Game of Life

Mario De Simone

1 Introduzione

L'obiettivo della seguente relazione è quello di studiare il comportamento il Game of Life inventato da John Conway negli anni sessanta.
A partire da un'implementazione efficiente, andremo a definire lo stato asintotico e come varia al rilassamento delle regole del gioco o eventuali perturbazioni.

2 Gioco

Il Game of Life è un gioco autonomo, ovvero data uno stati iniziale, la sua evoluzione non dipende in alcun modo dall'utent e consiste in una griglia dove ogni cellula che la compone è viva o morta, l'evoluzione di ogni cellula dipende dalle 8 che la circondano. Data una disposizione iniziale di vita, il gioco segue le regole:

- Una cellula morta diventa viva se è circondata da esattamente 3 cellule vive
- Una cellula viva muore se è circondata da più di 3 cellule vive
- Una cellula viva muore se è circondata da meno di 2 cellule vive
- Una cellula sopravvie alla generazione successiva se è circondata da 2 o 3 cellule vive

Usualmente la griglia la si considera continua, nel senso che le estremità sono collegate fra loro.

3 Strutture

Affinchè il codice fosse ottimizzato dal punto di vista della memoria che impiega sono state prese due decisioni di natura progettuale che incidono nella rappresentazione della cellula e nel meccanisco necessario a rintracciare se l'evoluzione del gioco è in loop:

- la cellula viene rappresentata mediante un unsigned char di cui manipolando i bit sfruttiamo LSB per rappresentare lo stato (0 morta, 1 viva) e i successivi 3 bit per rappresentare il numero di vicini vivi.
  Così facendo con gli 8 bit di un unsigned char è possibile rappresentare tutta l'informazione che ci interessa cellula per cellula, lasciando uno spreco di soli 4 bit.

- come verrà chiarito nel seguito è necessario comprendere quando si raggiunge un loop di stati, a tal fine si tiene memoria di quelli incontrati durante l'evoluzione mediante una tabella hash

Con un'impostazione di questo tipo era possibile scrivere un codice ancora più ottimizzato di quello ottenuto sfruttando a dovere l'operazione XOR, sacrificando però la leggibilità del codice, come suggerito in "General Algorithm for Totahstic Cellular Automata" di F. Bagnoli, R. Rechtman e S. Ruffo

4 Documentazione del codice

Il codice è stato scritto sfruttando c/c++, nella fattispecie dal c++ è stat presa l'implementazione delle unordered-map utilizzate come tabelle hash, non essendo stato utilizzata la programmazione ad oggetti (più costosa rispetto all'implementazione precedentemente discussa per le cellule), il codice è distribuito in un unico file che presenta i seguenti metodi:

- SetCell date delle coordinate x e y, imposta lo stato della cellula ad 1 e incrementa il numero di vicini delle celle vicine

- ClearCell date delle coordinate x e y, imposta lo stato della cellula a 0 e decrementa il numero di vicini delle celle vicine

- Draw consente di stampare su console la griglia di gioco

- NextGeneration effettua un ciclo su tutte le cellule al fine di eseguire una iterazione del gioco seguendo le regole base

- NextGenerationZero richiede in input un parametro di probabilità, con probabilità p il metodo farà morire la singola cellula ignorando le regole, mentre con probabilità 1-p seguirà le regole del gioco base

- NextGenerationIdentity richiede in input un parametro di probabilità, con probabilità p il metodo lascerà inalterata la cellula ignorando le regole, mentre con probabilità 1-p seguirà le regole del gioco base

- PatternReduce utilizzata per cercare loop di stati, riduce lòa griglia di unsigned char ad un'unica stringa di stati, che rappresenta univocamente lo stato complessivo del gioco

- Permutation controlla se ci siano cellule vive, in caso positivo cambia lo stato di una cellula morta con vicini vivi.

- Visualize permette di visualizzare lo stato del gioco sfruttando gnuplot
- Graph permette di visualizzare un grafico contenente il numero medio delle cellule vive nel tempo (rispetto al numero di ripetizioni richiesto dal codice)
- Main inizializza i file che verrano usati per Graph e Visualize, inizializza con una configurazione casuale di densità scelta la griglia infine esegue un loop che si occupa di effettuare le iterazioni del gioco fino a quando o si raggiunge lo stato asintotico o si eccedono il numero di iterazioni massimo possibile

Il codice oltre alla sopracitata libreria unordered-map sfrutta anche la random per la gestione delle estrazioni probabilistiche.
Sempre in intestazione al codice sono presenti una serie di variabili che consentono cambiare le impostazioni del programma, tali variabili hanno volutamente scope globale e sono tenute separate dal resto del codice in modo da separare parte di controllo da parte operativa.

