**Metodi Computazionali della Fisica**

# Pile di Sabbia

Marcello Melone

8 ottobre 2019

# Indice

# 1 introduzione

In questo progetto ci siamo occupati delle pile di sabbia abeliane, che rappresentano un semplice modello della dinamica delle pile di sabbia reali.

In questo modello, si assume che i granelli di sabbia vengano distribuiti a caso uno alla volta sui nodi di un grafo indiretto completo, come il reticolo mostrato in Figura 1.1.

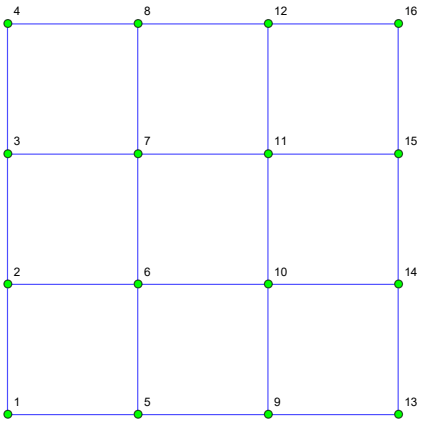



Figura 1.1: Reticolo quadrato di lato 4.

Ogni nodo è caratterizzato da un'altezza critica $h_c(i)$ che rappresenta il numero massimo di granelli $h(i)$ che possono accumularsi sull' $i$-esimo nodo prima che la pila si rovesci. Quando la pila si rovescia i granelli presenti sul sito vengono distribuiti sui siti vicini, uno per ogni vicino, e il processo continua fino a quando tutti i siti sono stabili. Questo processo prende il nome di "valanga", durante una valanga non vengono aggiunti granelli di sabbia, in questo modo si assume che la velocità di rilassamento del sistema sia molto maggiore della frequenza di aggiunta della sabbia. Affinché le valanghe non durino per un tempo infinito è necessario fare in modo che i granelli di sabbia escano dal reticolo, a questo scopo vengono definiti dei siti "di bordo" del grafo, in corrispondenza dei quali, quando viene superata l'altezza critica alcuni granelli vengono tolti dal sito senza essere distribuiti ai vicini.

Una importante caratteristica di questo modello risiede nel fatto che l'ordine con cui vengono aggiunti i granelli di sabbia non conta ai fini del raggiungimento della configurazione stabile: aggiungendo prima un granello nella posizione $i$ e poi uno nella posizione $j$ risulta nella stessa configurazione stabile che si otterrebbe aggiungendo i granelli in ordine inverso, da qui il nome "abeliano".

Per lo studio delle proprietà di questo modello vengono analizzate le proprietà delle valanghe originate dall'aggiunta di un singolo granello di sabbia, quali:

- Grandezza ($s$): numero totale di rovesciamenti in una valanga,
- Area ($a$): numero di rovesciamenti provenienti da siti distinti,
- Durata ($t$): Durata della valanga,
- Raggio ($r$): Massima distanza tra un sito rovesciato e l'origine della valanga.

La simulazione del sistema consiste nel partire da un grafo privo di sabbia e distribuire a uno a uno, su siti scelti a caso, $n$ granelli tenendo traccia dei parametri $\{s, a, t, r\}$ per ogni valanga, al fine di studiarne le proprietà statistiche.

# 2 Implementazione

Per ottenere il risultato descritto precedentemente si è sviluppato un programma che prenda in ingresso il grafo e il numero di punti $n$ e che restituisca una lista di valori $\{i, s, a, t, r\}$, dove $i$ rappresenta l'$i$-esimo granello che ha provocato la valanga.

Per rappresentare il grafo si è fatto uso della lista di adiacenza, nella quale l'$i$-esima entrata consiste nella lista dei nodi ai quali l'$i$-esimo è collegato. Per tenere traccia del numero di granelli di sabbia e dell'altezza critica, si sono inoltre affiancati alla lista dei vicini i numeri $v(i)$, $h(i)$, $h_c(i)$. Dove $v(i)$ rappresenta il numero di granelli che escono dal nodo $i$-esimo quando la pila si rovescia. Ad esempio il reticolo (privo di sabbia) in Figura 1.1 è rappresentato dalla seguente lista:

$$\begin{pmatrix} \{2,5\} & 4 & 0 & 4 \\ \{3,1,6\} & 4 & 0 & 4 \\ \{4,2,7\} & 4 & 0 & 4 \\ \{3,8\} & 4 & 0 & 4 \\ \{6,9,1\} & 4 & 0 & 4 \\ \{7,5,10,2\} & 4 & 0 & 4 \\ \{8,6,11,3\} & 4 & 0 & 4 \\ \{7,12,4\} & 4 & 0 & 4 \\ \{10,13,5\} & 4 & 0 & 4 \\ \{11,9,14,6\} & 4 & 0 & 4 \\ \{12,10,15,7\} & 4 & 0 & 4 \\ \{11,16,8\} & 4 & 0 & 4 \\ \{14,9\} & 4 & 0 & 4 \\ \{15,13,10\} & 4 & 0 & 4 \\ \{16,14,11\} & 4 & 0 & 4 \\ \{15,12\} & 4 & 0 & 4 \end{pmatrix}$$

## 2.1 Simulazione del sistema

Il programma con il quale vengono simulate le valanghe è composto da diverse funzioni.

### sim

```
sim[adl_, np_] := (
   If[adl === 0, Return[0]];
   tut = adl;
   ini = Hash[Map[tut[[#, 1]] &, Range[Length[tut]]]];
   Return[Reap[Do[metti[o, Random[Integer, {1, Length[tut]}]];, {o, 1, np}]][[2, 1]]]
  );
```

Questa funzione prende in ingresso la lista di adiacenza del reticolo ***adl*** e il numero di punti ***np*** da distribuire a caso sul reticolo e restituisce la lista dei valori $\{i, s, a, t, r\}$ ottenuti in seguito alla simulazione. Le variabili evidenziate in blu rappresentano variabili globali che verranno usate dalle altre funzioni atte alla simulazione della valanga. La variabile **ini** è usata per memorizzare le particolari caratteristiche del reticolo sul quale viene eseguita la simulazione (senza tenere conto del numero dei punti al bordo), e sarà data come argomento ad alcune funzioni coinvolte

nel calcolo del raggio della valanga, per fare in modo che in successivi utilizzi del codice, alcune quantità proprie della geometria di un particolare reticolo non vadano ricalcolate. Il comando **Reap** è usato (accoppiato al comando **Sow** più avanti) per radunare in una lista l'insieme dei risultati. L'utilizzo di questi comandi permette di non fare uso del comando **AppendTo** che risulta poco efficiente quando si debbano aggiungere molti valori ad una lista.

### metti

```
metti[o_, p_] := (
   tut[[p, 3]] ++;
   If[tut[[p, 3]] ≥ tut[[p, 4]], val[o, p];];
  );
```

Questa funzione prende in ingresso il numero di punti ***o*** che sono stati distribuiti fino al momento presente e il punto ***p*** sul reticolo. Questa funzione è responsabile dell' aggiunta di un granello nel sito ***p*** e, quando l'altezza critica vene superata, della chiamata di **val** mediante la quale viene simulata la vera e propria valanga.

