UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Tesina Finale di

Algoritmi e Strutture Dati

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2023-2024

Dipartimento di Ingegneria

docente

Prof. Emilio Di Giacomo

A* Pathfinding

Implementazione di algoritmi per la generazione di grafi e ricerca del cammino minimo

Studente

311462 **Andrea Zonghi** andrea.zonghi@studenti.unipg.it

Data ultimo aggiornamento: 3 giugno 2024

0. Indice

1. Descrizione del problema

Nella teoria dei grafi la ricerca del cammino minimo tra due vertici di un grafo $G = (V, E)$ consiste nel trovare un percorso che va dal primo al secondo tale che la somma dei pesi degli archi che lo costituiscono sia minima. La proprietà su cui si fondano gli algoritmi per il calcolo dei cammini minimi è che il sottocammino di un cammino minimo è anch'esso un cammino minimo. Inoltre alcuni algoritmi restringono i pesi ad essere solamente positivi, oppure accettano archi con peso anche negativo ma poi segnalano la presenza di cicli negativi che impediscono di calcolare la soluzione corretta.

Abbiamo diverse varianti di questo problema, tra cui cercare: il cammino minimo tra un vertice e tutti gli altri appartenenti al grafo (sorgente unica), tra ciascun vertice ed una singola destinazione (destinazione unica, uguale al precedente invertendo la direzione degli archi), tra una coppia di vertici e tra tutte le coppie di vertici.

Il problema su cui ci concentreremo in questa tesina è quello del cammino minimo tra due vertici in grafi con diverse caratteristiche (sparsi, densi, orientati e non orientati).

1.1 Algoritmi di Pathfinding Analizzati

Esistono molti algoritmi di pathfinding tra due nodi con pesi non uniformi ed hanno tutti caratteristiche e realizzazioni differenti. Tra i più noti troviamo l'algoritmo di Dijkstra che si concentra sulla risoluzione del problema della sorgente unica, non accettando pesi degli archi negativi. Nella versione che utilizza heap di Fibonacci è al momento l'algoritmo più veloce (asintoticamente) conosciuto per questa computazione, considerando grafi diretti arbitrari e pesi potenzialmente illimitati, avendo una complessità temporale pari a $\Theta(|E|+|V|\log|V|)$. Per casi più particolari ci sono specializzazioni adatte dell'algoritmo che lo rendono più efficiente in quelle specifiche condizioni. L'idea generale del funzionamento dell'algoritmo è quella di partire dal nodo sorgente ed assegnare a questo un peso 0, mentre a tutti gli altri nodi si assegna una distanza inizialmente infinita. Poi ad ogni passaggio si seleziona il nodo con il peso minore dall'insieme dei nodi non ancora visitati e si aggiornano le distanze di tutti i nodi adiacenti a questo, applicando anche la tecnica del rilassamento [2], e si tiene traccia del percorso per raggiungere questi nodi. In seguito si marca il nodo scelto come visitato e si rimuove dall'insieme dei nodi non visitati. Si ripetono quindi questi passaggi finché tutti i nodi vengono visitati. Infine si possono ricostruire le distanze dai nodi alla sorgente utilizzando i dati raccolti durante l'aggiornamento della distanza.

Un altro algoritmo in parte simile per risolvere lo stesso problema è quello di Bellman e Ford. Questo è più versatile di Dijkstra sotto un certo punto di vista, in quanto accetta che alcuni archi abbiano peso negativo. La presenza di cicli negativi

raggiungibili dalla sorgente fa però sì che non esistano cammini minimi in quanto basterebbe attraversare ripetutamente il ciclo per diminuire il costo totale. In questo caso Bellman-Ford rileva il ciclo negativo e lo segnala. Il funzionamento è diverso da Dijkstra in quanto si effettua il rilassamento di ogni arco del grafo per $V - 1$ volte, dove V è il numero di nodi del grafo. Dopo questo processo si fa un ulteriore rilassamento di tutti gli archi e se si trova ancora un miglioramento nelle distanze questo significa che ci sono cicli di peso negativo raggiungibili dal nodo sorgente. Per quanto riguarda l'efficienza questo è sicuramente meno efficiente del precedente in quanto richiede, nel caso peggiore, $\Theta(|V||E|)$ operazioni, dove V è il numero dei nodi ed E è il numero degli archi.

Entrambi questi due algoritmi precedentemente analizzati si possono adattare per cercare il percorso minimo tra un nodo sorgente ed una destinazione, basta usare come condizione di terminazione l'aver o meno determinato la distanza minima del nodo di destinazione.

Il prossimo algoritmo invece è specifico per il problema del pathfinding tra tutte le coppie di vertici ed è l'algoritmo di Floyd e Warshall, che accetta anche archi con peso negativo (non cicli negativi però). Per quanto riguarda il funzionamento si costruisce una matrice delle distanze di dimensione VxV e si inizializzano tutte le distanze da un nodo a sé stesso uguali a zero. La distanza tra due nodi collegati da un arco è rappresentata come l'elemento [i][j] della matrice ed è impostata al peso conosciuto di quell'arco. Nel caso in cui non ci sia un arco tra due nodi, la distanza è inizialmente impostata ad infinito. L'algoritmo quindi esegue un numero V di iterazioni ed in ciascuna iterazione k considera tutti i nodi *i* e *j*. Per ogni coppia *(i,j)* si controlla se passando per il nodo k si ha un percorso più breve rispetto a quello già conosciuto e in caso si aggiorna. Al termine delle V operazioni quindi la matrice delle distanze contiene la distanza minima tra tutte le coppie dei nodi. Per quanto riguarda la complessità abbiamo un tempo $O(V^3)$ ed un'occupazione di spazio pari a $O(V^2)$.

Anche quest'ultimo algoritmo si può usare per calcolare il percorso minimo tra due vertici, ma sarebbe particolarmente inefficiente in quanto abbiamo una sola coppia nodi.