5 Esperimenti eseguiti

L'automa ha differenti proprietà per cui è possibile fare innumerevoli esperimenti differenti, quelli analizzati in questa sede riguardano lo stato asintotico.
Definiamo stato asintotico, lo stato assunto dal sistema al limite delle iterazioni del gioco, usualmente si tratta di un loop di piccoli configurazioni, i nostri test vertono sull'analizzare la quantità di vita media che arriva allo stato asintotico in funzione della probabilità con cui vengono seguite le regole del gioco, del numero di iterazioni (tempo) e del numero di prove sul quale effettuiamo la media.
Per questa tipologia di esperimenti abbiamo studiato il comportamento di NextGenerationZero e NextGenerationIdentity, mentre per lo studio del ritardo asintotico, dove prendiamo e permutiamo lo stato asintotico cosiì da farlo uscire dal loop e verificare come impatta, è stato usato NextGeneration.
È necessario sottolineare infine che tutte le prove sono state fatto su una griglia di dimensione 100x100.

6 Risultati esperimenti

Verifichiamo il numero di iterazioni necessarie a raggiungere stao asintotico al variare della densità iniziale e della probabilità di applicazione di regola zero e identità.
Riportiamo tali tabelle per dare un'idea del costo temporale affrontato nella computazione di una sola occorrenza del gioco.

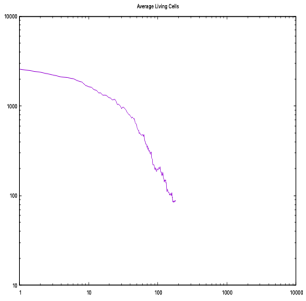
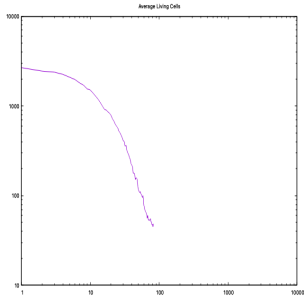
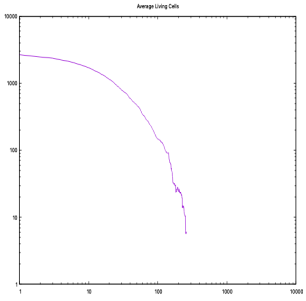
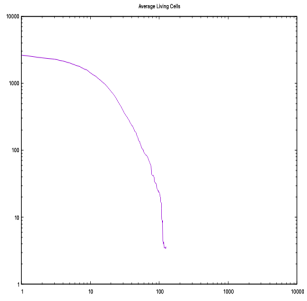
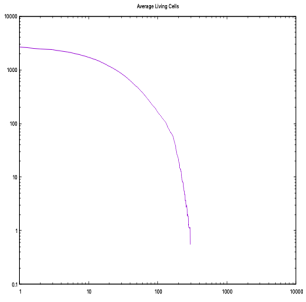
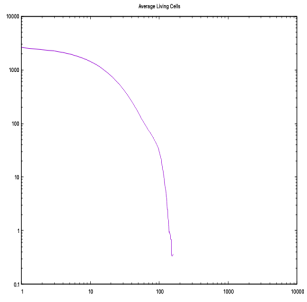
Probabilità → Densità ↓	p = 0.025	p = 0.05	p = 0.075
0.25	203	100	69
0.5	271	116	84
0.75	53	19	52

Numero iterazioni per stato asintotico con regola zero

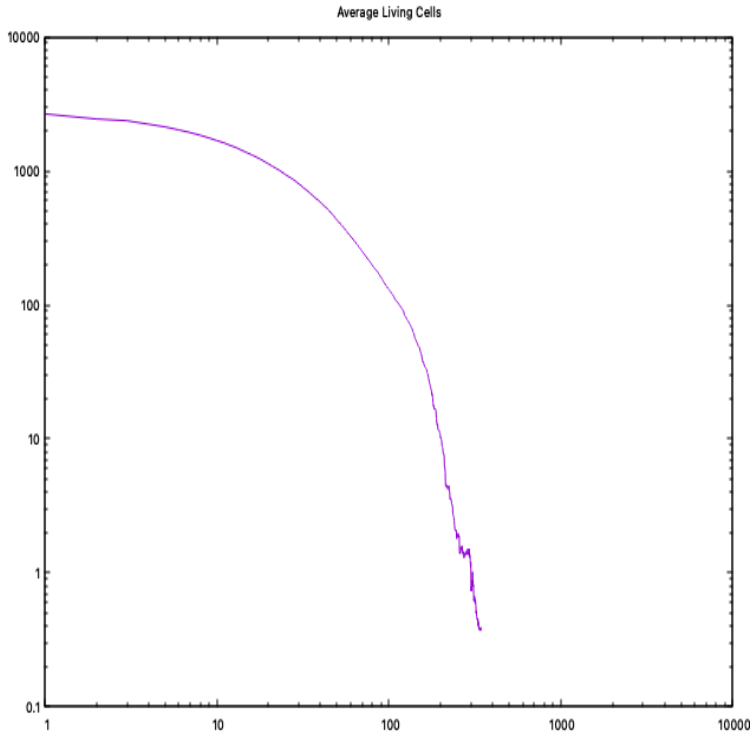
Probabilità → Densità ↓	p = 0.025	p = 0.05	p = 0.075
0.25	1280	1640	1740
0.5	1753	2592	1750
0.75	544	7355	2733