### val

```
val[o_, p_] := Module[{cad = {p}, si, t = 0, bu},
   si = Flatten[
     Reap[
       Sow[p];
       While[Length[cad] =!= 0,
        Scan[distribuisci, cad];
        cad = Flatten[Reap[Scan[prepara, cad]][[2]]];
        cad = DeleteDuplicates[cad];
        t++;
        Sow[cad];
       ];
      ][[2]]];
   bu = DeleteDuplicates[si];
   Sow[{o, Length[si], Length[bu], t, raggio[bu]}];
  ];

in[p_] := tut[[p, 3]] ++;
prepara[p_] := Scan[controlla, tut[[p, 1]]];
controlla[p_] := If[tut[[p, 3]] ≥ tut[[p, 4]], Sow[p]];
distribuisci[p_] := (
   tut[[p, 3]] += -tut[[p, 2]];
   Scan[in, tut[[p, 1]]];
  );
```

Questa funzione simula la valanga originata a partire dal sito instabile ***p*** e restituisce la lista dei valori $\{i, s, a, t, r\}$. All' interno di questa funzione vengono calcolati i parametri $s, a, t$, mentre $r$ viene calcolato separatamente. Il calcolo dei parametri $s$ e $a$ viene effettuato a partire dalla lista **si** contenente l'elenco di tutti i siti che si sono rovesciati nel corso della valanga, $s$ rappresenta la lunghezza della lista, mentre $a$ la lunghezza della lista priva degli elementi duplicati[1]. Il calcolo di

[1] Si è usato **DeleteDuplicates** invece di **Union** perché il primo non ordina la lista permettendo di velocizzare il tempo di esecuzione.

$t$ viene effettuato assumendo che il rovesciamento di alcuni siti possa avvenire simultaneamente, in quanto i granelli fuoriescono contemporaneamente da un sito rovesciato. Si pone quindi uguale a 1 il tempo che i granelli impiegano a raggiungere il sito vicino a quello appena rovesciato, e si assume che il processo di rovesciamento di un sito sia istantaneo. La funzione **val** deve quindi eseguire la valanga per passi di tempo 1, il numero di tali passi rappresenta il valore di $t$. L'algoritmo funziona nel seguente modo:

1. Si parte dalla lista dei siti instabili **cad**, la quale inizialmente contiene solo il sito ***p***.
2. Mediante la funzione **distribuisci** i granelli vengono distribuiti dai siti instabili ai loro vicini.
3. Mediante la funzione **prepara** viene aggiornata la lista **cad** inserendovi i vicini dei siti appena rovesciati che risultano instabili. Notiamo come il nuovo elenco di siti instabili possa contenere degli elementi duplicati, in quanto più siti potrebbero avere riversato sabbia sullo stesso punto, il quale verrebbe contato più di una volta nell'applicazione della funzione **prepara**. Vengono quindi eliminati gli elementi duplicati presenti in **cad**.
4. I punti contenuti nella lista **cad** vengono aggiunti alla lista **si** che conterrà tutti i punti che si sono rovesciati nel corso della valanga, e il valore di **t** viene incrementato di uno. Notiamo come il punto di partenza della valanga risulti essere il primo elemento di **si**.
5. Se **cad** non è vuota significa che ci sono ancora dei punti instabili, quindi si riparte dal punto 1.
6. Quando **cad** è vuota la valanga è terminata.

## 2.2 Calcolo del raggio delle valanghe

Il raggio di una valanga è definito come la distanza tra l'origine della valanga e il punto "più lontano" che si sia rovesciato nel corso della valanga. La distanza tra due punti di un grafo è definita come il numero di archi presenti nel cammino minimo che collega i due punti. Ad esempio per il grafo in Figura 1.1 la distanza tra i punti 11 e 5 è 3. Per il calcolo del raggio si sono sviluppate due diverse funzioni, basate su diverse strategie, che presentano performance differenti.

Entrambi i metodi ricorrono all'uso di "circonferenze di raggio $r$", che rappresentano l'insieme di punti del grafo che hanno distanza $r$ da un punto centrale. Esponiamo quindi ora l'algoritmo usato per determinarle.

### circ

```
circ[has_, centro_, r_] := circ[has, centro, r] = cir[centro, r];

cir[centro_, r_] := Module[{core, pat},
   If[r === 0, Return[{centro}]];
   If[r === -1, Return[{0}]];
   core = Flatten[Reap[Scan[Sow[tut[[#, 1]]] &, circ[ini, centro, r - 1]]][[2]]];
   pat = Join[circ[ini, centro, r - 1], circ[ini, centro, r - 2]];
   Return[DeleteDuplicates[pat ~ Join ~ core] ~ Drop ~ Length[pat]]
  ];
```

La funzione **circ** calcola, mediante la funzione **cir**, la circonferenza di raggio ***r*** centrata in ***centro*** e la memorizza per utilizzi futuri, a questo scopo si è aggiunto il parametro in ingresso ***has*** per diversificare le circonferenze calcolate per reticoli di geometria diversa, risulta ora chiaro lo scopo della variabile **ini**. La funzione **cir** calcola la circonferenza nel seguente modo:

1. Se `r` è uguale a 0 o $-1$ la circonferenza contiene il solo punto centrale, `centro` oppure 0. Questa condizione rende possibile il funzionamento ricorsivo dell'algoritmo, è inoltre fondamentale che un punto non compaia in più di una circonferenza, per questa ragione si è inserito il punto 0 nella circonferenza di raggio $-1$.
2. Si salvano in `core` tutti i punti vicini a quelli appartenenti alla circonferenza di raggio `r - 1`[2], tra questi sono presenti, per costruzione, quelli appartenenti alla circonferenza di raggio `r`. Notiamo che un punto potrebbe essere stato contato più volte, gli elementi duplicati andranno quindi eliminati in modo che la circonferenza non contenga più volte lo stesso punto. Se ciò non avvenisse si avrebbe un drastico aumento, in funzione di `r`, del numero di punti presenti nelle circonferenze (moltissimi sarebbero duplicati), riducendo l'efficienza dell'algoritmo.
3. Si selezionano da `core` i soli punti appartenenti alla circonferenza di raggio `r` e si eliminano gli elementi duplicati. In `core` sono infatti presenti tutti i punti delle circonferenze di raggio `r - 2` e `r` oltre che alcuni (in generale non tutti) punti della circonferenza di raggio `r - 1`; per ottenere i punti della circonferenza di raggio `r` è quindi sufficiente eliminare da `core` tutti i punti, contenuti in `pat`, appartenenti alle circonferenze di raggio `r - 1` e `r - 2`. Per fare ciò si è sfruttato il fatto che in `pat` non sono presenti elementi duplicati, chiamando quindi `DeleteDuplicates` sull'unione di `pat` e `core` si ottiene una lista i cui primi elementi sono tutti quelli di `pat`, mentre i restanti sono quelli di `core` privi di duplicati e di quegli elementi contenuti in `pat`.

Dal momento che `DeleteDuplicates` ha complessità temporale lineare nella dimensione della lista in ingresso, è possibile determinare la complessità temporale del calcolo di tutte le circonferenze centrate in un punto, che è la stessa del calcolo della circonferenza di raggio massimo. Una semplice analisi rivela che la complessità è $\mathcal{O}(V + E)$, dove $V$ è il numero di nodi e $E$ è il numero di archi. Infatti al punto 2 il numero di punti presenti in `core` è pari al numero di archi che collegano i punti di una circonferenza di raggio `r - 1` ai loro vicini, mentre al punto 3 il numero di elementi in `pat` è pari al numero di punti presenti nelle due circonferenze di raggio `r - 1` e `r - 2`. La somma del numero totale di elementi presenti in `core` e `pat` nel corso di tutta l'esecuzione è quindi pari a $2V + 2E$[3], assumendo che le altre operazioni richiedano ad ogni passo un tempo di ordine non superiore a quello richiesto da `DeleteDuplicates` al punto 3 si arriva al risultato.

### 2.2.1 Primo metodo

Avendo a disposizione le circonferenze centrate in un punto il raggio può essere determinato cercando la circonferenza di raggio massimo centrata nel punto di partenza della valanga. Cioè il raggio della più grande circonferenza che abbia intersezione non vuota con l'insieme dei punti della valanga[4]. Questo procedimento è implementato nella funzione `raggio`.

[2] Per fare ciò si è usata, come anche al punto 3, la funzione `circ` in modo da trarre vantaggio dai calcoli già effettuati in precedenza.

