POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

VALUTAZIONE SPERIMENTALE DI STRATEGIE DI ESPLORAZIONE PER SISTEMI MULTIROBOT CON VINCOLI DI COMUNICAZIONE

Laboratorio di Robotica e Intelligenza Artificiale
del Politecnico di Milano

Relatore: Prof. Francesco AMIGONI
Correlatore: Ing. Alberto QUATTRINI LI

Tesi di Laurea di:
Mattia ORNAGHI, matricola 799031

Anno Accademico 2014/2015

Ai miei genitori...

Sommario

L'esplorazione di un ambiente sconosciuto da parte di robot autonomi è un compito fondamentale in molte applicazioni, come quelle di ricerca e soccorso di vittime in ambienti disastrati. In letteratura, esistono diverse strategie di esplorazione che portano una squadra di robot a scoprire l'ambiente in modo incrementale. Alcuni lavori, che includono i vincoli di comunicazione all'interno della progettazione delle strategie di esplorazione, hanno considerato gli effetti di una comunicazione realistica tra i robot. Tuttavia, un confronto tra questi diversi approcci non è ancora stato effettuato nell'attuale stato dell'arte.

In questa tesi, presentiamo un confronto quantitativo di alcune strategie di esplorazione che considerano vincoli di comunicazione che pongono requisiti diversi sulla tempistica dei collegamenti fra i robot ed una base station fissa (BS). Le tre strategie che abbiamo scelto e valutato sono state testate con prove ripetute all'interno di un simulatore in cui erano già inclusi i metodi necessari alla comunicazione, alla navigazione e al mapping. In questo modo ci siamo potuti concentrare soltanto sull'implementazione delle strategie.

Gli esperimenti sono stati portati a termine in simulazione su ambienti di tipologie differenti e variando alcuni parametri. I risultati ottenuti hanno confermato che, a parità di tempo, più è rigido il vincolo di comunicazione, minore è l'area esplorata e conosciuta dalla BS, ma maggiore è la percentuale di area conosciuta dalla BS rispetto a quella visitata dai robot. Bisogna, quindi, trovare un compromesso tra la connettività e la copertura dell'ambiente a seconda dell'applicazione specifica.

Le motivazioni principali della tesi sono quelle di evidenziare i punti di forza e di debolezza dei metodi confrontati e di capire meglio come i vincoli di comunicazione influenzano le prestazioni dell'esplorazione, in modo che i risultati della nostra analisi sperimentale possano aiutare lo sviluppo futuro di strategie di esplorazione migliori.

Ringraziamenti

Desidero ringraziare, innanzitutto, il professor Francesco Amigoni per la grande disponibilità e cortesia dimostratemi, e per tutto l'aiuto che mi ha fornito.
Desidero ringraziare, inoltre, l'ingegnere Alberto Quattrini Li che mi ha seguito per tutto il lavoro di tesi consigliandomi ed aiutandomi durante le fasi critiche.
Un sentito ringraziamento va ai miei genitori ed ai miei fratelli, che, con il loro sostegno, mi hanno permesso di raggiungere questo traguardo.
Ringrazio tutti gli amici che mi sono stati vicino in questi anni.
Infine, ringrazio tutti i compagni di studi per il tempo passato insieme.

