Tesina Finale di

Algoritmi e strutture dati

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2021-2022

Dipartimento di Ingegneria

docente

Prof. Emilio Di Giacomo

Implementazione dell'algoritmo A*

studenti

321310 **Alex Ardelean** alexnicolae.ardelean@studenti.unipg.it

Data ultimo aggiornamento: 27 agosto 2022

0. Indice

1. Descrizione del problema

Il cammino minimo tra due vertici di un grafo è il cammino di peso minimo tra tutti i cammini che connettono i due vertici.

Il problema della determinazione del cammino a costo minimo attraverso un grafo è di interesse in molte aree dell'informatica. Ad esempio: instradamento del traffico telefonico/internet, layout di circuiti stampati, ...

Nel 1968 in [1] "*Formal Basis for the Heuristic Determination of Minimum Cost Paths*" è stato presentato l'*algoritmo A** che risolve tale problema. Esso utilizza informazioni sul dominio del problema al fine di trovare il cammino a costo minimo a partire da un nodo sorgente fino ad un nodo destinazione attraverso un grafo. É possibile dimostrare che tale algoritmo, sotto certe condizione, risulta essere ottimale nel senso che esso esamina il minor numero di nodi necessari a garantire una soluzione al problema.

Da notare che tale risultato poteva essere ottenuto anche con l'*algoritmo di Dijkstra (1959)*, con esso infatti è possibile ottenere non solo il percorso minimo tra un punto di partenza e uno di arrivo ma l'intero albero dei cammini minimi, cioè tutti i percorsi minimi tra un punto di partenza e tutti gli altri punti del grafo. A* però, nel trovare il singolo percorso da sorgente a destinazione, risulta essere più efficiente dato che esamina meno nodi rispetto a Dijkstra.

1.1 A* su reti stradali

In [1] viene presentato come tipica applicazione dell'algoritmo A* la risoluzione del seguente problema:

Dato un insieme di città e strade che connettono coppie di città si trovi la sequenza di strade che rappresenta il percorso più corto a partire da una città sorgente ad una città goal.

Per questo tipo di problema infatti l'algoritmo sfrutta il fatto che per ogni coppia di città il percorso stradale più breve non può essere più corto della distanza in linea d'aria tra di esse.

Per poter risolvere tale problema tramite l'algoritmo A* è necessario rappresentare l'insieme di città e strade tramite un grafo. In particolare si possono utilizzare grafi non orientati e pesati (con pesi non negativi) in cui:

- i nodi rappresentano le città, ogni nodo avrà perciò una coordinata (x, y) nel piano.
- gli archi rappresentano le strade e hanno peso pari alla distanza euclidea tra i due estremi.

Per valutare l'efficacia e l'efficienza dell'algoritmo A* sulla tipologia di grafo precedentemente descritta si effettueranno varie prove sia su grafi generati casualmente che su reti stradali reali [3].

1.2 A* su Occupancy Grid

Oltre che nel trovare il cammino minimo tra due città in una rete stradale, l'algoritmo A* è ampiamente utilizzato nella robotica. Ad esempio esso viene utilizzato per generare il cammino minimo all'interno di *Occupancy Grid*. Le Occupancy Grid sono matrici bidimensionali che vengono utilizzate dai robot mobili per rappresentare la mappa dell'ambiente costruita a partire da dati provenienti dai sensori (ad esempio sensori laser). La mappa dell'ambiente viene quindi generata utilizzando una griglia che rappresenta la discretizzazione dello spazio continuo. Ogni cella della griglia indica la presenza o meno di un ostacolo in quella determinata posizione.

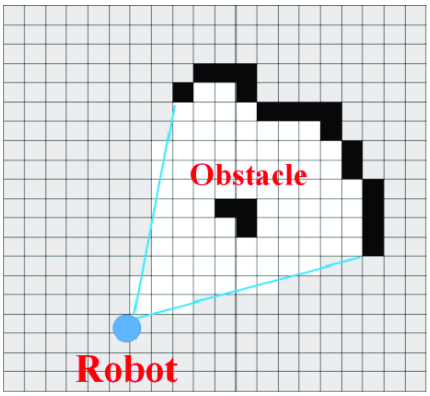

Figura 1.1: Esempio di Occupancy Grid

A* utilizza la posizione del robot nella griglia come cella di partenza e la cella destinazione come punto di arrivo e genera il cammino più breve e privo di ostacoli dalla cella di partenza alla cella goal.

Si vuole quindi valutare l'efficacia e l'efficienza dell'algoritmo A* applicato su Occupancy Grid.

2. Algoritmi e strutture dati implementate

2.1 Strutture dati

2.1.1 Grafi

Definizione

Un *grafo* (non orientato) G è una coppia (V, E) in cui V è l'insieme finito di *vertici* ed E è l'insieme di *archi*, ovvero coppie (u, v) non ordinate, in cui $u, v \in V$. Un grafo $G = (V, E)$ si dice *pesato* se sui suoi archi è definita una funzione di peso $w : E \to V$. Tale funzione associa ad ogni arco $e \in E$ un numero reale $w(e)$ che è detto *peso* dell'arco e.

I vertici hanno associate delle coordinate (x, y) che rappresentano la loro posizione nel piano. Essi inoltre presentano i seguenti parametri:

- d: stima del cammino minimo da s a v;
- h: stima del costo del cammino minimo da v alla destinazione;
- $f = d + h$;

Poiché ci interessa ricomporre il cammino minimo è inoltre necessario memorizzare un riferimento π al suo nodo predecessore.

La funzione di peso utilizzata $w(e)$ consiste nella distanza euclidea tra i due vertici estremi $w(e) = \sqrt{(v.x - u.x)^2 + (v.y - u.y)^2}$ dove $e = (u, v)$.

Implementazione

La classe *Graph* implementa la struttura dati rappresentante il grafo precedentemente descritto. Si è scelto di utilizzare la rappresentazione con liste di adiacenza. La classe *Graph* presenta quindi due attributi: *nodes* e *adj*. Il primo è di tipo *java.util.ArrayList<Node>* e contiene tutti i nodi $v \in V$, il secondo è di tipo *java.util.ArrayList<LinkedList<Edge>>* e associa al nodo i-esimo di *nodes* la sua lista di adiacenza, ovvero tutti gli archi in cui è presente tale nodo.

La classe *Graph* presenta i metodi *getNodes()* e *getAdjListOf(Node n)* che vengono utilizzati per ottenere la lista dei nodi presenti nel grafo e la lista di adiacenza di un determinato nodo n. Il metodo *getNodes()* ha complessità costante dato che restituisce il riferimento all'array dei nodi *nodes*. Il metodo *getAdjListOf(Node n)* ha anch'esso complessità costante dato che restituisce la lista di adiacenza del nodo n che è possibile accedere in tempo costante essendo memorizzata in un array. Il metodo *addEdge(Node n1, Node n2, double weight)* consente di aggiungere l'arco $(n1, n2)$ con peso *weight* all'interno del grafo. Tale metodo ha complessità $\mathcal{O}(m)$, dove m è la lunghezza della lista di adiacenza del nodo $n1$, ed è dovuta alla necessità di controllare che un determinato arco non sia già presente nel grafo. Da notare che se il grafo fosse stato implementato tramite matrice di adiacenza tale metodo avrebbe avuto complessità costante. Si è tuttavia preferita la rappresentazione con liste di adiacenza dato che i grafi utilizzati per rappresentare reti stradali di solito sono grafi sparsi e quindi la rappresentazione con matrice di adiacenza avrebbe costituito un enorme spreco di memoria.

La classe *Node* implementa la struttura dati rappresentante il vertice precedentemente descritto e presenta gli attributi: x, y, d, h, f e π. Tale classe inoltre implementa l'interfaccia *java.lang.Comparable* al fine di poter confrontare due nodi tramite il parametro f.

La classe *Edge* implementa la struttura dati rappresentante gli archi. Essa presenta due attributi: *connectedNode* e *weight*. Da notare la scelta di non memorizzare entrambi i nodi. Dato che uno dei due nodi che compongono l'arco è memorizzato nell'array *nodes*, non è quindi necessario memorizzarlo esplicitamente all'interno dell'arco.

