Simulated Annealing applicato al TSP ed al modello di Edwards-Anderson

Francesco Caporali
supervisione del Professor Giacomo Di Gesù

11 maggio 2021

Abstract

La seguente relazione prende in analisi il noto metodo del *Simulated Annealing* per l'approssimazione del minimo globale di una data funzione $f(x)$ per mezzo di catene di Markov. L'obiettivo è quello di determinare probabilisticamente un minimo costruendo una catena con una prefissata legge invariante tale che *privilegi*, attribuendo maggiore peso, il minimo globale.
La vera potenza di questo metodo euristico, che giustifica il suo diffuso utilizzo, è data dalla possibilità di approssimare soluzioni di problemi anche molto complessi, tramite algoritmi semplici e diretti.
Il primo esempio che si andrà a trattare è il famoso problema *NP-Hard*: *il commesso viaggiatore*, mentre nell'ultima sezione si descriverà un possibile approccio al problema della ricerca di minimi globali della funzione energia del modello di *Edwards-Anderson* in dimensione 2.

1 Introduzione

Questa relazione ha lo scopo di esporre uno schema risolutivo per una classe di problemi generali strutturati nel seguente modo[1]:

$$\text{dato un insieme finito } E \text{ e una funzione } f : E \to \mathbb{R} \text{ trovare } x_0 \in E$$
$$\text{tale che } \forall x \in E \text{ sia } f(x_0) \leq f(x)$$

Se l'insieme E su cui è definita la nostra $f(x)$ è piccolo, il problema è di immediata soluzione[2], tuttavia se $\#E$ è un numero molto grande i metodi semplici diventano computazionalmente troppo pesanti e perciò il problema risulta intrattabile.
Un esempio classico di questo fenomeno è dato dal problema del *commesso viaggiatore* che seppur presentando una enunciazione semplice (come si vedrà nel seguito della relazione) e risultando triviale nei casi in cui il numero di città considerate è piccolo, è a tutti gli effetti un problema *NP-Hard*[3].
In maniera del tutto analoga si prende in analisi, nella parte successiva della relazione, un ulteriore esempio: il modello di Edwards-Anderson. Anche se in questo secondo caso la struttura dell'esercizio risulta più complessa, il problema da risolvere è paragonabile al precedente; infatti si cerca di trovare una configurazione che minimizzi una specifica funzione al fine di raggiungere uno stato in cui l'energia sia il più bassa possibile.
La domanda che ci si pone a questo punto è se sia possibile, in generale, trovare un minimo globale, o almeno una sua buona approssimazione in un tempo ragionevole.
Uno dei metodi più diffusi per la risoluzione di problemi quali quelli sopra citati è il *Simulated Annealing (SA).*

2 Richiami di teoria relativa alle catene di Markov

Le implementazioni numeriche e algoritmiche eseguite necessitano di alcune definizioni e teoremi che permettano di dare una giustificazione formale al lavoro svolto.
Si introduce di seguito la distanza con la quale sono enunciati i teoremi di convergenza ed unicità delle catene di Markov.

Definizione 2.1. [1, p. 29] Siano $\mu = (\mu_1, \ldots, \mu_n)$ e $\nu = (\nu_1, \ldots, \nu_n)$ probabilità sullo spazio $E = \{x_1, \ldots, x_n\}$, definiamo la *distanza della variazione totale* tra μ e ν come segue:

$$d_{TV}(\mu, \nu) = \frac{1}{2} \sum_{i=1}^{n} |\mu_i - \nu_i|$$

Con la seguente definizione è possibile enunciare il seguente risultato fondamentale.

[1] nel seguito verrà analizzato solo il problema nella forma di *minimizzazione* ma, in maniera del tutto analoga, quanto visto può essere applicato alla ricerca di un massimo globale considerando invece della funzione f, la sua opposta $-f$
[2] per esempio utilizzando algoritmi di tipo *brute force*
[3] *NP-Hard*[4]: non deterministico in tempo polinomiale

Teorema 2.1 (Teorema di convergenza delle catene di Markov [1, p. 34])**.** *Data* $(X_0, X_1, \dots)$ *una catena di Markov irriducibile e aperiodica su uno spazio di stati finito* $E = \{x_1, \dots, x_n\}$*, con matrice di transizione* P *e una distribuzione iniziale* μ_0*, allora per ogni distribuzione* π *stazionaria per* P*, si ha*

$$\mu_n \to \pi$$

secondo la d_{TV}*.*

Da questo teorema e dal *Teorema di esistenza di una distribuzione stazionaria*[1, p. 29] segue facilmente

Teorema 2.2 (Teorema di unicità della distribuzione stazionaria[1, p. 37])**.** *Data una catena di Markov irriducibile e aperiodica, allora ha esattamente una unica distribuzione stazionaria.*

Un modo per verificare in maniera diretta se una distribuzione è stazionaria per una catena con buone proprietà è introdurre la nozione di reversibilità.

Definizione 2.2 (Proprietà di bilancio dettagliato)**.** [1, p. 39] Sia $(X_0, X_1, \dots)$ una catena di Markov irriducibile e aperiodica su uno spazio di stati finito $E = \{x_1, \dots, x_n\}$, con matrice di transizione P. Una probabilità π su S si dice *reversibile* per la catena (o per la matrice di transizione) se $\forall i, j \in [n]$ si ha

$$\pi(x_i)P_{i,j} = \pi(x_j)P_{j,i}$$

Una catena di Markov si dice reversibile se esiste una distribuzione che risulti reversibile per la stessa. Una tale catena ha questo nome in quanto assume lo stesso comportamento se osservata mentre il tempo scorre linearmente o in senso opposto.
Per maggiore precisione si introduce la seguente nozione:

Inversione temporale. [1, p. 44] Sia $(X_0, X_1, \dots)$ una catena di Markov reversibile su uno spazio di stati finito E con matrice di transizione P e distribuzione reversibile π. Se la catena viene lanciata con distribuzione iniziale π, allora $\forall n \in \mathbb{N}$ e $\forall x_1, \dots, x_n \in E$ si ha:

$$\mathbb{P}(X_0 = x_0, X_1 = x_1, \dots, X_n = x_n) = \mathbb{P}(X_0 = x_n, X_1 = x_{n-1}, \dots, X_n = x_0)$$

In altre parole la catena ha uguale probabilità di visitare gli stati $\{x_0, \dots, x_n\}$ in ordine diretto o inverso.

Data questa nozione si può enunciare il seguente teorema:

Teorema 2.3. [1, p. 40] *Sia* $(X_0, X_1, \dots)$ *una catena di Markov irriducibile e aperiodica su uno spazio di stati finito* $E = \{x_1, \dots, x_n\}$*, con matrice di transizione* P*. Se* π *è una distribuzione reversibile per la catena, allora è anche una distribuzione stazionaria per la stessa.*

Dimostrazione. Si deve dimostrare che π è stazionaria per la catena ovvero:

1. $\forall i \in [n],\ \pi(x_i) \geq 0$ e $\sum_{i=1}^{n} \pi(x_i) = 1$
2. $\pi P = \pi$

La condizione 1 è ovvia essendo π una probabilità per ipotesi, mentre la condizione 2 va verificata.
Dire che π è un autovettore sinistro per la matrice di transizione P equivale a dire

$$\forall j \in [n],\ \pi(x_j) = \sum_{i=1}^{n} \pi(x_i) P_{i,j}$$

Tuttavia si ha, essendo π reversibile per P, che $\forall j \in [n]$ vale

$$\pi(x_j) = \pi(x_j)1 = \pi(x_j) \sum_{i=1}^{n} P_{j,i} = \sum_{i=1}^{n} \pi(x_j) P_{j,i} = \sum_{i=1}^{n} \pi(x_i) P_{i,j}$$

□

3 Simulated Annealing

L'idea alla base del *Simulated Annealing*[3] è di cercare di avvicinarsi progressivamente alla soluzione, senza computarla mai direttamente, sfruttando le proprietà delle catene di Markov.
Il nome di questo metodo deriva dal termine metallurgico *temprare (in inglese "anneal")*, una tecnica che prevede il riscaldamento ed il raffreddamento controllato di diversi materiali per migliorarne la cristallizzazione e ridurne le impurità.
Il principio di funzionamento di *SA* è lo stesso su cui si basa *Markov Chain Montecarlo (MCMC)*: si supponga di poter costruire una catena di Markov aperiodica $(X_0, X_1, \dots)$ con (unica) distribuzione stazionaria π; se si lancia tale catena con arbitraria distribuzione iniziale (per esempio partendo in uno stato iniziale fissato), per il Teorema 2.1, la distribuzione al tempo t converge a π per $t \to \infty$.
Questa rimane comunque una approssimazione, ma certamente se si aspetta per un tempo t sufficientemente lungo si avrà che la distribuzione della variabile X_t diventerà molto vicina a π.
Questo principio è sfruttato da *SA* in maniera diretta: si supponga di lanciare una catena di Markov sullo spazio degli stati E con una distribuzione stazionaria che concentra la maggior parte della probabilità sugli stati $x \in E$ tali che $f(x)$ è molto piccolo. Se la catena viene lanciata per una quantità sufficientemente lunga di tempo, è molto probabile che si finisca in uno stato x tra quelli appena definiti.
Se poi si passa ad una seconda catena con distribuzione stazionaria che concentra ancora più peso sugli stati x che minimizzano $f(x)$ allora sarà ancora più probabile che aspettando una quantità ragionevole di tempo ci si trovi in uno stato x che minimizza f. Procedendo iterativamente in tal modo è ragionevole sperare che la probabilità di trovarci in un minimo tende a 1 per $t \to \infty$.
Quanto descritto sul piano teorico funziona perfettamente anche perchè esistono molti modi canonici di costruire catene di Markov con distribuzione stazionaria la cosiddetta *distribuzione di Boltzmann*; disponendo di tali catene è facile, facendo tendere a 0 il parametro *temperatura*, ottenere una distribuzione limite che attribuisce tutto il peso al minimo globale.
Ciò si può vedere in modo semplice, infatti:

Definizione 3.1. [1, p. 100] La *distribuzione di Boltzmann* $\pi_{f,T}$ sullo spazio degli stati finito E, con energia $f : E \to \mathbb{R}$ e temperatura $T > 0$ è la distribuzione di probabilità su E che associa ad ogni $x \in E$ una probabilità

$$\pi_{f,T}(x) = \frac{1}{Z_{f,T}} e^{\frac{-f(x)}{T}}$$

dove

$$Z_{f,T} = \sum_{x \in E} e^{\frac{-f(x)}{T}}$$

è la costante di normalizzazione che assicura $\sum_{x \in E} \pi_{f,T}(x) = 1$

Vale dunque il seguente risultato:

Esercizio 3.1. [1, p. 101] *Sia E uno spazio degli stati finito, $f : E \to \mathbb{R}$ una arbitraria funzione energia. Sia data una temperatura $T : \mathbb{N} \to \mathbb{R}$ tale che $T \xrightarrow{t\to\infty} 0$. Data $\pi_{f,T}(x) := \pi(x)$ distribuzione di Boltzmann vale:*

$$\pi \xrightarrow{t\to\infty} \mu(x) = \begin{cases} \frac{1}{\#A} & \text{se } x \in A \\ 0 & \text{altrimenti} \end{cases}$$

dove $A = \{x \in E \mid \forall y \in E,\ f(x) \leq f(y)\}$

Dimostrazione. Sia $x_0 \in A$. $\forall x \in A$ vale $f(x) = f(x_0)$, allora:

$$\pi(x_0) = \frac{e^{\frac{-f(x_0)}{T}}}{\#A e^{\frac{-f(x_0)}{T}} + \sum\limits_{x\in E\setminus A} e^{\frac{-f(x)}{T}}} = \\ = \frac{1}{\#A + \sum\limits_{x\in E\setminus A} e^{\frac{f(x_0)-f(x)}{T}}}$$

dunque

$$\pi(x_0) \xrightarrow{t\to\infty} \frac{1}{\#A + \sum\limits_{x\in E\setminus A} 0} = \frac{1}{\#A}$$

infatti essendo

- $f(x_0) - f(x) < 0\ \forall x \in E$
- $T \xrightarrow{t\to\infty} 0$

segue

$$\frac{f(x_0) - f(x)}{T} \xrightarrow{t\to\infty} -\infty \implies e^{\frac{f(x_0)-f(x)}{T}} \xrightarrow{t\to\infty} 0$$

Sia x_1 tale che $\exists x \in E$ per cui $f(x_1) > f(x)$, per passaggio al complementare (si noti che $\sum\limits_{x\in A} \mu(x) = 1$, quindi essendo μ misura di probabilità $\forall y \in E \setminus A$ risulta $\mu(y) = 0$):

$$\pi(x_1) \xrightarrow{t\to\infty} 0$$

□

Detto ciò basta costruire una catena di Markov $(X_i)_{i\in\mathbb{N}}$ con distribuzione stazionaria $\pi_{f,T}$ di *Boltzmann* e fissare una funzione temperatura T tale che $T \xrightarrow{t\to\infty} 0$ (per esempio $T(t) = \frac{1}{t}$), per definire correttamente un algoritmo che implementi *SA*.
Ciò può essere fatto in molti modi, per esempio attraverso l'uso della catena definita da *Metropolis (M)*[2, p. 410] o da *Barker (B)*[2, p. 411], due tra le più utilizzate per i modelli

MCMC.
Allo scopo di introdurre le catene sopra citate si precisa che il modo più diretto per definire una catena di Markov è partire dalla sua matrice di transizione, poichè è quella che verrà utilizzata nell'implementazione algoritmica.
Prima di scendere nei dettagli della costruzione delle catene si veda lo schema algoritmico di *SA*, così da poter introdurre tutti gli strumenti necessari per definire *Metropolis* e *Barker*.
La prima definizione di cui si necessita è quella di *vicinato*:

Definizione 3.2. Dato E spazio di stati finito, definiamo $\forall x \in E$ un sottoinsieme $N(x) \subset E$ detto *vicinato di x in E*.

Definizione 3.3. Dato E spazio di stati finito, sia $\{N(x), x \in E\}$ una collezione di sottoinsiemi di E che soddisfano la condizione

$$x \notin N(x).$$

Una tale collezione è definita *struttura di vicinato*.
Se poi per ogni coppia di stati $x, y \in E$ esiste un *percorso* da x a y, ovvero una sequenza di stati $z_1 \dots z_m \in E$, tale che $z_1 \in N(x), z_2 \in N(z_1), \dots, z_m \in N(z_{m-1}), y \in N(z_m)$, allora la *struttura di vicinato* è detta *comunicante*.

Queste strutture possono essere costruite in diversi modi, il più semplice è di pensare E come un grafo in cui ogni *nodo* è collegato ai suoi vicini tramite un *arco*.
Detto ciò, quel che effettivamente viene fatto durante una *discesa algoritmica* di *SA* è riassunto nei seguenti passi:

1. si supponga di iniziare trovandosi in uno stato $x \in E$
2. alla prima iterazione viene esaminato un $y \in N(x)$, scelto secondo specifiche regole (nella trattazione che segue sarà estratto con probabilità uniforme)
3. viene poi confrontato il valore della funzione f in x e y:
   - se $f(x) \geq f(y) \implies y$ diventa l'elemento su cui si sposta l'attenzione, generalmente con una probabilità abbastanza alta, o comunque proporzionale al valore di $f(x) - f(y)$
   - se invece $f(x) < f(y) \implies$ si lascia comunque una piccola possibilità ad y di diventare l'elemento su cui spostare l'attenzione (in accordo con la distribuzione della catena scelta)

   Nel caso in cui non si sposta l'attenzione, al successivo passo viene scelto nuovamente un $y \in N(x)$, secondo le stesse regole precedenti, altrimenti si itera il procedimento con y che prende il posto di x.

Si possono ora introdurre le costruzioni formali di *Metropolis* e *Barker*.
Si inizia ribadendo che *Metropolis* e *Barker* hanno come obiettivo quello di costruire

catene con distribuzione stazionaria di Boltzmann, $\pi_{f,T}(x)$ con funzione energia f (nel *SA* è la funzione da minimizzare) e funzione temperatura T tale che $T \xrightarrow{t\to\infty} 0$.
Entrambi i modelli partono da una predefinita matrice irriducibile di transizione

$$Q = (q_{ij})_{i,j\in[n]\times[n]} \in \mathbb{R}^{n\times n}$$

che ha lo scopo di modellare la *struttura di vicinato*. Se la *struttura di vicinato* è *comunicante* allora Q è *irriducibile* (e lo sarà anche la catena che si andrà a definire[2, p. 397 ex. 11.5.2]).
Nell'implementazione Q modellizza distribuzioni omogenee:

$$Q_{i,j} = \begin{cases} \frac{1}{\#N(i)} & \text{se } j \in N(i) \\ 0 & \text{se } j \notin N(i) \end{cases}$$

Inoltre per ogni valore della funzione T viene definita una probabilità

$$\alpha_{i,j}(T(t)) \quad \forall t \in \mathbb{N}$$

Definiti tutti questi oggetti vale

$$P_{i,j} = \begin{cases} \frac{1}{\#N(i)}\alpha_{i,j}(T) & \text{se } j \in N(i) \\ 0 & \text{se } j \notin N(i) \text{ e } j \neq i \\ 1 - \sum\limits_{j\in N(i)} \frac{1}{\#N(i)}\alpha_{i,j}(T) & \text{se } j = i \end{cases}$$

Si precisa che se $f(x)$ è non costante allora questi metodi costruiscono catene aperiodiche[2, p. 397 ex. 11.5.2]. Ciò permette di affermare che sono soddisfatte le ipotesi del Teorema 2.1 e dunque è possibile usare tali catene per *SA* in quanto ammettono una unica distribuzione stazionaria.
Nel caso di *Metropolis* tale $\alpha_{i,j}(T)$ è

$$\alpha_{i,j}(T) = \min\{1, e^{\frac{f(i)-f(j)}{T}}\} = e^{-\frac{(f(j)-f(i))^+}{T}}$$

e allora

$$P_{i,j} = \begin{cases} \frac{1}{\#N(i)} e^{-\frac{(f(j)-f(i))^+}{T}} & \text{se } j \in N(i) \\ 0 & \text{se } j \notin N(i) \text{ e } j \neq i \\ 1 - \sum\limits_{j\in N(i)} \frac{1}{\#N(i)} e^{-\frac{(f(j)-f(i))^+}{T}} & \text{se } j = i \end{cases}$$

mentre nel caso di *Barker* è

$$\alpha_{i,j}(T) = \frac{1}{1 + e^{-\frac{f(i)-f(j)}{T}}}$$

ed analogamente si ha

$$P_{i,j} = \begin{cases} \frac{1}{\#N(i)} \frac{1}{1+e^{-\frac{f(i)-f(j)}{T}}} & \text{se } j \in N(i) \\ 0 & \text{se } j \notin N(i) \text{ e } j \neq i \\ 1 - \sum\limits_{j \in N(i)} \frac{1}{\#N(i)} \frac{1}{1+e^{-\frac{f(i)-f(j)}{T}}} & \text{se } j = i \end{cases}$$

Suppondendo di trovarsi nel caso Q simmetrica[4], per entrambi i metodi c'è convergenza alla distribuzione di *Boltzmann* $\pi_{f,T}$, indipendentemente dalla Q scelta.
Ciò segue dal fatto che sia *Metropolis* che *Barker* definiscono catene reversibili[5]. In particolare, proprio la distribuzione di *Boltzmann* è reversibile rispetto alle catene, dunque per il Teorema 2.3 è stazionaria per entrambe.

