Progetto di sistemi complessi: modelli e simulazione

Pietro Brenna (781840)
Alessandro Bregoli (780711)

Indice

1 Introduzione

La modellazione ad agenti è senza dubbio un metodo valido per simulare l'esplorazione di uno spazio da parte di robot; ciò che distingue i vari approcci studiati è l'architettura dell'agente-robot, i metodi di interazione, il modello decisionale.

Nei modelli esistenti si cerca l'ottimalità dell'esplorazione (come tempo, come lunghezza dei percorsi), ed emergono due approcci fondamentali:

- Un solo robot costoso (computazionalmente) che ricerca cerca di approssimare l'esplorazione ottimale
- Diversi robot poco costosi coordinati da un nodo centrale (costoso computazionalmente)

Entrambi gli approcci soffrono di una caratteristica indesiderabile: hanno un single point of failure che rende inutilizzabile l'intero modello.

Il nostro lavoro rinuncia all'ottimalità dell'esplorazione: accontentandosi di un comportamento "buono" anche se non ottimale, è possibile eliminare il coordinamento centrale, mantenere i vari robot poco costosi e ottenere robustezza di comportamento.

2 Stato dell'arte

Nel campo dell'esplorazione multi-robotica si trovano in letteratura approcci diversi che coprono problematiche simili. Troviamo ad esempio [6] guarda al modello decisionale degli agenti sotto l'aspetto della task allocation, approccio da cui abbiamo tratto spunto per il nostro lavoro. Non mancano tuttavia esempi di modelli, come in [2], che analizzano il problema con gli strumenti della teoria dell'informazione; tale lavoro dà anche soluzione al problema di navigazione, stabilire la propria posizione nell'ambiente, mentre in [4] si guarda al problema da tutt'altra prospettiva ed il solo posizionamento relativo tra robot vicini viene usato per condurre a termine l'esplorazione.

Vale la pena sottolineare che nel nostro lavoro non ci occupiamo della navigazione e la posizione assoluta degli agenti sulla griglia è sempre nota con precisione.

[6] Sviluppando un sistema composto da robot multipli che devono cooperare una delle domande chiave è: "quale robot deve eseguire che task?"; a questa domanda tenta di rispondere la branca della ricerca che si occupa di multi-robot task allocation (MRTA). Per task si intende un sottogoal necessario per ottenere l'obiettivo generale.

2.1 Funzione da ottimizzare

Per trattare MRTA in un contesto di ottimiziazione è necessario decidere cosa deve essere ottimizzato. Idealmente il goal è di ottimizzare le performance del sistema, ma questo valore è spesso difficile da misurare durante l'esecuzione del sistema. Inoltre, quando si sceglie tra diverse possibilità, l'impatto sulle performance del sistema di ogni opzione solitamente non è conosciuto; quindi è necessario una stima delle performance. Viene dunque introdotto il concetto di "utility" definito come quel valore che:

1. Rappresenta la qualità dell'esecuzione del compito assegnato dati il metodo e l'equipaggiamento.

2. Rappresenta il costo in risorse dati i requisiti spazio-temporali del lavoro assegnato.

2.2 Un approccio collaborativo

[3] L'obiettivo di un processo di esplorazione è quello di esplorare l'intero ambiente nel minor tempo possibile. Questo modello utilizza un ambiente discretizzato attraverso una griglia. Durante la scelta degli obiettivi per i robot vengono tenunte in considerazione le celle di frontiera (celle esplorate con almeno un vicino non esplorato) e si calcola una funzione di costo che è inversamente proporzionale alla distanza dal robot per cui si sta cercando l'obiettivo e direttamente proporzionale al numero di robot che si stanno

diriginedo verso quella cella. Un altro dato da considerare è l'utilità di una cella di frontiera. Tale valore è difficile da calcolare tuttavia possiamo aspettarci che una cella obiettivo di un robot avrà un valore di utilità inferiore per gli altri.
La selezione del target appropriato per ogni robot tiene in considerazione il costo dello spostamento verso il target e l'utilità della cella target. In particolare per ogni robot i cerchiamo un compromesso tra il costo V_t^i di muoversi verso la cella t e l'utilità U_t di t.

2.3 Complessità delle mappe

Sono state sviluppate diverse metriche per caratterizzare la "difficoltà" di una mappa; la maggior parte degli approcci in letteratura si dedicano al problema affine di calcolare la difficoltà di un labirinto con una uscita e un ingresso; per dare una soluzione elegante a tale problema ad esempio, [1] e [7] si basano su una rappresentazione a grafo dell'ambiente e ne calcolano le proprietà; in più in [1] si fa riferimento all'"absorption time", ovvero la probabilità che un cammino randomico arrivi dall'entrata all'uscita. Tuttavia per il nostro approccio non è sufficiente considerare la mappa un labirinto.

2.4 Legge di Amdhal

La legge di Amdhal[9] in informatica è quella formula che rappresenta l'aumento teorico di velocità di esecuzione derivante da una parallelizzazione del problema.

$$S_{lantency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

dove:

- $S_{lantency}$: é il guadagno teorico di performance dell'intero task
- s: è il fattore che rappresenta le risorse del sistema
- p: rappresenta la proporzione tra la parte di sitema parallelizzabile e quelle non parallelizzabile

$$\begin{cases} S_{latency}(s) \leq \frac{1}{1-p} \\ \lim_{s\to\infty} S_{latency}(s) = \frac{1}{1-p}. \end{cases}$$

Da questo possiamo evincere che aumentando all'infinito le risorse di parallelizzazione il sistema raggiungerà un minimo rappresentato dalla sua parte non parallelizzabile. Una possibile derivazione di questa formula è l'equazione che ricava il tempo teorico di esecuzione del sistema:

$$T(s) = (1-p)T + \frac{p}{s}T$$

3 Modello

Il modello che andremo a presentare si prefigge lo scopo di esplorare un territorio in cui possono esserci degli ostacoli insormontabili. Si tratta di un modello distribuito; in questo modo si può contrastare il problema del single point of failure; tuttavia per poter sopperire a questo problema ogni robot deve possedere una certa capacità computazionale ed una buona capacità comunicativa. Questi robot dovranno dunque essere in grado di comunicare tra loro; dovranno inoltre essere in grado di selezionare obiettivi e percorsi; in fine dovranno avere modo di salvare la mappa.
Il modello è scalabile ma bisogna tenere in considerazione la quantità di dati che vengono scambiati tra i robot all'aumentare degli stessi.

3.1 Classificazione in quanto sistema di robot

Secondo la tassonomia di Dudek et al [4] il nostro modello ricade nelle categorie:

- per quantità, SIZE-LIM: i robot sono pochi rispetto alla dimensione dell'ambiente
- per raggio di comunicazione, COM-INF: ogni robot può comunicare con qualsiasi altro robot.
- per topologia di comunicazione, TOP-BROAD: ogni messaggio è ricevuto da ogni robot (broadcast)
- per banda di comunicazione, BAND-INF: non modelliamo il costo delle comunicazioni, anche se vedremo che si ricava facilmente un limite superiore
- per flessibilità di riposizionamento, ARR-DYN: ogni robot si muove indipendentemente dai movimenti degli altri robot.
- per potenza elaborativa, PROC-TME: il calcolo del goal richiede una macchina di turing
- per composizione del gruppo, COMP-IDENT: i robot sono identici tra loro

3.2 Ambiente

I robot si muovono in una griglia rettangolare. L'ambiente è:

- discreto: è diviso in celle che si classificano come
  - Empty
  - Obstacle

- statico: una volta esplorata, una cella non cambia stato
- accessibile: il robot conosce lo stato delle celle circostanti
- deterministico: non c'è incertezza sullo stato del sistema

La griglia impedisce che due agenti si trovino stessa cella; tale scelta di modellazione è data dal fatto che la compresenza sarebbe inutile ai fini esplorativi, dato che i robot sono intercambiabili e identici.