Arriviamo quindi infine all'algoritmo A* che risolve specificatamente il problema di pathfinding che stiamo analizzando. Questo combina l'algoritmo di Dijkstra con la ricerca best-first, utilizzando una funzione euristica per la stima della distanza del nodo dalla destinazione. Questa in pratica aiuta l'algoritmo a muoversi efficientemente verso la destinazione. La scelta dell'euristica è fondamentale e questa deve essere ammissibile, cioè non deve mai sovrastimare il reale costo per garantire che il percorso stimato sia ottimo. Il funzionamento è praticamente lo stesso di Dijkstra, ma per ogni nodo *n* si calcola il costo stimato *f(n)* come somma del costo esatto del percorso dal nodo iniziale al nodo n più la stima euristica del costo dal nodo n alla destinazione. Si prende quindi dall'insieme dei nodi da visitare quello con il costo stimato *f(n)* minore e si esplora, aggiornando i nodi ad esso adiacenti. Per ogni nodo vicino *m* si calcola un nuovo valore *g(m)* come somma del costo esatto del percorso del nodo iniziale al nodo n più il costo dell'arco che collega n ad m. Se questo nuovo valore è inferiore a quello precedentemente noto per m, si aggiorna g(m) e si calcola f(m), impostando anche n come predecessore di m per poter ricostruire il

percorso. Infine si aggiunge (o aggiorna) m nell'insieme dei nodi da visitare. L'algoritmo termina quando il nodo di destinazione è quello con il costo stimato f(n) minore e viene scelto come prossimo nodo da esplorare oppure quando non ci sono più percorsi da visitare. La complessità temporale dipende fortemente dalla qualità dell'euristica utilizzata e, con una buona funzione, può essere significativamente più veloce di Dijkstra. Se il grafo è un albero e con una funzione f che rispetta la condizione $|f(x) - f^*(x)| = O(\log f^*(x))$, dove f^* è l'euristica ottimale cioè l'esatta distanza per andare da x all'obiettivo, la complessità temporale è **polinomiale**. Un problema potrebbe però essere la complessità spaziale $O(|V|)$, in quanto nel caso peggiore può essere necessario tenere tutti i nodi in memoria.

Utilizzeremo quindi questo ultimo algoritmo, A*, per il problema della ricerca del cammino minimo tra due nodi che stiamo studiando.

1.2 Algoritmi di Generazione Grafi Analizzati

Un primo approccio alla generazione di grafi potrebbe essere quello di, dato un numero N^2 di nodi, generare un grafo a griglia creando una matrice NxN dove ogni cella (i, j) rappresenta un singolo vertice di coordinate i e j. Inoltre possiamo aggiungere le celle (i-1,j), (i,j+1), (i+1,j) e (i,j+1) in una lista o matrice di adiacenza per rappresentare gli archi ai quattro nodi vicini. In questo caso potremmo anche decidere arbitrariamente di rimuovere degli archi per ridurre la densità del grafo e bloccare alcuni passaggi.

Sarebbe sicuramente un metodo efficace, ma volendo utilizzare un tipo di grafo più generico analizziamo il modello di Erdős e Rényi [9] [6] (modello ER) per la generazione casuale di grafi. Ci sono due varianti del modello ER che sono per alcuni tratti simili:

- il modello $G(n, m)$ dove un grafo è scelto randomicamente dalla collezione contenente tutti i grafi che hanno **n** nodi e **m** archi e permutazioni diverse dei vertici sono considerate grafi diversi. Definito **T** il numero di grafi nella collezione, ogni grafo ha la stessa probabilità $P = 1/T$ di essere scelto.

- il modello $G(n, p)$ dove un grafo è costruito connettendo due nodi a caso. Ogni arco viene incluso nel grafo con probabilità indipendente **p**. Questo parametro di probabilità **p** può essere visto come una funzione di ponderazione in quanto, aumentando p da 0 ad 1, il modello includerà con probabilità crescente grafi con più archi e con probabilità sempre più bassa grafi con meno archi.

Una importante proprietà di quest'ultimo modello è che se $p < (1 - \epsilon)ln(n)/n$ allora un grafo $G(n, p)$ conterrà quasi sicuramente vertici isolati e quindi non sarà connesso mentre se $p > (1-\epsilon)ln(n)/n$ allora un grafo $G(n, p)$ sarà quasi sicuramente connesso. In entrambi i casi più è grande il numero di nodi n e più sarà preciso il risultato.

Vediamo quindi che $ln(n)/n$ è un punto critico per per la connettività di $G(n, p)$.

Utilizzeremo quindi questo secondo modello ER per quando sarà necessario generare grafi.

2. Algoritmi e Strutture Dati Implementati

Abbiamo quindi l'implementazione di due diversi algoritmi, uno per trovare il cammino minimo tra due nodi, e l'altro per la generazione di grafi a partire dal numero di vertici **n** e da una probabilità **p**. Viene inoltre implementato un metodo per creare un grafo a partire da due file di testo contenenti nodi ed archi in una specifica formattazione. Questo verrà poi usato per eseguire set di esperimenti con A*. Tale funzione sarà utile in quanto viene implementata anche l'opzione di creare un singolo grafo con il modello ER e salvarlo in due file di testo nella formattazione usata dal metodo precedente.

Il grafo che viene generato, letto o su cui si lavora è rappresentato da un array di nodi, implementati nel progetto nella classe *Node*. Gli archi invece sono modellati dalla classe *Edges*.

In *Nodes* troviamo gli attributi di base del nodo che sono: un identificativo, la coordinata X, la coordinata Y ed una lista di archi. Quest'ultima è la lista di adiacenza che serve a rappresentare il grafo ed è stata scelta al posto della matrice di adiacenza perché nell'implementazione poi successiva di A* si avrà necessità di scorrere tutti i nodi adiacenti ad uno dato, operazione in cui è più veloce della sua controparte.

Viene scelto di utilizzare una lista di archi invece della più canonica lista di adiacenza fatta di nodi perché in questa implementazione abbiamo i pesi solamente sugli archi. Questo potrebbe essere utile nel caso in cui, per esempio, dovremmo utilizzare l'algoritmo su grafi che rappresentano strade e ci permetterebbe di cambiare il peso di uno specifico tratto di strada in base alle variabili del caso (eg: condizioni di viabilità, limiti di velocità bassi) in modo piuttosto semplice. La lista di nodi viene implementata usando un oggetto della classe *ArrayList* di Java [3].

Per quanto riguarda la classe *Edge* abbiamo i nodi di inizio, di fine ed il costo.

Viene poi implementata una classe *Result*, utilizzata poi per ritornare i risultati dell'esecuzione di A*. In essa troviamo il percorso minimo cercato, modellato come lista di nodi, ed il suo costo complessivo.