[3] In realtà, siccome non viene calcolata la circonferenza di raggio $r_{max}+1$ (che sarebbe vuota), andrebbe sottratto da $2V+2E$ due volte il numero di punti nella circonferenza di raggio $r_{max}$, una volta il numero di punti presenti nella circonferenza di raggio $r_{max} - 1$ e il numero di archi che collegano i punti della circonferenza di raggio $r_{max}$ ai loro vicini; tuttavia nella maggioranza dei casi questi numeri sono trascurabili. Una eccezione è data dal grafo completo $K_n$ per il quale a $n^2 + n$ bisognerebbe sottrarre $n^2$.

[4] La certezza che questo metodo sia corretto deriva dal fatto che tutti i punti che sono stati coinvolti nella valanga formano un insieme "connesso".

**raggio**

```
raggio[val_] := Module[{r = 0, pat, centro = val[[1]]},
   pat = Alternatives @@ circ[ini, centro, r];
   While[MemberQ[val, pat],
    r++;
    pat = Alternatives @@ circ[ini, centro, r];
   ];
   Return[r - 1]
  ];
```

Osserviamo che con questo metodo non viene esplicitamente misurata la distanza tra due punti del grafo, informazione che è implicitamente contenuta nelle circonferenze. Notiamo inoltre che l'informazione sulla distanza tra punti presente nelle circonferenze non viene "riciclata' completamente nel calcolo del raggio di valanghe diverse: infatti quando si calcola il raggio di una nuova valanga con un centro diverso dalle precedenti le nuove circonferenze che vengono calcolate potrebbero contenere parte dell'informazione relativa alla distanza tra due punti già presente in altre circonferenze con un centro diverso, le stesse informazioni vengono quindi ricalcolate più volte. Le considerazioni precedenti si possono sommarizzare osservando che, dal momento che le circonferenze vengono memorizzate, l'efficienza del calcolo del raggio diviene massima quando si siano calcolate *tutte* le circonferenze, di *ogni* possibile raggio, centrate in *tutti* i punti del reticolo. In altre parole il programma fa uso dei calcoli precedenti soltanto quando il centro di una valanga è lo stesso di una precedente valanga il cui raggio sia già stato calcolato.

### 2.2.2 Secondo metodo

Il secondo metodo per il calcolo del raggio ha il pregio di sprecare meno informazione rispetto al metodo precedente, e, come mostrato successivamente, risulta essere più efficiente. In questo metodo il raggio viene calcolato a partire dalla distanza tra due punti.

Dato un grafo di $n$ punti la distanza tra ogni punto può essere rappresentata da una matrice $n \times n$ (simmetrica) $M$ tale per cui $[M]_{ij} = d(i, j)$, dove $d$ indica la distanza. Di seguito è mostrata la matrice $M$ per il reticolo in Figura 1.1. Osserviamo come la matrice sia simmetrica rispetto a entrambe le diagonali, questo è un caso peculiare dei reticoli ed è dovuto alla loro invarianza per traslazioni, in generale $M$ è soltanto simmetrica. Per determinare $M$ ne calcoleremo quindi soltanto la parte triangolare superiore.

$$M = \begin{pmatrix}
0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 2 & 3 & 4 & 5 & 3 & 4 & 5 & 6 \\
1 & 0 & 1 & 2 & 2 & 1 & 2 & 3 & 3 & 2 & 3 & 4 & 4 & 3 & 4 & 5 \\
2 & 1 & 0 & 1 & 3 & 2 & 1 & 2 & 4 & 3 & 2 & 3 & 5 & 4 & 3 & 4 \\
3 & 2 & 1 & 0 & 4 & 3 & 2 & 1 & 5 & 4 & 3 & 2 & 6 & 5 & 4 & 3 \\
1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 2 & 3 & 4 & 5 \\
2 & 1 & 2 & 3 & 1 & 0 & 1 & 2 & 2 & 1 & 2 & 3 & 3 & 2 & 3 & 4 \\
3 & 2 & 1 & 2 & 2 & 1 & 0 & 1 & 3 & 2 & 1 & 2 & 4 & 3 & 2 & 3 \\
4 & 3 & 2 & 1 & 3 & 2 & 1 & 0 & 4 & 3 & 2 & 1 & 5 & 4 & 3 & 2 \\
2 & 3 & 4 & 5 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 & 1 & 2 & 3 & 4 \\
3 & 2 & 3 & 4 & 2 & 1 & 2 & 3 & 1 & 0 & 1 & 2 & 2 & 1 & 2 & 3 \\
4 & 3 & 2 & 3 & 3 & 2 & 1 & 2 & 2 & 1 & 0 & 1 & 3 & 2 & 1 & 2 \\
5 & 4 & 3 & 2 & 4 & 3 & 2 & 1 & 3 & 2 & 1 & 0 & 4 & 3 & 2 & 1 \\
3 & 4 & 5 & 6 & 2 & 3 & 4 & 5 & 1 & 2 & 3 & 4 & 0 & 1 & 2 & 3 \\
4 & 3 & 4 & 5 & 3 & 2 & 3 & 4 & 2 & 1 & 2 & 3 & 1 & 0 & 1 & 2 \\
5 & 4 & 3 & 4 & 4 & 3 & 2 & 3 & 3 & 2 & 1 & 2 & 2 & 1 & 0 & 1 \\
6 & 5 & 4 & 3 & 5 & 4 & 3 & 2 & 4 & 3 & 2 & 1 & 3 & 2 & 1 & 0
\end{pmatrix}$$

Anche in questo caso l'algoritmo è stato diviso in varie funzioni:

**riga**

```
riga[has_, p_] := riga[has, p] = rig[p];

rig[p_] := Module[{l = Length[tut], i = 0, r = 1, out, ci},
   out = ConstantArray[0, l - p];
   While[i < (l - p),
    ci = circ[ini, p, r];
    ci = Pick[ci, UnitStep[ci - p - 1], 1];
    Scan[(out[[#]] = r) &, ci - p];
    i += Length[ci];
    r++;
   ];
   Return[out]
  ];
```

Questa funzione calcola, mediante la funzione **rig**, e memorizza la *p*-esima riga della componente triangolare superiore della matrice $M$ senza includere gli elementi sulla diagonale principale che sono tutti nulli. La riga viene calcolata utilizzando **circ** nel modo seguente:

1. Si considera la circonferenza di raggio *r*, inizialmente posto uguale a 1, centrata in *p*.
2. Si eliminano i punti la cui distanza da *p* non sarà presente nella riga che si vuole calcolare, cioè tutti i punti minori di *p* + 1.
3. I siti di cui al punto 2 sono tutti a distanza *r* dal punto *p*, gli elementi della riga corrispondenti a questi punti vengono quindi posti uguali ad *r*
4. Il numero di tali punti viene sommato al valore (inizialmente 0) della variabile **i**, che tiene traccia del numero di elementi della riga finora calcolati.
5. Se il numero di punti inseriti **i** è minore del numero totale di elementi della riga, *r* viene incrementato di uno e si riparte da 1, in caso contrario il calcolo della riga è completo.

Notiamo che, nell' effettuare il calcolo della *p*-esima riga, non vengono necessariamente calcolate tutte le possibili circonferenze centrate in *p*.

### disz

```
disz[has_, a_, b_] := disz[has, a, b] = disz[has, b, a] =
    If[a === b, 0, riga[has, Min[a, b]][[Abs[a - b]]]];
```

Questa funzione estrae dalla **riga** appropriata della matrice $M$ la distanza tra due punti e la memorizza insieme alla distanza con gli argomenti scambiati, in quanto la distanza tra due punti è simmetrica per scambio degli argomenti. Osserviamo che quando viene richiesta la distanza di un punto da se stesso la funzione non comporta la determinazione della **riga** corrispondente in quanto la distanza è nulla.

### raggio1

```
raggio1[val_] := Module[{li},
   li = Map[(disz[ini, val[[1]], #]) &, val];
   Return[Max[li]]
  ];
```

Il raggio viene calcolato determinando il massimo della distanza tra il punto di partenza da tutti gli altri punti della valanga.