3.3 Interazione

3.3.1 Tra agenti

I robot hanno un'interazione diretta attraverso una comunicazione in broadcast. Ogni robot ha conoscenza a priori dell'esistenza di tutti gli altri e invia un messaggio circa la sua posizione e le sue scoperte ad ogni step.

3.3.2 Con l'ambiente

I robot interagiscono con l'ambiente muovendosi in una delle celle dell'8-vicinato. Dato il modello sensoriale, lo stato della cella in cui si muoveranno è noto.

3.4 Percezione

Ogni robot percepisce lo stato di tutte le celle comprese nel suo 8-vicinato, che può essere Obstacle o Empty. Ai fini della simulazione, lo stato viene aggornato (ovvero, i robot broadcastano lo stato loro circostante) alla fine di ogni turno, cioè dopo che tutti i robot hanno compiuto la loro mossa

3.5 Complessità delle mappe

Per poter caratterizzare le proprietà degli approcci da noi testati è necessario avere una misura che renda conto della difficoltà di esplorare una mappa. Le metriche esistenti affrontano il problema dal punto di vista della risoluzione di un labirinto con un ingresso e un'uscita. Tale approccio non si adatta al nostro caso, poiché

- Tutta la mappa deve essere esplorata
- Non ha senso definire un ingresso e un'uscita

Si rende pertanto necessario trovare una metrica alternativa che sia:

- indipendente dalla dimensione della mappa: una mappa senza ostacoli deve presentare la stessa metrica qualunque sia la sua dimensione.

- proporzionale all'incremento di lunghezza dei percorsi causato dalla morfologia della mappa.

Chiameremo questa misura *intralcio medio*; date due celle i e j a distanza di Chebychev d_{ij}, sia p_{ij} il percorso minimo che le collega, $|p_{ij}|$ la sua lunghezza e definiamo $|p_{ij}| = +\infty$ se tale percorso non esiste; k_m è l'intralcio medio sulla mappa m se in media

$$|p_{ij}| = d_{ij} * k_m$$

Sia $C = \{(i,j)|i,j \in Cells, i \neq j\}$, allora

$$k_m = \frac{1}{|C|} \sum_{(i,j)\in C} \frac{|p_{ij}|}{d_{ij}}$$

Si deducono facilmente le seguenti proprietà:

- $\forall m, k_m \geq 1$
- Se m non contiene ostacoli, $k_m = 1$
- Se una zona non è raggiungibile in m da un'altra, $k_m = +\infty$

È chiaro che calcolare il percorso minimo tra ogni coppia di celle diventa presto dispendioso man mano che cresce la mappa, occorre quindi approssimare k_m con una stima $\hat{k_m}$. Supponiamo di calcolare, data una cella i,

$$k_m^i = \frac{1}{|Cells| - 1} \sum_{j\in Cells\setminus\{i\}} \frac{|p_{ij}|}{d_{ij}}$$

Si può mostrare che la media dei k_m^i è k_m:

$$k_m = \frac{1}{|Cells|} \sum_{i\in Cells} k_m^i$$

Date le regole di movimento del nostro modello notiamo che

- se i è raggiungibile da j, vale anche l'inverso
- se quindi esiste una coppia (a, b) di celle non raggiungibili tra loro, presa una cella qualsiasi i, almeno una tra a e b sarà irraggiungibile da i; se ciò non fosse vero (da i raggiungo sia a che b), potrei costruire un percorso $a \to i \to b$ valido.
- dunque se esiste una coppia (a, b) di celle non raggiungibili tra loro, $\forall i, k_m^i = +\infty$.

Questo assicura che è sufficiente un solo k_i per stabilire se $k_m = +\infty$. Se d'altro canto la quantità k_m è finita, occorrerà calcolare più k_m^i per approssimarla.

Dai nostri test risulta che per generare mappe di complessità arbitraria secondo questa metrica, un buon metodo è generare una spirale; più alto è il numero di spire, più alto sarà l'intralcio medio. Sebbene questa sia una situazione del tutto artificiale, è interessante dal punto di vista teorico.

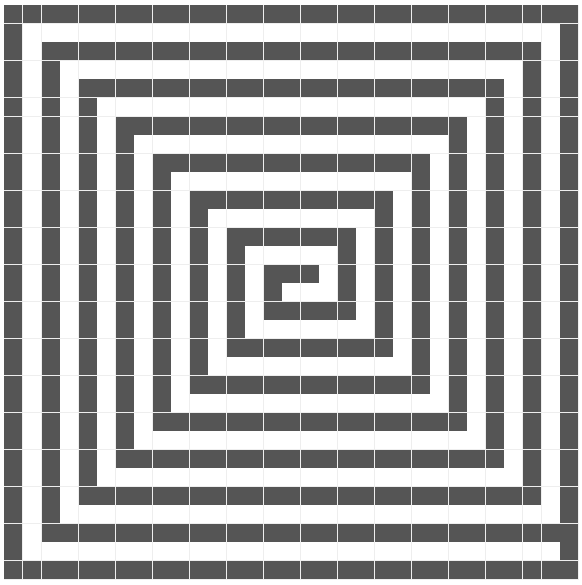


Figura 1: Una mappa a spirale con 7 avvolgimenti e intralcio medio di 12.9

3.6 Cattivi comportamenti

Su mappe che presentano una complessità elevata l'algoritmo simple può esibire comportamenti periodici. Facciamo qualche considerazione sul modello:

- lo stato dell'esplorazione è completamente definito dalla posizione degli agenti e dall'insieme delle celle, ciascuna nei 3 stati possibili per le celle
- lo stato successivo è derminato univocamente a partire dall'attuale
- il numero di celle è considerato finito

Da ciò segue che l'esplorazione può trovarsi in un numero finito di stati; questo ci assicura di non avere a che fare con un Halting Problem [10], e che anzi sicuramente, in presenza di un comportamento periodico, il sistema passerà più volte da uno stesso stato; ciò fornisce un metodo per poter rilevare con certezza tali comportamenti ciclici, seppure non di pratico utilizzo.

3.7 Strumenti utilizzati

Per poter simulare e testare il modello è stato utilizzato il linguaggio Python unito con il framework mesa appositamente progettato per simulazioni multi-agente.

3.8 Discrepanze implementative

A differenza di quanto descritto nel modello a livello implementativo la mappa è unica; tuttavia questo non modifica il comportamente descritto. Un'altra differenza tra modello e implementazione è il fatto che nel framework i robot si muovono uno per volta mentre nel modello dovrebbero muoversi contemporaneamente. Tuttavia questa differenza dovrebbe essere vista come l'introduzione di una priorità di movimento necessaria nel caso di contese di una cella.