Proposizione 3.1. *La distribuzione di Boltzmann $\pi_{f,T}$ è reversibile rispetto alla catena definita da Metropolis.*

Dimostrazione. Si vuole verificare la seguente proprietà $\forall i, j \in [n]$

$$\pi_{f,T}(x_i) P_{i,j} = \pi_{f,T}(x_j) P_{j,i}$$

con

$$\pi_{f,T}(x) = \frac{1}{Z_{f,T}} e^{\frac{-f(x)}{T}} \quad , \quad Z_{f,T} = \sum_{x \in E} e^{\frac{-f(x)}{T}}$$

Si hanno 3 casi:

1. se $i = j$ è banalmente vero
2. se $i \neq j$ e $x_j \notin N(x_i)$ (vale anche $x_i \notin N(x_j)$ per ipotesi di simmetria dei *vicinati*) si ha
   $$\pi_{f,T}(x_i) 0 = \pi_{f,T}(x_j) 0 \iff 0 = 0$$
   che risulta banalmente vero
3. se $i \neq j$ e $x_j \in N(x_i)$ si ha
   $$P_{i,j} = \frac{1}{\#N(x_i)} \min\{1, e^{\frac{f(x_i)-f(x_j)}{T}}\}$$
   dunque
   $$\pi_{f,T}(x_i) P_{i,j} = \pi_{f,T}(x_j) P_{j,i}$$
   diventa
   $$\frac{e^{\frac{-f(x_i)}{T}}}{Z_{f,T}} \frac{\min\{1, e^{\frac{f(x_i)-f(x_j)}{T}}\}}{\#N(x_i)} = \frac{e^{\frac{-f(x_j)}{T}}}{Z_{f,T}} \frac{\min\{1, e^{\frac{f(x_j)-f(x_i)}{T}}\}}{\#N(x_j)}$$

[4] è il caso preso in analisi nelle implementazioni numeriche di questa relazione in quanto i *vicinati* sono simmetrici

[5] ovvero ammettono probabilità reversibili

vero se e solo se[6]

$$\frac{\min\{e^{\frac{-f(x_i)}{T}}, e^{\frac{-f(x_j)}{T}}\}}{\#N(x_i)Z_{f,T}} = \frac{\min\{e^{\frac{-f(x_j)}{T}}, e^{\frac{-f(x_i)}{T}}\}}{\#N(x_i)Z_{f,T}}$$

che è chiaramente un'identità.

□

Proposizione 3.2. *La distribuzione di Boltzmann $\pi_{f,T}$ è reversibile rispetto alla catena definita da Barker.*

Dimostrazione. La dimostrazione è analoga alla precedente. □

Indipendentemente dalla costruzione della catena, vanno comunque fatte delle considerazioni sulla funzione *temperatura* $T(t)$. Si è osservato che se il tempo t tende a ∞, la distribuzione della catena converge alla distribuzione di *Boltzmann* e contemporaneamente tende a 0 anche la temperatura. Questo secondo fatto, per l'Esercizio 3.1, permetterebbe di dire che la distribuzione stazionaria delle due catene converge puntualmente a μ, misura di probabilità su E tale che:

$$\mu(x) = \begin{cases} \frac{1}{\#A} & \text{se } x \in A \\ 0 & \text{altrimenti} \end{cases}$$

dove $A = \{x \in E \mid \forall y \in E,\ f(x) \leq f(y)\}$.
Va però notato che in realtà quanto detto non permette di concludere che, qualunque sia la funzione *temperatura*, la probabilità di trovarsi in un minimo globale di f converge ad 1 per $t \to \infty$.
Esistono teoremi che affermano che se la $T(t)$ converge a 0 in misura sufficientemente lenta allora accade quanto detto; tuttavia, sfortunatamente, usando tali funzioni la convergenza è, nella maggior parte dei casi, talmente lenta da non riuscire a raggiungere il minimo globale in tempo ragionevole.
Il primo risultato di convergenza di *SA* è stato dimostrato dai fratelli Geman nel 1984 e viene riportato di seguito:

Teorema 3.1 (Geman & Geman[8])**.** *Supponendo di trovarci nelle ipotesi fatte precedentemente per SA, se la temperatura $T(t)$ al tempo t, converge a 0 abbastanza lentamente da rispettare la relazione*

$$T(t) = \frac{k\left(\max_{x\in E} f(x) - \min_{x\in E} f(x)\right)}{\log t}$$

$\forall$ t sufficientemente grande, allora la probabilità di trovarsi in un minimo di f al tempo t converge ad 1 per $t \to \infty$.

[6] se $x_i \in N(x_j) \implies \#N(x_i) = \#N(x_j)$ poichè i *vicinati* sono simmetrici e la matrice Q è simmetrica

Per tale motivo nelle implementazioni reali di *SA* si rischia di avere *raffreddamenti* troppo rapidi che causano, in alcune circostanze, il fallimento dell'algoritmo, dando come risultato minimi locali, non globali.
Di seguito un esempio di *SA* fallimentare[1, p. 116].

***SA* fallimentare.** Sia $E = \{x_1, x_2, x_3, x_4\}$ e $f : E \to \mathbb{R}$ così definita:

$$f(x) = \begin{cases} 1 & \text{se } x = x_1 \\ 2 & \text{se } x = x_2 \\ 0 & \text{se } x = x_3 \\ 2 & \text{se } x = x_4 \end{cases}$$

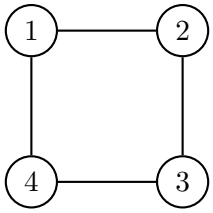

Figura 1: grafo associato ad insieme di stati E

Sia E munito della struttura di grafo in Figura 1 e si applichi *SA* con la catena definita da *Metropolis* .
Definita una generica funzione temperatura $T(t)$ si ha che al tempo t la matrice di transizione della catena è la seguente[7]

$$\begin{bmatrix} 1 - e^{-\frac{1}{T(t)}} & \frac{1}{2}e^{-\frac{1}{T(t)}} & 0 & \frac{1}{2}e^{-\frac{1}{T(t)}} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2}e^{-\frac{2}{T(t)}} & 1 - e^{-\frac{2}{T(t)}} & \frac{1}{2}e^{-\frac{2}{T(t)}} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \end{bmatrix}$$

Sia ora A l'evento $A = \{$*la catena rimane per sempre nello stato* $x_1\}$ (in particolare se ci si trova in A non verrà mai trovato il minimo di f ovvero x_3).

[7] si ricava direttamente dalla struttura di grafo di E e dalle regole di *Metropolis*

Risulta che

$$\begin{aligned}\mathbb{P}(A) &= \mathbb{P}(X_1 = x_1,\ X_2 = x_1, \dots) = \\ &= \lim_{n\to\infty} \mathbb{P}(X_1 = x_1, \dots, X_n = x_1) = \\ &= \lim_{n\to\infty} \mathbb{P}(X_1 = x_1 | X_0 = x_1) \cdots \mathbb{P}(X_n = x_1 | X_{n-1} = x_1) = \\ &= \lim_{n\to\infty} \prod_{i=1}^{n} \left(1 - e^{-\frac{1}{T(i)}}\right) = \\ &= \prod_{i=1}^{\infty} \left(1 - e^{-\frac{1}{T(i)}}\right)\end{aligned}$$

ma $\prod_{i=1}^{\infty} \left(1 - e^{-\frac{1}{T(i)}}\right) = 0 \iff \sum_{i=1}^{\infty} \left(e^{-\frac{1}{T(i)}}\right) = \infty$, quindi se $T(t)$ converge a 0 abbastanza rapidamente da far convergere la serie $\sum_{i=1}^{\infty} \left(e^{-\frac{1}{T(i)}}\right)$ (per esempio se $T(t) = \frac{1}{t}$), allora $\mathbb{P}(A) > 0$, e di conseguenza la catena potrebbe rimanere bloccata in x_1 per sempre.
Sono due i fattori che danno luogo a tale risultato:

1. la funzione $T(t)$ converge a 0 troppo rapidamente;
2. è presente un minimo locale in cui il *SA* potrebbe rimanere *incastrato*;

Per tale motivo si cerca di ottenere è un bilanciamento tra buona velocità di convergenza e buona probabilità di ottenere un minimo globale, obiettivo non semplicemente raggiungibile.

4 Commesso viaggiatore

Dopo aver introdotto formalmente il *Simulated Annealing* si espone, in questa sezione, un'implementazione pratica di quanto visto sotto forma di algoritmo euristico.
Il problema che si andrà ad analizzare è il celebre *Problema del commesso viaggiatore (TSP dall'inglese "Travelling Salesman Problem")*[5], che si può formulare nel seguente modo:

> "Dato un insieme di n città, e note le distanze tra ciascuna coppia di esse, trovare il tragitto di minima percorrenza che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta e ritornare alla città di partenza."

È possibile riscrivere tale problema nella forma di minimizzazione di una funzione, in maniera analoga a quanto fatto nell'introduzione.
Anzitutto si prenderà come spazio degli eventi E l'insieme delle possibili strade percorribili caratterizzato come il gruppo delle permutazioni di n elementi: S_n. Successivamente si potrà scegliere la f come la funzione che associa ad ogni percorso la sua lunghezza[8], dunque il minimo tragitto sarà proprio il minimo di f su tutti i possibili percorsi.
Detto ciò per applicare il metodo *SA* si ha ancora bisogno di definire una *struttura di vicinato* che sia anche simmetrica e comunicante così da poter applicare la teoria esposta nella sezione dedicata.
Questo obiettivo può essere raggiunto attraverso la definizione delle cosiddette *2-change*.

Definizione 4.1 (2-change)**.** Se le n città sono ordinate, un percorso x può essere identificato come una permutazione $\sigma \in S_n$ dove $\sigma(\alpha)$ è l'ordine in cui la città α è stata visitata. Si supponga ora, senza perdita di generalità[9], che dato un generico percorso p questo sia associato all'identità ($\sigma_p = id$), si definisce una *2-change* di p che coinvolge le città α e β un nuovo percorso $p_{\alpha,\beta}$ ottenuto da p tagliando gli archi $(\alpha, \alpha+1)$ e $(\beta, \beta-1)$ e rimpiazzandoli con $(\alpha, \beta-1)$ e $(\alpha+1, \beta)$.