Questo secondo metodo, nonostante ricicli in maniera più efficiente l'informazione, ha il difetto di calcolare un intera riga della matrice $M$ anche quando è richiesto un solo punto della riga.[5] Questo metodo raggiunge la massima efficienza quando siano state calcolate tutte le righe (e memorizzate tutte le distanze) di $M$, questo accade *senza* che vengano calcolate necessariamente tutte le circonferenze possibili. Notiamo infatti che l'informazione sulla distanza tra due punti che si ottiene calcolando la circonferenza centrata in un punto $p$ può essere usata non solo nella misura del raggio della valanga originata in $p$, ma anche nella misura del raggio di quelle valanghe che contengono $p$ come uno (non il primo) dei punti rovesciati. Questo accorgimento è stato implementato quando si è calcolata solo la metà della matrice $M$ sfruttandone la simmetria.

Nonostante questo metodo faccia un uso più intelligente dell'informazione contenuta nelle circonferenze potremmo notare che i nodi eliminati al punto 2, che provengono dal calcolo di una circonferenza, vengano determinati "per niente" dal momento che vengono rimossi dalla circonferenza stessa oltre che contenere informazione già presente in altre righe di $M$; è quindi naturale chiedersi se sia possibile ovviare a questo inconveniente e rendere l'algoritmo ancora più efficiente. La risposta a questo interrogativo è negativa, infatti ogni qual volta bisogna determinare una circonferenza, dal momento che non si conosce l'ubicazione dei punti, bisogna controllare "in tutte le direzioni". Inoltre il fatto che le circonferenze contengano talvolta punti non richiesti al fine del calcolo di una riga risulta indispensabile per il corretto funzionamento della funzione **circ**, infatti se le circonferenze non contenessero alcuni punti, in base al funzionamento ricorsivo della funzione, i vicini di quei punti trascurati non verrebbero inseriti nella circonferenza successiva, ma non si ha la certezza che i punti vicini ad un punto "indesiderato" siano anch'essi superflui.

Assumendo che nel caso peggiore la funzione **rig** richieda il calcolo di tutte le circonferenze centrate su ogni punto del reticolo possiamo stimare la complessità temporale del calcolo della matrice $M$ ottenendo una complessità dell' ordine $\mathcal{O}(V^2 + VE)$.[6]

[5] Tentativi di ovviare a questo problema sono risultati leggermente meno efficienti del presente algoritmo.

[6] Come osservato nella nota 3 un'eccezione è data dal grafo completo $K_n$ per il quale la complessità è $\mathcal{O}(n^2)$ invece che $\mathcal{O}(n^3)$.

## 2.3 Generazione di grafi

Presentiamo ora alcuni metodi per generare la lista di adiacenza di diversi tipi di grafi sui quali eseguire le simulazioni.

### 2.3.1 Reticoli quadrati $d$-dimensionali

Un ostacolo che si presenta nella generazione di questi grafi è dato dal fatto che il "sistema di coordinate" con il quale identificare i punti del reticolo non sono le coordinate cartesiane. Infatti per strutturare la lista di adiacenza è necessario che i punti del grafo siano identificati con numeri interi (coordinate "unidimensionali"), che rappresentano una posizione nella lista di adiacenza. I reticoli sono tuttavia definiti in termini di coordinate cartesiane, mediante l'uso delle quali i collegamenti tra i vari punti risultano immediati. Determiniamo quindi una funzione biettiva $f$ che implementi un cambio di coordinate da quelle cartesiane a quelle unidimensionali, per comodità imponiamo che i punti del reticolo vengano identificati con un numero intero da 1 a $n^d$, dove $n$ è il numero (maggiore di uno) di punti presenti sul lato del reticolo. La $f$ deve quindi prendere in ingresso un punto $p$ del reticolo in coordinate cartesiane, espresso da una sequenza di numeri $p = (p_1, p_2, \dots, p_d)$ con $p_i \in \{1, 2, \dots, n\}$, e restituire un numero intero $m = f(p) \in \{1, 2, \dots, n^d\}$. Una possibile scelta di $f$ è data da:

$$f(p_1, p_2, \dots, p_d) = p_1 + \sum_{i=2}^{d} (p_i - 1) n^{i-1} = \frac{n - n^d}{n - 1} + \sum_{i=1}^{d} p_i n^{i-1} \tag{2.3.1}$$

Osserviamo subito come il minimo valore assunto da $f$ sia 1 ($p_i = 1$) e il massimo $n^d$ ($p_i = n$), come stabilito dalle richieste iniziali per $f$.

Dimostriamo ora che $f$ è iniettiva, a questo scopo determiniamo le coordinate dei punti $x$ e $y$ tali che $f(x) = f(y)$, e mostriamo che $x = y$. Si ha

$$f(x) = f(y) \Rightarrow \sum_{i=1}^{d} x_i n^{i-1} = \sum_{i=1}^{d} y_i n^{i-1} \tag{2.3.2}$$

l'equazione si risolve valutando le due quantità a destra e a sinistra dell'uguale modulo $n$, da cui si ottiene $x_1 = y_1$ in quanto $x_1$ e $y_1$ sono minori o uguali di $n$. Ripetendo il procedimento[7] si trova che $x = y$, quindi $f$ è iniettiva.

Dal fatto che la funzione è iniettiva, che la cardinalità del dominio è $n^d$ e che il codominio è contenuto in $\{1, 2, \dots, n^d\}$ si deduce che $f$ è anche suriettiva. Quindi $f$ è biettiva.

Possiamo ora determinare l'inversa di $f$ che, come mostrato più avanti, è fondamentale per identificare i punti sul bordo del reticolo. Dividendo $f(p)$ per $n^j$, con $j \in \{0, 1, \dots, d-1\}$, si ottiene:

$$\frac{f(p)}{n^j} = \begin{cases} p_1 + \sum_{i=2}^{d} (p_i - 1) n^{i-1} & j = 0 \\ \frac{p_1}{n} + p_2 - 1 + \sum_{i=3}^{d} (p_i - 1) n^{i-2} & j = 1 \\ \frac{p_1}{n^j} + \sum_{i=2}^{j} \frac{p_i - 1}{n^{j-i+1}} + p_{j+1} - 1 + \sum_{i=j+2}^{d} (p_i - 1) n^{i-1-j} & 2 \le j \le d-2 \\ \frac{p_1}{n^{d-1}} + \sum_{i=2}^{d-1} \frac{p_i - 1}{n^{d-i}} + p_d - 1 & j = d-1 \end{cases} \tag{2.3.3}$$

da cui[8]

$$\left\lceil \frac{f(p)}{n^j} \right\rceil = \begin{cases} p_{j+1} + \sum_{i=j+2}^{d} (p_i - 1) n^{i-1-j} & 0 \le j \le d-2 \\ p_d & j = d-1 \end{cases} \tag{2.3.4}$$

[7] Semplificando $x_i$ e $y_i$, dividendo per $n^i$ entrambi i membri e valutandoli modulo $n$ si ha $x_{i+1} = y_{i+1}$.

[8] Si trova facilmente che $0 < \frac{p_1}{n^j} + \sum_{i=2}^{j} \frac{p_i - 1}{n^{j-i+1}} \le \frac{n}{n^j} + \sum_{i=2}^{j} \frac{n-1}{n^{j-i+1}} = 1$.