2.1 Algoritmo A*, coda di priorità e mappa hash

Passando ad A* abbiamo innanzitutto una classe ausiliaria, chiamata *NodeComp*, che implementa l'interfaccia di Java *Comparable* [4], necessaria per ordinare gli elementi nella coda di priorità usata dall'algoritmo. Qui troviamo un nodo, il suo predecessore, il costo dal nodo di partenza calcolato fino a quel momento ed il costo dal nodo di partenza stimato dall'euristica utilizzata. Altro punto degno di nota in questa classe è l'override del metodo *compareTo()*, necessario a far funzionare correttamente l'ordinamento della coda di priorità. Un altro metodo ausiliario utilizzato

nel codice che esegue A* è il metodo statico *euclideanDistance()* che rappresenta la nostra euristica e calcola la distanza euclidea [7] tra due punti in uno spazio bidimensionale. Per altre necessità si potrebbero usare altre euristiche, come ad esempio la distanza di Manhattan [10] se si utilizzasse il grafo a griglia visto nella sezione 1.2.

Arriviamo quindi ad implementare l'algoritmo A*, che come visto prima, è praticamente una versione dell'algoritmo di Dijkstra ottimizzato per la ricerca verso un solo punto e che utilizza una funzione euristica per dirigersi nella giusta direzione. Come openSet viene scelta una coda di (min-)priorità, implementata con la classe *PriorityQueue* di Java in quanto gli elementi al suo interno vengono ordinati secondo il costo stimato, attributo usato come termine di paragone nell'override del metodo *compareTo()*. L'openSet invece viene realizzato utilizzando la classe *HashMap* di Java in quanto le operazioni *get()* e *put()* sono realizzate in tempi costanti con questa struttura dati. La *get()* viene utilizzata sotto forma di *containsKey()* per vedere se di un sono già stati presi in considerazione tutti i nodi ad esso adiacenti, mentre la *put()* viene utilizzata per aggiungere elementi alla mappa. Vediamo in dettaglio lo pseudocodice nella pagina successiva.

Algorithm 1 Trova Percorso con A*

```
1: Input Nodo iniziale start, nodo finale goal
2: Output Percorso più breve path, costo totale totalPathCost
3: inizializzo una PriorityQueue vuota
4: inizializzo una HashMap vuota
5: startNode = nuovo NodeComp con costo accumulato=0 e costo stimato
6: aggiungo startNode all'openSet
7: while openSet non è vuoto do
8:     next = nodo in cima all'openSet
9:     if next è già nel closedSet then
10:        continue
11:    end if
12:    if next è goal then
13:        totalPathCost = costo fino a goal
14:        path = ricostruisco il percorso e lo inverto
15:        return path e totalPathCost
16:    end if
17:    for ogni arco ∈ archi di next do
18:        inzializzo nodo adiacente corrispondente
19:        calcolo costo accumulato e costo stimato per il nodo adiacente
20:        if il nodo adiacente è già in closedSet then
21:            passo alla prossima iterazione del for
22:        end if
23:        toNode = nuovo NodeComp con costo accumulato e costo stimato
24:        aggiungo toNode all'openSet
25:    end for
26:    metto next nel closedSet
27: end while
28: return null
```

L'algoritmo (Algorithm 1) viene implementato nel metodo centrale della classe *AStar*, nel metodo *findPath()*. La prima cosa che si fa è inizializzare le due strutture dati che usiamo per tenere traccia dei nodi da visitare (*openSet*) e dei nodi già visitati (*closedSet*) (righe 3,4 dello pseudocodice). Quindi si crea *startNode*, un nuovo NodeComp a partire dal nodo start, con un costo accumulato uguale a 0 e un costo stimato uguale alla distanza euclidea tra *start* e *goal* (riga 5). Poi si aggiunge *startNode* all'*openSet* e si entra nel ciclo while, che finirà nel momento in cui la coda di priorità sarà vuota. Qui si controlla prima se il nodo estratto dalla cima della coda, *next*, sia già nel *closedSet* e, in caso, si salta alla prossima iterazione del ciclo (righe 9-11).

Si controlla poi se si è raggiunto il *goal*, nel qual caso viene ricostruito il percorso da *start* a *goal* e viene ritornato, insieme al costo totale per arrivarci (righe 12-16). Se invece non si è arrivati all'obiettivo allora si prendono in considerazione tutti i nodi adiacenti, uno ad uno. Per il nodo adiacente corrente si calcolano il costo accumulato e stimato, si controlla quindi che non sia già nel *closedSet* e si inserisce nell'*openSet* sotto forma di NodeComp con costo accumulato e stimato

appena calcolati (righe 17-24).

A questo punto si procede fino a quando non sono finiti i nodi adiacenti e in quel caso si esce dal ciclo for e si inserisce il nodo di cui si consideravano gli adiacenti nel *closedSet* (righe 25,26). Tutto questo viene ripetuto per ogni nodo estratto dalla coda di priorità e, nel caso questa arrivi ad essere vuota senza che si sia mai raggiunto il *goal*, si esce dal ciclo while e si ritorna **null**.

2.2 Algoritmo di generazione grafi con modello ER

La generazione dei grafi viene implementata nel metodo *generateErdosRenyiGraph()* della classe *GraphGeneration* che, dati n nodi ed una probabilità p, per ogni coppia di nodi decide in base a p se generare un arco che li colleghi. Non vengono utilizzate particolari strutture dati in questo processo, solo le classi di base di Java, la classe *Random* dal package *util* di Java e le classi *Node* ed *Edges* viste in precedenza. Vediamo brevemente lo pseudocodice:

Algorithm 2 Genera Grafo di Erdős-Rényi

```
1:  Input Numero di nodi n, probabilità p
2:  Output Array di nodi nodes
3:  if check sugli input n e p then
4:      throw IllegalArgumentException
5:  end if
6:  rand = nuovo Random
7:  nodes = nuovo Node[n]
8:  for i = 0 to n − 1 do
9:      nodes[i] = nuovo nodo con coordinate casuali (entro bound definiti)
10: end for
11: for j = 0 to n − 1 do
12:     for k = j + 1 to n − 1 do
13:         if rand.nextDouble() < p then
14:             calcolo del costo dell'arco come distanza euclidea tra i due nodi
15:             if arco non è già presente then
16:                 creiamo l'arco tra i nodi
17:             end if
18:         end if
19:     end for
20: end for
21: return nodes
```

L'algoritmo inizia con dei controlli sui valori inseriti dall'utente, n e p, e si ritorna un'eccezione nel caso in cui non siano validi (righe 3-5). Si inizializzano quindi un oggetto della classe *Random* e l'array di nodi *nodes* che verrà poi ritornato al termine dell'esecuzione (righe 6,7). Vengono poi assegnate ad ogni nodo delle coordinate casuali ed un'etichetta (righe 8-10). In seguito per ogni coppia di nodi scegliamo se

creare o meno un arco tra di loro in base a p e, in caso l'arco non sia già presente, lo aggiungiamo ad entrambi i nodi (righe 11-19). Infine ritorniamo l'array di nodi.