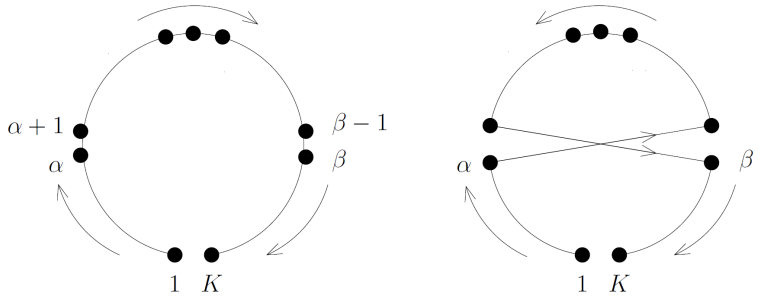

Figura 2: 2-change descritta nella definizione

[8] ovvero la somma delle lunghezze degli archi percorsi per attraversare tutte le n città e tornare alla partenza
[9] si possono rinumerare le città a piacimento in quanto l'ordine è aleatorio

Nella costruzione che si sta facendo saranno considerate solo le *2-change* in cui $\beta \geq \alpha+3$ oppure $\beta \leq \alpha$ così da ottenere un totale di $n(n-3)$ possibili *vicini* di un dato percorso. Da notare che la computazione di $f(p_{\alpha,\beta})$ a partire da $f(p)$ coinvolge solo le distanze tra 4 città.
Sulla base di questa struttura è stato elaborato il seguente algoritmo scritto in linguaggio `python` che esegue iterativamente la procedura di *SA* su degli input di *TSP*.
Di seguito il `main.py` che contiene il "nucleo" della procedura: vengono richiamate le funzioni principali dai vari file e viene eseguita l'iterazione della procedura di *annealing* su dati di input generati dal file `input.py` (allegato). Vengono successivamente richiamate le funzioni per il plot dei risultati dal file `plot.py` (anch'esso allegato).

```python
#previene creazione di cache
import sys
sys.dont_write_bytecode = True

import os
import numpy as np
import random

#funzioni importate
from input import *
from function import *
from output import *
#parent directory
sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))
from plot.plot import*


#set random
random.seed()


#activate or not
y_n = activate_or_not()

output_costs_flag = y_n[0]
plotfig = y_n[1]
numbers = y_n[2]


#input
inp = input_main()

n = inp[0]
k = inp[1]
cities = inp[2]
begin = inp[3]
```

Figura 3: tsp/main.py (parte 1)

```
#first computation
dist = distances(n, cities)

f_begin = f(begin, n, dist)

route = begin
f_route = f_begin


#iteration
it_type = input_iteration()

it = it_type[0]
ty = it_type[1]
tyt = it_type[2]


#start-timer
import time
tot = time.time()


#y_n (0)
if output_costs_flag == 1:
    output_costs = np.zeros(it + 1)
    output_costs[0] = f_route


for t in range(0, it):

    #seleziona 2-change aleatoriamente
    prob = probability(k)

    #trova la 2-change estratta
    alpha_beta = find_two_change(prob, n)
    alpha = alpha_beta[0]
    beta = alpha_beta[1]

    #costruisce la 2-change estratta
    newroute = build_two_change(route, alpha, beta, n)

    #computa f_newroute
    f_newroute = f_short(f_route, route, alpha, beta, n, dist)

    #esegue scelta in accordo con la catena
    out = input_mb(ty, route, f_route, newroute, f_newroute, t, tyt)

    route = out[0]
    f_route = out[1]
```

Figura 4: tsp/main.py (parte 2)

```
90      #y_n (0)
91      if output_costs_flag == 1:
92          output_costs[t + 1] = f_route
93
94      if t%10000 == 0:
95          print("Iteration: " , t)
96          print("Time: ", time.time() - tot)
97
98  #separa i print delle iterazioni dall'output
99  print()
100
101
102 #output
103 text_output(begin, route, f_begin, f_route)
104
105
106 #end-timer
107 tot = time.time() - tot
108 print()
109 print("Time: ", tot)
110
111
112 #plot
113
114 #y_n (1)
115 if plotfig == 1:
116     #y_n (1.1)
117     plot_figure(begin, n, cities, 0, numbers)
118     plot_figure(route, n, cities, 1, numbers)
119
120 if output_costs_flag == 1:
121     #y_n (0)
122     plot_descent(it + 1, output_costs)
123
124 if plotfig == 1 or output_costs_flag == 1:
125     plt.show()
```

Figura 5: tsp/main.py (parte 3)

Per completezza si riporta anche il codice che comprende le principali funzioni: `function.py`. All'interno del seguente codice sono implementate le seguenti procedure:

- definizione della matrice delle distanze dalla funzione `distances`
- computo di $f(p)$ con p percorso iniziale nella funzione f
- computo di $f(p_{\alpha,\beta})$ a partire da $f(p)$
- estrazione aleatoria e costruzione di una 2-change di p, $p_{\alpha,\beta}$
- calcolo di $a_{i,j}$ con i metodi di *Metropolis* e *Barker* con varie funzioni *temperatura*

```python
import numpy as np
import math as mt
import random
import itertools as it


#definisce una matrice delle distanze tra ciascuna coppia di nodi
def distances(n, cities):
    dist = np.zeros((n, n))

    for i in range(0, n):
        for j in range(0, n):
            dist[i, j] = np.sqrt((cities[0, i] - cities[0, j])**2 + (cities[1, i] - cities[1, j])**2)

    return dist


#computa la lunghezza del percorso route
def f(route, n, dist):
    f_route = 0

    app = int(route[0])
    for i in route[1:]:
        i = int(i)
        f_route += dist[app, i]
        app = i
    f_route += dist[int(route[n - 1]), int(route[0])]

    return f_route


#computa la lunghezza del percorso newroute a partire da route, alpha e beta
def f_short(f_route, route, alpha, beta, n, dist):
    alpha_less = dist[int(route[alpha]), int(route[(alpha + 1)%n])]
    beta_less = dist[int(route[beta]), int(route[(beta - 1)%n])]
    alpha_add = dist[int(route[alpha]), int(route[(beta - 1)%n])]
    beta_add = dist[int(route[beta]), int(route[(alpha + 1)%n])]

    return (f_route - alpha_less - beta_less + alpha_add + beta_add)


#estrae con distribuzione uniforme un valore da 0 a (k - 1)
def probability(k):
    return random.randint(0, k - 1)
```

Figura 6: tsp/function.py (parte 1)

```
47 #estrae aleatoriamente una 2-change nella neighborhood
48 def find_two_change(prob, n):
49     c = -1
50     flag = 0
51     for alpha in range (0, n):
52         if (alpha + 3 >= n):
53             for beta in range ((alpha + 3)%n, alpha):
54                 c = c + 1
55                 if c == prob:
56                     flag = 1
57                     break
58         else:
59             #it.chain() unisce e concatena i due range
60             for beta in it.chain(range (alpha + 3, n), range (0, alpha)):
61                 c = c + 1
62                 if c == prob:
63                     flag = 1
64                     break
65         if (flag == 1):
66             break
67 
68     return alpha, beta
69 
70 
71 #costruisce la 2-change estratta aleatoriamente nella funzione precedente
72 def build_two_change(route, alpha, beta, n):
73     newroute = np.zeros((n))
74 
75     if (beta >= alpha + 3):
76         for i in range (0, alpha + 1):
77             newroute[i] = route[i]
78         for i in range (alpha + 1, beta):
79             newroute[i] = route[beta + alpha - i]
80         for i in range (beta, n):
81             newroute[i] = route[i]
82     elif (beta <= (alpha - 1)):
83         for i in range (beta, alpha + 1):
84             newroute[i] = route[i]
85         for i in range (alpha + 1, n):
86             newroute[i] = route[(beta + alpha - i)%n]
87         for i in range (0, beta):
88             newroute[i] = route[(beta + alpha - i)%n]
89 
90     return newroute
91 
92 
93 #Metropolis con Temperatura 1/sqrt(t)
94 def function_m1(f_route, f_newroute, t):
95     return np.exp((f_route - f_newroute)*np.sqrt(t))
```

Figura 7: tsp/function.py (parte 2)

```
98  #Metropolis con Temperatura 1/t
99  def function_m2(f_route, f_newroute, t):
100     return np.exp((f_route - f_newroute)*t)
101
102
103 #Metropolis con Temperatura 0.95^t
104 def function_m3(f_route, f_newroute, t):
105     return np.exp((f_route - f_newroute)*((1/(0.95))**t))
106
107
108 #Metropolis
109 def metropolis(route, f_route, newroute, f_newroute, t, tyt):
110     if f_route >= f_newroute:
111         route = newroute
112         f_route = f_newroute
113     else:
114         accept = random.uniform(0, 1)
115
116         if tyt == 0:
117             aij_t = function_m1(f_route, f_newroute, t)
118         elif tyt == 1:
119             aij_t = function_m2(f_route, f_newroute, t)
120         elif tyt == 2:
121             aij_t = function_m3(f_route, f_newroute, t)
122
123         if accept <= aij_t:
124             route = newroute
125             f_route = f_newroute
126
127     return route, f_route
128
129
130 #Barker con Temperatura 1/sqrt(t)
131 def function_b1(f_route, f_newroute, t):
132     return 1/(1 + np.exp(-(f_route - f_newroute)*np.sqrt(t)))
133
134
135 #Barker con Temperatura 1/t
136 def function_b2(f_route, f_newroute, t):
137     return 1/(1 + np.exp(-(f_route - f_newroute)*t))
138
139
140 #Barker con Temperatura 0.95^t
141 def function_b3(f_route, f_newroute, t):
142     return 1/(1 + np.exp(-(f_route - f_newroute)*((1/(0.95))**t)))
```

Figura 8: tsp/function.py (parte 3)

```
145 #Barker
146 def barker(route, f_route, newroute, f_newroute, t, tyt):
147     #sopprime warning per underflow e overflow (numpy usa 0 e +infty
        automaticamente)
148     np.seterr(all='ignore')
149 
150     accept = random.uniform(0, 1)
151 
152     if tyt == 0:
153         aij_t = function_b1(f_route, f_newroute, t)
154     elif tyt == 1:
155         aij_t = function_b2(f_route, f_newroute, t)
156     elif tyt == 2:
157         aij_t = function_b3(f_route, f_newroute, t)
158 
159     if accept <= aij_t:
160         route = newroute
161         f_route = f_newroute
162 
163     return route, f_route
164 
165 
166 #Metropolis o Barker
167 def input_mb(ty, route, f_route, newroute, f_newroute, t, tyt):
168     if ty == 0:
169         out = metropolis(route, f_route, newroute, f_newroute, t, tyt)
170     elif ty == 1:
171         out = barker(route, f_route, newroute, f_newroute, t, tyt)
172 
173     return out
```

Figura 9: tsp/function.py (parte 4)

L'elaborazione delle *2-change* costituisce la parte computazionalmente più costosa dell'algoritmo. Di seguito uno script che mostra come vengono computati tutti i *vicini* di un percorso con 5 città.
In particolare si hanno in input vertici di un pentagono ed il percorso iniziale è il suo perimetro. Ci aspettiamo di trovare $n(n-3)$ percorsi, ovvero nel caso in esame 10.

```
n = 5

k = n*(n - 3)

cities = np.zeros((2, n))

cities[0,0] = 0.31
cities[1,0] = 0

cities[0,1] = 1.31
cities[1,1] = 0

cities[0,2] = 1.62
cities[1,2] = 0.95

cities[0,3] = 0.81
cities[1,3] = 1.54

cities[0,4] = 0
cities[1,4] = 0.95

begin = np.array([0, 1, 2, 3, 4])
label = "start"
route = begin
plot_figure(begin, n, cities, 0, label)

for prob in range (0, k):

    alpha_beta = find_two_change(prob, n)
    alpha = alpha_beta[0]
    beta = alpha_beta[1]

    newroute = build_two_change(route, alpha, beta, n)

    a = str(alpha)
    b = str(beta)
    label = "alpha: " + a + " beta: " + b

    plot_figure(newroute, n, cities, prob + 1, label)

plt.show()
```

Figura 10: tsp/check.py

Gli output del codice sono i seguenti grafi (si noti che con la costruzione effettuata ciascuna 2-change compare due volte, tuttavia ciò non influenza l'estrazione aleatoria che rimane uniforme). L'ordinamento dell'output è lessicografico: prima in relazione al nodo alpha scelto, poi al nodo beta.