Indice

Sommario V

Ringraziamenti VII

Indice IX

Indice delle figure XIII

1 Introduzione 1

2 Stato dell'arte 5

2.1 Processo di scoperta dell'ambiente....5

2.1.1 Percezione dell'ambiente....6

2.1.2 Mappa dell'ambiente....8

2.1.3 Esplorazione dell'ambiente....9

2.1.3.1 Strategie di esplorazione....10

2.1.3.2 Metodi di coordinamento....11

2.2 Influenza della comunicazione sulle strategie di esplorazione multirobot....12

3 Impostazione del problema di ricerca 19

3.1 Definizione dei vincoli di comunicazione adottati....19

3.1.1 Vincoli di comunicazione rigidi....19

3.1.2 Vincoli di comunicazione morbidi....21

3.2 Ambiente, rappresentazioni e loro relazioni....21

3.2.1 Rappresentazione a griglia....22

3.2.2 Rappresentazione topologica....26

4 **Descrizione dei metodi utilizzati** **31**
4.1 Programmazione lineare intera....31
4.2 Stump....33
4.3 Birk....36
4.4 Utility....38
5 **Architettura del sistema** **41**
5.1 MRESim....41
5.1.1 Package e classi principali....43
5.1.2 Ciclo di simulazione....49
5.1.3 Movimento degli agenti e percezione dell'ambiente....50
5.1.4 Simulazione della comunicazione....54
5.1.5 Controllo della terminazione....57
5.2 Ilp Exploration....57
5.2.1 IlpExploration.takeStep()....57
5.2.2 IlpExploration.replan()....59
5.2.3 IlpExploration.chooseFrontiers()....60
5.2.4 Ilp.callIlp()....61
5.3 Stump Exploration....62
5.3.1 StumpExploration.chooseFrontiers()....62
5.3.2 Stump.callStump()....65
5.4 Birk Exploration....66
5.4.1 BirkExploration.chooseFrontier()....67
5.5 Utility Exploration....69
5.5.1 UtilityExploration.takeStep()....69
5.5.2 UtilityExploration.takeStep_Initial()....70
5.5.3 UtilityExploration.takeStep_Explore()....70
5.5.4 UtilityExploration.takeStep_returnToParent()....73
6 **Realizzazioni sperimentali e valutazione** **75**
6.1 Impostazione dell'attività sperimentale....75
6.2 Risultati ottenuti....79
6.2.1 Office....79
6.2.2 Open....91

6.2.3 Maze......101
6.2.4 Sommario dei risultati......110

7 Conclusioni e sviluppi futuri **113**

Bibliografia **117**

A Manuale d'installazione e d'uso **123**

Indice delle figure

2.1 Esempi di sensori....7
2.2 Esempi di rappresentazioni dell'ambiente....9
2.3 Esempio di albero gerarchico di comunicazione....13
2.4 Esempio di aggiornamento della rappresentazione topologica dell'ambiente....15
2.5 Esempi di esplorazione....17

3.1 Esempio di strategia di esplorazione con vincolo di comunicazione rigido....20
3.2 Esempio di strategia di esplorazione con vincolo di comunicazione morbido....21
3.3 Esempio di rappresentazione dell'ambiente all'interno del simulatore....23
3.4 Confronto tra rappresentazione interna a simulatore e interna a robot....24
3.5 Esempio di rappresentazione dell'ambiente all'interno del robot....26
3.6 Confronto tra rappresentazione a griglia e rappresentazione topologica....29
3.7 I tre livelli di rappresentazione dell'ambiente....30

4.1 Applicazione dell'albero di comunicazione nella rappresentazione a griglia....36
4.2 Movimenti possibili del robot per la strategia di esplorazione Birk....37
4.3 Confronto tra due esplorazioni....38

5.1 Interfaccia grafica di MRESim....43
5.2 Package agents e relazioni tra le classi....44
5.3 Package exploration....46
5.4 Alcuni package interni ad MRESim....48
5.5 Relazioni tra le classi più importanti del simulatore....48
5.6 Rappresentazione grafica del funzionamento del metodo writeStep()....52
5.7 Esempio di errori di integrazione nella mappa del robot....53

6.1 Ambienti di test e disposizione iniziale della squadra di robot....................................78
6.2 Risultati aggregati ottenuti nell'ambiente office (con tempo di replan)..................84
6.3 Risultati aggregati ottenuti nell'ambiente office (senza tempo di replan)..............86
6.4 Risultati ottenuti nell'ambiente office con 6 robot (con tempo di replan)..............88
6.5 Risultati ottenuti nell'ambiente office con 6 robot (senza tempo di replan)...........90
6.6 Risultati aggregati ottenuti nell'ambiente open (con tempo di replan)....................94
6.7 Risultati aggregati ottenuti nell'ambiente open (senza tempo di replan)................96
6.8 Risultati ottenuti nell'ambiente open con 6 robot (con tempo di replan)................98
6.9 Risultati ottenuti nell'ambiente open con 6 robot (senza tempo di replan)..........100
6.10 Risultati aggregati ottenuti nell'ambiente maze (con tempo di replan)................103
6.11 Risultati aggregati ottenuti nell'ambiente maze (senza tempo di replan).............105
6.12 Risultati ottenuti nell'ambiente maze con 6 robot (con tempo di replan).............107
6.13 Risultati ottenuti nell'ambiente maze con 6 robot (senza tempo di replan)..........109
6.14 Confronto dell'area conosciuta dalla BS adottando strategie diverse..................111