4 Algoritmi

Per poter analizzare il modello è stato necessario implementare alcuni algoritmi che soddisfacessero due obiettivi fondamentali:

- Ricerca dell'obiettivo
- Path finding

4.1 Ricerca dell'obiettivo

Per selezionare il miglior obiettivo da raggiungere è stato progettato un algoritmo che seleziona la cella di bordo con il minor valore calcolato con la seguente equazione:

$$target\_value = \delta(pos, border\_cell)^2 - \sum_{x \in other\_robot} \delta(x, border\_cell)^2$$

dove le celle di bordo sono state definite come quelle celle che sono transitabili e che confinano con almeno una cella non esplorata. Questa euristica tenta dunque di selezionare un obiettivo che sia vicino al robot in analisi e lontano dagli altri. Il concetto di "lontano" è stato implementato come distanza euclidea; questa scelta è molto efficiente a livello implementativo, tuttavia potrebbe risultare subottimale per quanto riguarda la scelta dei goal in quanto non tiene conto di possibili ostacoli durante il percorso. Per rendere più stabile la scelte di un goal e non farlo cambiare ad ogni tick, deve verificarsi una delle seguenti condizioni perché possa essere modificato:

- La cella selezionata non è più obiettivo in quanto qualcuno (anche il robot stesso) ha esplorato tutti i suoi dintorni
- Viene trovato un obiettivo molto migliore; in questo caso la soglia di "testardaggine" del robot viene definita da un parametro che determina di quanto un obiettivo deve essere migliore di quello attuale per poterlo sostituire

```
def find_goal(analized_agent):
    for x,y in frontier_cells:

        #Viene calcolata la distanza geometrica da ogni cella
        #di confine con l'agente di cui si sta cercando
        #il goal

        score = geometric_distance((x,y), analized_agent.pos)^2
        for agent in agents:
            if agent.unique_id == analized_agent.unique_id:
                continue

            #Per ogni agente che non sia l'agente in analisi si va a penalizzare
            #il punteggio della cella inversamente alla loro distanza

            score -= distanza_geometrica((x,y), agent.pos)^2
        if score < best_score:
            best_score = score
            best_goal = (x,y)
        if (x,y) == old_goal:
            recalculated_old_goal_score = score

    #Facendo uso della variabile "stubborness" si va a pesare l'obiettivo
    #vecchio prima di confrontarlo con quello
    #nuovo; più l'agente è testardo meno sarà propenso a cambiare obiettivo

    if (recalculated_old_goal_score * analized_agent.stubborness < best_score
          or analized_agent.old_goal == best_goal):
        return analized_agent.old_goal

    return best_goal
```

Andiamo ora ad analizzare la complessità dell'argoritmo: Sia N il numero di celle e A il numero di agenti:

1. Ricerca delle celle di frontiera; complessità $O(N)$
2. Calcolo del punteggio di ogni cella rispetto all'agente; complessità $O(N \cdot A)$

Di conseguenza possiamo dire che l'algoritmo per la ricerca dell'obiettivo ha complessità:

$$O(N) + O(N \cdot A) = O(N \cdot A)$$

4.2 Path finding

Il problema del path finding è un problema ampiamente trattato dalla letteratura sia dal punto di visto videoludico che da quello robotico; per questo motivo sono stati provati diversi algoritmi; alcuni suggeriti dalla letteratura mentre altri progettati da zero.

Ogniqualvolta il "territorio" non è conosciuto a priori bisogna tener conto della necessità di ricalcolare il percorso quando viene scoperto un ostacolo; troviamo in [5] una descrizione dettagliata degli algoritmi di planning - replanning; inoltre gli algoritmi "anytime" permettono di raffinare nel tempo un path subottimale, e tale procedimento è applicabile ad A*. Tuttavia ciò non è presente nella nostra implementazione.

4.2.1 Algoritmo di Dijkstra

La mappa su cui si muovono i robot è una griglia ma può essere facilmente vista come un grafo dove ogni cella è un nodo connesso solo con le celle adiacenti. A questo punto è facile applicare l'algoritmo di Dijkstra per poter trovare un miglior percorso dalla posizione del robot fino all'obiettivo selezionato.

4.2.2 Algoritmo A^*

L'algoritmo A^* [8] è ben noto nel campo delle simulazioni ad agenti perché può sfruttare in maniera euristica la struttura spaziale della griglia: l'algoritmo si comporta come una ricerca depth-first sull'albero dei percorsi, ma dà la precedenza alla ricerca lungo i percorsi migliori secondo un'euristica specificabile; nel nostro caso utilizziamo la distanza euclidea.

```
1  def A*(start, goal)
2      # L'insieme dei nodi analizzati
3      closedSet = {}
4
5      # L'insieme dei nodi scoperti ma non ancora analizzati.
6      # All'inizio contiene solo il nodo di start.
7      openSet = {start}
8
9      # Per ogni nodo, memorizziamo il nodo da cui può essere raggiunto più
10     # efficientente. Se un nodo può essere raggiunto da più nodi, cameFrom
11     # conterrà il più efficiente.
12     cameFrom = the empty map
13
14     # Per ogni nodo, il costo di raggiungerlo dal nodo iniziale.
15     gScore = map with default value of Infinity
16
17     # Il costo per andare dallo start allo start è 0.
18     gScore[start] = 0
19
20     # Per ogni nodo, il costo totale per arrivare dal nodo iniziale al goal
```

```
    # passando da quel nodo. Questo valore è parzialmente noto, e in parte
    # stimato.
    fScore = map with default value of Infinity

    # Per il primo nodo, tale valore è del tutto euristico.
    fScore[start] = heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current = the node in openSet having the lowest fScore[] value
        if current = goal
            # siamo arrivati a destinazione: ricostruiamo il path
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue        # Ignora i vicini già analizzati.

            if neighbor not in openSet    # Scoperto un nuovo nodo
                openSet.Add(neighbor)

            # Distanza dallo start al vicino analizzato
            tentative_gScore = gScore[current] + dist_between(current, neighbor)
            if tentative_gScore >= gScore[neighbor]
                continue        # Il path scoperto non è migliore.

            # Il path è il migliore
            cameFrom[neighbor] = current
            gScore[neighbor] = tentative_gScore
            fScore[neighbor] = gScore[neighbor] +
                                        heuristic_cost_estimate(neighbor, goal)

    return failure

def reconstruct_path(cameFrom, current)
    total_path = [current]
    while current in cameFrom.Keys:
        current = cameFrom[current]
        total_path.append(current)
    return total_path
```

Segue la struttura dello step che utilizza l'algoritmo $A*$.

```
def step_astar(agent):
    goal = find_goal(agent)
    if goal == None:
        return
    pos = get_position(position)
    while True:
        #quando cambia il goal, bisogna ricalcolare il path
        if goal != get_old_goal(agent) or path is None:
            path = A*(pos, goal)
```

```
10              if path is None:
11                  return
12          direction = path.pop()
13          if the cell at direction is empty:
14              if direction == goal:
15                  #il goal è raggiunto
16                  goal = find_goal(agent)
17          else:
18              path = ricalcola path evitando la cella
19              continue
20          model.grid.move_agent(agent, direction)
```

La complessità dell'algoritmo A^* nel caso peggiore è : $O(b^d)$ dove b è il numero di possibili successori per ogni cella e d è la lunghezza dello shortest path.