Numero iterazioni per stato asintotico con regola identità

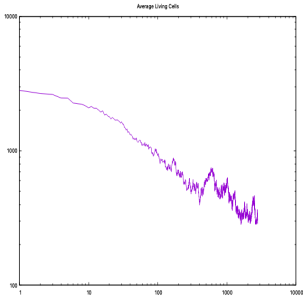
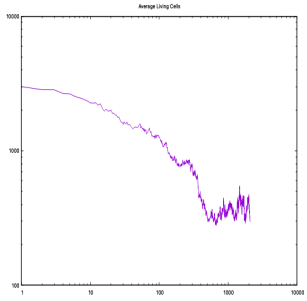
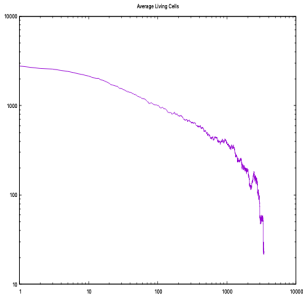
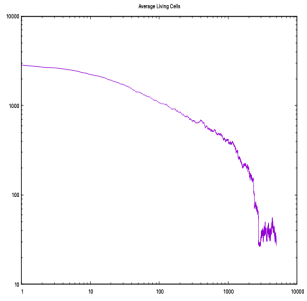
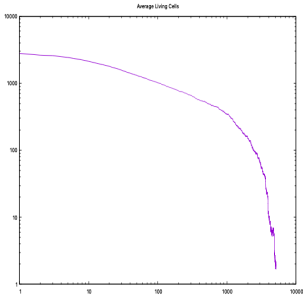
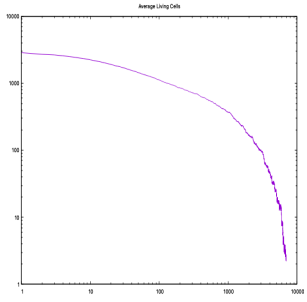
I risultati della prima tabella confermano che all'aumentare della densità iniziale il numero di iterazioni richiesto per arrivare allo stato asintotico generalmente aumenta.
Per la seconda tabella invece il discorso è più complicato questo perchè c'è una dipendenza non banale dalla configurazione iniziale, sicuramente il tempo richiesto è maggiore rispetto al metodo con regola zero, la possibilità di mantenere inalterato un sito implica la possibilità di continuare a generare vita ad un ritmo maggiore della versione a regola zero o base. Adesso vediamo i grafi di vita media in funzione di probabilità e numero di occorrenze, le prove sono state effettuate con densità iniziale di 0.5.
Iniziando dalle prove con regola zero:

	Probability = 0.025	Probability = 0.05
Prove = 1		
Prove = 10		
Prove = 100		

Innanzitutto possiamo notare che per probabilità maggiori la curva si piega più rapidamente e prima, mentre all'aumentare del numero di prove rende il grafico più preciso andando a abbassare l'importanza di eventuali prove "sporche".
Un caso particolare lo si ha per p $\simeq$ 0.028 come segue:



In questo caso prima di piegarsi si ha un tratto quasi rettilineo, nel dettaglio uno studio approfondito della transizione di fase del gioco è riportato in "Phase transitions near the game of Life" di Jacob Nordfalk e Preben Alstro, mentre si riporta l'articolo "Some Facts of Life" di Franco Bagnoli, Radl Rechtman e Stefano Ruffo in merito al perchè per $p = 0.028$ il comportamento sia differente. Proseguiamo con i relativi grafici nel caso di regola identità:

	Probability = 0.025	Probability = 0.05
Prove = 1		
Prove = 10		
Prove = 100		

In questo caso il tempo necessario alla conclusione è decisamente maggiore (come già accennato nelle tabelle), il fatto interessante è che tendenzialmente la discesa si mantiene poco ripida per quasi tutto il corso dell'esecuzione per poi avere un drastico calo in prossimità della conclusione, questo fatto potrebbe essre giustificato considerando che intrinsecamente la regola identità alla lunga innesca una serie di morti per sovrappopolazione.
Concludiamo con un semplice ma interessante dato, eseguendo 100 ripetizioni della versione base del gioco permutandolo all'ottenimento dello stato asintotico, facendo la media del numero di iterazioni di ritardo è stato ottenuto il valore 156.169998.
Una spiegazione potrebbe essere data dal fatto che l'impatto che ha la permutazione può essere di due tipi:

- la permutazione avviene in prossimità di una zona densa quindi assume connotati globali, questo porta ad avere uno stravolgimento dello stato

asintotico, richiede diverse iterazioni

- la permutazione ha effetto solo locale, servono poche iterazioni per tornare allo stato asintotico

La media di queste due situazioni ci dona il valore precedente.

7 Miglioramenti

Per scopi che esulano da questa breve relazione, potrebbe essere interessante svolgere altri tipi di prove come misurare il numero di correlazioni dovute ad una permutazione, studiare il comportamento di una determinata zona di cellule.
Mentre per una maggior chiarezza e precisione dei dati in gioco potrebbe essere funzionale aumentare le dimensioni della griglia e aumentare il numero di occorrenze su cui si effettua la media delle misure.