2.3 Altre classi degne di nota

Le ultime due classi implementate sono *Utilities*, in cui vengono scritte subroutine utilizzate frequentemente, ed *Experiments*, in cui si trova il *main()* dell'intero progetto.

Nella prima troviamo tutti i metodi che servono ad ottenere un input dall'utente a tempo di esecuzione: *getInputFileNames()*, *getInputNodesNumber()*, *getInputProbability()* e *getInputExperimentsNumber()*. In questi vengono implementati loop che servono a far reinserire l'input richiesto nel caso in cui l'utente inserisca qualcosa di errato o non valido. Troviamo inoltre il metodo *euclideanDistance()* che calcola l'euristica utilizzata in A*, la distanza euclidea tra una coppia di punti nel piano. Ci sono poi i due metodi *writeResultToFile()* e *WriteErdosRenyiGraphToFile()* che servono a scrivere su file di testo, rispettivamente, i risultati di un set di esperimenti ed il grafo generato tramite l'algoritmo visto in precedenza. Infine vi troviamo il metodo *printGraph()* usato per stampare, a partire da un array di nodi, il grafo rappresentato con liste di adiacenza e *loadGraphFromFiles()* che serve ad estrapolare un grafo da due file di testo.

Nella classe *Experiments* invece troveremo il *main()* che realizza il modo in cui viene chiesto all'utente di inserire una scelta e, in base a questa, utilizzare una delle tre funzionalità del programma. Vi troviamo inoltre alla fine i calcoli dei risultati degli esperimenti che verranno utilizzati in seguito.

3. Analisi della Complessità

Analizziamo ora la complessità dello pseudocodice dei due algoritmi, il path-finding con A* e la generazione dei grafi con il modello ER. In entrambi i casi si useranno n per il numero di nodi ed m per il numero di archi.

3.1 Complessità di A*

Partiamo analizzando la complessità di A*, di cui si riporta lo pseudocodice per comodità.

Algorithm 3 Trova Percorso con A*

```
1: Input Nodo iniziale start, nodo finale goal
2: Output Percorso più breve path, costo totale totalPathCost
3: inizializzo una PriorityQueue vuota
4: inizializzo una HashMap vuota
5: startNode = nuovo NodeComp con costo accumulato=0 e costo stimato
6: aggiungo startNode all'openSet
7: while openSet non è vuoto do
8:     next = nodo in cima all'openSet
9:     if next è già nel closedSet then
10:        continue
11:    end if
12:    if next è goal then
13:        totalPathCost = costo fino a goal
14:        path = ricostruisco il percorso e lo inverto
15:        return path e totalPathCost
16:    end if
17:    for ogni arco ∈ archi di next do
18:        inzializzo nodo adiacente corrispondente
19:        calcolo costo accumulato e costo stimato per il nodo adiacente
20:        if il nodo adiacente è già in closedSet then
21:            passo alla prossima iterazione del for
22:        end if
23:        toNode = nuovo NodeComp con costo accumulato e costo stimato
24:        aggiungo toNode all'openSet
25:    end for
26:    metto next nel closedSet
27: end while
28: return null
```

Le inizializzazioni della PriorityQueue, della HashMap e del nodo di partenza hanno tutte costo $O(1)$, ma aggiungere *startNode* alla coda ha costo $O(log(n))$ [5]. Si passa ora al ciclo while principale e l'estrazione del nodo con priorità più alta (costo stimato più basso) è fatta con l'operazione *openSet.poll()* che ha un costo $O(log(n))$. Le successive verifiche, fatte con i metodi *containsKey()* e *equals()* hanno entrambe costo $O(1)$.

La ricostruzione del percorso ha, nel peggiore dei casi, costo $O(n)$ perché potrebbe dover attraversare tutti i nodi, mentre invertire la lista di nodi appena trovata utilizzando *Collections.reverse()* avrà ugualmente costo $O(n)$ nel peggiore dei casi. Si arriva quindi al ciclo for che per ogni arco incidente inizializza il nodo adiacente corrispondente e ne calcola il costo cumulativo e quello stimato con costo complessivo $O(1)$. Anche la verifica nella mappa per essere sicuri che il nodo non sia già stato esplorato fatta con il metodo *containsKey()* ha costo $O(1)$ come visto precedentemente.

Infine l'inizializzazione di un nuovo NodeComp ha costo $O(1)$, l'aggiunta del nodo alla coda di priorità ha costo $O(log(n))$ e, uscendo dal ciclo while principale, l'aggiunta della coppia $\langle$*Node,NodeComp*$\rangle$ alla mappa hash ha costo $O(1)$.

Quindi riassumendo si ha che, nel caso peggiore, ogni nodo verrà aggiunto alla coda di priorità una volta con costo totale $O(nlog(n))$, mentre l'esplorazione ed eventuale aggiunta di tutti i nodi adiacenti alla coda avrà un costo $O(mlog(n))$, dipendente dal numero di archi. In conclusione essendo l'istruzione dominante l'aggiunta di un elemento alla coda di priorità si avrà che il costo complessivo dell'algoritmo, nel caso peggiore, sarà $O((n+m)log(n))$.

È interessante notare che la complessità temporale nel caso peggiore è la stessa dell'algoritmo di Dijkstra[8] in quanto l'algoritmo si sarà effettivamente comportato allo stesso modo di quest'ultimo. Come è stato però discusso nel paragrafo sull'analisi degli algoritmi di pathfinding la complessità temporale di A* è polinomiale nel caso in cui il grafo sia un albero e si utilizzi una funzione euristica f che rispetti la condizione $|f(x) - f^*(x)| = O(\log f^*(x))$, dove f* è l'esatta distanza dall'obiettivo.

3.2 Complessità della generazione del grafo

Viene di seguito riportato lo pseudocodice dell'algoritmo di generazione per comodità.