4.2.3 Algoritmo greedy (simple)

Questo algoritmo è stato progettato per essere di facile computazione; per questa ragione non calcola un path ma seleziona una mossa per volta utilizzando come metrica la distanza euclidea. Scegliendo ad ogni mossa la cella più libera più vicina però si richia di incorrere in loop che impediscono ad un robot di superare determinati ostacoli. Per questa ragione si è deciso di aggiunre come ulteriore clausola il fatto che fino a quando un robot non cambia obiettivo non può passare due volte sulla stessa casella. Nel caso in cui però il robot non si possa più muovere perchè circondato da celle già percorse vengono rese nuovamente percorribili delle celle già percorse partendo da quelle più recenti e andando indietro; le celle vengono rese percorribili se hanno almeno un vicino percorribile.

```
def step_simple(analized_agent):
        goal = find_goal(analized_agent)
        analized_agent.old_goal = analized_agent.goal = goal

        direction_found = False
        #Si cerca una mossa da eseguire (anche stare fermi è un'opzione)
        while direzione non trovata:
            direction_found = True
            x_range, y_range = vicinato di actual_agent
            for x in x_range:
                for y in y_range:
                    #Le smelly cells sono quelle celle su cui il robot è già
                    #passato perseguendo il goal attuale
                    if (x,y) in analized_agent.smelly_cells:
                        continue
                    #Si cerca la cella adiacente libera che minimizza la
                    #distanza geometrica dal goal
                    if (x,y) è vuoto:
                        distance = geometric_distance((x,y), goal)
                        if distance < best_distance:
                            best_distance = distance
                            best_direction = (x,y)

            #Se le smelly_cells impediscono qualunque movimento si tenta di
            #simulare un cammino all'indietro fino a trovare una cella dove è
            #possibile scegliere un nuovo percorso
            if best_direction == None and goal != None and \
                    len(analized_agent.smelly_cells) != 0:
                cleaned = False
                for cella in analized_agent.smelly_cells[1:-1]:
                    x_range, y_range = vicinato di cella:
                    for x in x_range:
                        for y in y_range:
                            if Se la cella(x,y) è vuota e non è una smelly_cell:
                                cleaned = True
                    if cleaned:
                        analized_agent.smelly_cells.remove(cella)
                        break
```

```
                if cleaned:
                    continue

                analized_agent.old_goal = None
                goal = analized_agent.find_goal()
                analized_agent.old_goal = analized_agent.goal = goal
                direction_found = False
        if è stata trovata una direzione:
            analized_agent.smelly_cells.append(analized_agent.pos)
            analized_agent.move_agent(analized_agent, best_direction)
```

La complessità di questo algoritmo dipende prevalentemente dal numero di celle già percorse; si potrebbe dunque dire che si tratta di un algoritmo a complessità :

- $\Omega(1)$ nel caso migliore
- $O(s^2)$ nel caso peggiore

dove s è il numero di celle già percorse durante il perseguimento dell'obiettivo attuale.

5 Risultati

5.1 Misurazioni

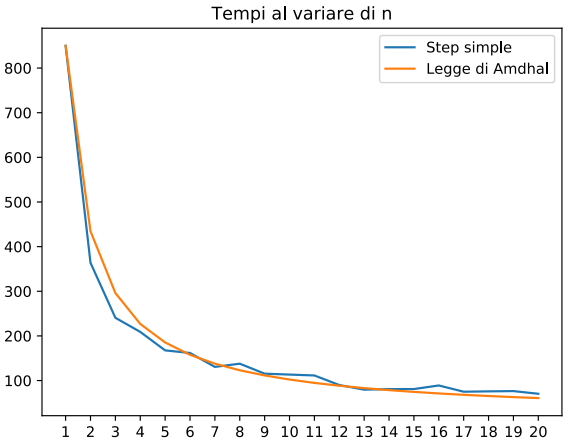
Per giudicare la bontà degli algoritmi sono state utilizzate varie mappe di diversa complessità (calcolata come in sezione 3.5). Per valutare i risultati abbiamo raccolto le seguenti misure:

- Tempi medi di esplorazione (funzione del numero di robot)
- Efficienza media di esplorazione (funzione del tempo)
- Quantità media di comunicazione (funzione del tempo)
- La deviazione standard dei tempi ottenuti
- La regressione rispetto alla funzione $T(s) = (1-p)T + \frac{p}{s}T$ dei valori dei tempi; forniamo il parametro p e l'errore di regressione come deviazione standard dalla funzione "fittata".

5.2 Mappa vuota

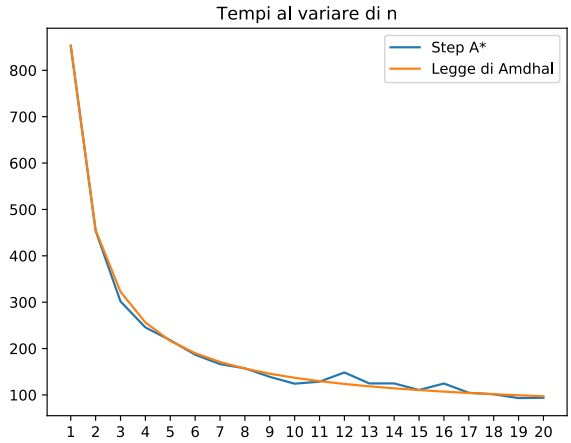
Figura 2: Questa mappa ha complessità 1.0

Utilizziamo la mappa vuota per verificare che il modello funzioni nel caso più semplice e per confrontarla con mappe più complicate. La mappa senza ostacoli mantiene un intralcio medio di 1.0 anche cambiandone le dimensione. Sulla mappa vuota possiamo notare che l'algoritmo simple sia leggermente migliore sia per quanto riguarda i tempi che per quanto riguarda la stabilità. Questo si riflette anche sulla media di celle esplorate. Dato che l'algoritmo A^* esplora mediamente meno, ha anche una comunicazione media inferiore.



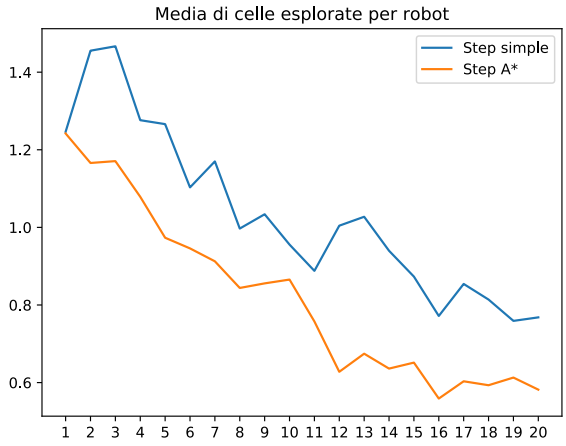
Risultati del fit: $p = 0.98, std = 22.61$

(a) Tempi dell'algoritmo simple sulla Mappa vuota.

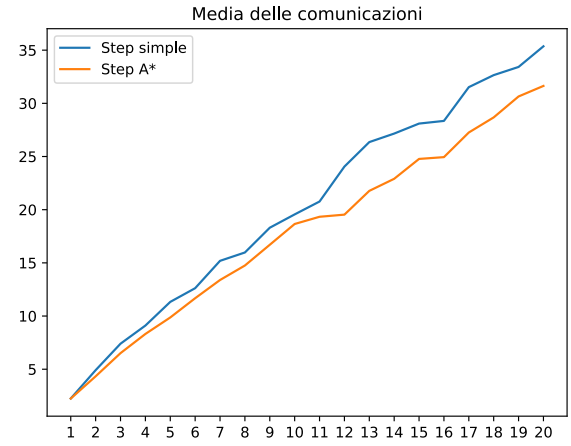


Risultati del fit: $p = 0.93, std = 9.78$

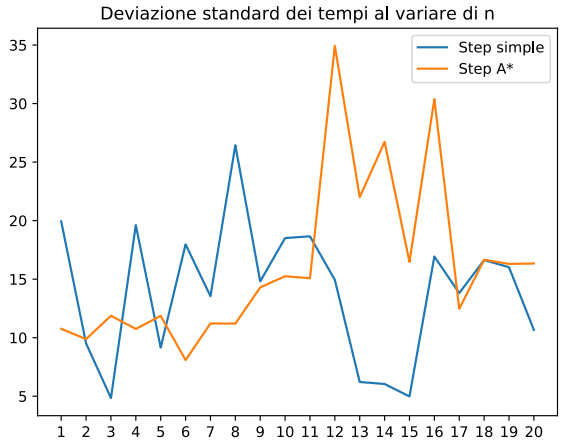
(b) Tempi dell'algoritmo $A*$ sulla Mappa vuota.



