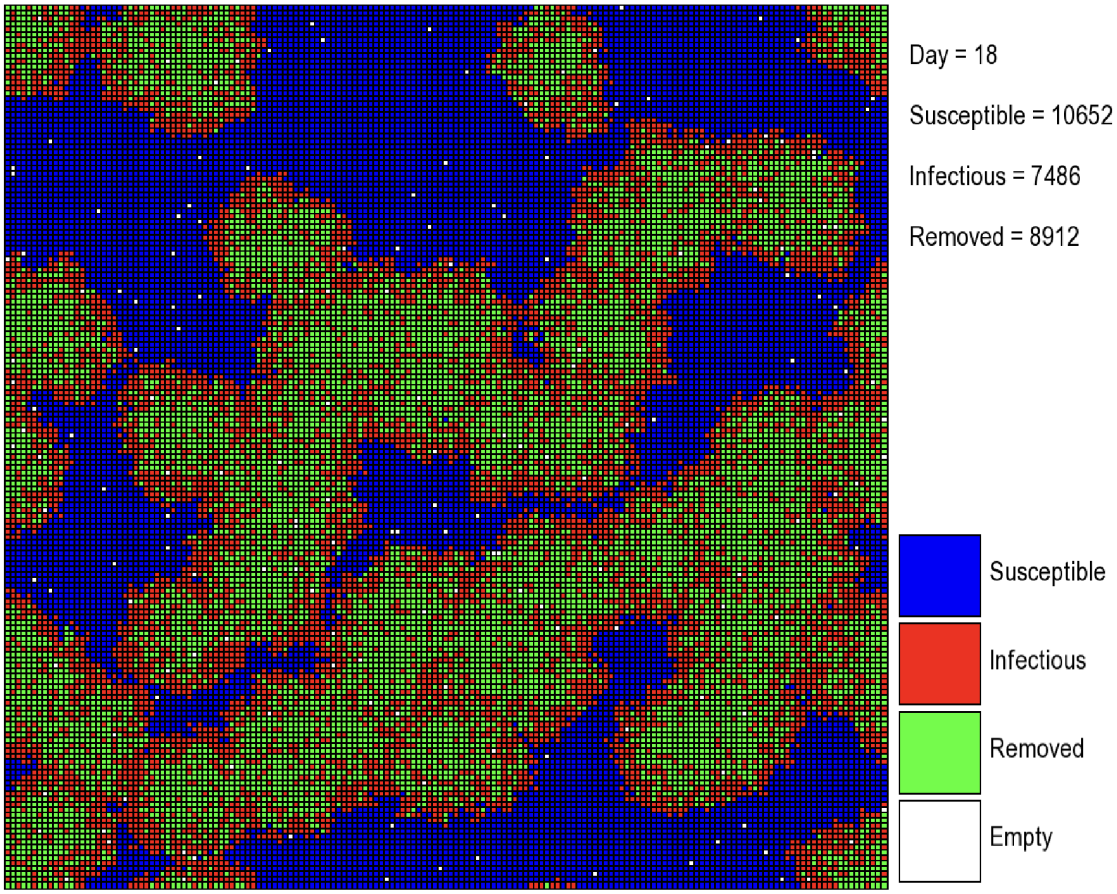

Figure 2: Una delle infinite configurazioni di un automa cellulare relativo all'epidemia con parametri iniziali $\beta = 0.25$, $\gamma = 0.14$, $S_0 = 27000$, $I_0 = 50$, $R_0 = 0$, $T = 60$

aspetterebbe. Si nota che aumentando il valore del parametro β il numero di infetti cresce più rapidamente e diminuendo γ il numero di rimossi aumenta più lentamente. A partire dagli stessi dati iniziali il ruolo che ciascuna casella assume nella griglia varia ogni volta che si esegue il programma, così come desiderato, dal momento che la configurazione iniziale deve essere casuale. Inoltre le caselle che si trovano a bordi opposti della griglia si influenzano a vicenda così da simulare un oggetto con geometria sferica. L'automa cellulare si comporta quindi come atteso.