2.1.2 Occupancy Grid

Descrizione

Il sistema *ROS (Robot Operating System)* per rappresentare le Occupancy Grid [4] utilizza una matrice di interi a 8-bit dove ogni cella (i, j) della matrice assume un valore $P(i, j)$ con il seguente significato:

- $P(i, j) = -1$ la cella rappresenta spazio inesplorato;
- $P(i, j) \geq 0$ la cella rappresenta spazio esplorato e il valore $P(i, j)$ è la probabilità che nella cella (i, j) ci sia un ostacolo;

Oltre alla matrice vengono inoltre memorizzate altre informazioni come ad esempio la *risoluzione* della mappa e la sua posizione nel sistema di riferimento assoluto, in modo da poter convertire coordinate della griglia (i, j) in coordinate reali (x, y).

Tale approccio risulta superfluo per gli scopi di questo progetto. Si utilizzeranno infatti celle che memorizzano un valore binario che assumerà valore *True* nel caso in cui la cella fosse occupata e *False* in caso contrario. Si assumerà inoltre che le coordinate della griglia (i, j) corrispondano perfettamente alle coordinate reali (x, y), evitando così di dover effettuare eventuali conversioni.

Implementazione

La classe *OccupancyGrid* implementa la struttura dati Occupancy Grid e presenta l'attributo *grid*, ovvero una matrice di booleani.

Per poter eseguire l'algoritmo A* su Occupancy Grid è necessario convertire le Occupancy Grid in grafi della tipologia precedentemente descritta. Per fare ciò è possibile considerare ogni cella libera come un nodo del grafo avente coordinate pari alle coordinate della cella nella griglia e come archi coppie di celle libere adiacenti, dove per celle adiacenti si intende coppie di celle $\{(i_1, j_1), (i_2, j_2)\}$ tali che $|i_1 - i_2| + |j_1 - j_2| = 1$. La funzione *convertToGraph* permette di fare tale conversione.

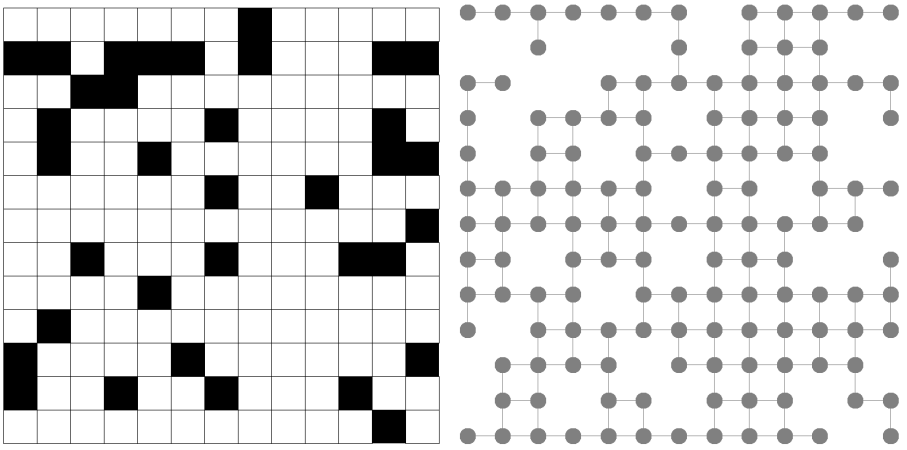

Figura 2.1: Conversione di *OccupancyGrid* in *Graph*

É stata inoltre implementata la classe *OccupancyGridFactory* che permette di generare *OccupancyGrid* in maniera casuale tramite il parametro *occupancyProb*. Tale parametro rappresenta la probabilità che una cella di *grid* sia *True* (occupata). La generazione di Occupancy Grid avviene in maniera semplice: si scorre l'intera matrice *grid* e ad ogni cella si assegna il valore *true* se un numero estratto casualmente nel range $[0-1]$ risulta minore del parametro *occupancyProb*, *false* altrimenti. La complessità totale è quindi $\Theta(n \times m)$, dove n ed m sono rispettivamente il numero di righe e il numero di colonne della matrice *grid*.

2.2 Algoritmi

2.2.1 Generazione casuale di grafi

Descrizione

Al fine di generare casualmente grafi connessi con n nodi e m archi si segue la seguente procedura:

1. si genera una sequenza casuale di n nodi connettendo il nodo i-esimo con il nodo $(i+1)$-esimo per $i \in \{1, \ldots, n-1\}$, in questo modo si ottiene un grafo connesso con n nodi e $n-1$ archi;
2. si generano casualmente i restanti $m-(n-1)$ archi;

Affinché tale procedura funzioni è necessario avere $m \in \left[n-1, \frac{n(n-1)}{2}\right]$ dato che un grafo non orientato connesso ha almeno $n-1$ archi e al massimo $\frac{n(n-1)}{2}$.

Per illustrare meglio tale procedimento conviene utilizzare una matrice di adiacenza. Ricordando che in una matrice di adiacenza nel posto (i, j) si trova un 1 se e solo se esiste nel grafo un arco che va dal vertice i al vertice j altrimenti si trova uno 0, il passo 1 della procedura precedentemente descritta fornisce un grafo avente una matrice di adiacenza del seguente tipo:

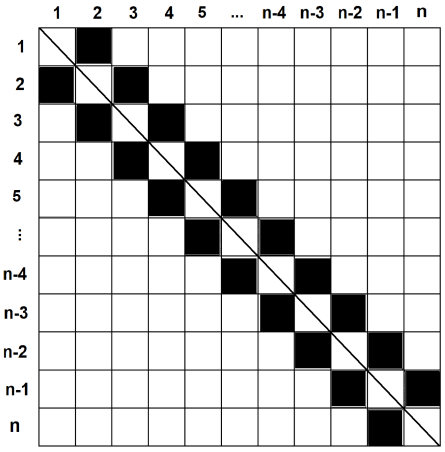

Figura 2.2: Matrice di adiacenza rappresentante il grafo generato al passo 1

Da notare che essendo il grafo non orientato tale matrice risulta simmetrica, non è inoltre possibile avere un arco che connette un nodo con se stesso.

Al passo 2 non si fa altro che scegliere casualmente $m - (n - 1)$ coppie (i, j) disponibili e creare l'arco che connette il nodo i con il nodo j. Ad esempio per $n = 8$ e $m = 11$ supponendo che i $11 - (8 - 1) = 4$ archi generati casualmente siano gli archi $(1, 4), (2, 5), (3, 5), (5, 8)$ si avrebbe la seguente matrice di adiacenza:

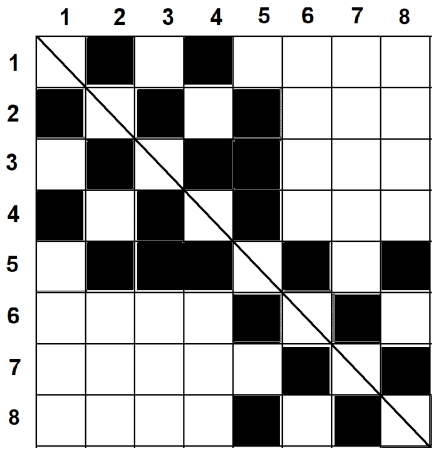

Figura 2.3: Matrice di adiacenza per un grafo con 8 nodi e 11 archi generato casualemente

Al fine di implementare tale procedura si è tuttavia deciso di non utilizzare una matrice di adiacenza, si è invece preferito utilizzare due array *availableNodes* e *candidateNodes* con il seguente significato:

- *availableNodes*: contiene i nodi per cui è ancora possibile inserire archi, sono quindi i nodi tali che hanno una lista di adiacenza con una dimensione $< n-1$.
- *candidateNodes*: dato un nodo $n1 \in availableNodes$ esso contiene i nodi $n2$ per cui è possibile formare l'arco $(n1, n2)$, tali nodi sono i seguenti $\{\{nodes \setminus n1\} \setminus adj\}$, dove *nodes* sono i nodi del grafo e *adj* sono i nodi presenti nella lista di adiacenza di $n1$.

In questo modo per implementare il passo 2 della procedura precedentemente descritta basta scegliere casualmente $m-(n-1)$ nodi da $availableNodes$, per ognuno di essi scegliere casualmente un nodo dall'array $candidateNodes$ corrispondente e generare l'arco tra i due nodi.