(c) Esplorazione della Mappa vuota



(d) Comunicazioni



(e) Deviazione standard Mappa vuota

5.3 Mappa 1

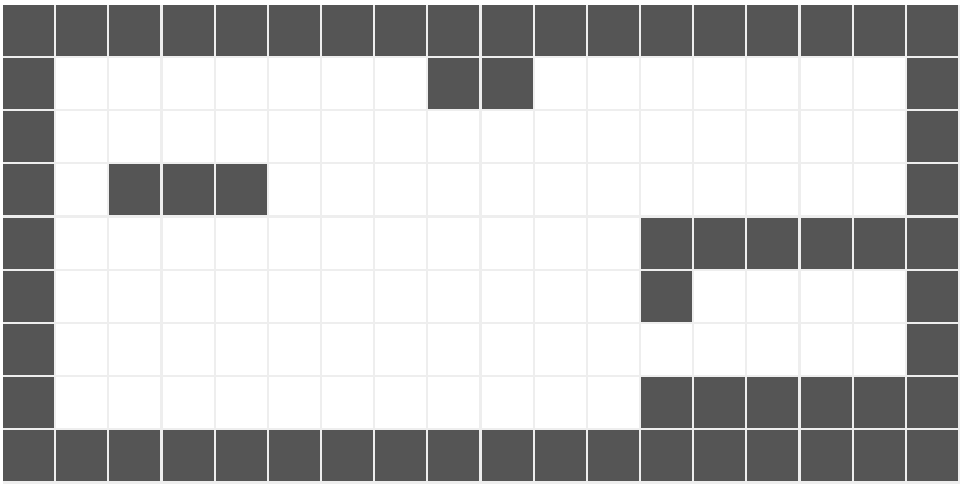
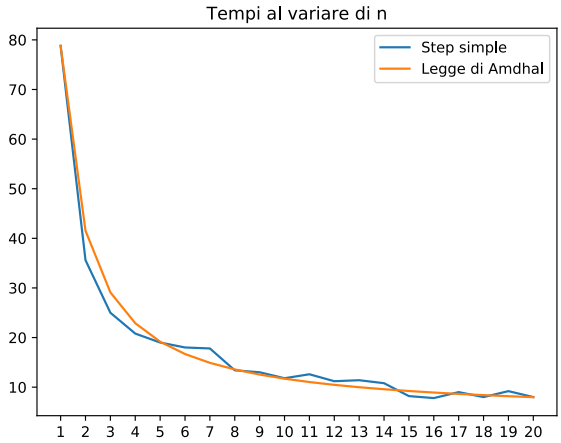


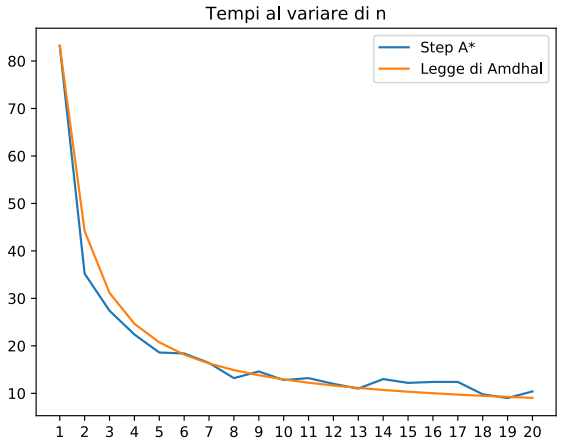
Figura 3: Questa mappa ha complessità 1.05

Questa mappa ha intralcio medio prossimo a 1; si tratta dunque di una mappa in cui una tecnica greedy può dare buoni risultati in quanto una cella vicina sarà probabilmente anche una cella facilmente raggiungibile. Possiamo infatti notare come l'algoritmo "simple" dia in media risultati migliori dal punto di vista del numero di iterazioni. Inoltre prestenta mediamente valori migliori anche per quanto rigurada l'esplorazione media dei singoli robot. Tuttavia si può notare come con tanti robot l'algoritmo simple risulta meno stabile dal punto di vista dei tempi.



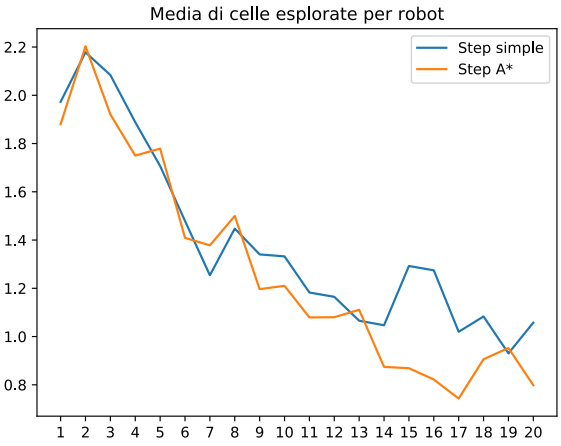
Risultati del fit: $p = 0.95, std = 1.96$

(a) Tempi dell'algoritmo simple sulla Mappa 1.

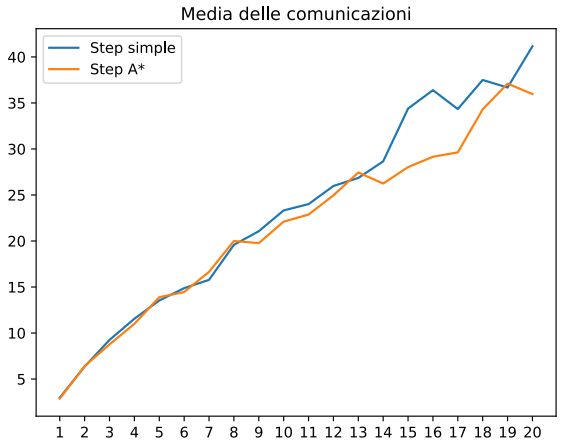


Risultati del fit: $p = 0.94, std = 2.57$

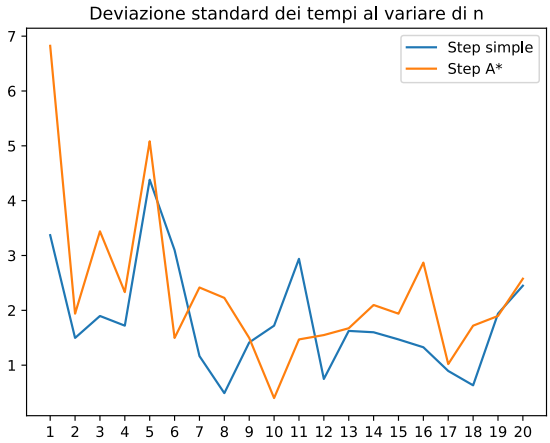
(b) Tempi dell'algoritmo $A*$ sulla Mappa 1.