2-change di un grafo di 5 nodi con perimetro come percorso iniziale

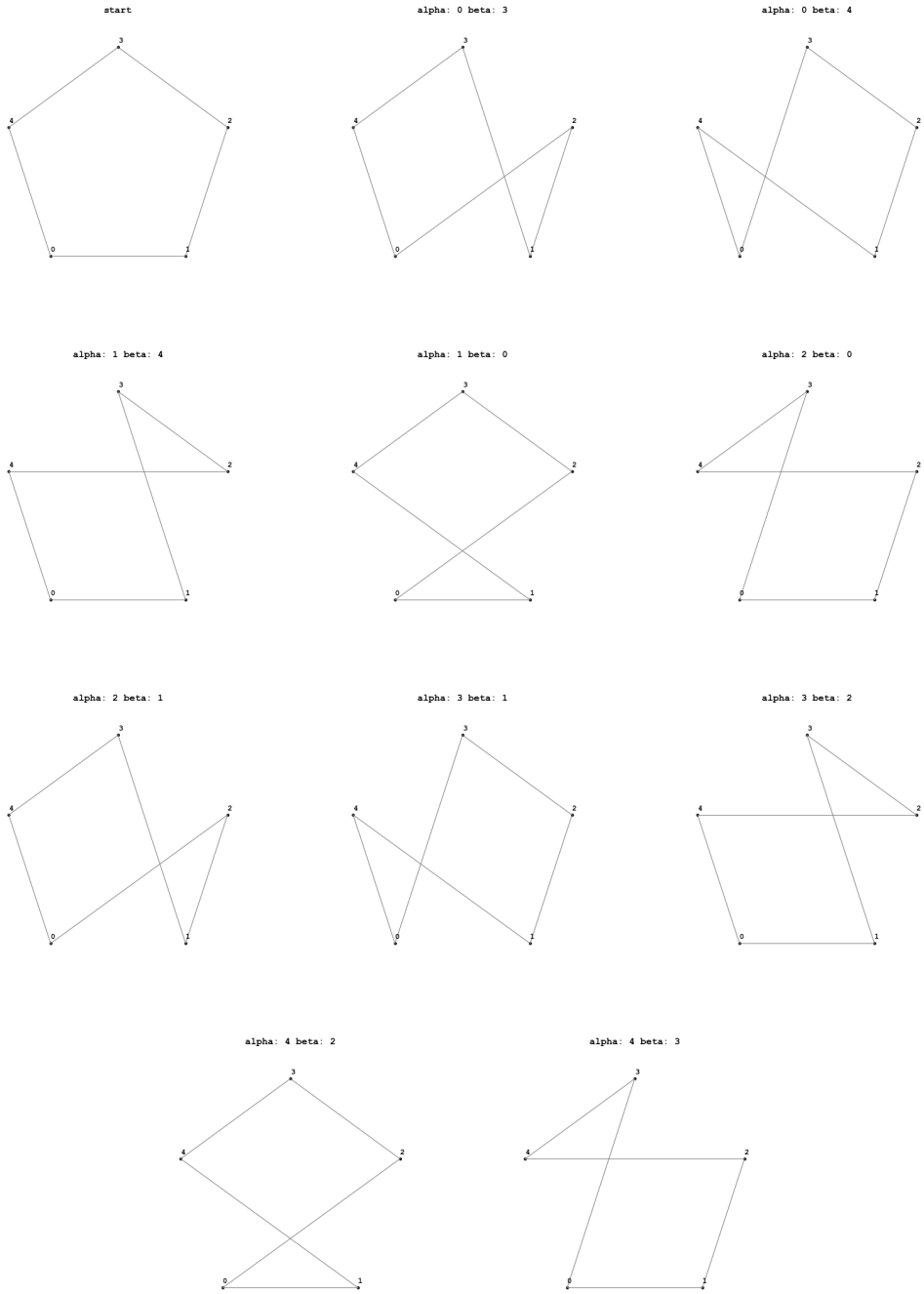

Figura 11: 2-change del perimetro di un pentagono

Per concludere si presenta un esempio di output del codice principale su input aleatorio di 1000 città con percorso iniziale anch'esso aleatorio. L'elaborazione è stata fatta scegliendo la catena di *Metropolis* ed utilizzando come funzione *temperatura* $T(t) = \frac{1}{t}$.
Oltre ai plot dei grafi iniziale e finale si ha in output anche il plot della discesa di $f(p)$ al variare del percorso p allo scorrere delle iterazioni, in funzione del tempo t.

`Plot della discesa di` f

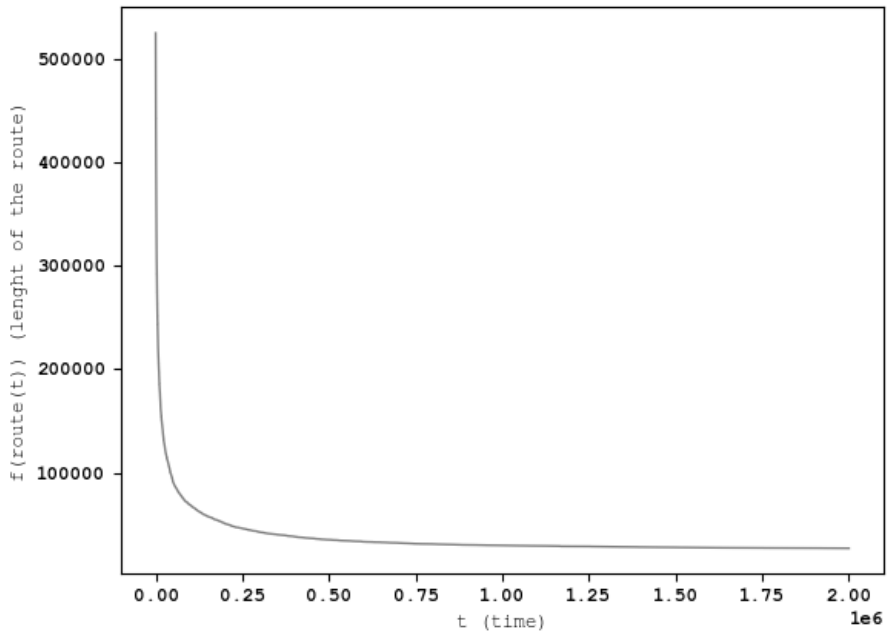

Figura 12: plot di $f(p(t))$ con $p(t)$ percorso al tempo t

Output testuale del codice

```
Activate or not:
    0. Record costs during iterations and plot descent: 1
    1. Figure plot: 1
        1.1. Write numbers in figure plot: 0

Insert scenario:
    0. Random scenario
    1. Octagon
    2. 50 fixed points
    Scenario: 0

Insert number of cities: 1000
Insert number of iterations: 2000000
Insert type of iterations:
    0. Metropolis
    1. Barker
    Type: 0
    Insert type of cooling schedule:
        0. T(t) = 1/sqrt(t)
        1. T(t) = 1/t
        2. T(t) = (0.95)^t
        Temperature: 1

Iteration:  0
Time:  0.006005525588989258
Iteration:  10000
Time:  254.12885189056396
Iteration:  20000
Time:  505.41855788230896
...
Iteration:  1970000
Time:  49033.379962444305
Iteration:  1980000
Time:  49280.80493712425
Iteration:  1990000
Time:  49523.15043449402

Start route:  [0, 1, 2 ... 997, 998, 999]

Final route:  [161, 205, 718 ... 759, 553, 588]

Start cost:  525488.2398075429

Final cost:  26838.925599942555

Time:  49765.64771270752
```

Figura 13: Output completo, comprensivo di scelte di input e lunghezze dei percorsi

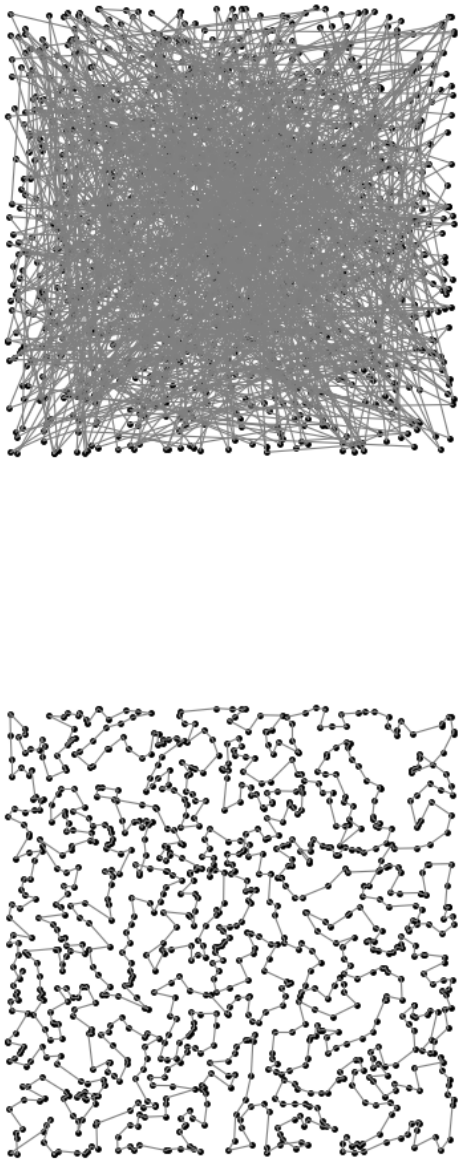

Figura 14: in alto il grafo aleatorio in input, in basso il risultato di 2000000 iterazioni di *SA*

5 Vetri di Spin

Uno *vetro di spin (SG dall'inglese "spin glass")*[6] è un modello per certi tipi di magnete in cui legami ferromagnetici e antiferromagnetici sono distribuiti in modo casuale. Questi oggetti, molto studiati in fisica, hanno anche un notevole interesse matematico.
Fissata a due la dimensione dello spazio su cui definiamo i *vetri di spin* per il nostro esercizio, possiamo descrivere l'oggetto che stiamo prendendo in analisi come una lastra magnetica discreta in cui ciascuna particella può trovarsi in una di due possibili configurazioni: *spin* ± 1.
In particolare nella presente relazione è presa in analisi la struttura di base con la quale solitamente i *vetri di spin* sono descritti: il modello di *Edwards-Anderson (EA)*, una variante di quello di *Ising*, che aggiunge una componente aleatoria alla classica funzione energia di quest'ultimo.
Di seguito si riporta una descrizione formale del modello di *Edwards-Anderson*[10] su un sottoinsieme del reticolo $\mathbb{Z}^2$.