Pseudocodice

Algorithm 1 Generazione casuale di grafi

```
GEN_RANDOM_CONNECTED_GRAPH(n, m)

g = inizializza un nuovo grafo
n1 = crea un nuovo nodo con coordinate (x, y) casuali
availableNodes = inizializza un nuovo array contenente n1

// crea un grafo connesso con n nodi e n-1 archi
for i=1 to n-1
    n2 = crea un nuovo nodo con coordinate (x, y) casuali
    if n>3
        availableNodes.add(n2)
    g.addEdge(n1, n2)
    n1 = n2

// crea i restanti m-(n-1) archi
for i=1 to m-(n-1)
    n1 = nodo casuale da availableNodes

    adj = g.getAdjListOf(n1)
    candidateNodes = g.getNodes()
    candidateNodes.remove(n1)
    for e in adj
        candidateNodes.remove(e.getConnectedNode())

    n2 = nodo casuale da candidateNodes
    g.addEdge(n1, n2)

    rimuovi n1 da availableNodes se non ha piu' archi disponibili
    rimuovi n2 da availableNodes se non ha piu' archi disponibili
```

2.2.2 Algoritmo A*

Cammino minimo

Un *cammino* è una sequenza di nodi $u_1, u_2, \ldots, u_k$ tali che per $i = 1, 2, \ldots, k-1$ esistono gli archi (u_i, u_{i+1}). Il numero di archi è la *lunghezza del cammino*. Il *cammino minimo* tra due vertici di un grafo è quel percorso che collega i due vertici e che minimizza la somma dei costi $\sum_{i=1}^{k} w(u_i)$ associati a ciascun arco.

Descrizione dell'algoritmo A*

In [1] viene descritto un generico algoritmo di ricerca del cammino minimo da un nodo s ad un nodo t su un grafo G. Tale algoritmo partendo dal nodo s ricostruisce il cammino minimo espandendo in maniera incrementale un sottografo di G applicando iterativamente l'operatore *successore* Γ. Se su un nodo è stato applicato l'operatore Γ si dice che tale nodo è stato *espanso*. Per mantenere traccia del percorso a costo minimo da s verso tutti i nodi incontrati, ogni volta che si espande un nodo, per ogni suo successore si memorizza:

- il costo del cammino minimo da s a lui noto in quel momento;
- il nodo predecessore;

L'algoritmo termina quando si arriva al nodo t e non ci sono più nodi da espandere. Il cammino a costo minimo può essere ricostruito a partire dal nodo t seguendo la catena di predecessori fino ad s.

In [1] un algoritmo di questo tipo viene detto *ammissibile* se garantisce di trovare il cammino a costo minimo, a patto che il grafo su cui viene applicato sia un δ *grafo*, ovvero un grafo per cui il costo degli archi è $w > \delta$, dove δ è una costante > 0.

Al fine di espandere il minor numero di nodi un algoritmo di ricerca del tipo descritto precedentemente deve necessariamente avere un buon criterio di decisione su quale nodo espandere. Se vengono espansi troppi nodi non appartenenti al cammino minimo si spreca tempo, se invece si ignorano nodi appartenenti al cammino minimo l'algoritmo potrebbe fallire e non essere quindi ammissibile. La decisione su quale nodo espandere viene quindi presa in base ad una funzione $\hat{f}(n)$, il nodo n che ha il valore più basso di $\hat{f}(n)$ viene espanso.

L'algoritmo A* fa uso di tale funzione e può essere riassunto come segue:

1. segna s "open" e calcola $\hat{f}(s)$;
2. seleziona il nodo n con il valore $\hat{f}(n)$ minore;
3. se $n == t$ segna n "closed" e termina l'algoritmo;
4. altrimenti segna n "closed" e applica l'operatore successore Γ a n. Calcola f per ogni successore di n e segna "open" tutti i successori tranne quelli segnati come "closed". Segna "open" tutti i successori di n segnati come "closed" che hanno un nuovo valore di $\hat{f}()$ minore del precedente. Torna al passo 2.

Rimane solo da stabilire come definire la funzione $\hat{f}()$. Per fare ciò si può notare che è possibile definire una funzione $f(n)$ che rappresenta il costo del cammino

minimo passante per il nodo n dalla sorgente s alla destinazione t. Tale funzione può essere espressa nel seguente modo:

$$f(n) = d(n) + h(n) \tag{2.1}$$

dove $d(n)$ è il costo del cammino minimo da s a n e $h(n)$ è il costo del cammino minimo da n a s. Sfortunatamente $f(n)$ non è nota a priori, la cosa migliore che si può fare è quindi definire delle stime $\hat{d}(n)$ e $\hat{h}(n)$ tali che:

- $\hat{d}(n)$ è definita come il costo del cammino minimo trovato fino ad ora da s a n;
- $\hat{h}(n)$ definita in base a informazioni sul grafo, ad esempio in grafi rappresentati reti stradali si può usare la distanza in linea d'aria come stima di $h(n)$ dato che tale distanza rappresenta un limite inferiore per la distanza reale, è infatti importante che essa non venga sopravvalutata;

In [1] viene dimostrata la seguente proprietà:

Se $\hat{h}(n) \leq h(n)$ per ogni nodo n allora A è ammissibile.*

La precedente proprietà potrebbe far pensare che la scelta migliore per $\hat{h}(n)$ sia quindi $\hat{h}(n) = 0$ dato che in questo modo l'algoritmo risulterebbe sempre ammissibile per qualunque *grafo* δ. Risulta però vero che maggiore è il limite inferiore che si trova per $h(n)$ e minore sarà il numero di nodi da espandere. La funzione $\hat{h}(n) = 0$ risulta quindi la migliore per garantire l'ammissibilità ma la peggiore in termini di prestazioni ottenute.

Ottimalità di A*

Si è visto che con la condizione $\hat{h}(n) \leq h(n)$ l'algoritmo A* risulta essere ammissibile, garantisce quindi di trovare il cammino a costo minimo. Si vuole ora fare in modo che A* non soltanto sia ammissibile ma che sia anche ottimo, nel senso che esamina il minor numero di nodi necessari a garantire una soluzione al problema. Per fare ciò in [1] viene aggiunta l'ipotesi di *consistenza* per $\hat{h}(n)$, ovvero

$$\hat{h}(m) \leq \hat{h}(n) + w(m, n) \tag{2.2}$$

dove $w(m, n)$ indica il costo di un qualsiasi arco (m, n). Sotto tale condizione è vero anche che i valori di $\hat{f}(n) = \hat{d}(n) + \hat{h}(n)$ lungo un qualsiasi cammino sono non decrescenti, infatti se n' è un successore di n allora $\hat{d}(n') = \hat{d}(n) + w(n, n')$ e quindi

$$\hat{f}(n') = \hat{d}(n') + \hat{h}(n') = \hat{d}(n) + w(n, n') + \hat{h}(n') \geq \hat{d}(n) + \hat{h}(n) = \hat{f}(n) \tag{2.3}$$

Sotto tale condizione in [1] si dimostra che A* è ottimo.

Implementazione

In [1] viene dimostrata la seguente proprietà:

Supponendo che la condizione di consistenza sia soddisfatta e che n sia un nodo segnato "closed" da A, allora $\hat{d}(n) = d(n)$.*

Tale proprietà garantisce che quando A* espande un nodo il percorso minimo verso tale nodo è stato trovato, inoltre elimina la necessità di dover riconsiderare nodi segnati "closed".

La seguente implementazione non fa quindi esplicitamente uso di una closed list dato che assume che sia valida la condizione di consistenza:

Algorithm 2 Algoritmo A*

```
0  ASTAR(G, source, dest)
1
2  openSet = {source}
3  for v in G
4      v.h = heuristicFunction(v, dest)
5      v.d = ∞
6      v.f = ∞
7      v.π = NIL
8  source.d = 0
9  source.f = source.h
10
11 while openSet is not empty
12     current = EXTRACT_MIN(openSet)
13
14     if current == dest
15         return true
16
17     for neighbor in G.adj[u]
18         tentativeDScore = current.d + w(current, neighbor)
19         if tentativeDScore < neighbor.d
20             neighbor.π = current
21             neighbor.d = tentativeDScore
22             neighbor.f = tentativeDScore + neighbor.h
23             // decrease key
24             openSet.remove(neighbor)
25             openSet.add(neighbor)
26
27 return false
```

Per ricostruire la sequenza di nodi che compongono il cammino basta partire dal nodo destinazione e seguire la catena di predecessori.