(c) Esplorazione della Mappa 1



(d) Comunicazioni



(e) Deviazione standard Mappa 1

5.4 Mappa 2

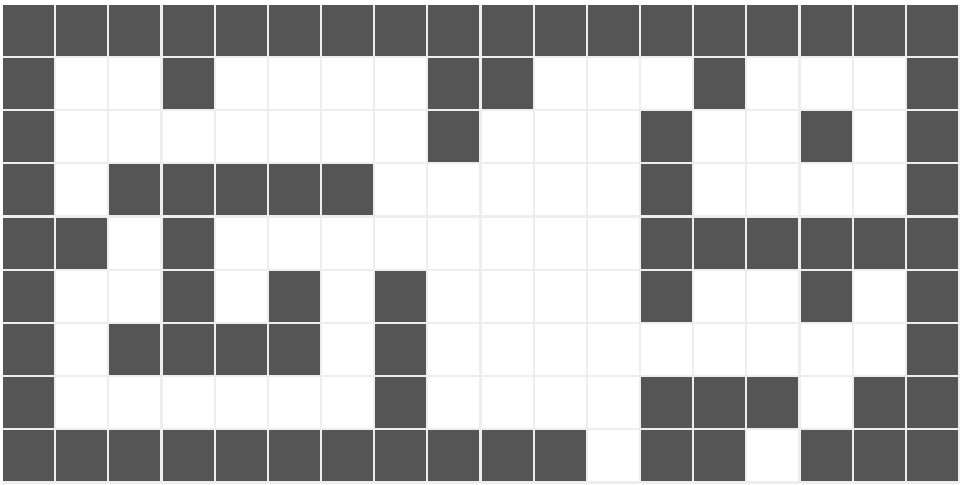
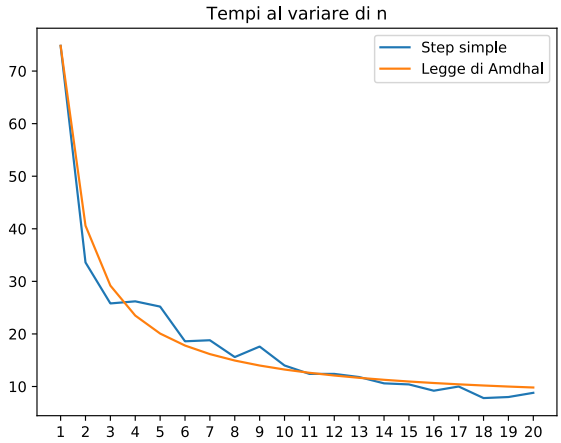


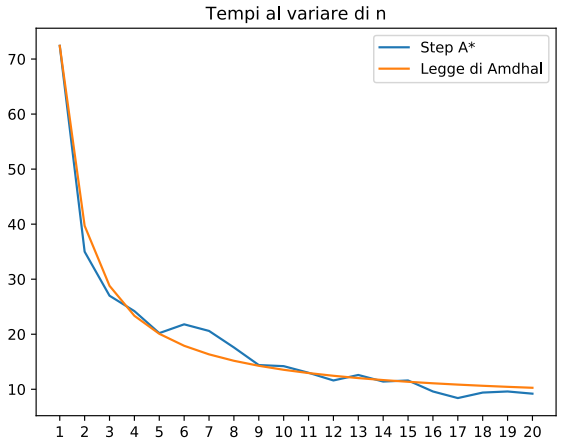
Figura 4: Questa mappa ha complessità 1.20

La mappa 2 è leggermente più complessa della mappa 1 secondo la metrica dell'intralcio medio. In questa mappa possiamo notare come l'algoritmo A^* surclassi l'algoritmo simple sia per tempi che per stabilità.



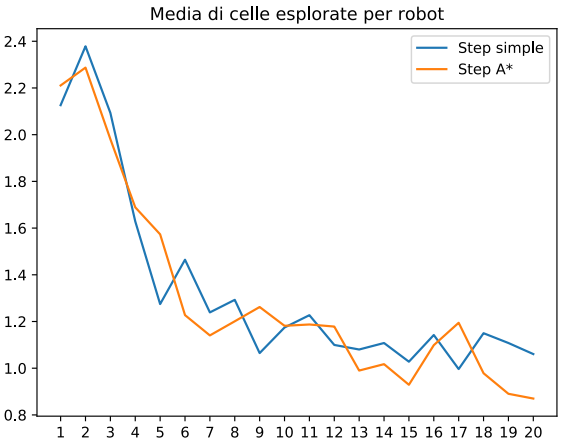
Risultati del fit: $p = 0.91, std = 2.55$

(a) Tempi dell'algoritmo simple sulla Mappa 2.

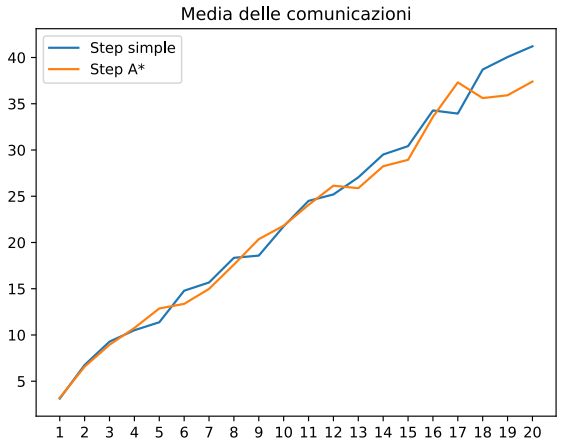


Risultati del fit: $p = 0.90, std = 1.98$

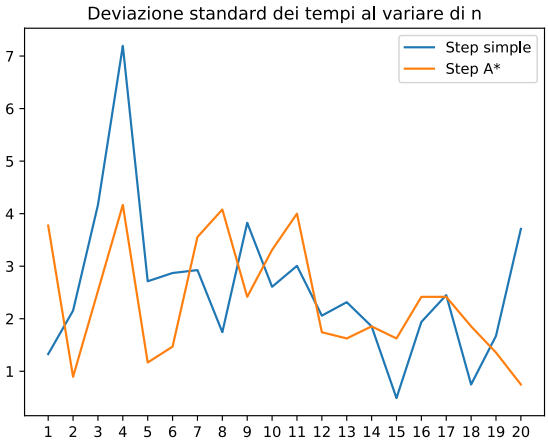
(b) Tempi dell'algoritmo $A*$ sulla Mappa 2.



(c) Esplorazione della Mappa 2



(d) Comunicazioni



(e) Deviazione standard Mappa 2

5.5 Mappa 3

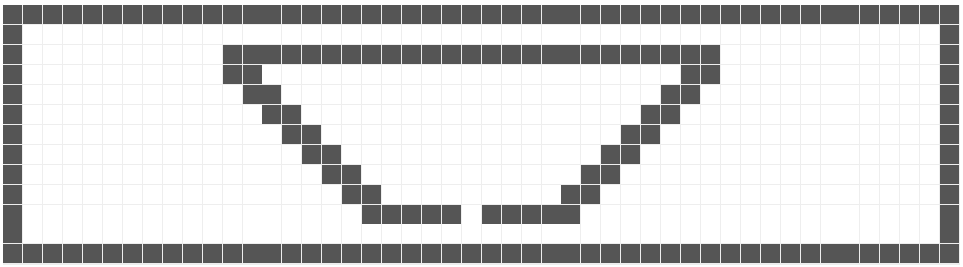
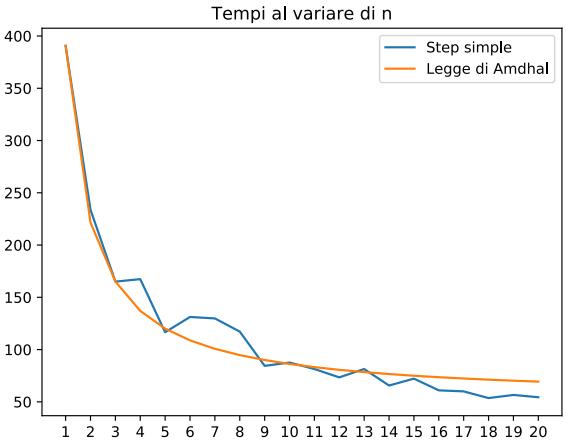