A.1 Finestra per la creazione di un nuovo progetto Java..124
A.2 Finestra per l'importazione di un progetto in Eclipse..124
A.3 Finestra per l'importazione di un archivio in Eclipse..125
A.4 Simulatore importato in Eclipse...125
A.5 Finestra per l'importazione delle librerie..126
A.6 Esecuzione del codice del simulatore..127
A.7 Interfaccia grafica di MRESim..127
A.8 Finestra per la scelta della strategia di esplorazione..129
A.9 Finestra per la scelta del metodo di comunicazione..129
A.10 Finestra per scelta dell'ambiente di esplorazione..130
A.11 Finestra per la scelta delle impostazioni iniziali della squadra di robot.............130
A.12 Finestra per la scelta delle informazioni di cui salvare i log..............................130
A.13 Mappa dell'ambiente..131
A.14 Informazioni dei robot a runtime..133
A.15 Pulsanti presenti nella parte destra dell'interfaccia grafica.................................133
A.16 Pulsanti presenti nella parte bassa dell'interfaccia grafica.................................133
A.17 Console di Eclipse a runtime..133

Capitolo 1

Introduzione

L'esplorazione di un ambiente inizialmente sconosciuto attraverso l'utilizzo di sistemi multirobot è la base per molte applicazioni, incluse, per esempio, quelle di costruzione della mappa dell'ambiente [36] e quelle di ricerca e soccorso [35].

In questi scenari, il processo di scoperta incrementale delle caratteristiche sconosciute dell'ambiente può essere modellato secondo le seguenti macrofasi che descrivono le operazioni svolte dal singolo robot:

1. percezione dell'ambiente che lo circonda;
2. integrazione dei dati percepiti all'interno della mappa che rappresenta l'ambiente conosciuto fino a quel momento;
3. decisione della prossima zona da esplorare mediante l'utilizzo di una strategia di esplorazione;
4. movimento verso la zona selezionata.

In letteratura, molte strategie di esplorazione (passo 3) sono state proposte, fra cui quelle con l'obiettivo di mantenere in ogni momento la comunicazione tra la squadra di robot ed una base station [24, 29] e quelle dove i robot esplorano l'ambiente in modo greedy senza comunicare l'informazione alla base station fino al termine dell'esplorazione [10, 32]. Questa varietà di approcci ha portato al bisogno di un confronto sperimentale con lo scopo di identificare i metodi più significativi e sottolineare i loro punti di forza e di debolezza valutando come lavorano in tipi di scenario diversi. Esempi di lavori che illustrano questo tipo di confronto, non focalizzati sulla comunicazione, sono [1, 19].

Nella formulazione più comune, le strategie di esplorazione assumono che i robot possano sempre comunicare tra loro con un'elevata larghezza di banda e che siano sempre connessi (direttamente o indirettamente) ad una base station (alcuni esempi sono gli approcci [8, 41]). Tuttavia, queste ipotesi difficilmente trovano riscontro nella pratica, ponendo il bisogno di incorporare gli aspetti di comunicazione all'interno della progettazione della strategia di esplorazione. Consideriamo, per esempio, uno scenario di ricerca e soccorso, in cui i robot sono utilizzati per trovare le vittime in un edificio collassato. Solitamente è presente una base station fissa che deve comunicare continuamente con i robot in modo che un operatore umano possa vedere in tempo reale le immagini provenienti dalle telecamere montate sui robot. In questo caso, il vincolo di comunicazione è rigido, in quanto i robot devono rimanere connessi alla base station in qualunque momento. Un'altra applicazione è quella relativa ad uno scenario di monitoraggio ambientale, dove i robot raccolgono dati dell'ambiente e li inviano ad una base station che li elabora. In questo caso il vincolo di comunicazione è più morbido, in quanto non è richiesta la connessione continua tra i robot e la base station, ma solo ad intervalli più o meno regolari. Alcuni lavori recenti, come [26, 29, 32], presentano strategie di esplorazione che incorporano tecniche diverse per trovare dei percorsi di esplorazione efficienti avendo a che fare, allo stesso tempo, con i vincoli di comunicazione. Un confronto sperimentale quantitativo di questi metodi, così come una valutazione dell'influenza dei diversi vincoli di comunicazione sulle prestazioni di esplorazione, deve essere ancora effettuato.