3. Analisi della complessità

3.1 Generazione casuale di grafi

Le istruzioni

```
g = inizializza un nuovo grafo
n1 = crea un nuovo nodo con coordinate (x,y) casuali
availableNodes = inizializza un nuovo array contenente n1
```

hanno tutte costo costante.

Per creare il grafo connesso con n nodi e $n-1$ archi

```
for i=1 to n-1
    n2 = crea un nuovo nodo con coordinate (x,y) casuali
    if n>3
        availableNodes.add(n2)
    g.addEdge(n1, n2)
    n1 = n2
```

si spende $\mathcal{O}(n)$ dato che in questo caso l'istruzione *addEdge(n1, n2)* ha costo costante essendo le liste di adiacenza dei vari nodi vuote, anche le altre istruzioni hanno tutte costo costante e vengono eseguite n volte.

Per generare i restanti $m-(n-1)$ archi

```
for i=1 to m-(n-1)
    n1 = nodo casuale da availableNodes

    adj = g.getAdjListOf(n1)
    candidateNodes = g.getNodes()
    candidateNodes.remove(n1)
    for e in adj
        candidateNodes.remove(e.getConnectedNode())
```

```
    n2 = nodo casuale da candidateNodes
    g.addEdge(n1, n2)

    rimuovi n1 da availableNodes se non ha piu' archi disponibili
    rimuovi n2 da availableNodes se non ha piu' archi disponibili
```

si spende:

- l'estrazione casuale dei nodi da *availableNodes* e *candidateNodes* hanno costo costante dato che basta generare casualmente un indice valido per l'array e accedere all'elemento indicizzato da tale indice.
- la rimozione di nodi (che non hanno più archi disponibili da creare) dalla lista *availableNodes* costa nel caso peggiore $\mathcal{O}(n)$ dato che c'è la necessità di rimuovere un elemento da un array. La necessità di rimuovere un nodo è facilmente verificabile in tempo costante controllando la lunghezza della sua lista di adiacenza.
- per creare l'array contenente i nodi $\{\{g.getNodes() \setminus n1\} \setminus adj\}$ si spende:
  - l'inizializzazione di *candidatesNodes* con i nodi del grafo *g.getNodes()* costa $\mathcal{O}(n)$;
  - la rimozione da *candidatesNodes* del nodo $n1$ costa $\mathcal{O}(n)$;
  - la rimozione da *candidatesNodes* dei nodi presenti nella lista di adiacenza *adj* costa nel caso peggiore $\mathcal{O}(n^2)$ dato che il *for* viene eseguito nel caso peggiore $n-1$ volte (massimo numero di archi di un nodo) e la rimozione dei singoli nodi da *candidatesNodes* costa $\mathcal{O}(n)$.

  Il costo complessivo di tale operazione è perciò $\mathcal{O}(n^2)$.

Il costo della generazione dei $c = m - (n-1)$ archi è quindi $\mathcal{O}(cn^2)$.

Il costo totale della procedura *GEN_RANDOM_CONNECTED_GRAPH(n, m)* è $\mathcal{O}(cn^2 + n)$.

Conviene tuttavia analizzare alcuni casi specifici, in particolare si può notare che in media il ciclo *for* più annidato viene eseguito un numero di volte pari al grado medio dei nodi. Essendo il grafo non orientato il grado medio dei nodi è $\frac{2m}{n}$ e quindi il ciclo *for* viene eseguito mediamente $\frac{2m}{n}$ volte.

Tenendo conto di ciò e ricordando che l'operazione eseguita all'interno del ciclo *for* più annidato ha costo $\mathcal{O}(n)$ si può analizzare il seguente codice

```
for i=1 to m-(n-1)
    n1 = nodo casuale da availableNodes
```

```
adj = g.getAdjListOf(n1)
candidateNodes = g.getNodes()
candidateNodes.remove(n1)
for e in adj
    candidateNodes.remove(e.getConnectedNode())

n2 = nodo casuale da candidateNodes
g.addEdge(n1, n2)

rimuovi n1 da availableNodes se non ha piu' archi disponibili
rimuovi n2 da availableNodes se non ha piu' archi disponibili
```

come segue:

- nel caso in cui $m = n - 1$ non si entra mai nel ciclo *for* meno annidato, il costo è $\mathcal{O}(1)$;
- nel caso in cui $m = n + k$, con $k \geq 0$ costante
  - il ciclo *for* meno annidato viene eseguito $m-(n-1) = n+k-(n-1) = k+1$ volte;
  - il ciclo *for* più annidato viene eseguito in media $\frac{2m}{n} = \frac{2(n+k)}{n} = 2 + \frac{2k}{n}$ volte;

  e quindi il costo è $\mathcal{O}(n)$;
- nel caso in cui $m = kn$, con $k > 1$ costante
  - il ciclo *for* meno annidato viene eseguito $m - (n-1) = kn - (n-1) = n(k-1) + 1$ volte;
  - il ciclo *for* più annidato viene eseguito in media $\frac{2m}{n} = \frac{2(kn)}{n} = 2k$ volte;

  e quindi il costo è $\mathcal{O}(n^2)$;
- nel caso in cui $m = kn^2$, con $k \geq 1$ costante
  - il ciclo *for* meno annidato viene eseguito $m-(n-1) = kn^2-(n-1) = kn^2 - n + 1$ volte;
  - il ciclo *for* più annidato viene eseguito in media $\frac{2m}{n} = \frac{2(kn^2)}{n} = 2kn$ volte;

  e quindi il costo è $\mathcal{O}(n^4)$;

La seguente tabella riassume i risultati precedentemente ottenuti:

Tipo di grafo	Costo
m generico	$\mathcal{O}(cn^2 + n)$
$m = n - 1$	$\mathcal{O}(n)$
$m = n + k,\ k \geq 0$	$\mathcal{O}(n)$
$m = kn,\ k > 1$	$\mathcal{O}(n^2)$
$m = kn^2,\ k \geq 1$	$\mathcal{O}(n^4)$

Tabella 3.1: Costo della procedura *GEN_RANDOM_CONNECTED_GRAPH(n, m)*

3.2 A*

L'inizializzazione

```
openSet = {source}
for v in G
    v.h = heuristicFunction(v, dest)
    v.d = ∞
    v.f = ∞
    v.π = NIL
source.d = 0
source.f = source.h
```

ha costo $\mathcal{O}(n)$ dato che il ciclo *for* viene eseguito n volte e il costo delle operazioni di assegnazione è costante.

Utilizzando una coda di priorità (min-heap) è possibile implementare le operazioni *EXTRACT_MIN* e *DECREASE_KEY* con complessità $\mathcal{O}(\log n)$. Il seguente codice:

```
while openSet is not empty
    current = EXTRACT_MIN(openSet)

    if current == dest
        return reconstructPath(current)

    for neighbor in G.adj[u]
        tentativeDScore = current.d + w(current, neighbor)
        if tentativeDScore < neighbor.d
            neighbor.π = current
            neighbor.d = tentativeDScore
            neighbor.f = tentativeDScore + neighbor.h
            // decrease key
            openSet.remove(neighbor)
            openSet.add(neighbor)
```

```
return NIL
```

può dunque essere analizzato come segue:

- un nodo viene estratto dalla coda al massimo una volta, il ciclo *while* viene quindi eseguito nel caso peggiore n volte;
- il costo totale delle operazioni di $EXTRACT\_MIN$ è quindi $\mathcal{O}(n \log n)$;
- il ciclo *for* viene eseguito al massimo m volte perchè ogni arco viene valutato al massimo una volta;
- il costo totale delle operazioni di $DECREASE\_KEY$ è quindi $\mathcal{O}(m \log n)$

Si può quindi dire che il costo totale dell'algoritmo A* è $\mathcal{O}(n \log n + m \log n)$.

Da notare che la classe *java.util.PriorityQueue* mette a disposizione le operazioni *decrease_key*, *remove* e *add*, tuttavia la *remove* per tale implementazione ha costo lineare. Si è quindi scelto di implementare l'algoritmo tramite la classe *java.util.TreeSet* che fornisce le tre operazioni necessarie con costo logaritmico.