Modello di Edwards-Anderson. Sia $\Lambda_n = \{(x, y) \in \mathbb{Z}^2 \mid -n \leq x, y \leq n\}$ la porzione di reticolo considerata, la si munisce di struttura di grafo definendo come nodi gli $x \in \Lambda_n$ e andando ad aggiungere come archi solo quelli che congiungono nodi *adiacenti*, ovvero $\langle x, y\rangle$ è un arco per il nostro grafo $\iff$ $d_2(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} = 1$.
Dunque il grafo è $G = (\Lambda_n, A)$, dove A è l'insieme di archi appena descritto.
Lo spazio delle possibili configurazioni degli spin che il nostro modello può assumere è definito come $E = \{-1, 1\}^{\Lambda_n}$.
Si può definire l'energia $H(\sigma)$ di una configurazione $\sigma \in E$ con la seguente formula

$$H(\sigma) = -\sum_{\langle x,y\rangle \in A} J(\langle x, y\rangle)\sigma_x\sigma_y \tag{1}$$

dove $\sigma_x := \{\text{lo spin del nodo } x \in \Lambda_n\} \in \{-1, 1\}$.
La $J(\langle x, y\rangle)$ è l'elemento che distingue il modello di *EA* da quello di *Ising*. Nel caso trattato $J(\langle x, y\rangle)$ è una variabile aleatoria con distribuzione uniforme su $\{-1, 1\}$.

Questo modello è in realtà generalizzabile a d dimensioni per $d \in \mathbb{N}$. Chiaramente scegliendo dimensioni maggiori la funzione energia si complica notevolmente.
Sarà analizzato esclusivamente il caso $d = 2$ sopra descritto in quanto risulta essere il modello più *semplice* possibile per il quale il risultato è non scontato.
Si presenta di seguito una breve descrizione del modello di *EA* in dimensione 1 per mostrare la semplicità della soluzione diretta e giustificare la scelta di non implementare un *SA* per questo problema, in quanto risulterebbe superfluo.

[10] in realtà si sta utilizzando una specifica versione, particolarmente semplice del modello, che invece può prevedere scelte diverse sia per la costruzione della struttura di grafo sia per la scelta della variabile aleatoria J

***EA* in dimensione 1.** Se venisse considerato $d = 1$ il problema si potrebbe risolvere in maniera diretta. Scegliendo ad esempio $n = 3$ si avrebbe questa situazione:

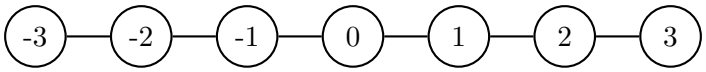

Figura 15: modello di *EA* per $d = 1$ ed $n = 3$

Supponiamo che J assuma i seguenti valori[11]

$$J(\langle x, y\rangle) = \begin{cases} -1 & \text{se } (x, y) = (-3, -2) \\ -1 & \text{se } (x, y) = (-2, -1) \\ +1 & \text{se } (x, y) = (-1, 0) \\ -1 & \text{se } (x, y) = (0, 1) \\ +1 & \text{se } (x, y) = (1, 2) \\ -1 & \text{se } (x, y) = (2, 3) \end{cases}$$

Basta assegnare al nodo -3 spin 1 e scegliere di conseguenza tutti gli altri spin, al fine di forzare ad 1 ogni termine della somma (1), per garantire la minimizzazione di H.

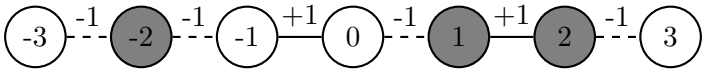

Figura 16: modello di *EA* per d = 1, risoluzione diretta (grigio sta per spin -1 e bianco per spin 1)

In questo caso la soluzione quella indicata in Figura 16 e realizza il minimo globale per H, pari a -6.

Definito il modello si può procedere alla formulazione del problema.
L'esercizio da risolvere[12] è il seguente: trovare i minimi globali della funzione energia di *EA* (nella versione appena definita).
Anche questo, come il *TSP*, è un problema *NP-Hard*[7], perciò ha senso provare ad applicare *SA* in quanto non ci sono algoritmi *esatti* per calcolare la soluzione che abbiano un costo computazionale ragionevole.
Per poter implementare una versione di *SA* per questo quesito occorre definire una *struttura di vicinato* che, come già specificato, sia simmetrica e comunicante in modo da poter disporre del supporto teorico sopra descritto. Si definiscono perciò come *vicini* di una configurazione tutti i suoi *primi vicini*, ovvero le altre possibili configurazioni di spin tale per cui valga la seguente proprietà:

[11] si potrebbe ripetere l'esempio in maniera analoga con una qualsiasi altra scelta di J
[12] in realtà essendo *SA* un algoritmo euristico sarebbe più corretto dire "l'esercizio da approssimare"

Definizione 5.1. Dati $\sigma, \sigma' \in E$, si dicono *primi vicini* se

$$\sigma_x = \sigma'_x \quad \forall x \in \Lambda_n \text{ tranne al più un elemento}$$

Come per il *TSP*, è stato elaborato un algoritmo scritto in linguaggio `python` che esegue iterativamente la procedura di *SA* su degli input coerenti con il modello *EA*.
La parte di codice seguente è il `main.py`, ovvero, in maniera analoga al *TSP*, la parte centrale dell'algoritmo che richiama tutte le routine principali dai file `function.py`, `input.py`, `output.py` e `plot.py`.

```
#previene creazione di cache
import sys
sys.dont_write_bytecode = True

import os
import numpy as np
import random

#funzioni importate
from input import *
from function import *
from output import *
#parent directory
sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))
from plot.plot import*


#set random
random.seed()


#activate or not
y_n = activate_or_not()

output_costs_flag = y_n[0]
plotfig = y_n[1]
edges = y_n[2]


#input
inp = input_main()

begin = inp[0]
n = inp[1]


#computation of J
computation = J(n)

J_0 = computation[0]
J_1 = computation[1]
```

Figura 17: sg/main.py (parte 1)

```
#first computation
H_begin = H(begin, n, J_0, J_1)

#viene fatto np.matrix(begin) altrimenti l'assegnazione sarebbe
#eseguita come puntatore e avrei delle modifiche di begin ogni qual volta
#modifico sigma_0 (anche dopo)
sigma_0 = np.matrix(begin)
H_sigma_0 = H_begin


#iteration
it_type = input_iteration()

it = it_type[0]
ty = it_type[1]
tyt = it_type[2]


#start-timer
import time
tot = time.time()


#y_n (0)
if output_costs_flag == 1:
    output_costs = np.zeros(it + 1)
    output_costs[0] = H_sigma_0


for t in range(0, it):

    #seleziona neighbor aleatoriamente
    prob = probability(n)

    line = prob[0]
    column = prob[1]

    sigma_1 = np.matrix(sigma_0)
    sigma_1[line, column] = -sigma_0[line, column]

    H_sigma_1 = H_short(sigma_0, H_sigma_0, line, column, n, J_0, J_1)

    #esegue scelta in accordo con la catena
    out = input_mb(ty, sigma_0, H_sigma_0, sigma_1, H_sigma_1, t, tyt)

    sigma_0 = np.matrix(out[0])
    H_sigma_0 = out[1]


    #y_n (0)
    if output_costs_flag == 1:
        output_costs[t + 1] = H_sigma_0
```

Figura 18: sg/main.py (parte 2)

```
98 #output
99 text_output(sigma_0, H_sigma_0, begin, H_begin, J_0, J_1)
100
101
102 #end-timer
103 tot = time.time() - tot
104 print()
105 print("Time: ", tot)
106
107
108 #plot
109
110 if n >= 21:
111     plotfig = 0
112
113 if n >= 6:
114     edges = 0
115
116 if plotfig == 1:
117     #y_n (1)
118     plot_figure(begin, n, 0)
119     plot_figure(sigma_0, n, 1)
120
121     if edges == 1:
122         #y_n (1.1)
123         plot_edges(J_0, J_1, n, 2)
124
125 if output_costs_flag == 1:
126     #y_n (0)
127     plot_descent(it + 1, output_costs)
128
129 if plotfig == 1 or output_costs_flag == 1:
130     plt.show()
```

Figura 19: sg/main.py (parte 3)

Come effettuato per *TSP*, per completezza si riporta anche il codice che comprende le principali funzioni: `function.py`.
All'interno del seguente codice sono implementate le seguenti procedure:

- costruzione delle variabili aleatorie $J(\langle x, y \rangle)$
- computo di $H(\sigma)$ con σ configurazione di spin
- computo di $H(\sigma')$ a partire da $H(\sigma)$
- estrazione aleatoria con distribuzione uniforme di un valore $\in [0, 2n]$
- calcolo di $a_{i,j}$ con i metodi di *Metropolis* e *Barker* con varie funzioni *temperatura* (funzioni identiche a quelle di *TSP*)

```python
import numpy as np
import math as mt
import random
import itertools as it


#computa l'energia di sigma
def H(sigma, n, J_0, J_1):
    H_sigma = 0

    #aggiungo contributo di archi orizzontali (identificati con 0)
    #i mi dice la riga in cui si trova l'arco, j il nodo da cui parte (sulla riga, da sx verso dx)
    #la componente i denota le y, la j le x (si vede nel plot)
    for i in range(0, 2*n + 1):
        for j in range(0, 2*n):
            H_sigma +=  J_0[i, j]*sigma[i, j]*sigma[i, j + 1]

    #aggiungo contributo di archi verticali (identificati con 1)
    #j mi dice la colonna in cui si trova l'arco, i il nodo da cui parte (sulla colonna, dall'up verso il down)
    #la componente i denota le y, la j le x (si vede nel plot)
    for j in range(0, 2*n + 1):
        for i in range(0, 2*n):
            H_sigma +=  J_1[i, j]*sigma[i, j]*sigma[i + 1, j]

    return H_sigma


#computa l'energia di sigma_1 sapendo quella di sigma_0
def H_short(sigma_0, H_sigma_0, line, column, n, J_0, J_1):
    H_sigma_1 = H_sigma_0

    if (line != 0):
        H_sigma_1 += -2*J_1[line - 1, column]*sigma_0[line - 1, column]*sigma_0[line, column]
    if (line != 2*n):
        H_sigma_1 += -2*J_1[line, column]*sigma_0[line, column]*sigma_0[line + 1, column]
    if (column != 0):
        H_sigma_1 += -2*J_0[line, column - 1]*sigma_0[line, column - 1]*sigma_0[line, column]
    if (column != 2*n):
        H_sigma_1 += -2*J_0[line, column]*sigma_0[line, column]*sigma_0[line, column + 1]

    return H_sigma_1
```