Prendendo ora la (2.3.4) modulo $n$ si ottiene:

$$\left\lceil \frac{f(p)}{n^j} \right\rceil \mod n = p_{j+1} \mod n = \begin{cases} 0 & p_{j+1} = n \\ p_{j+1} & p_{j+1} \neq n \end{cases} \tag{2.3.5}$$

Ponendo quindi

$$f_{j+1}^{-1}(m) \coloneqq \begin{cases} n & \left\lceil \frac{m}{n^j} \right\rceil \mod n = 0 \\ \left\lceil \frac{m}{n^j} \right\rceil & \left\lceil \frac{m}{n^j} \right\rceil \mod n \neq 0 \end{cases} \tag{2.3.6}$$

si ottiene infine la funzione inversa:

$$f^{-1}(m) = \left(f_1^{-1}(m), f_2^{-1}(m), \ldots, f_d^{-1}(m)\right) \tag{2.3.7}$$

Avendo a disposizione $f$ e $f^{-1}$ è ora semplice, partendo dalle coordinate cartesiane, determinare i vicini di un punto $p$ in coordinate unidimensionali. In coordinate cartesiane i vicini di un punto $p = (p_1, p_2, \ldots, p_d)$ sono dati dai $2d$ punti $q$ che hanno la coordinata $i$-esima traslata di $\pm 1$: $q = (p_1, p_2, \ldots, p_i \pm 1, p_{i+1}, \ldots, p_d)$. In coordinate unidimensionali i vicini del punto $p$ sono quindi della forma $q = p \pm n^{i-1}$. Osserviamo però che quando $p_i = 0$ o $p_i = n$ non è possibile traslare la coordinata in una delle due direzioni, questi punti sono infatti punti di bordo del reticolo che hanno un minor numero di vicini e in coordinate unidimensionali possono essere identificati usando $f^{-1}$. Tutto ciò è facilmente implementabile nella funzione **vic** che determina tutti i vicini di un dato punto del reticolo espresso in coordinate unidimensionali. Usando **vic** si determina immediatamente la lista di adiacenza del reticolo $d$-dimensionale.

**ret**

```
ret[n_, d_] := Table[{vic[i, n, d], 2 * d, 0, 2 * d}, {i, 1, n^d}];
```

```
vic[p_, n_, d_] := Module[{lis},
   lis = Flatten[Reap[
       Do[
         If[Mod[Ceiling[p / (n^j)], n] =!= 0, Sow[p + n^j]];
         If[Mod[Ceiling[p / (n^j)], n] =!= 1, Sow[p - n^j]];
         , {j, 0, d - 1}
        ];
      ][[2]]];
   Return[lis]
  ];
```

A partire da un reticolo $d$-dimensionale è possibile generare altri tipi di grafi ad esso imparentati, un esempio è dato dal "reticolo $t$-ripiegato" ottenuto collegando tra loro $t$ lati opposti del reticolo. Per esempio il reticolo quadrato 1-ripiegato può essere immaginato come la superficie di un cilindro, mentre quello 2-ripiegato come la superficie di un toro. Per determinare questo grafo è necessario identificare i punti presenti su lati opposti del reticolo, osserviamo come in coordinate cartesiane i punti sul lato "sinistro" del reticolo sono dati da tutti i punti che hanno la coordinata $i$-esima (fissata) uguale a 1, quelli sul lato "destro" avranno quindi coordinata $i$-esima uguale a $n$. Per identificare tali punti a partire dalle coordinate unidimensionali si ricorre quindi a $f^{-1}$. Questo procedimento è stato implementato nella funzione **tret**.

**tret**

```
tret[n_, d_, t_, est_: 1] := Module[{ou = ret[n, d], st = Range[1, n^d], sx, dx, r},
   If[t > d, Return[0]];
   If[n === 2, Return[ou]];
   Do[
    sx = Cases[st, a_Integer /; Mod[Ceiling[a / (n^j)], n] === 1];
    dx = Cases[st, a_Integer /; Mod[Ceiling[a / (n^j)], n] === 0];
    Scan[
      (
       AppendTo[ou[[sx[[#]], 1]], dx[[#]]];
       AppendTo[ou[[dx[[#]], 1]], sx[[#]]];
      ) &
     , Range[1, Length[sx]]];
    , {j, 0, t - 1}];
   Do[
    r = Random[Integer, {1, Length[ou]}];
    ou[[r, 2]]++;
    ou[[r, 4]]++;
    , {est}];
   Return[ou]
  ];
```

Questa funzione prende in ingresso i parametri lato, dimensione e $t$ del grafo e restituisce la lista di adiacenza del reticolo $t$-ripiegato. Osserviamo che quando $t$ è uguale a $d$ il numero di vicini di ciascun nodo è uguale a $2d$, in questo caso è necessario aggiungere dei punti al bordo, il numero di tali punti è infatti l'ultimo parametro preso in ingresso dalla funzione; per semplicità si è deciso di scegliere a caso la posizione di questi punti.

### 2.3.2 Grafo random di Erdős–Rényi

I grafi random sono tipologie di grafi le cui caratteristiche sono determinate dall'esito di processi aleatori, a questa famiglia appartengono i grafi random di Erdős–Rényi. Un grafo di Erdős–Rényi viene ottenuto scegliendo a caso tra tutti i grafi che presentano $V$ vertici ed $E$ archi, questi grafi sono quindi identificati dai parametri $V$ ed $E$. Nel nostro caso i grafi generati devono essere connessi, per questa ragione si è generato il grafo a partire da un reticolo unidimensionale con $V$ vertici aggiungendo a questo $E - V + 1$ archi ($V - 1$ sono gli archi già presenti nel reticolo unidimensionale di partenza) in modo casuale. Notiamo che fissato $V$ il numero di archi $E$ non può essere scelto arbitrariamente alto in quanto ogni nodo può essere collegato al massimo con $V - 1$ altri nodi; il massimo numero di archi risulta pertanto essere $\frac{V(V-1)}{2}$[9]. Nel nostro caso il massimo numero di archi sarà $\frac{(V-1)(V-2)}{2}$. Il grafo random di Erdős–Rényi è generato mediante la funzione **err**.

[9] Questo risultato si ottiene notando che il massimo numero di archi si ha quando la matrice di adiacenza (che è simmetrica) presenti il numero 1 in tutta la parte triangolare superiore.

```
err
```

```
err[V_, E_, est_: 1] := Module[{r, c, lis},
   If[E > (V - 2) * (V - 1) / 2, Return[0]];
   lis = ret[V, 1];
   lis[[1, 2]] = lis[[1, 4]] = lis[[V, 4]] = lis[[V, 2]] = 1;
   Do[
    r = Random[Integer, {1, V}];
    c = Random[Integer, {1, V}];
    While[Length[lis[[r, 1]]] ≥ (V - 1), r = Random[Integer, {1, V}];];
    While[MemberQ[lis[[r, 1]], c] || r === c, c = Random[Integer, {1, V}];];
    lis[[r, 1]] = Append[lis[[r, 1]], c];
    lis[[r, 2]]++;
    lis[[r, 4]]++;
    lis[[c, 4]]++;
    lis[[c, 2]]++;
    lis[[c, 1]] = Append[lis[[c, 1]], r];
    , {E}];
   Do[
    r = Random[Integer, {1, V}];
    lis[[r, 2]]++;
    lis[[r, 4]]++;
    , {est}];
   Return[lis]
  ];
```

Questa funzione prende in ingresso il numero di nodi, il numero di archi e il numero di punti al bordo e restituisce la lista di adiacenza del grafo. Per stabilire a quali nodi collegare un nuovo arco si sceglie a caso un punto del reticolo che sia collegato a meno di $V-1$ altri punti e a questo si collega uno dei nodi (scelto a caso) che non sia un vicino del punto appena scelto.

### 2.3.3 Grafo random ad albero

Un grafo ad albero è un grafo connesso che presenti $V$ nodi e $V-1$ archi. Il grafo ad albero random è stato generato mediante la funzione **alr**.

**alr**

```
alr[n_, m_, est_: 1] := Module[{adl = {{{}, 0, 0, 0}}, dove, r},
   Do[
    dove = Random[Integer, {1, Length[adl]}];
    adl = appenditanti[adl, dove, Random[Integer, {1, m}]];
    , {n}];
   Do[
    r = Random[Integer, {1, Length[adl]}];
    adl[[r, 2]] ++;
    adl[[r, 4]] ++;
    , {est}];
   Return[adl]
  ];

appendi[adl_, pu_] := Module[{gr = {{pu}, 1, 0, 1}, ad = adl, l},
   l = Length[ad] + 1;
   AppendTo[ad[[pu, 1]], l];
   ad[[pu, 2]] ++;
   ad[[pu, 4]] ++;
   Return[Join[ad, {gr}]]
  ];

appenditanti[adl_, pu_, quanti_] := Module[{ad = adl},
   Do[ad = appendi[ad, pu];, {quanti}];
   Return[ad]
  ];
```

Questa funzione genera il grafo collegando per *n* volte, a nodi generati a caso, un numero casuale tra 1 e *m* di nuovi punti. I punti al bordo vengono, come per gli altri grafi, distribuiti in posizioni casuali.