4. Alcuni esempi di applicazione

Segue una carrellata di esempi che mostrano l'efficacia degli algoritmi implementati. Per fare ciò e visualizzare i risultati graficamente è stata implementata la classe *Demo* che fa uso del package *gfx* capace di rappresentare grafi, Occupancy Grid e percorsi generati con A*.

4.1 Generazione casuale di grafi

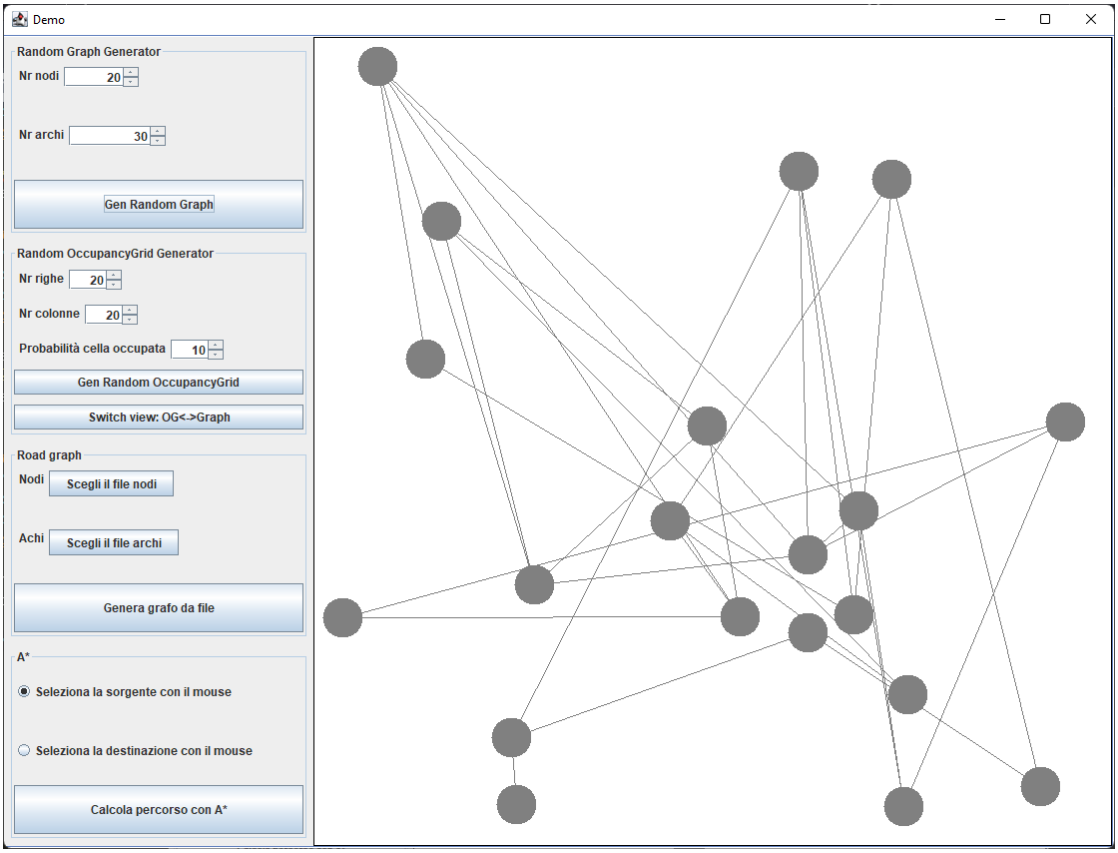

Figura 4.1: Generazione di un grafo con 20 nodi e 30 archi

4.2 A* su grafi generati casualmente

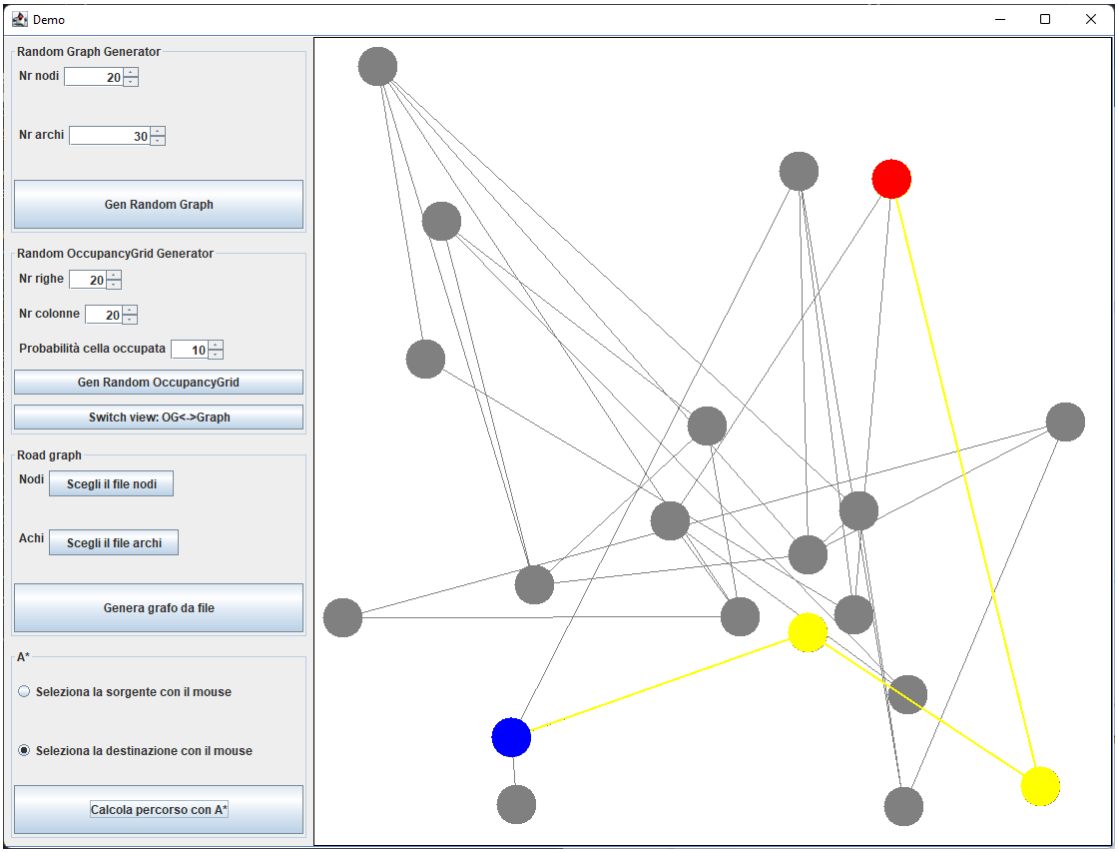

Figura 4.2: Algoritmo A* applicato su un grafo generato casualmente

4.3 A* su reti stradali

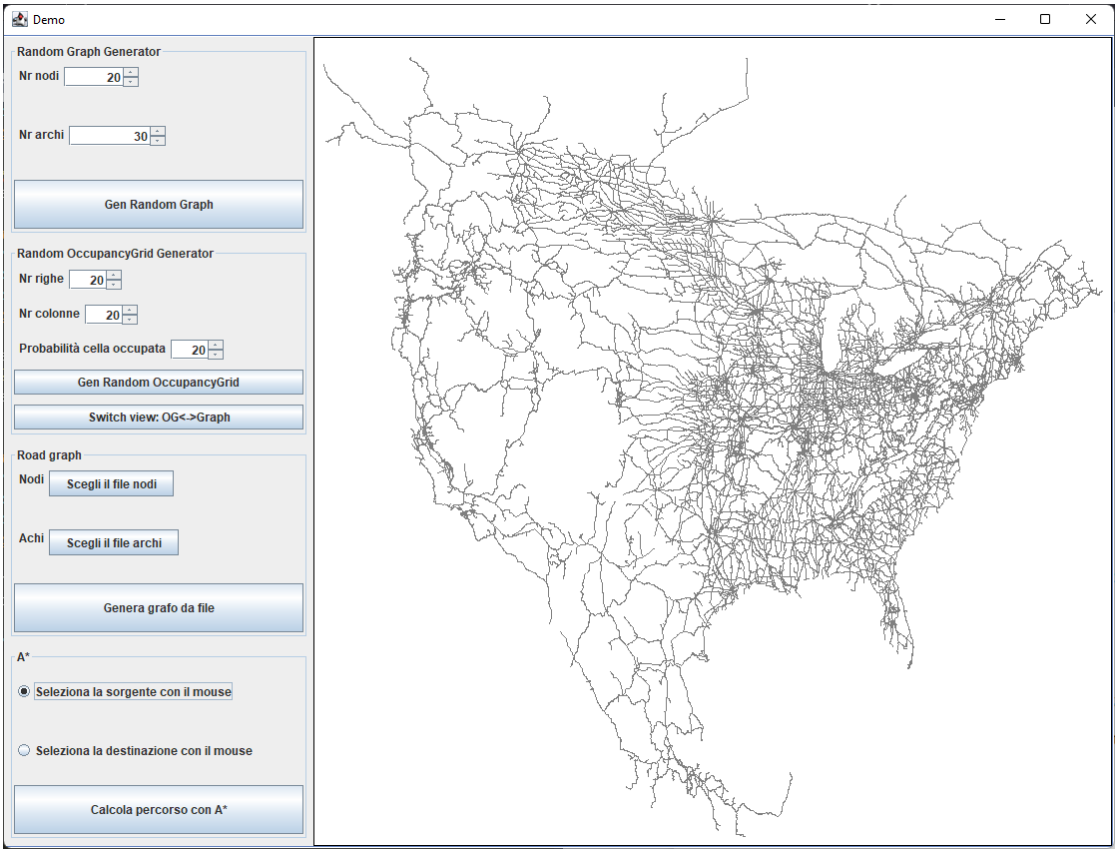

Figura 4.3: Grafo generato a partire da dati di reti stradali

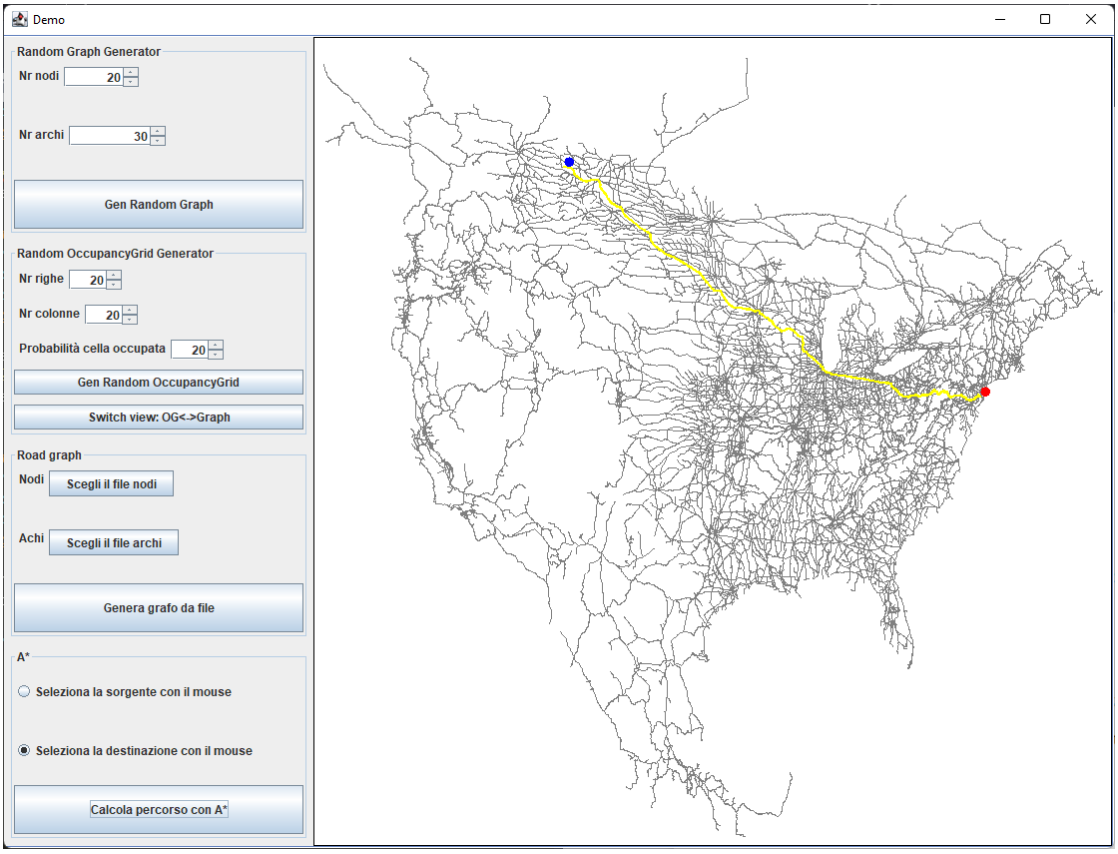

Figura 4.4: Algoritmo A* applicato su reti stradali

