

PC-2022/23 Maze Labyrinth Solver

Francesco Bellezza

E-mail address

`francesco.bellezza@stud.unifi.it`

Abstract

All'interno di questo progetto abbiamo costruito in linguaggio Python un algoritmo che crea e risolve i labirinti mediante il movimento casuale di una particella che viene generata all'entrata del labirinto e si muove in maniera casuale fino a quando non raggiunge la fine del percorso. Abbiamo realizzato anche una versione parallela del programma e l'abbiamo confrontata con quella sequenziale. I test sono stati effettuati su un pc contenente un Intel Core i5-8250U 1.60 GHz, 1801 Mhz, 4 core e 8 processori logici. Solitamente l'algoritmo parallelo performa meglio, ma l'aumento di performance è molto più evidente per labirinti di dimensione più grande.

1. L'algoritmo

Per creare l'intero processo abbiamo sfruttato una suddivisione in classi. Le classi principali utilizzate in questo progetto sono le seguenti:

- Labyrinth, classe preposta alla definizione della struttura di un labirinto. Un labirinto viene rappresentato come una griglia di LabyrinthCell.
- LabyrinthCell, classe adibita alla definizione di una cella del labirinto. Una cella all'interno del labirinto può avere un massimo di 4 muri: la presenza di un muro in una delle possibili 4 direzioni (Nord,Sud,Est,Ovest) viene definita mediante un apposita variabile booleana per ciascun punto cardinale.
- GenerateLabyrinth, classe adibita alla creazione del labirinto. Abbiamo adottato una semplificazione dell'algoritmo di Wilson.
- DrawLabyrinthSolve, classe adibita al disegnare il labirinto e alla sua risoluzione attraverso il movimento delle particelle.

1.1. Generazione del labirinto

Per la generazione del labirinto è stata adoperata una versione semplificata dell'algoritmo di Wilson. Esso consiste nei seguenti passaggi:

- Inserire tutte le celle del labirinto come celle non visitate.
- Scegliere una cella casuale da inserire nel labirinto.
- Scegliere un'altra cella casuale tra quelle che non sono presenti nel labirinto. Da questa cella partiamo per costruire un percorso. Il percorso termina quando raggiunge una cella che è già presente nel labirinto.
- Quando ciò accade, bisogna aggiungere tutte le celle del percorso all'interno del labirinto e riselectare una cella casuale all'interno del labirinto che non fa ancora parte delle celle visitate e ricostruire un altro percorso come fatto precedentemente. L'algoritmo termina quando tutte le celle sono state visitate.

1.2. Il codice sequenziale

Il metodo di interesse che implementa la creazione della particella che si muove all'interno del labirinto si chiama `find_exit_with_random_particles`. Il fulcro del metodo consiste nel ciclo `while`, dove si esce da tale ciclo solo ed esclusivamente se si raggiunge l'uscita del labirinto. La variabile `FOUND_PATH` che si trova in un file apposito viene utilizzata per l'approccio parallelo e verrà spiegata nel prossimo capitolo. La particella prende una decisione e genera una delle possibili direzioni, (compito del metodo `get_random_point_direction`). Dopo aver generato la direzione (che può essere una qualunque dei 8 punti cardinali), si genera il movimento che viene visto come un incremento della posizione della particella nel labirinto - che viene rappresentata tramite coppia (ascissa,ordinata) -. Il metodo `get_allowed_movements` ritorna la terna di possibili incrementi o decrementi della (x,y) della particella in base alla direzione. Le direzioni sono definite nel file `python directions.py`. Successivamente il movimento di x e di y viene stabilito scegliendo casualmente un elemento

dalla lista generata dal metodo `get_allowed_movements`. Se il movimento della particella è consentito, allora aggiungiamo il punto al percorso compiuto dalla particella (che si trova all'interno della lista `path_particle`). Poi è necessario fare un ulteriore controllo importante: controllare se incrementando la coordinata della particella si raggiunge o meno una nuova cella del labirinto (compito del metodo `is_particle_go_to_new_cell`). Questo è fondamentale perché, per capire se la particella incontra un muro oppure no, è necessario che venga aggiornata correttamente la posizione spaziale all'interno della griglia. Se per caso ci muoviamo in una nuova cella, aggiorniamo `path_grid` (che contiene il percorso fatto dalla particella relativa alla griglia) e aggiorniamo la posizione corrente. Per controllare se una particella entra o meno in una nuova cella, è sufficiente controllare se il valore della `x` o della `y` è maggiore dei delimiter definiti ad inizio metodo. I delimiter sono determinati mediante la lunghezza del singolo muro. Per controllare se la particella si possa o meno muovere in una direzione, abbiamo utilizzato la seguente logica: se nella posizione corrente all'interno della griglia la particella si sta muovendo a Nord, allora se per caso supera il delimiter e la particella sta andando a Nord, significa che tale movimento non è contemplato. Questo si applica a tutte le direzioni. Tale controllo però non è sufficiente, infatti bisogna avere particolare attenzione per i movimenti in diagonale. Infatti può succedere che se la particella va a Nord-Est, bisogna anche controllare che nella posizione nella griglia a Nord-Est abbia un muro a Sud-Ovest. Questo viene controllato tramite il metodo `diagonal_movement_is_possible`. Alla fine del metodo principale, usciti dal `while`, si disegna il percorso che è stato fatto dalla particella, (disegnandolo dall'uscita all'entrata).