### 2.3.4 Grafo random di Barabási–Albert

Questo tipo di grafo è stato ideato con lo scopo di modellizzare sistemi quali internet, social network, world wide web e altri. Tutti questi siatemi infatti presentano delle caratteristiche comuni: vi sono pochi nodi che sono collegati ad un numero considerevolmente alto ad altri nodi. Il grafo random di Barabási–Albert presenta queste caratteristiche ed è stato implementato nella funzione **baa**.

**baa**

```
baa[in_, t_, m_, es_: 1] := Module[{adl = in, lis, r},
   If[Length[in] < m, Return[0]];
   Do[
    lis = dove[adl, m];
    adl = Join[adl, {{{}, m, 0, m}}];
    Scan[(adl = connetti[adl, #, Length[adl]]) &, lis];
    , {t}];
   Do[
    r = Random[Integer, {1, Length[adl]}];
    adl[[r, 2]]++;
    adl[[r, 4]]++;
    , {es}];
   Return[adl]
  ];

connetti[adl_, uno_, altro_] := Module[{ad = adl},
   AppendTo[ad[[uno, 1]], altro];
   ad[[uno, 2]]++;
   ad[[uno, 4]]++;
   AppendTo[ad[[altro, 1]], uno];
   Return[ad]
  ];

dove[adl_, m_] := Module[{lis, usc = {}, p},
   lis = Flatten[Map[adl[[#1, 1]] &, Range[Length[adl]]]];
   Do[
    p = RandomChoice[lis];
    While[MemberQ[usc, p], p = RandomChoice[lis];];
    AppendTo[usc, p];
    , {m}];
   Return[usc]
  ];
```

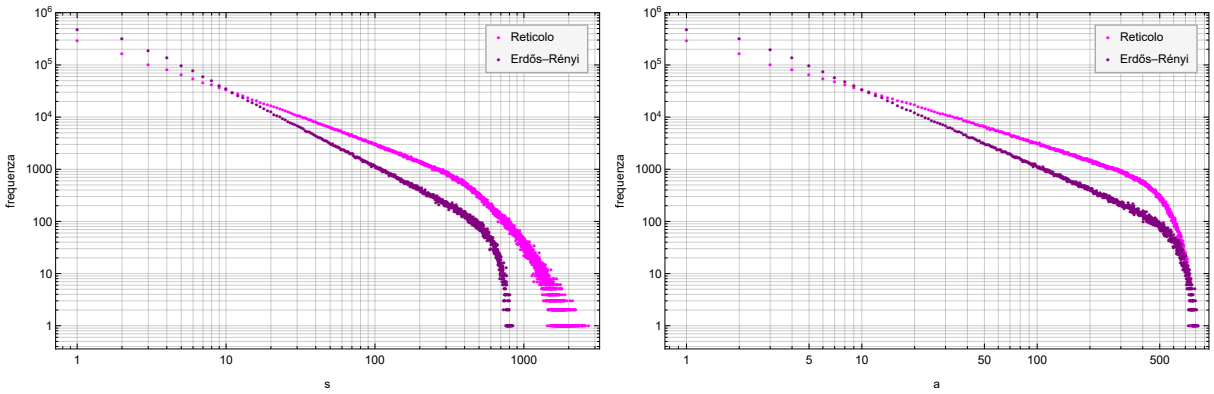
Il grafo random di Barabási–Albert viene generato a partire da un grafo di partenza, ***in***, al quale vengano progressivamente aggiunti **t** nuovi nodi i quali vengono collegati ad ***m*** dei nodi già presenti nel reticolo. La caratteristica di questo grafo è data dal fatto che un nuovo nodo viene collegato al nodo $i$-esimo con probabilità $p_i = \frac{k_i}{\sum_j k_j}$, dove $k_i$ indica il grado del nodo $i$-esimo. In questo modo si ottiene la proprietà descritta in precedenza secondo la quale nel grafo vi sono un numero ristretto di nodi che detengono la maggioranza dei collegamenti. La funzione **dove** é responsabile della selezione dei nodi ai quali verranno collegati i nuovi punti del grafo. Essa opera selezionando a caso ***m*** elementi distinti da **lis**, la quale contiene tutti gli elementi presenti nella lista di adiacenza del grafo, il numero di occorrenze di un dato vertice è quindi pari al suo grado.

# 3 Analisi statistiche

É noto che in questo modello il parametri $\{s, a, t, r\}$ delle valanghe si distribuiscono in accordo con delle leggi di potenza, al fine di verificare il funzionamento dell'algoritmo analizziamo quindi la distribuzione statistica del parametri in corrispondenza di diversi tipi di grafi, i risultati sono stati esposti in scala log-log per meglio evidenziare distribuzioni che seguono una legge di potenza.

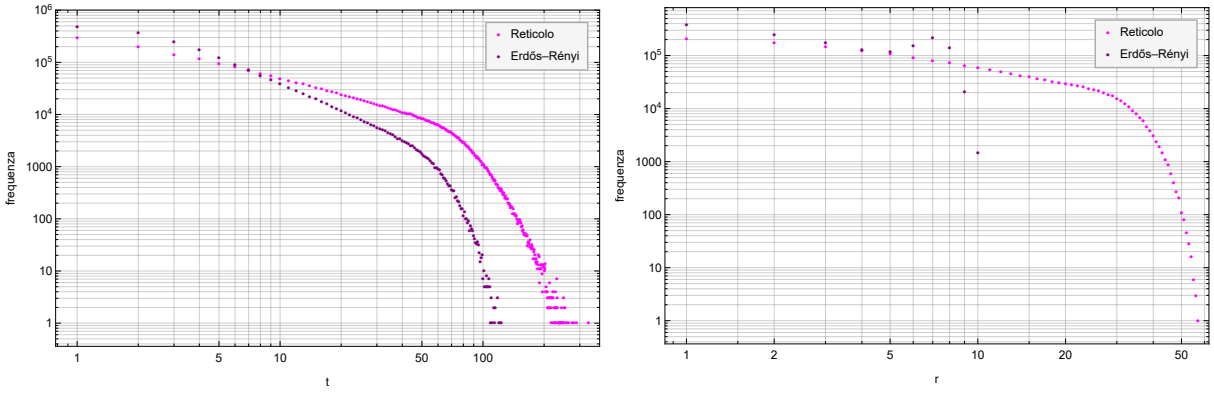
Dalla Figura 3.1 notiamo come la forma rettilinea dei grafici indichi una distribuzione che segue una legge di potenza. Il fatto che l'andamento si abbassi velocemente dopo una certa soglia è dovuto alla limitatezza dei grafi, le dimensioni delle valanghe sono quindi limitate dalla dimensione del grafo. Tutto ciò è molto chiaro in Figura 3.1b dove si nota che la massima area è pari al numero di punti del grafo.

In Figura 3.3 è analizzata la distribuzione dei parametri $s$ e $a$ per due tipi diversi di grafo ad albero con lo stesso numero di vertici, si nota che l'andamento non segue una legge di potenza nel caso del grafo ad albero random contrariamente al grafo di Barabási–Albert per il quale l'andamento risulta essere lineare per il tratto iniziale. Concludiamo che la distribuzione dei parametri non segue sempre una legge di potenza.