4.4 Generazione casuale di Occupancy Grid

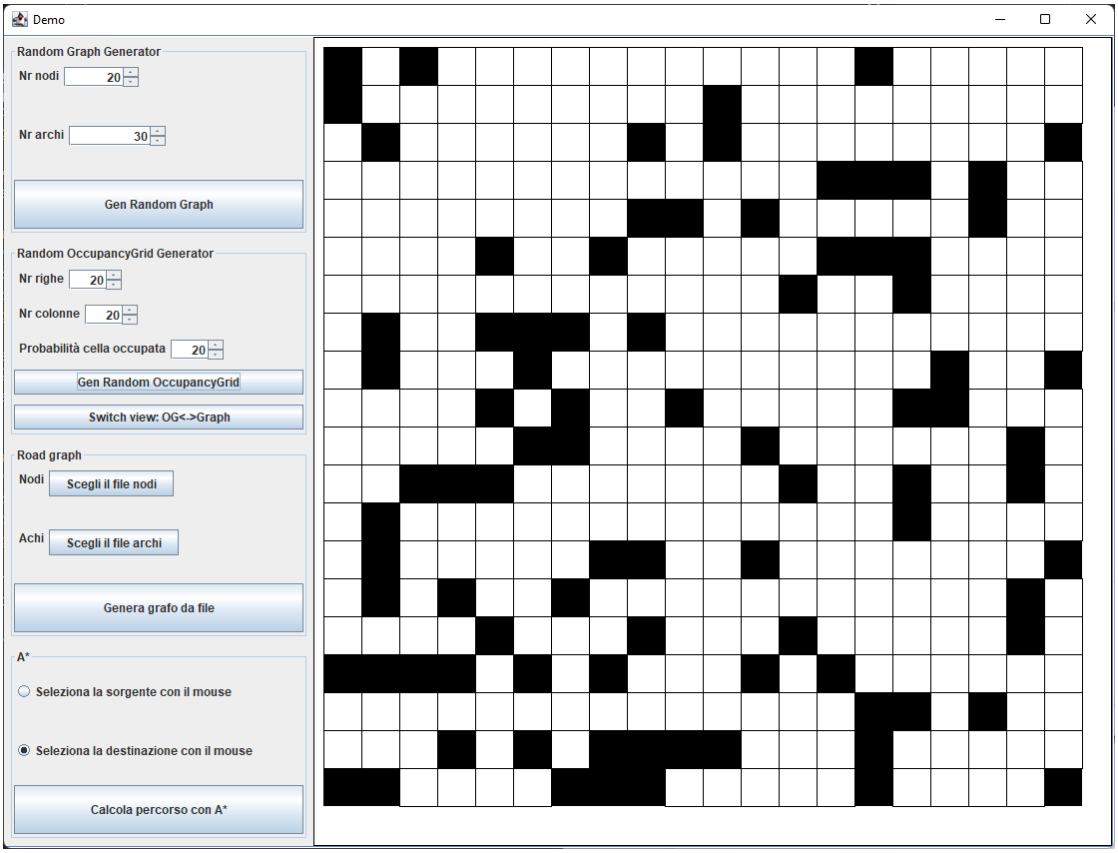

Figura 4.5: Generazione di una Occupancy Grid con 20 righe e 20 colonne

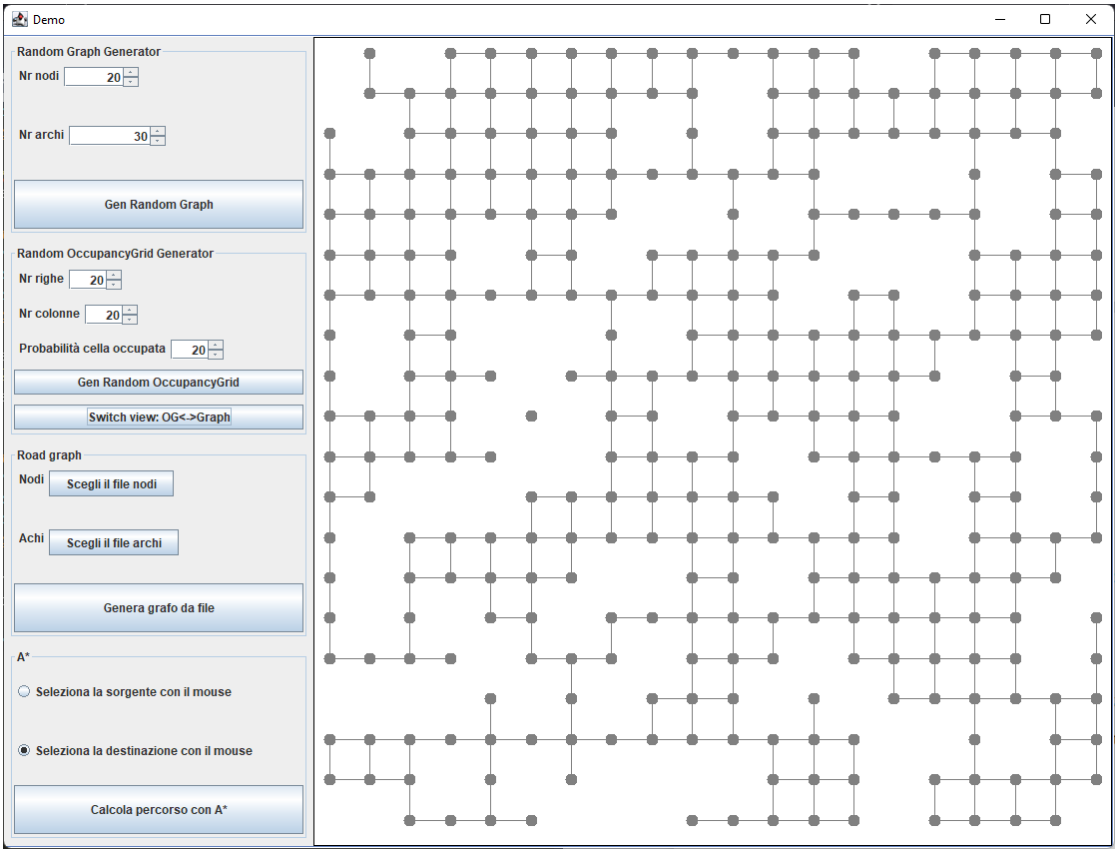

Figura 4.6: Grafo rappresentante la Occupancy Grid

4.5 A* su Occupancy Grid

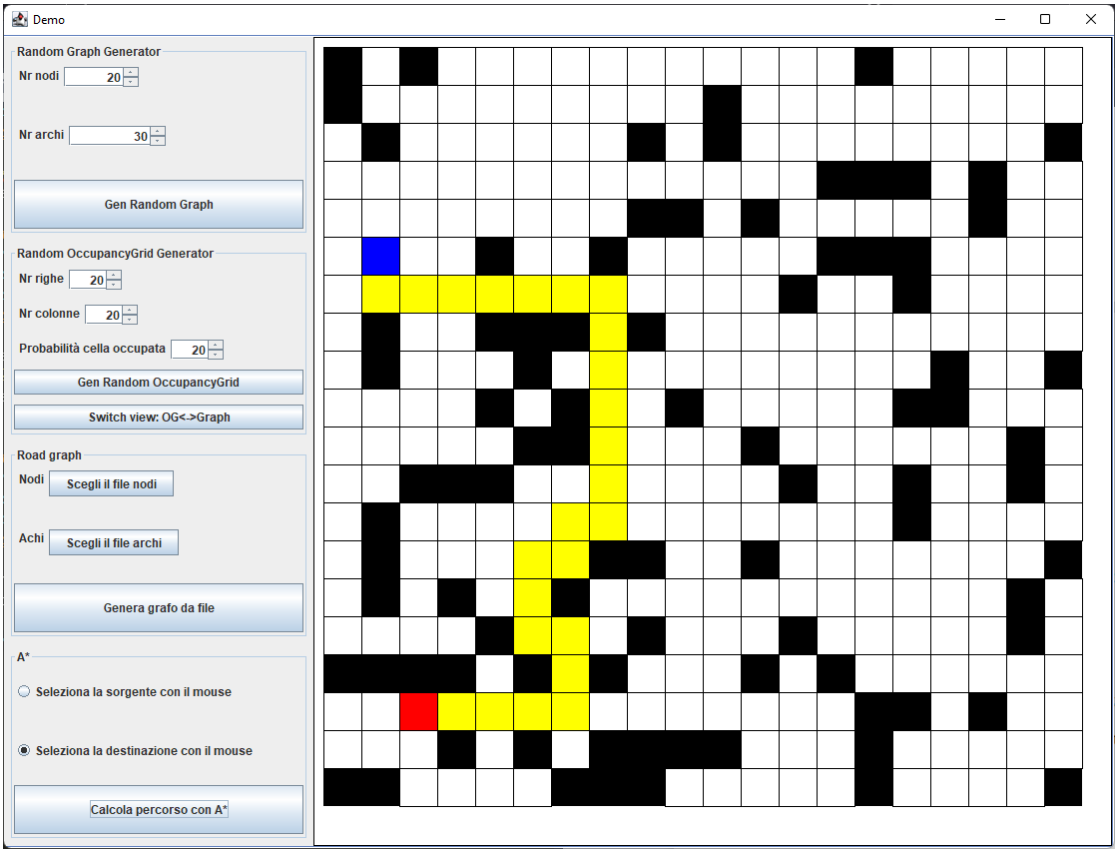

Figura 4.7: Algoritmo A* applicato su una Occupancy Grid generata casualmente

5. Dati Sperimentali

5.1 Generazione casuale di grafi

Al fine di verificare l'analisi della complessità svolta per l'algoritmo di generazione casuale di grafi riassunta dalla seguente tabella

Tipo di grafo	Costo
m generico	$\mathcal{O}(cn^2 + n)$
$m = n - 1$	$\mathcal{O}(n)$
$m = n + k$, $k > 0$	$\mathcal{O}(n)$
$m = kn$, $k > 1$	$\mathcal{O}(n^2)$
$m = kn^2$, $k > 1$	$\mathcal{O}(n^4)$