Algorithm 4 Genera Grafo di Erdős-Rényi

```
1: Input Numero di nodi n, probabilità p
2: Output Array di nodi nodes
3: if check sugli input n e p then
4:     throw IllegalArgumentException
5: end if
6: rand = nuovo Random
7: nodes = nuovo Node[n]
8: for i = 0 to n − 1 do
9:     nodes[i] = nuovo nodo con coordinate casuali (entro bound definiti)
10: end for
11: for j = 0 to n − 1 do
12:     for k = j + 1 to n − 1 do
13:         if rand.nextDouble() < p then
14:             calcolo del costo dell'arco come distanza euclidea tra i due nodi
15:             if arco non è già presente then
16:                 creiamo l'arco tra i nodi
17:             end if
18:         end if
19:     end for
20: end for
21: return nodes
```

I controlli sugli input hanno costo $O(1)$, mentre la creazione dell'array di nodi avrà costo $O(n)$ ed il conseguente ciclo for per assegnare le coordinate a tutti gli elementi avrà ugualmente costo $O(n)$.

Per quanto riguarda invece il ciclo annidato successivo abbiamo che l'istruzione dominante è la creazione dell'arco che ha costo $O(1)$ e viene eseguita per ogni coppia di nodi, avendo quindi costo complessivo $O(n^2)$. Il controllo con la probabilità p, il calcolo della distanza euclidea ed il controllo per l'esistenza dell'arco sono operazioni che hanno anch'esse tutte probabilità $O(1)$ ed essendo che vengono ripetute per ogni coppia di nodo avranno un costo complessivo $O(n^2)$.

Concludendo quindi il costo complessivo in termini di tempo sarà dato dalle operazioni nel ciclo annidato, per una complessità asintotica di $O(n^2)$.

4. Dati Sperimentali

Si vanno ora a condurre dei set di esperimenti per testare i le prestazioni degli algoritmi e strutture dati implementati ed utilizzati. Il numero di elementi per set verrà specificato ogni volta, così come le specifiche dell'esperimento.

4.1 Dataset reali di reti stradali

Si vanno ad analizzare i grafi estrapolati dai dataset forniti dall'università di Kahlert, Utah [1], che rappresentano reali reti stradali. In questo caso la probabilità p viene utilizzata per passare da un grafo orientato ad un grafo non orientato in quanto una $p = 0$ corrisponde ad un solo arco tra i due vertici indicati nel dataset, mentre $p = 1$ aggiungerà un arco nel senso opposto. Per i due dataset minori, relativi alle strade della città di Oldenburg (OL) e di San Joaquin County (TG) vengono effettuati set di 100000 esperimenti, mentre per quelli del North America (NA) e San Francisco Road Network i set sono da 10000 esperimenti ciascuno.

Si ricorda che in ogni singolo esperimento vengono scelti due nodi a caso, tra quelli nel grafo, come *start* e *goal* di A*. Questa prima parte di test viene fatta per vedere come variano i tempi di esecuzione di A* sia relativamente al numero di nodi nel grafo, che alla sua "direzionalità", descritta dalla variabilità della p. Si nota inoltre che tutti i grafi in analisi a questo punto sono sparsi, cioè in cui il numero di vertici è dello stesso ordine di grandezza del numero di archi.

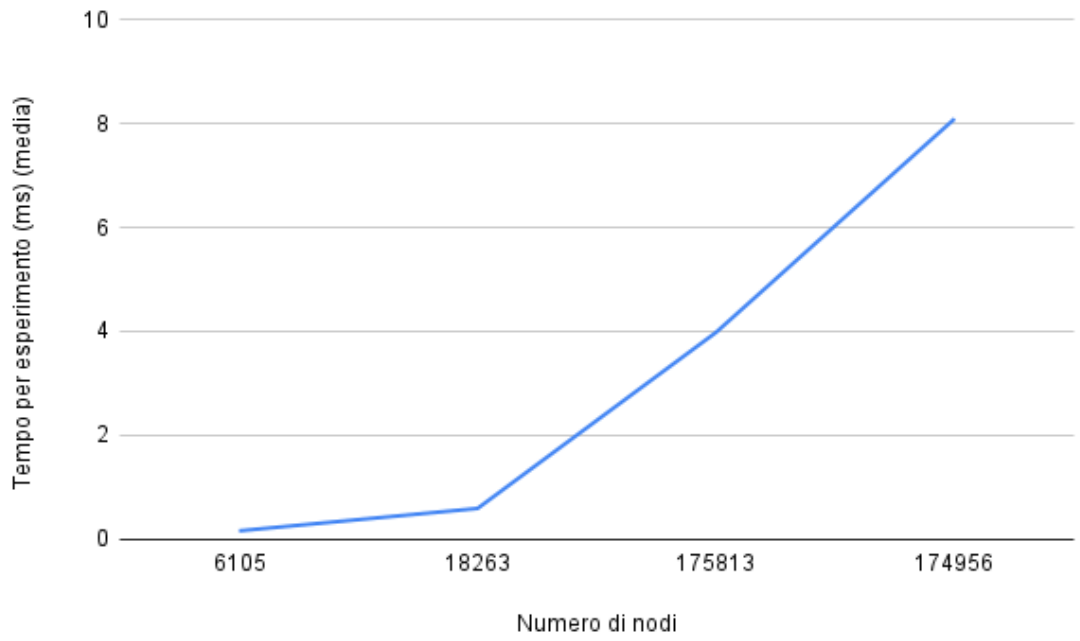

Figura 4.1: **Tempo per esperimento, in millisecondi, rispetto al numero di nodi**