Figura 20: sg/function.py (parte 1)

```python
#computo di J
def J(n):
    J_0  = np.zeros((2*n + 1, 2*n))

    for i in range(0, 2*n + 1):
        for j in range(0, 2*n):
            J_0[i, j] = random.choices([-1, 1])[0]

    J_1  = np.zeros((2*n, 2*n + 1))

    for i in range(0, 2*n):
        for j in range(0, 2*n + 1):
            J_1[i, j] = random.choices([-1, 1])[0]

    return J_0, J_1


#estrae con distribuzione uniforme un valore da 0 a 2*n
def probability(n):
    line = random.randint(0, 2*n)
    column = random.randint(0, 2*n)

    return line, column

#Metropolis con Temperatura 1/sqrt(t)
def function_m1(H_sigma_0, H_sigma_1, t):
    return np.exp((H_sigma_0 - H_sigma_1)*np.sqrt(t))


#Metropolis con Temperatura 1/t
def function_m2(H_sigma_0, H_sigma_1, t):
    return np.exp((H_sigma_0 - H_sigma_1)*t)


#Metropolis con Temperatura 0.95^t
def function_m3(H_sigma_0, H_sigma_1, t):
    return np.exp((H_sigma_0 - H_sigma_1)*((1/(0.95))**t))


#Metropolis
def metropolis(sigma_0, H_sigma_0, sigma_1, H_sigma_1, t, tyt):
    if H_sigma_0 >= H_sigma_1:
        sigma_0 = sigma_1
        H_sigma_0 = H_sigma_1
    else:
        accept = random.uniform(0, 1)

        if tyt == 0:
            aij_t = function_m1(H_sigma_0, H_sigma_1, t)
        elif tyt == 1:
            aij_t = function_m2(H_sigma_0, H_sigma_1, t)
        elif tyt == 2:
            aij_t = function_m3(H_sigma_0, H_sigma_1, t)

        if accept <= aij_t:
            sigma_0 = sigma_1
            H_sigma_0 = H_sigma_1

    return sigma_0, H_sigma_0
```

Figura 21: sg/function.py (parte 2)

```
105 #Barker con Temperatura 1/sqrt(t)
106 def function_b1(H_sigma_0, H_sigma_1, t):
107     return 1/(1 + np.exp(-(H_sigma_0 - H_sigma_1)*np.sqrt(t)))
108
109
110 #Barker con Temperatura 1/t
111 def function_b2(H_sigma_0, H_sigma_1, t):
112     return 1/(1 + np.exp(-(H_sigma_0 - H_sigma_1)*t))
113
114
115 #Barker con Temperatura 0.95^t
116 def function_b3(H_sigma_0, H_sigma_1, t):
117     return 1/(1 + np.exp(-(H_sigma_0 - H_sigma_1)*((1/(0.95))**t)))
118
119
120 #Barker
121 def barker(sigma_0, H_sigma_0, sigma_1, H_sigma_1, t, tyt):
122     #sopprime warning per underflow e overflow (numpy usa 0 e +infty
        automaticamente)
123     np.seterr(all='ignore')
124
125     accept = random.uniform(0, 1)
126
127     if tyt == 0:
128         aij_t = function_b1(H_sigma_0, H_sigma_1, t)
129     elif tyt == 1:
130         aij_t = function_b2(H_sigma_0, H_sigma_1, t)
131     elif tyt == 2:
132         aij_t = function_b3(H_sigma_0, H_sigma_1, t)
133
134     if accept <= aij_t:
135         sigma_0 = sigma_1
136         H_sigma_0 = H_sigma_1
137
138     return sigma_0, H_sigma_0
139
140
141 #Metropolis o Barker
142 def input_mb(ty, sigma_0, H_sigma_0, sigma_1, H_sigma_1, t, tyt):
143     if ty == 0:
144         out = metropolis(sigma_0, H_sigma_0, sigma_1, H_sigma_1, t, tyt)
145     elif ty == 1:
146         out = barker(sigma_0, H_sigma_0, sigma_1, H_sigma_1, t, tyt)
147
148     return out
```

Figura 22: sg/function.py (parte 3)

In maniera analoga alla sperientazione precedente si presenta un esempio di output del codice principale su input aleatorio con $n = 12$ (dunque la porzione di reticolo considerata è $[-12, 12]^2$). L'elaborazione è stata fatta scegliendo la catena di *Barker* ed utilizzando come funzione *temperatura* $\frac{1}{\sqrt{t}}$.
Sono presenti i plot della discesa e delle 2 configurazioni di spin, prima e dopo *SA*.
Si noti che il valore -1 è stato rappresentato con un pallino bianco mentre il valore 1 con uno nero (Figura 25).

`Plot della discesa di` H

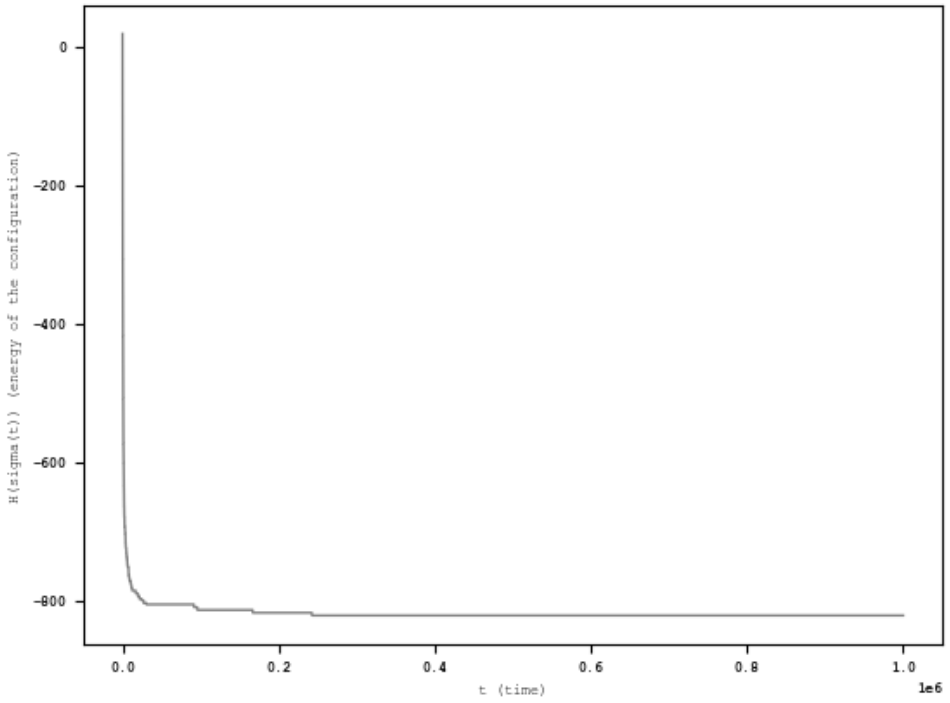

Figura 23: plot di $H(\sigma(t))$ con $\sigma(t)$ configurazione al tempo t

Output testuale del codice

```
Activate or not:
    0. Record costs during iterations and plot descent: 1
    1. Figure plot (possible only if n <= 20): 1
        1.1. Plot edges (possible only if n <= 5): 0

Insert scenario:
    0. Random scenario
    Scenario: 0

Insert n for size of lattice (-n, n): 12
Insert number of iterations: 1000000
Insert type of iterations:
    0. Metropolis
    1. Barker
    Type: 1
    Insert type of cooling schedule:
        0. T(t) = 1/sqrt(t)
        1. T(t) = 1/t
        2. T(t) = (0.95)^t
        Temperature: 0

Start configuration:
[-1.  1.  1. ...  1.  1. -1.]
[ 1. -1. -1. ...  1. -1.  1.]
[ 1.  1.  1. ...  1.  1. -1.]
...
[ 1. -1. -1.  ... -1. -1. -1.]
[ 1. -1.  1.  ...  1. -1.  1.]
[ 1.  1.  1.  ... -1.  1.  1.]

Start configuration energy:  20.0

Final configuration:
[ 1. -1. -1. ... -1.  1. -1.]
[-1.  1. -1. ...  1. -1.  1.]
[ 1. -1.  1. ...  1.  1.  1.]
...
[ 1. -1. -1. ... -1. -1. -1.]
[ 1. -1. -1. ...  1.  1.  1.]
[-1.  1.  1. ...  1.  1.  1.]

Final configuration energy:  -820.0

Time:  21.69878125190735
```

Figura 24: Output completo, comprensivo di scelte di input e energia delle varie configurazioni