Tabella 5.1: Costo della procedura *GEN_ RANDOM_ CONNECTED_ GRAPH(n, m)*

sono state svolte diverse prove tramite la classe *Experiment1*. In particolare sono stati generati una sequenza di grafi incrementando n e utilizzando un numero di archi m dipendente da n come specificato nella tabella precedente.

Per $m = n - 1$ i risultati sperimentali confermano l'andamento lineare della procedura *GEN_ RANDOM_ CONNECTED_ GRAPH(n, m)*.

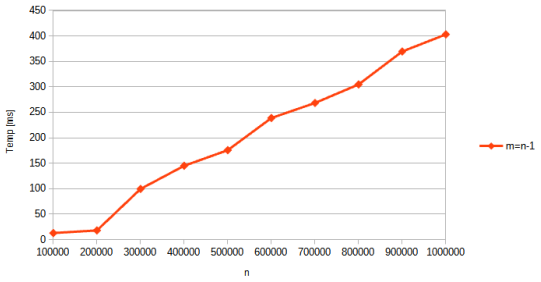

Figura 5.1: Andamento lineare per $m = n - 1$

Per $m = n + k$ i risultati sperimentali confermano l'andamento lineare della procedura *GEN_RANDOM_CONNECTED_GRAPH(n, m)*.

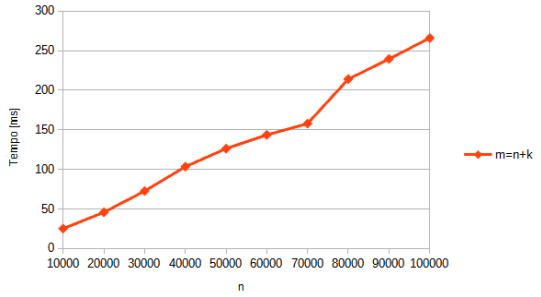

Figura 5.2: Andamento lineare per $m = n + k$

Per $m = kn$ i risultati sperimentali confermano l'andamento quadratico della procedura *GEN_RANDOM_CONNECTED_GRAPH(n, m)*.