In questa tesi abbiamo comparato, in modo sperimentale, alcuni esempi rappresentativi di strategie di esplorazione presenti in letteratura che considerano vincoli di comunicazione diversi. Il confronto è stato portato a termine utilizzando il simulatore MRESim [9] in ambienti differenti e variando alcuni parametri. Le motivazioni principali del nostro lavoro sono quelle di trovare i punti di forza e di debolezza di questi metodi e di capire meglio come i vincoli di comunicazione influiscono sulle prestazioni dell'esplorazione, per aiutare lo sviluppo futuro di strategie di esplorazione migliori.

I risultati che abbiamo ottenuto evidenziano che, a parità di tempo, più è rigido il vincolo di comunicazione, minore è l'area esplorata e conosciuta dalla BS, ma è anche minore il tempo di disconnessione dalla BS. Tuttavia, i metodi caratterizzati da vincoli

di comunicazione rigidi permettono di avere alla BS gran parte della conoscenza che i robot possiedono riguardo all'ambiente.
Inoltre, la struttura dell'ambiente influenza le prestazioni delle strategie di esplorazione qualunque sia il vincolo di comunicazione che queste adottano. Infatti, con l'aumentare della complessità dell'ambiente, ovvero aumentando il numero di ostacoli all'interno dell'ambiente, si nota un peggioramento in termini di area esplorata per qualsiasi strategia testata. Inoltre, le strategie di esplorazione che adottano vincoli di comunicazione morbidi presentano un peggioramento in termini di tempo di disconnessione. Le strategie che adottano vincoli di comunicazione rigidi, invece, scalano meglio all'interno di ambienti caratterizzati da grandi spazi aperti e delle volte raggiungono prestazioni simili a quelle ottenute da strategie che adottano vincoli di comunicazione morbidi.

La tesi è strutturata nel modo seguente. Nel Capitolo 2, introduciamo alcuni lavori che sono stati portati a termine fino ad oggi relativi alle strategie di esplorazione. Nel Capitolo 3, diamo delle definizioni relative ai vincoli di comunicazione e descriviamo i diversi livelli di rappresentazione dell'ambiente utilizzati in questa tesi. Nel Capitolo 4, forniamo una descrizione dei modelli delle strategie di esplorazione che abbiamo confrontato. Nel Capitolo 5, presentiamo le componenti principali del simulatore MRESim utilizzato per le simulazioni ed entriamo nel dettaglio dell'implementazione delle strategie di esplorazione. Nel Capitolo 6, presentiamo e commentiamo i risultati ottenuti dagli esperimenti. Nel Capitolo 7, tiriamo le somme del lavoro svolto e suggeriamo ulteriori direzioni per i lavori futuri. Nell'Appendice A descriviamo come installare ed utilizzare MRESim.

Capitolo 2

Stato dell'arte

Questo capitolo è diviso in due parti. Nella prima parte, descriviamo il processo di esplorazione dell'ambiente, soffermandoci su alcune fasi che lo compongono e ponendo particolare attenzione alle strategie di esplorazione ed ai metodi di coordinamento. Nella seconda parte, presentiamo diversi lavori portati a termine fino ad oggi, differenziati in base ai vincoli di comunicazione adottati durante l'esplorazione di un ambiente sconosciuto.

2.1 Processo di scoperta dell'ambiente

L'esplorazione di un ambiente inizialmente sconosciuto da parte di una squadra di robot riguarda il problema di costruire, in modo incrementale, la mappa dell'ambiente. Affinché il processo di esplorazione possa essere portato a termine da parte di alcuni robot senza dipendere da un operatore umano, la squadra di robot deve riuscire ad eseguire le operazioni seguenti [12]:

- percezione tramite sensori: il singolo robot deve possedere dei sensori che gli permettano di percepire l'ambiente che sta esplorando, in modo da potere, per esempio, individuare ostacoli, oggetti o vittime di un incidente;
- localizzazione e mapping: chiamato anche SLAM (*Simultaneous Localization and Mapping*), consiste nel costruire una mappa dell'ambiente in cui il robot si deve muovere e nel localizzare il robot al suo interno. La costruzione di tale mappa è importante, per esempio, per ottimizzare i movimenti del robot

all'interno dell'ambiente;

- pianificazione del percorso: chiamato anche *path planning*, consiste nel calcolo del cammino che un robot deve seguire per spostarsi all'interno della mappa dell'ambiente conosciuto da una posizione iniziale ad una posizione finale;
- esplorazione autonoma: i robot devono essere in grado di svolgere questi compiti in completa autonomia senza l'ausilio di un operatore umano ed eventualmente interagendo tra loro.