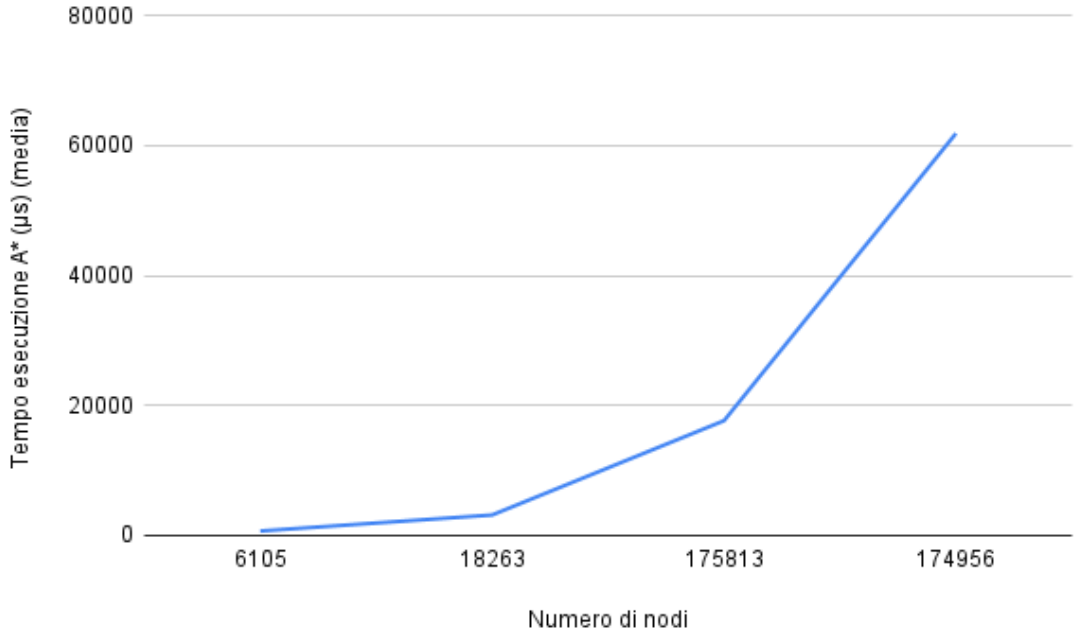

Figura 4.2: **Tempo per una singola esecuzione di A*, in microsecondi, rispetto al numero di nodi**

Si può notare come i grafici nelle figure 4.1 e 4.2 descrivano un andamento simile per gli stessi valori dei nodi, ad evidenziare quindi che il comportamento asintotico del codice scritto, in questo caso, sia quello dell'algoritmo di pathfinding stesso.
Per quanto riguarda la complessità si può vedere come il tempo di esecuzione si trovi tra una curva del tipo $y = nlog(n)$ ed un'altra del tipo $y = n$, ma servirebbero più dataset con numeri di vertici differenti per confermare.
Volendo vedere ora come cambiano i tempi di esecuzione relativamente all'utilizzo di un grafo orientato o non orientato, si vanno ad analizzare brevemente le figure di seguito.

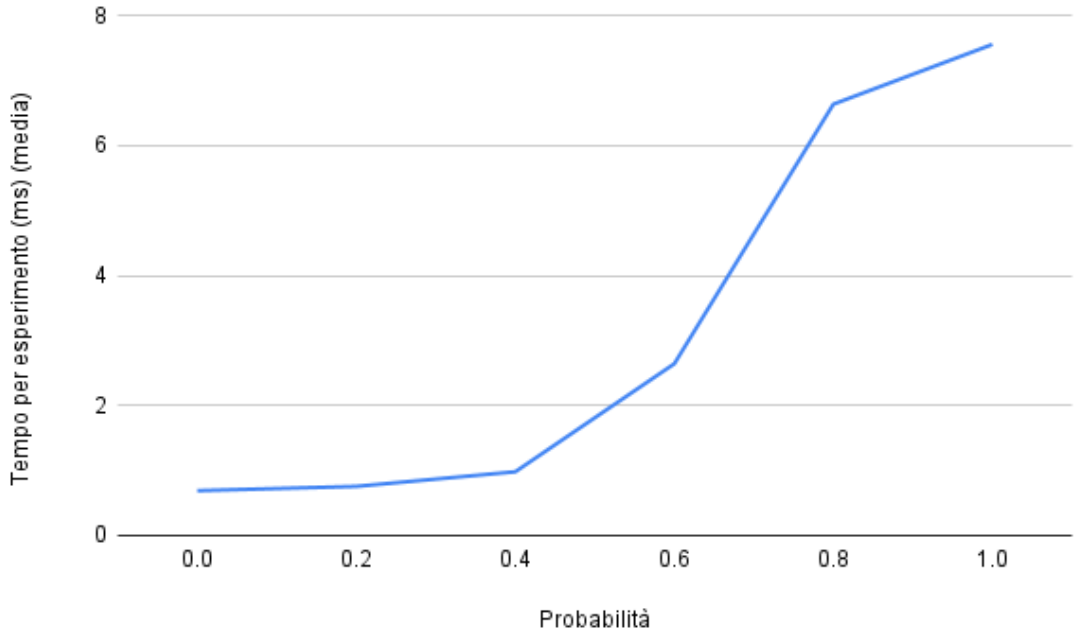

Figura 4.3: **Tempo per esperimento, in millisecondi, rispetto alla probabilità, aggregata tra dataset**

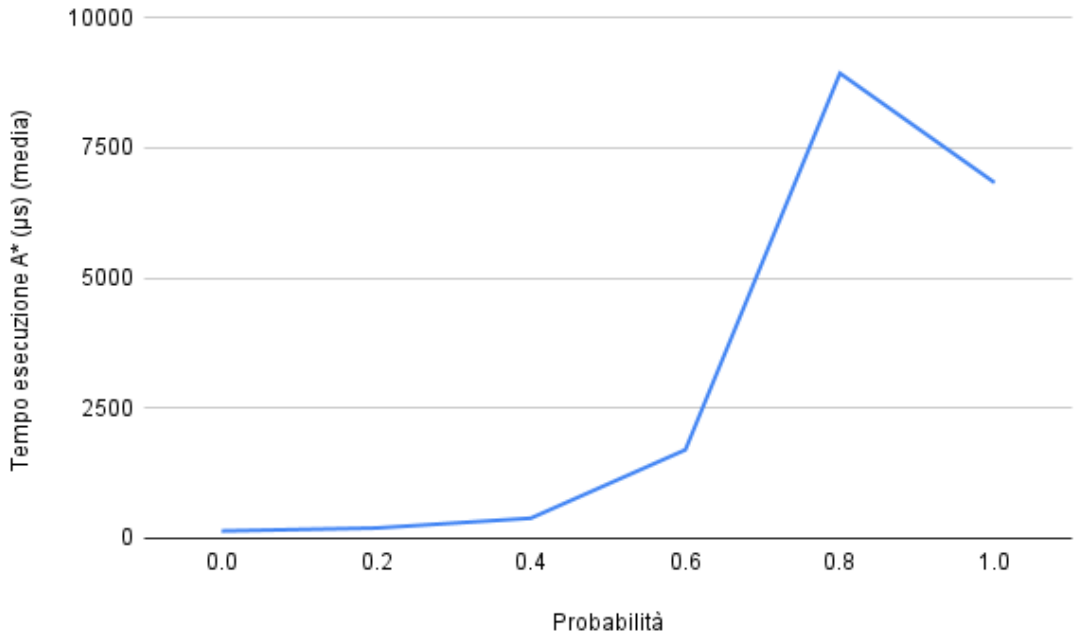

Figura 4.4: **Tempo per una singola esecuzione di A*, in microsecondi, rispetto alla probabilità, aggregata tra dataset**