1.3. Il codice parallelo

Per quanto riguarda il codice in parallelo, si utilizza il solito metodo. Solo che stavolta, invece che chiamarlo direttamente, si sfrutta la libreria di python `Joblib` per creare più processi che eseguono lo stesso metodo. La variabile `FOUND_PATH` viene utilizzata per quando la prima particella riesce effettivamente a completare il labirinto, infatti, qualora dovesse accadere, bisogna accedere a tale variabile condivisa e controllare se è stata già impostata al valore vero. Se così non fosse, allora si imposta a vero, e tutti gli altri programmi terminano. Quando si controlla il contenuto di tale variabile e si vuole sovrascrivere tale valore, per evitare inconsistenze, si usa un `Lock`, così che l'accesso a quella sezione di codice è consentito solo da un thread per volta.

2. Analisi della correttezza

Per verificare o meno la correttezza di tale codice sono state effettuate numerose prove e test su molti labirinti. E'

possibile verificare l'andamento della particella all'interno del labirinto vedendo il file video che viene generato nella apposita directory.

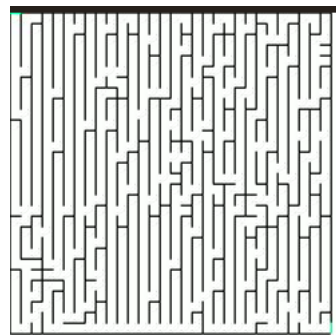
3. Analisi delle prestazioni

3.1. Hardware

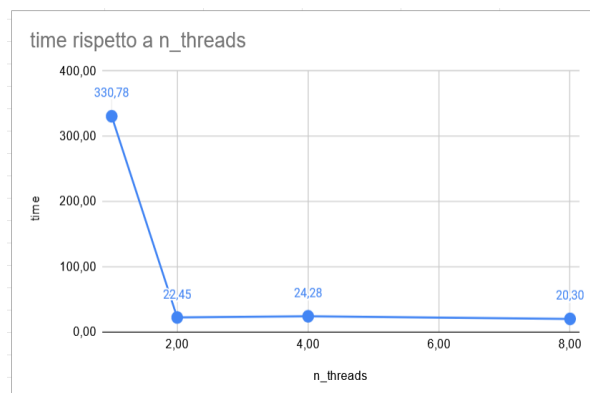
Abbiamo testato questo programma python su un computer con CPU Intel Core i5-8250U 1.60 GHz, 1801 Mhz, 4 core e 8 processori logici. I labirinti vengono generati casualmente e le singole prove sequenziali e quelle con un numero `n` di thread vengono effettuate sullo stesso labirinto.

3.2. Test su un labirinto di dimensione 30x30

Per il seguente labirinto 30x30:

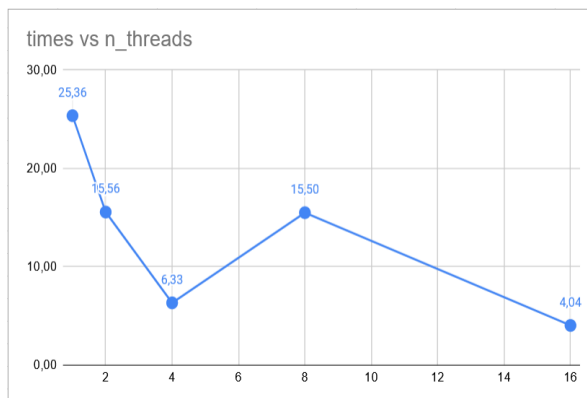
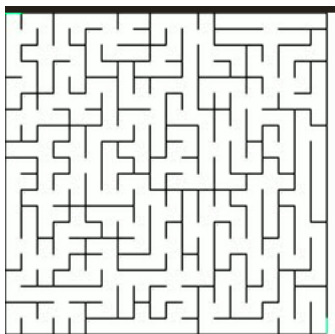


Abbiamo utilizzato prima il metodo sequenziale, successivamente il metodo multithreading e abbiamo ottenuto i seguenti risultati. Sull'asse delle ordinate abbiamo il valore dei secondi, sull'asse delle `x` il numero dei thread utilizzati. In questo caso il risparmio da un punto di vista temporale è considerevole.