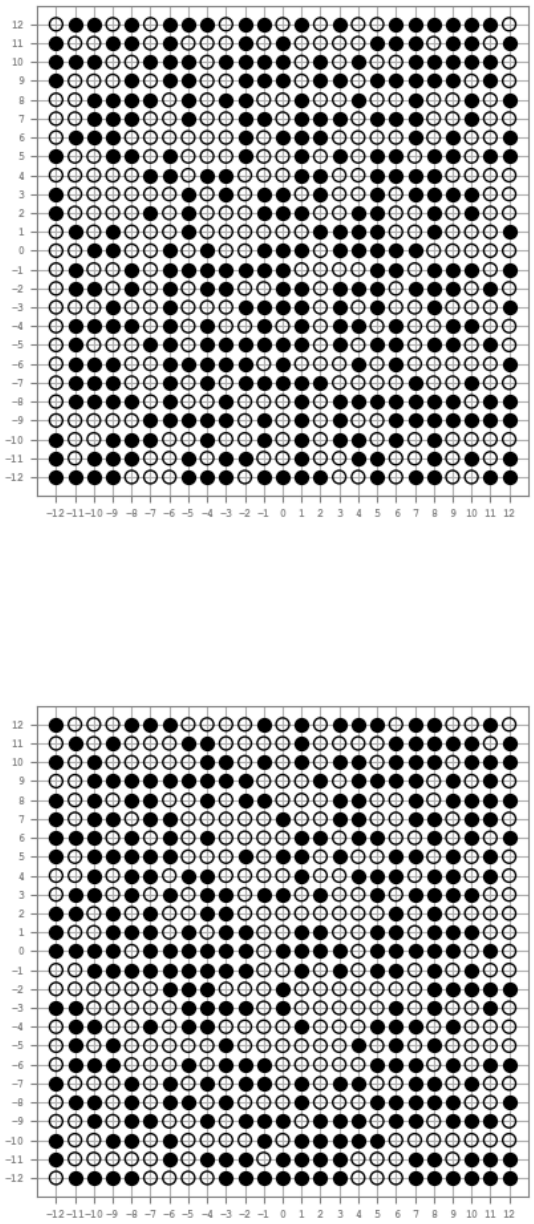

Figura 25: in alto la configurazione aleatoria in input, in basso il risultato di 1000000 iterazioni di *SA*

Si presenta infine un ulteriore esempio di risultato per un valore di n più piccolo, pari a 5, per poter dare una esemplificazione grafica di come le variabili aleatorie $J(\langle x, y\rangle)$ influenzano gli archi. In figura si vede che nella sperimentazione la J ha assegnato casualmente legami *ferromagnetici* ($J(\langle x, y\rangle) = +1$, indicati con linee continue) o *anti-ferromagnetici* ($J(\langle x, y\rangle) = -1$ indicati con linee tratteggiate) al nostro modello, così da modificare il minimo della funzione H (Figura 28).
In questo caso l'elaborazione è stata eseguita scegliendo la catena di *Barker* ed utilizzando come funzione *temperatura* 0.95^t.

`Plot della discesa di` H

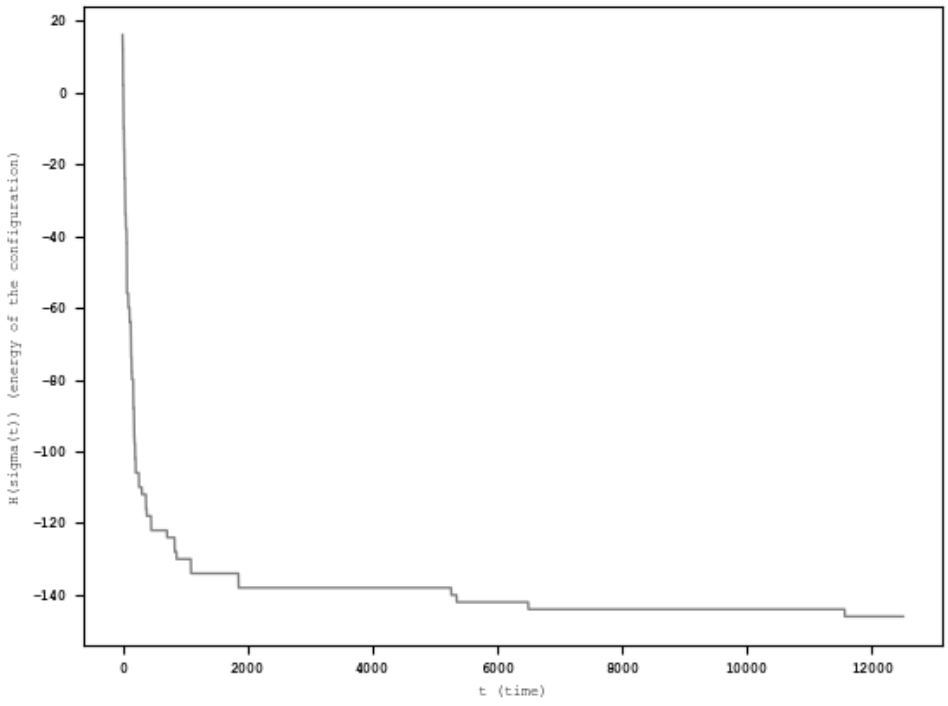

Figura 26: plot di $H(\sigma(t))$ con $\sigma(t)$ configurazione al tempo t

Output testuale del codice

```
Activate or not:
    0. Record costs during iterations and plot descent: 1
    1. Figure plot (possible only if n <= 20): 1
        1.1. Plot edges (possible only if n <= 5): 1

Insert scenario:
    0. Random scenario
    Scenario: 0

Insert n for size of lattice (-n, n): 5
Insert number of iterations: 12500
Insert type of iterations:
    0. Metropolis
    1. Barker
    Type: 1
    Insert type of cooling schedule:
        0. T(t) = 1/sqrt(t)
        1. T(t) = 1/t
        2. T(t) = (0.95)^t
        Temperature: 2

Start configuration:
[ 1.  1.  1.  1.  1.  1. -1.  1.  1. -1.  1.]
[ 1.  1. -1. -1. -1. -1. -1.  1.  1. -1. -1.]
[ 1. -1. -1. -1. -1.  1. -1. -1.  1.  1.  1.]
[ 1.  1.  1.  1. -1. -1. -1. -1. -1.  1. -1.]
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  1.]
[-1. -1. -1.  1. -1. -1.  1.  1. -1. -1. -1.]
[-1.  1. -1. -1.  1. -1. -1.  1.  1.  1.  1.]
[ 1.  1. -1. -1. -1.  1. -1. -1. -1.  1. -1.]
[ 1. -1. -1. -1.  1.  1.  1.  1. -1.  1. -1.]
[-1. -1.  1.  1. -1. -1.  1.  1. -1. -1. -1.]
[-1.  1.  1. -1.  1. -1.  1.  1.  1. -1.  1.]

Start configuration energy:  16.0

Final configuration:
[ 1.  1. -1.  1.  1.  1. -1. -1.  1.  1.  1.]
[ 1. -1. -1.  1.  1.  1.  1. -1.  1. -1. -1.]
[-1. -1. -1. -1.  1. -1. -1.  1.  1.  1. -1.]
[ 1.  1. -1. -1. -1.  1.  1.  1. -1.  1. -1.]
[-1.  1. -1. -1. -1. -1. -1. -1.  1. -1.  1.]
[-1.  1. -1. -1. -1. -1. -1.  1. -1. -1. -1.]
[ 1.  1.  1.  1.  1.  1. -1. -1. -1. -1.  1.]
[ 1. -1. -1.  1.  1.  1. -1. -1.  1.  1. -1.]
[ 1.  1. -1.  1. -1. -1. -1. -1.  1. -1. -1.]
[-1. -1.  1.  1. -1. -1.  1.  1. -1. -1.  1.]
[ 1.  1. -1.  1.  1.  1. -1. -1. -1. -1.  1.]

Final configuration energy:  -146.0

Time:  0.4059450626373291
```

Figura 27: Output completo, comprensivo di scelte di input e energia delle varie configurazioni

Plot degli archi generati da $J(\langle x, y \rangle)$

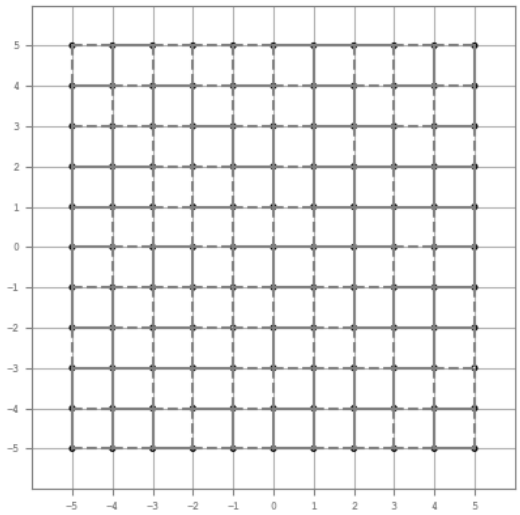

Figura 28: archi ferromagnetici $(-)$ o antiferromagnetici $(--)$ generati aleatoriamente

Configurazione prima e dopo l'applicazione di *SA*

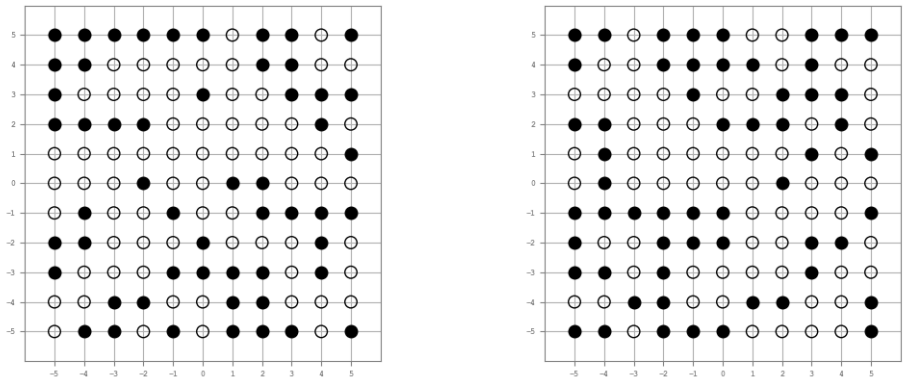

Figura 29: a destra la configurazione aleatoria in input, a sinistra il risultato di 12500 iterazioni di *SA*

Riferimenti bibliografici

[1] Olle Häggström, *Finite Markov Chains and Algorithmic Applications*, Cambridge University Press, 2002

[2] Pierre Brémaud, *Markov Chains: Gibbs Fields, Monte Carlo Simulation and Queues*, 2^{nd} edition, Springer, 2020

[3] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, *Optimization by simulated annealing*, Science 220, 1983, p. 671–680

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein, *Introduction to Algorithms*, McGraw-Hill, 1990

[5] William J. Cook, *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*, Princeton University Press, 2012

[6] Marc Mezard, Giorgio Parisi, Miguel Angel Virasoro, *Spin glass theory and beyond: an introduction to the replica method and its applications*, World Scientific Publishing Company, 1987

[7] F. Barahona, *On the computational complexity of Ising spin glass models*, Journal of Physics A: Mathematical and General, 1982

[8] S. Geman, D. Geman, *Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images*, IEEE Transactions on Pattern Analysis and Machine Intelligence 6, 1984, p. 721–741.