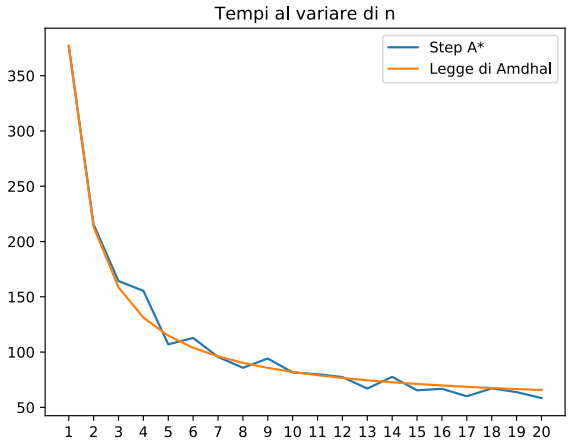
Figura 5: Questa mappa ha complessità 1.29

La mappa 3 risulta avere un intralcio medio superiore alle precendenti mappe. In questo caso si ha un'ulteriore complessità data dalla stanza interna che presenta un solo ingresso di piccole dimensioni in modo da mettere in difficoltà gli algoritmi greedy. Possiamo vedere come i tempi medi di esecuzione siano comparabili; tuttavia l'algoritmo A^* risulta significativamente più stabile. È inoltre da notare il fatto che l'algoritmo simple può, in rari casi, entrare in loop e dunque non terminare; questi casi non sono presi in considerazione ai fini delle statistiche.



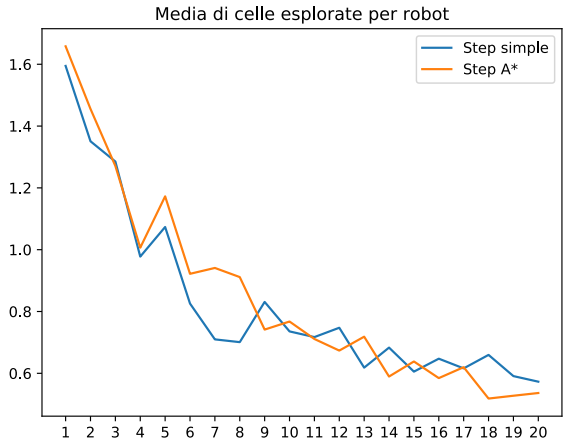
Risultati del fit: $p = 0.87, std = 14.49$

(a) Tempi dell'algoritmo simple sulla Mappa 3.

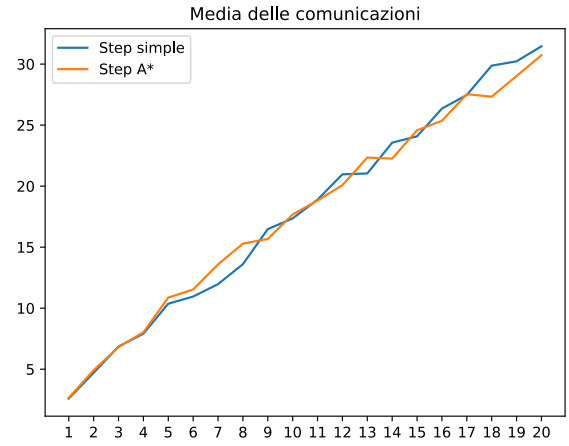


Risultati del fit: $p = 0.87, std = 7.46$

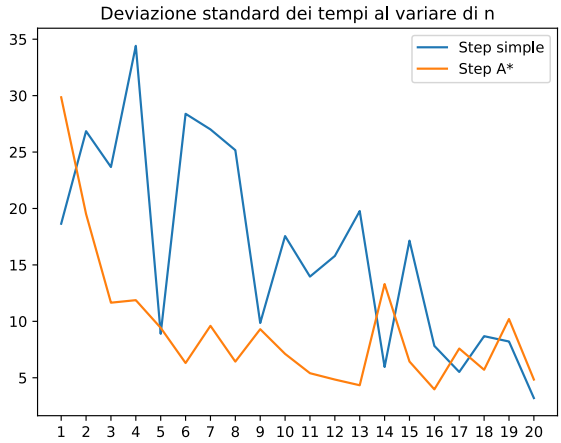
(b) Tempi dell'algoritmo $A*$ sulla Mappa 3.



(c) Esplorazione della Mappa 3



(d) Comunicazioni



(e) Deviazione standard Mappa 3

5.6 Mappa con corridoio

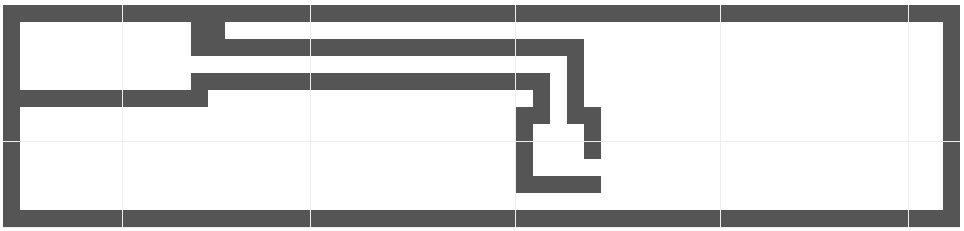
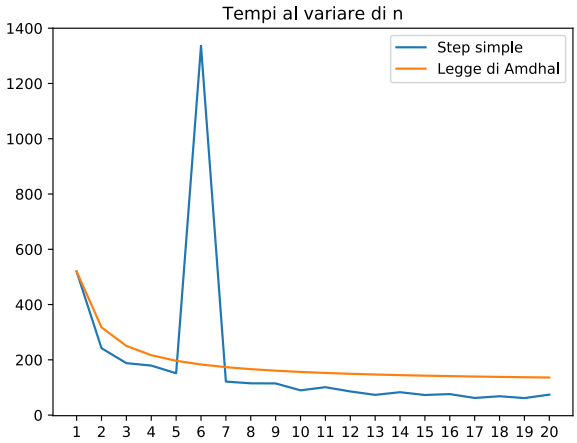


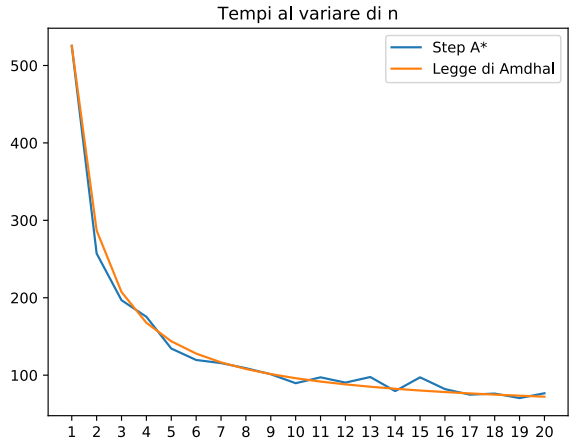
Figura 6: Questa mappa ha complessità 1.6

Questa mappa risulta avere una complessità considerevole derivante dal lungo corridoio che rende difficile un'esplorazione greedy. In questa mappa viene evidenziata l'instabilità del modello "simple" che mediamente si comporta bene quanto $A*$, tuttavia può avere un comportamento arbitrariamente peggiore; in questo test è chiaramente visibile la presenza di un'esecuzione dalle caratteristiche outlier.