Avendo ora sull'asse delle ascisse la probabilità, che in pratica rappresenta quanto il grafo sia vicino all'essere orientato o non orientato, si nota immediatamente come il doppio arco del grafo non orientato ($p = 1$) migliori i tempi della ricerca del cammino minimo mostrati in figura 4.4. Guardando anche alla figura 4.3 si vede

come il passaggio da $p = 0.8$ a $p = 1$ cambi l'ordine della curva, riducendone la complessità.

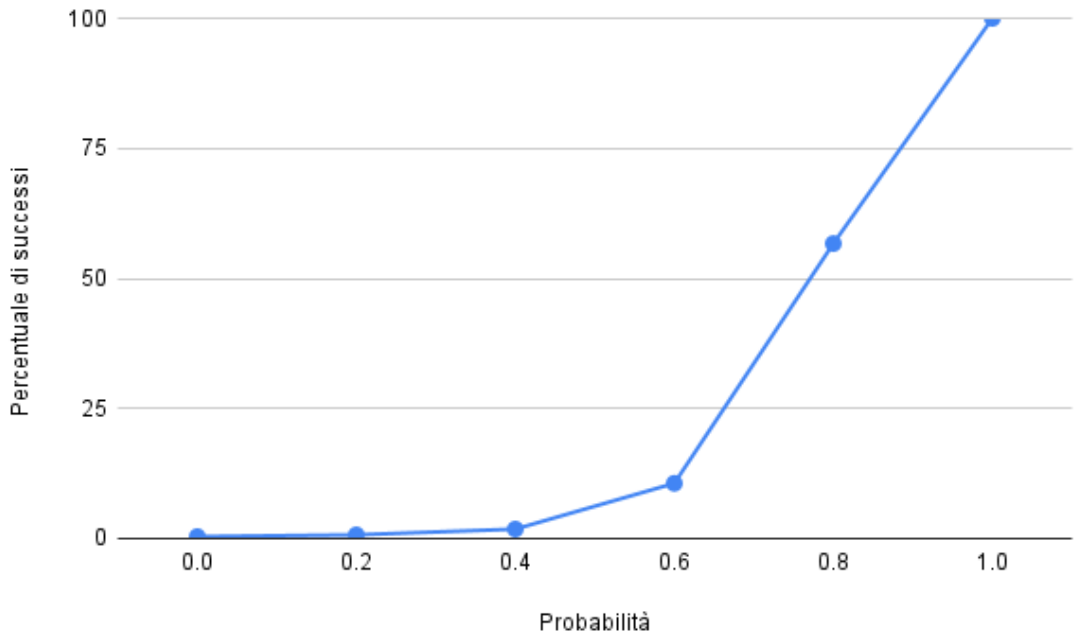

Figura 4.5: **Percentuale di successi rispetto alla probabilità, aggregata tra dataset**

Infine si analizza il cambiamento della percentuale di successi, definiti come il riuscire a trovare un percorso tra due nodi presi casualmente, in relazione alla probabilità. In questo caso si può notare il cambiamento sostanziale che ha passando da una $p = 0.6$ a $p = 0.8$ e ancora da $p = 0.8$ a $p = 1$.

4.2 Grafi da modello ER

In questa ultima sezione vengono invece analizzati i dati sperimentali relativi alla generazione dei grafi con il modello ER. In questo caso la probabilità p rappresenta la densità del grafo, con 0 per un grafo sparso ed 1 per archi tra tutte le coppie di nodi.
Dati i valori di p ed n, numero dei nodi nel grafo, per ogni set vengono eseguiti 1000 esperimenti che qui significano 1000 generazioni di grafi $G(n, p)$ diversi con conseguente scelta randomica dei nodi *start* e *goal*. Il valore di n varia da 25 a 500, con incrementi di 25, mentre p va da 0 ad 1 con incrementi di 0.1.
Con questi valori di p si nota che non c'è interesse nell'analizzare la percentuale di successo di A* in quanto, come visto nella sezione sugli algoritmi di generazione di grafi, per n abbastanza grandi la probabilità (densità) necessaria a far sì che il grafo sia connesso è molto piccola.

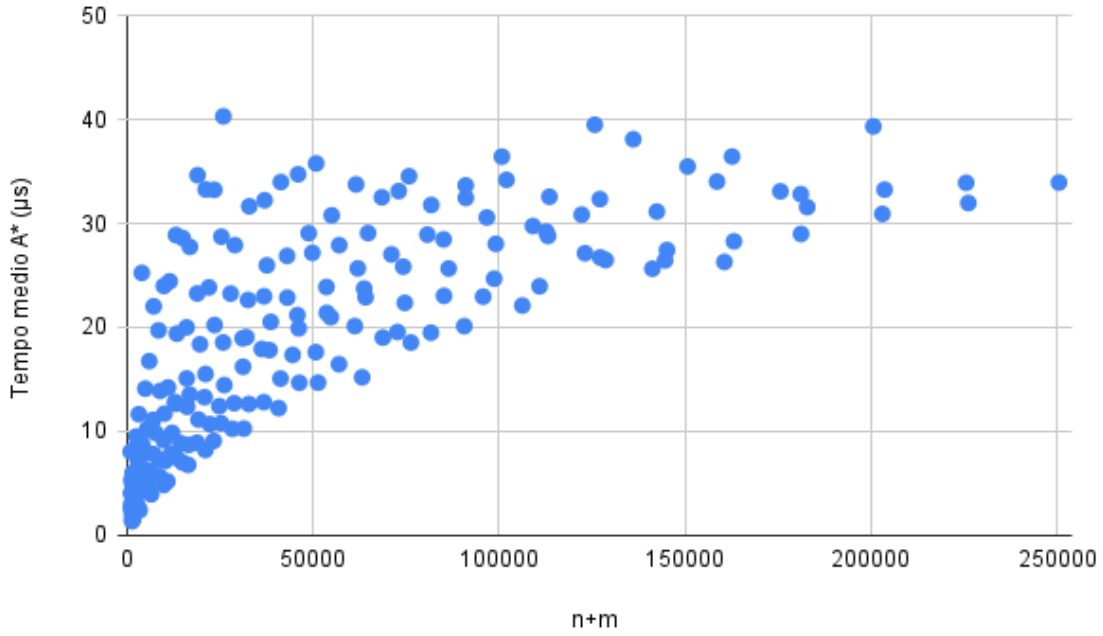

Figura 4.6: **Tempo per una singola esecuzione di A*, in microsecondi, rispetto alla somma di nodi e archi**