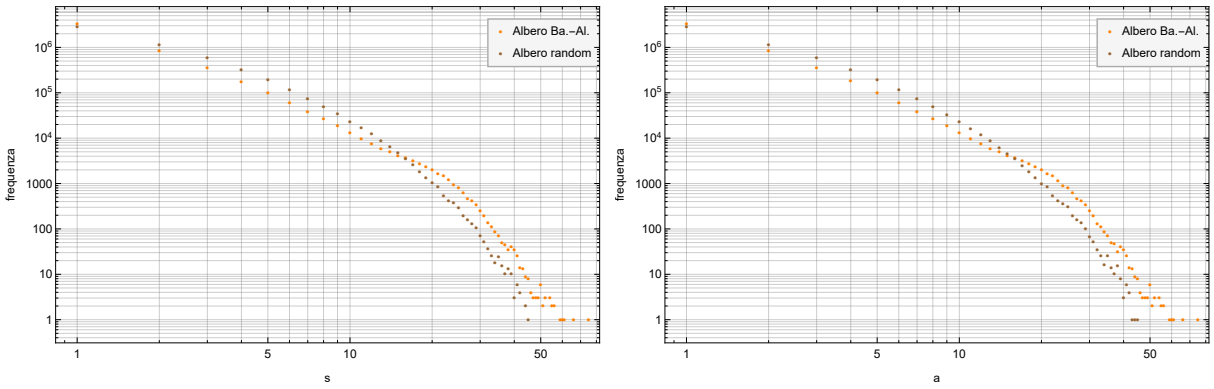

(a) Grandezza (b) Area

Figura 3.1: Il grafici mostrano in scala log–log la distribuzione del parametri grandezza ($s$) e area ($a$), il numero di punti distribuiti in entrambi i casi è $5 \times 10^6$. La simulazione è stata eseguita su un grafo random di Erdős–Rényi con 900 vertici, 1740 archi e 120 punti esterni e su un reticolo quadrato di lato 30.




(a) Tempo (b) Raggio

Figura 3.2: Il grafici mostrano in scala log–log la distribuzione del parametri tempo ($t$) e raggio ($r$), il numero di punti distribuiti in entrambi i casi è $5 \times 10^6$. La simulazione è stata eseguita su un grafo random di Erdős–Rényi con 900 vertici, 1740 archi e 120 punti esterni e su un reticolo quadrato di lato 30.




(a) Grandezza (b) Area

Figura 3.3: Il grafici mostrano in scala log–log la distribuzione del parametri grandezza ($s$) e area ($a$), il numero di punti distribuiti in entrambi i casi è $15 \times 10^6$. La simulazione è stata eseguita su un grafo random ad albero con 1001 vertici e su un grafo random di Barabási–Albert, generato a partire da un reticolo unidimensionale con due vertici, con 1001 vertici; in entrambi i casi i punti esterni sono stati posizionati in corrispondenza di vertici con grado 1 e 2.

# 4 Studio dell'efficienza

Studiamo ora il tempo di esecuzione della simulazione e confrontiamo le due strategie implementate per il calcolo del raggio delle valanghe. La simulazione verrà eseguita più volte aumentando progressivamente il numero di punti distribuiti sul grafo, al termine di ogni esecuzione la memoria delle funzioni **`circ`**, **`riga`** e **`disz`** verrà cancellata in quanto il programma è stato concepito per essere eseguito una sola volta lanciando sul grafo il massimo numero di punti richiesto. Al fine di garantire la ripetibilità dei risultati si è fissato un particolare seme random (1).

In Figura 4.1 sono mostrati i tempi di esecuzione simulando il sistema su un reticolo quadrato, osserviamo in primo luogo come il tempo di esecuzione risulti lineare nel numero di punti distribuiti sul grafo e come il secondo metodo per il calcolo del raggio risulti più efficiente del primo quando la quantità di punti immessi nel reticolo superi una carta soglia (nel caso specifico 30000 punti).

Il fatto che l'algoritmo abbia complessità temporale lineare nel numero di punti in ingresso è corroborato dai dati mostrati in Figura 4.2, qui la simulazione è stata eseguita (su più punti rispetto al caso precedente) su di un grafo random di Erdős–Rényi che presenta lo stesso numero di vertici, archi e punti esterni del reticolo utilizzato in precedenza, ancora una volta si nota il fatto che il secondo metodo sia più efficiente del primo dopo una certa soglia, soglia che differisce dal caso precedente (80000 punti).

Questa differenza tra i due metodi può essere compresa alla luce del loro funzionamento, infatti il primo metodo richiede inizialmente il calcolo di circonferenze di raggio piccolo, pari a quello delle prime valanghe che si verificano nel sistema. Il secondo metodo invece indipendentemente dal raggio della valanga calcola tutta una riga della matrice $M$ il che può richiedere l'uso di circonferenze di raggio molto grande. Tuttavia una volta che la matrice $M$ è stata calcolata le operazioni da eseguire per determinare il raggio sono meno dispendiose di quelle utilizzate dal primo metodo.

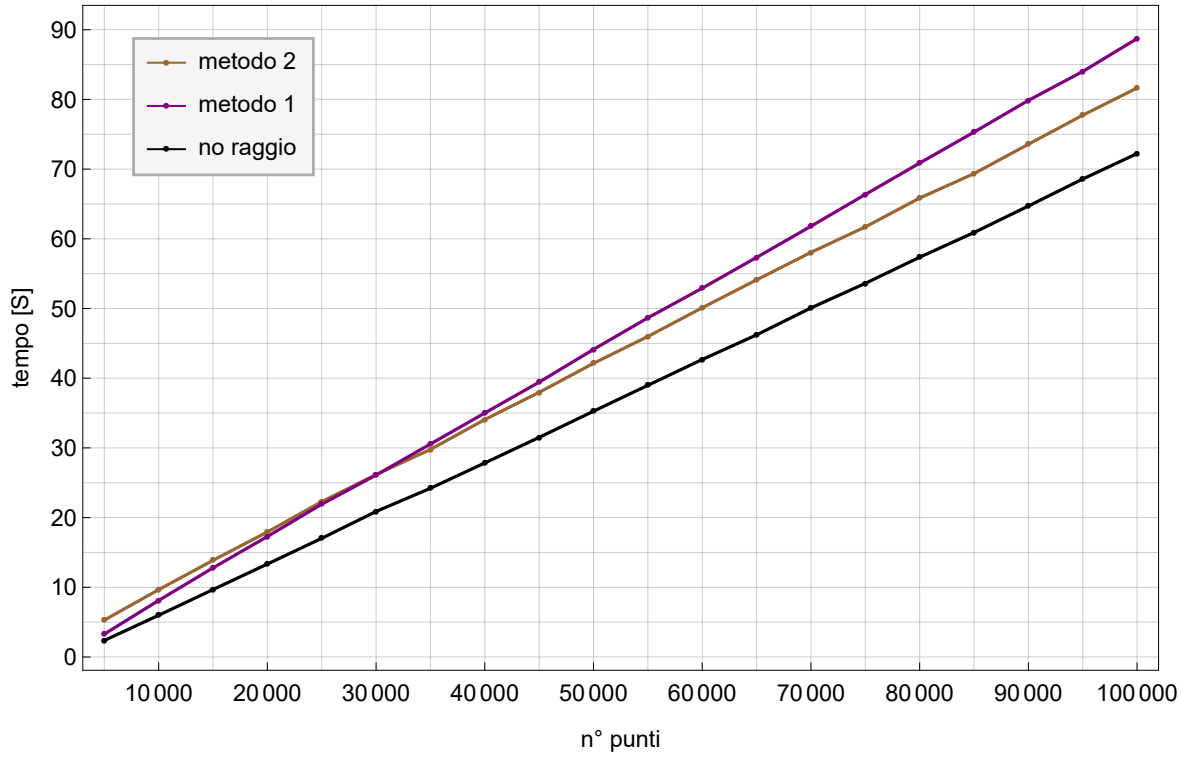



Figura 4.1: Il grafico confronta i tempi di esecuzione della simulazione in corrispondenza dell' utilizzo o meno delle due strategie per il calcolo del raggio delle valanghe. La simulazione è stata eseguita su un reticolo quadrato di lato 30.