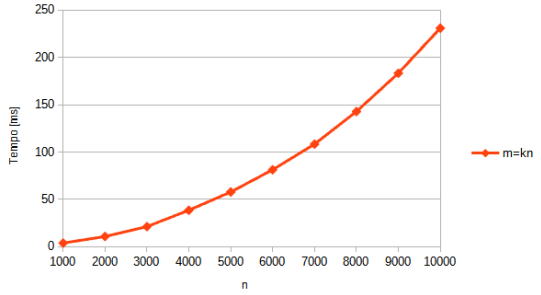

Figura 5.3: Andamento quadratico per $m = kn$

Per $m = kn^2$ i risultati sperimentali confermano l'andamento n^4 della procedura *GEN_RANDOM_CONNECTED_GRAPH(n, m)*.

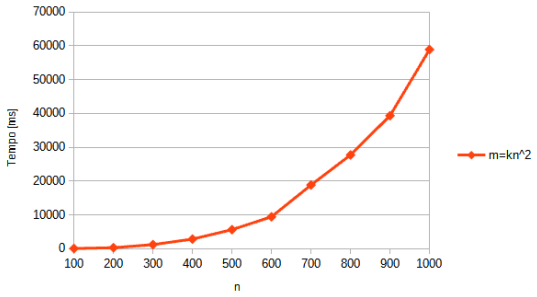

Figura 5.4: Andamento n^4 per $m = kn^4$

Utilizzando lo stesso numero di nodi n per $m \in \{n-1, n+k, kn, kn^2\}$ e confrontando i risultati su scala logaritmica si può notare l'enorme differenza di tempo per i vari andamenti n, n^2 e n^4.

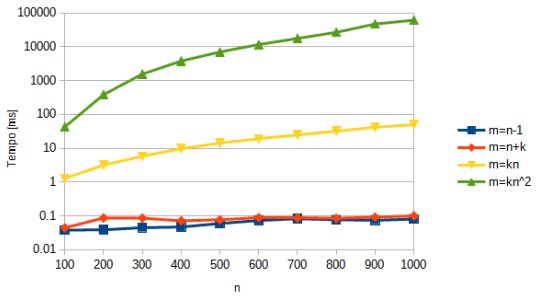

Figura 5.5: Confronto

Si è consapevoli che gli andamenti n^2 e n^4 sono poco accettabili, non si è tuttavia riuscito a fare di meglio.

5.2 A*

Al fine di verificare la complessità $\mathcal{O}(n \log n + m \log n)$ dell'algoritmo A* sono stati generati casualmente 10 grafi di dimensione crescente con $n \in \{50000, 100000, \ldots, 500000\}$ e $m = 2n$. Per ogni grafo sono stati scelti casualmente 100 sorgenti e 100 destinazioni e si è misurato il tempo di esecuzione. I risultati ottenuti confermano l'andamento linearitmico ottenuto in analisi.

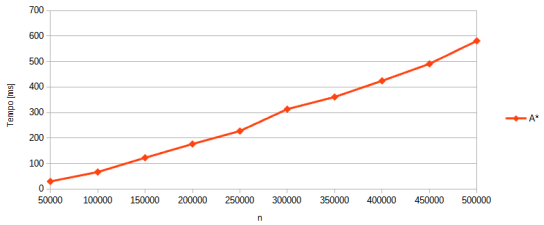

Figura 5.6: Andamento linearitmico dell'algoritmo A*

Utilizzando una funzione euristica $h(n) = 0$ l'algoritmo A* diventa sostanzialmente equivalente all'algoritmo di Dijkstra. Anche l'algoritmo di Dijkstra ha complessità $\mathcal{O}(n \log n + m \log n)$, si vuole però confrontare le loro prestazioni in modo da poter evidenziare il vantaggio ottenuto con A* avente come funzione euristica la distanza euclidea. Come ci si aspettava A* risulta nettamente migliore di Dijkstra, l'andamento asintotico è ovviamente lo stesso.

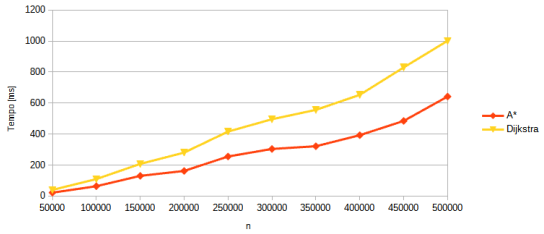

Figura 5.7: Confronto A*-Dijkstra

Si vuole ora confrontare la complessità di A* eseguito su una road network con quella di A* eseguita su un grafo generato casualmente con lo stesso numero di nodi e stesso numero di archi. L'algoritmo risulta più efficiente su road network reali.

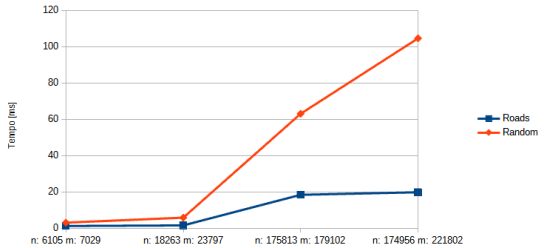

Figura 5.8: Confronto A* su Road Network - Grafi generati casualmente

In precedenza si è detto che maggiore è il limite inferiore che si trova per $h(n)$ e minore sarà il numero di nodi da espandere e quindi migliori saranno le prestazioni. Tale affermazione però non tiene conto della complessità nel calcolare la funzione $h(n)$. In alcuni casi infatti il numero di nodi in meno che vengono espansi potrebbe non compensare la complessità dovuta al dover calcolare la funzione $h(n)$. Nel caso di $h(n) = 0$ e grafi connessi con n nodi e $n - 1$ archi l'algoritmo con $h(n)$ è più prestante dato entrambi espandono lo stesso numero di nodi ma quello con $h(n) = 0$ non ha bisogno di calcolare la funzione $h(n)$.

Non si è tuttavia riusciti ad evidenziare tale proprietà dato che il costo per il calcolo dell'euristica è poco significativo, si può però comunque notare che in questo caso le due euristiche hanno le stesse prestazioni.

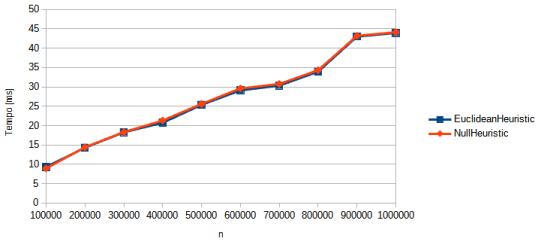

Figura 5.9: Confronto A*-Dijkstra su grafi con $m = n - 1$

Si vogliono ora valutare le prestazioni di A* su Occupancy Grid generate casualmente al variare della probabilità di occupazione.

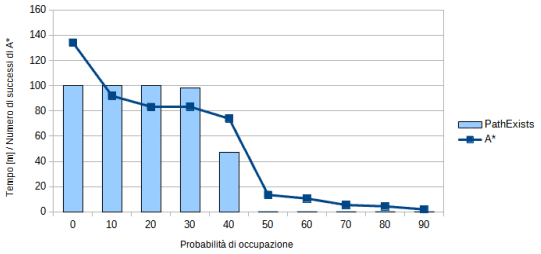

Figura 5.10: A* su Occupancy Grid

Dai dati ottenuti si può notare come con una probabilità di occupazione $> 50\%$ sia estremamente difficile che esista un percorso tra due nodi scelti casualmente. Il miglioramento delle prestazioni di A* all'aumentare della probabilità di occupazione è dovuto al fatto che all'aumentare della probabilità di occupazione diminuisce il numero di nodi espansi dato che l'algoritmo termina prima restituendo *false*.

6. Bibliografia

[1] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136.

[2] `https://en.wikipedia.org/wiki/A*_search_algorithm`

[3] `https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm`

[4] `http://docs.ros.org/en/lunar/api/nav_msgs/html/msg/OccupancyGrid.html`