Volendo qui evidenziare il rapporto tra densità e tempo di esecuzione, nella figura 4.6 si fa un'analisi leggermente diversa rispetto alla precedente e si vede come cambia il tempo necessario per una singola esecuzione di A* in base alla densità variabile. Si nota che la tendenza della curva, per n+m grandi, è simile a quella di $log(n)$.

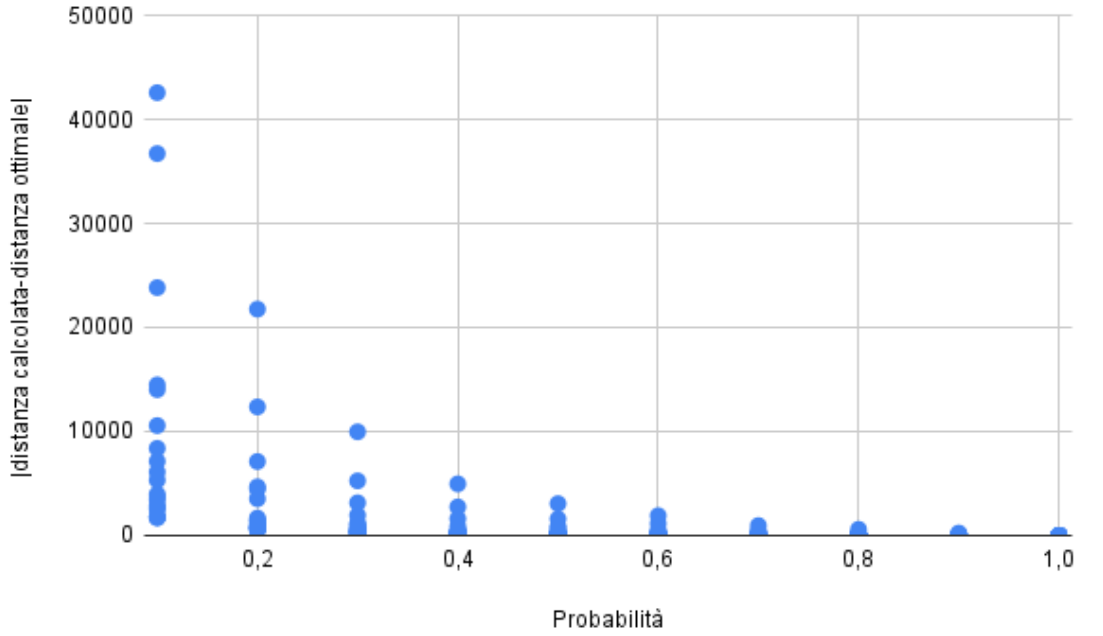

Figura 4.7: **Modulo della differenza tra distanza calcolata e distanza ottimale rispetto alla probabilità**

In figura 4.7 invece viene evidenziato come al crescere della probabilità, e quindi della densità del grafo, il modulo della differenza tra distanza calcolata e distanza ottimale tenda scendere fino ad annullarsi. In applicazioni reali sarebbe quasi impossibile trovarsi nella situazione dove questo valore è uguale a zero in quanto significherebbe che esiste una strada che, presi qualsiasi due punti, li collega. In questo caso il valore uguale a zero è dovuto allo spazio in cui vengono generate le coordinate X ed Y dei singoli nodi, ristretto 100000x100000 a tempo di esecuzione, che evidentemente causa un addensamento di vertici anche con un n massimo analizzato uguale a 500. In realtà come visto nella relativa sezione, la condizione $|f(x)-f^*(x)| = O(\log f^*(x))$ è parte dei requisiti per avere una complessità polinomiale con A*. Di conseguenza si ipotizza che, potendo utilizzare uno spazio abbastanza ampio ed un livello di dettaglio maggiore per la probabilità ed essendo la l'euristica utilizzata efficace, il valore del modulo della differenza tra le distanze tenderà al logaritmo della distanza stimata.

5. Bibliografia

[1] Real Datasets for Spatial Databases: Road Networks and Category Points — users.cs.utah.edu. https://users.cs.utah.edu/~lifeifei/SpatialDataset.htm. [Accessed 03-06-2024].

[2] baeldung. Edge Relaxation in Dijkstra's Algorithm — Baeldung on Computer Science — baeldung.com. https://www.baeldung.com/cs/dijkstra-edge-relaxation, 2024. [Controllato il 31-05-2024].

[3] Oracle. ArrayList (Java Platform SE 8) — docs.oracle.com. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. [Controllato il 01-06-2024].

[4] Oracle. Comparable (Java Platform SE 8) — docs.oracle.com. https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html. [Controllato il 01-06-2024].

[5] Oracle. PriorityQueue (Java Platform SE 8) — docs.oracle.com. https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html. [Controllato il 02-06-2024].

[6] A. Rényi P. Erdős. On random graphs i. https://www.renyi.hu/~p_erdos/1959-11.pdf, 1959. [Controllato il 30-05-2024].

[7] Wikipedia. Distanza euclidea - Wikipedia — it.wikipedia.org. https://it.wikipedia.org/wiki/Distanza_euclidea#Distanza_bidimensionale, 2023. [Controllato il 01-06-2024].

[8] Wikipedia. Dijkstra's algorithm - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, 2024. [Accessed 02-06-2024].

[9] Wikipedia. Erdős–Rényi model - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model, 2024. [Controllato il 30-05-2024].

[10] Wikipedia. Geometria del taxi - Wikipedia — it.wikipedia.org. https://it.wikipedia.org/wiki/Geometria_del_taxi#:~:text=In%20matematica%2C%20la%20geometria%20del,delle%20differenze%20delle%20loro%20coordinate., 2024. [Controllato il 30-05-2024].