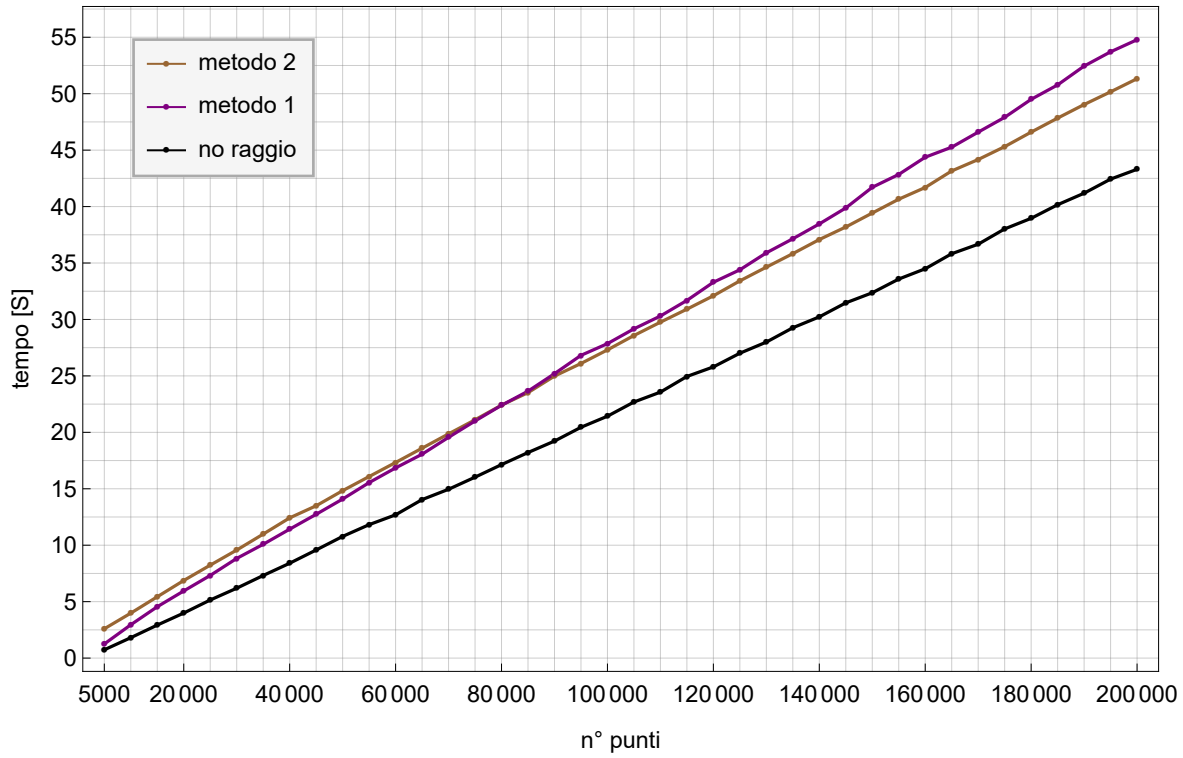



Figura 4.2: Il grafico confronta i tempi di esecuzione della simulazione in corrispondenza dell' utilizzo o meno delle due strategie per il calcolo del raggio delle valanghe. La simulazione è stata eseguita su un grafo random di Erdős–Rényi con 900 vertici, 1740 archi e 120 punti esterni.

Notiamo inoltre che il tempo di esecuzione della simulazione sul grafo di Erdős–Rényi è minore di quello richiesto sul reticolo, nonostante entrambi presentino le stesse caratteristiche in termini di numero di nodi, archi e punti esterni. Questo può essere spiegato osservando la Figura 4.3 in cui si nota come il numero di granelli che si rovesciano lungo tutta la simulazione è minore nel caso del grafo random.

Il fatto che l'andamento dei grafici in Figura 4.3 sia una retta suggerisce che il tempo di esecuzione sia proporzionale al numero di granelli rovesciato nel corso della simulazione. Questa ipotesi è avvalorata confrontando il rapporto tra i coefficienti angolari delle rette (ottenute mediante un fit lineare) delle Figure 4.1 e 4.2 con quello di quelle presenti nella Figura 4.3, in corrispondenza dell'uso del secondo metodo per il calcolo del raggio si ottengono i valori $2.35 \times 10^{-5}$ per il grafo random e $2.22 \times 10^{-5}$ per il reticolo.

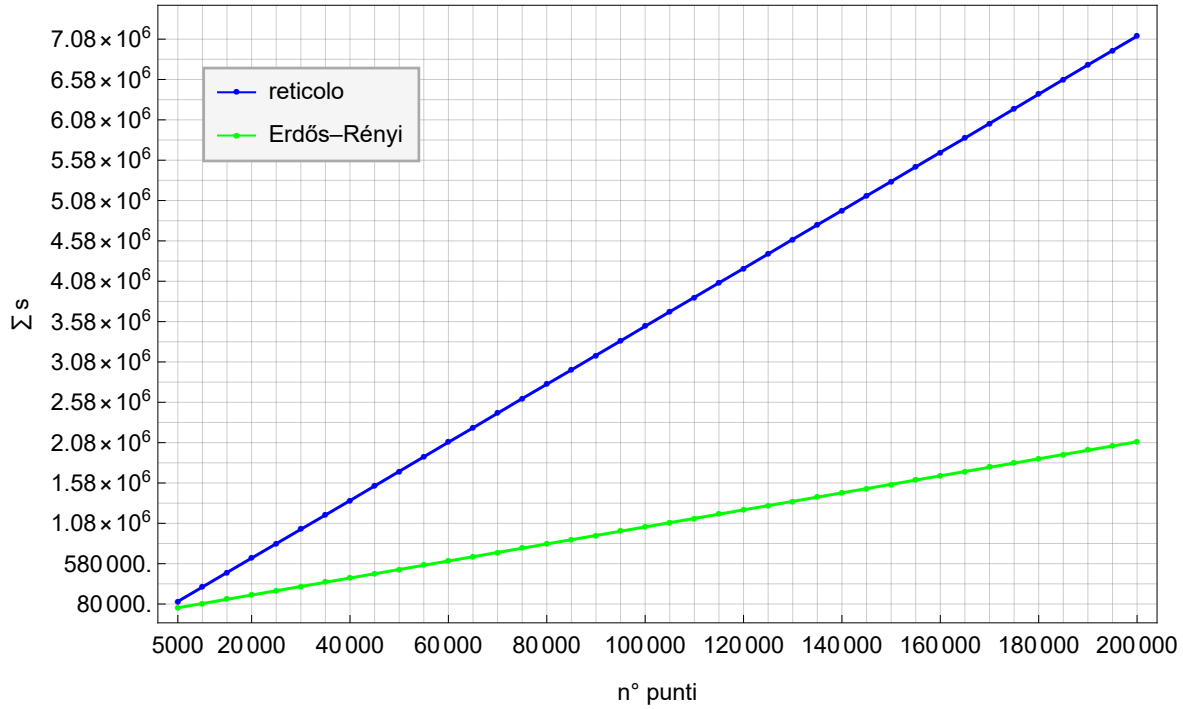



Figura 4.3: Il grafico mostra in numero totale di granelli rovesciati (parametro $s$) nel corso dell'intera simulazione in funzione del numero di punti distribuiti sul reticolo. I reticoli sui quali è stata eseguita la simulazione sono un reticolo quadrato di lato 30 e un grafo di Erdős–Rényi con 900 vertici, 1740 archi e 120 punti esterni.

# 5 Conclusioni

Per lo sviluppo di questo progetto si è usato il linguaggio "Wolfram Language", rispetto ad altri linguaggi, come ad esempio il C, permette di svolgere laboriose operazioni mediante un unico comando (si pensi ad esempio al comando `Cases`), questo ha permesso di rendere il codice più conciso.

Questo linguaggio presenta però degli aspetti negativi, essendo un linguaggio proprietario non permette la comprensione profonda dei comandi messi a disposizione degli utenti e questo complica non poco l'ottimizzazione del codice. In questo linguaggio risulta talvolta che il codice scritto in maniera più elegante e concisa non corrisponda ad un aumento delle prestazioni rispetto ad una implementazione meno sintetica.