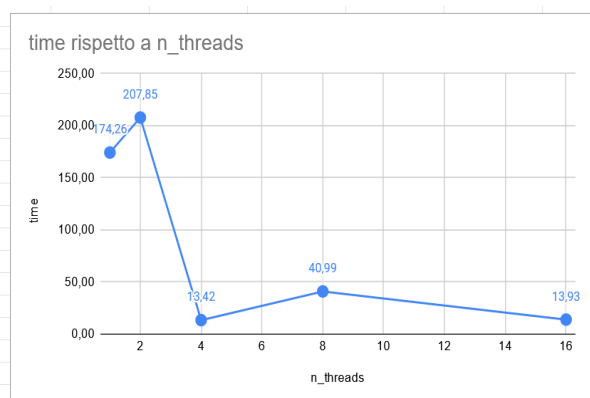


3.3. Test su un labirinto di dimensione 20x20

Proviamo adesso con un labirinto di dimensione più piccola (20x20):

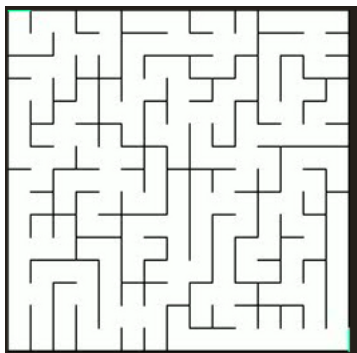


I risultati in questo caso, sono riportati nel seguente grafico:



In entrambi i casi che abbiamo visto precedentemente abbiamo prestazioni migliori se scegliamo un adeguato numero di thread.

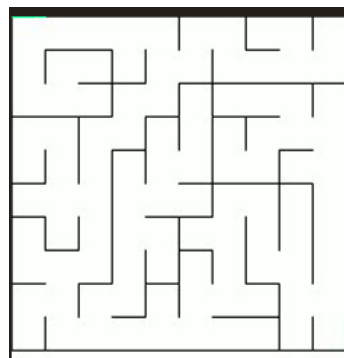
3.4. Test su un labirinto di dimensione 15x15



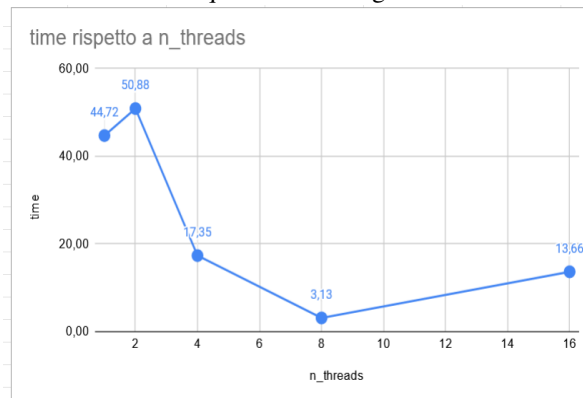
Che ha prodotto i risultati seguenti:

3.5. Test su un labirinto di dimensione 10x10

Proviamo adesso a effettuare gli stessi test per un labirinto 10x10:



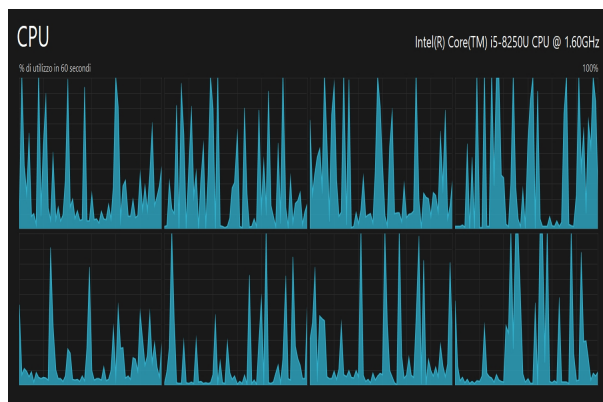
Abbiamo ottenuto questa volta i seguenti risultati:



4. Conclusion

L'approccio multithreading è considerevolmente più veloce rispetto all'approccio sequenziale. Questo risulta ragionevole, in quanto più particelle sono implicate nel processo, e più è plausibile supporre che l'algoritmo termini in minor tempo.

4.1. Esecuzione sequenziale



4.2. Esecuzione parallela