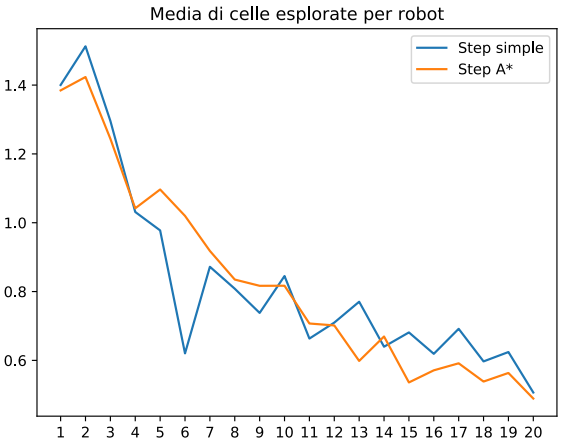
Risultati del fit: $p = 0.78, std = 264.59$

(a) Tempi dell'algoritmo simple sulla Mappa con corridoio.

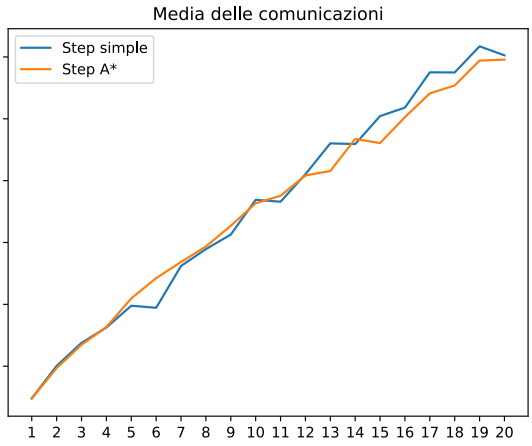


Risultati del fit: $p = 0.91, std = 9.47$

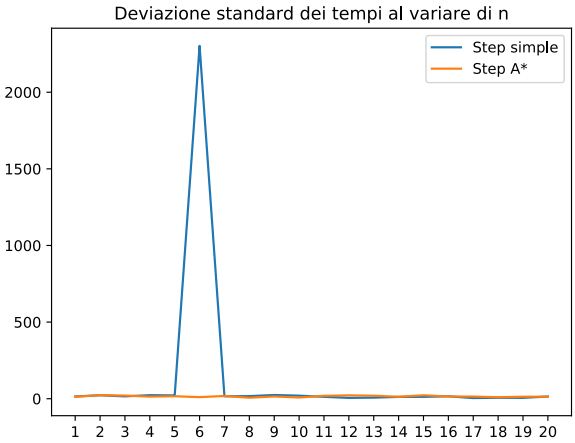
(b) Tempi dell'algoritmo $A*$ sulla Mappa con corridoio.



(c) Esplorazione della Mappa con corridoio



(d) Comunicazioni



(e) Deviazione standard Mappa con corridoio

5.7 Osservazioni sui risultati

Tutti i test sono stati svolti 5 volte per ogni possibile numero di agenti tra 1 e 20; tuttavia per l'algoritmo A^* è stata utilizzato un valore di stubborness pari a 0.5 mentre per l'algoritmo simple ne è stato utilizzato uno pari a 0.3; questa scelta deriva dal fatto che l'algoritmo simple essendo greedy entra con più facilità in loop con effetti quantomeno deleteri. Ciononostante è capitato, durante la campagna di simulazione, che alcune mappe causassero un loop nell'algoritmo simple.

6 Conclusioni

I risultati sono generalmente coerenti con le aspettative; tuttavia particolari conformazioni possono migliorare considerevolmente il comportamento dell'algoritmo simple. Spesso i due algoritmi sono comparabili; tuttavia è bene notare che la performance di $A*$ è probabilmente limitata dall'algoritmo per la scelta del goal, di natura greedy: infatti quest'ultimo non sfrutta l'ottimalità del path calcolato da $A*$.

Nei risultati non è mai stata presa in considerazione la complessità degli algoritmi come metrica, tuttavia, dato lo studio nelle sezioni 4.2.2 e 4.2.3, è chiaro che la scelta dell'algoritmo dovrebbe essere influenzata dalla disponibilità di potenza computazionale dei singoli strumenti.

6.1 Metrica di impedimento medio

La metrica da noi sviluppata ha lo svantaggio di essere computazionalmente molto onerosa, e di non predire fedelmente il comportamento del nostro algoritmo "simple"; tuttavia è anche da imputarsi dal fatto che tale algoritmo è solo parzialmente greedy, in quanto l'accorgimento delle "smelly cells" gli conferisce maggior resistenza contro i loop.

Sosteniamo tuttavia che la metrica sia effettivamente indicativa della difficoltà di spostarsi sulla mappa con un algoritmo greedy, e più generalmente fornisca una buona stima dell'aumento di lunghezza di un percorso dettata dalla topologia della mappa.

6.2 Possibili sviluppi futuri

Sarebbe interessante caratterizzare con precisione la formazione dei comportamenti periodici in relazione ai parametri di stubborness e della complessità della mappa; inoltre si potrebbe ideare una metodologia di detection e recovery dei loop, o comportamenti non del tutto deterministici volti a renderne improbabile la formazione.

Un ambito di ulteriore sviluppo è la funzione di selezione del goal; un'euristica che tenga maggior conto del territorio potrebbe aumentare l'efficienza dell'esplorazione.

Se da un lato l'intero modello è stato pensato come simulazione software, ci si può chiedere la fattibilità di un'implementazione hardware, e le sue caratteristiche; il modello richiede un sistema di comunicazione robusto, di capacità che dipende dal numero dei robot impiegati; inoltre la necessità di avere una comunicazione broadcast costante tra i robot richiede un raggio di comunicazione almeno pari alla distanza tra i 2 punti più lontani raggiungibili dagli agenti.

Riferimenti bibliografici

[1] DR Amancio, ON Oliveira e L da F Costa. "On the concepts of complex networks to quantify the difficulty in finding the way out of labyrinths". In: *Physica A: Statistical Mechanics and its Applications* 390.23 (2011), pp. 4673–4683.

[2] Frederic Bourgault et al. "Information based adaptive robotic exploration". In: *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*. Vol. 1. IEEE. 2002, pp. 540–545.

[3] Wolfram Burgard, Mark Moors e Frank Schneider. "Collaborative exploration of unknown environments with teams of mobile robots". In: *Lecture notes in computer science* 2466 (2002), pp. 52–70.

[4] Gregory Dudek et al. "A taxonomy for multi-agent robotics". In: *Autonomous Robots* 3.4 (1996), pp. 375–397.

[5] Dave Ferguson, Maxim Likhachev e Anthony Stentz. "A guide to heuristic-based path planning". In: *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)*. 2005, pp. 9–18.

[6] Brian P Gerkey e Maja J Matarić. "A formal analysis and taxonomy of task allocation in multi-robot systems". In: *The International Journal of Robotics Research* 23.9 (2004), pp. 939–954.

[7] Michael Scott McClendon et al. "The complexity and difficulty of a maze". In: *Bridges: Mathematical Connections in Art, Music, and Science*. Bridges Conference. 2001, pp. 213–222.

[8] Wikipedia. *A* search algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-October-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=804287050`.

[9] Wikipedia. *Amdahl's law — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-October-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=790799480`.

[10] Wikipedia. *Halting problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-October-2017]. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Halting_problem&oldid=801290298`.