In questa tesi consideriamo una squadra di robot in grado di raccogliere nuove informazioni dell'ambiente inizialmente sconosciuto e condividerle con una base station fissa. Il processo di scoperta dell'ambiente da parte della squadra di robot può essere modellato secondo le seguenti macrofasi:

1. percezione dell'ambiente circostante;
2. integrazione delle nuove informazioni all'interno della mappa dell'ambiente conosciuta fino a quel momento;
3. valutazione di possibili posizioni da raggiungere;
4. scelta della posizione da raggiungere;
5. tracciamento del percorso verso quella posizione;
6. movimento verso la posizione selezionata.

Di seguito, entriamo nel dettaglio di alcune di queste fasi che possono essere eseguite in modo centralizzato (presso una base station) o distribuito (sui robot).

2.1.1 Percezione dell'ambiente

Su ogni robot che compone la squadra di esplorazione sono montati dei sensori. Questi sensori sono divisi in due categorie principali [12]:

- sensori di stato interno: riguardano la misura di variabili che sono utilizzate per il controllo del robot, quali, per esempio, la velocità e l'accelerazione;
- sensori di stato esterno: riguardando la misura di variabili che sono utilizzate per la guida del robot, per l'individuazione degli ostacoli e per l'identificazione degli oggetti, quali, per esempio, la distanza, la forma ed il contatto.

I sensori di stato esterno possono essere ulteriormente classificati nel seguente modo:

- sensori di distanza: sono utilizzati per la guida del movimento e per evitare

ostacoli, dove l'interesse è nella valutazione della distanza dagli oggetti più vicini e nel conoscere la posizione e le caratteristiche generali della forma degli oggetti nell'ambiente di esplorazione (per esempio, telemetri laser);

- sensori di tatto: sono utilizzati per ottenere dati associati al contatto tra un robot e gli oggetti nell'ambiente. La percezione tattile può essere utilizzata, per esempio, per localizzare e per identificare un ostacolo (per esempio, microinterruttori);
- sensori di visione: sono utilizzati principalmente per avere informazioni riguardo alle caratteristiche di un oggetto che non siano relative soltanto alla sua forma geometrica e alla sua posizione, ma che possano essere utilizzate anche, per esempio, per capire effettivamente di che oggetto si tratti sulla base della sua texture (per esempio, telecamere);
- di posizione: sono utilizzati per tenere traccia della posizione del robot all'interno dell'ambiente durante l'esplorazione (per esempio, GPS).

In Figura 2.1 sono mostrati due esempi di tipi di sensori.

Il diverso tipo di sensori installati sul robot provoca una esplorazione dell'ambiente differente. Per esempio, utilizzando sensori di tatto il robot basa l'esplorazione sul contatto con gli ostacoli. Il robot esplora l'ambiente seguendo una certa traiettoria finché non entra in contatto con un ostacolo. A quel punto cambia la sua traiettoria per seguire il profilo dell'ostacolo. Un'esplorazione di questo tipo è eseguita, per esempio, dai pulitori di ambienti interni. Viceversa, utilizzando sensori di distanza il robot può evitare il contatto con gli ostacoli e rimanere all'interno di spazi liberi in modo da percepire più area possibile dell'ambiente che lo circonda. Un'esplorazione di questo tipo è eseguita, per esempio, dai robot utilizzati per le missioni di ricerca e soccorso.

Figura 2.1: due tipi di sensori utilizzati dai robot. A sinistra un sensore di visione (telecamera) e a destra un sensore di distanza ad infrarossi.

2.1.2 Mappa dell'ambiente

Vi sono diverse tipologie di mappe che permettono al robot di rappresentare l'ambiente in cui si sta muovendo [12]:

- paths: la mappa è formata da percorsi predefiniti (*path*) che il robot deve seguire. L'ambiente non viene modellizzato mediante la geometria, ma con dei grafi. Un arco che unisce due vertici all'interno del grafo indica la presenza di un percorso tra le locazioni rappresentate dai vertici. Il percorso che porta da una locazione all'altra è memorizzato all'interno dei vertici;
- free space: all'interno della mappa viene rappresentato soltanto lo spazio libero dell'ambiente. Durante il movimento del robot è possibile memorizzare i punti in cui il robot si ferma e cambia direzione (per esempio, per evitare un ostacolo) e le traiettorie percorse dal robot per spostarsi da un punto all'altro. Le regioni di ambiente racchiuse tra le traiettorie possono essere etichettate come libere o occupate. Il robot può spostarsi liberamente all'interno delle zone libere senza dover seguire obbligatoriamente le traiettorie già percorse;
- orientata agli ostacoli (object oriented): all'interno della mappa vengono rappresentati solo gli oggetti. In questo modo viene vietato al robot di raggiungere determinate posizioni. Gli oggetti vengono memorizzati, per esempio, come record in una lista. Ogni record descrive la posizione assoluta in coordinate x, y, θ di un punto di riferimento dell'oggetto e una lista di vertici in coordinate relative al punto di riferimento dell'oggetto;
- mista (composite space): l'ambiente viene rappresentato come una griglia in cui le celle hanno una forma regolare (per esempio, quadrata). A queste celle viene assegnato un valore che descrive la zona di ambiente relativa, per esempio, libera oppure occupata.

Il tipo di rappresentazione dell'ambiente influenza l'esplorazione da parte del robot. Per esempio, un robot che rappresenta l'ambiente secondo la tipologia path, può muoversi tra due locazioni soltanto seguendo il percorso che è stato memorizzato all'interno dei vertici del grafo. Viceversa, un robot che utilizza una rappresentazione di tipo composite space, può cambiare il percorso di spostamento tra due locazioni a seconda dell'informazione che ha a disposizione riguardo l'ambiente. Scoprendo nuove aree, infatti, può trovare nuovi percorsi che gli permettano di raggiungere la locazione che ha

come obiettivo percorrendo, per esempio, una distanza minore.

In Figura 2.2 sono mostrati due esempi di rappresentazioni dell'ambiente utilizzate dai robot.

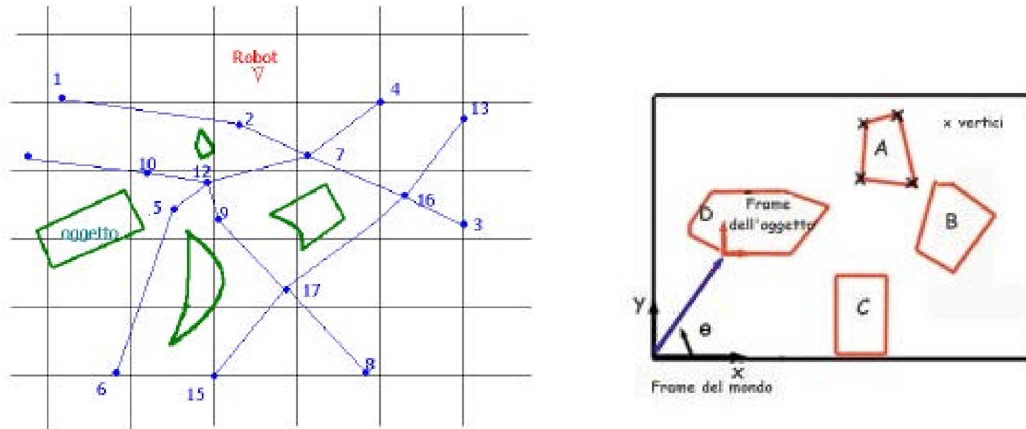


Figura 2.2: a sinistra è mostrata una rappresentazione dell'ambiente secondo una tipologia di mappa free space, mentre a destra è mostrata una rappresentazione di tipo object oriented [12].

2.1.3 Esplorazione dell'ambiente

L'utilizzo di più robot durante l'esplorazione porta molti benefici, dall'aumento della robustezza dell'intero sistema fino ad una costruzione in modo efficiente della mappa dell'ambiente [5, 6, 30]. L'efficienza della costruzione della mappa cambia a seconda del criterio di ottimalità che viene preso in considerazione. Se, per esempio, il criterio di ottimalità preso in considerazione è la distanza percorsa dalla squadra, allora una mappa viene costruita in modo efficiente se la distanza complessiva percorsa dai robot, per raggiungere delle posizioni obiettivo, è la minore possibile. Se, invece, viene preso in considerazione come criterio di ottimalità l'ammontare di area coperta dell'ambiente, allora la costruzione della mappa è efficiente se la squadra di robot mappa la più grande quantità di area dell'ambiente possibile in un certo intervallo di tempo.

L'esplorazione dell'ambiente in modo coordinato è stato studiato principalmente per scopi di costruzione della mappa [20, 21] e di ricerca e soccorso [23]. I due aspetti principali di un'esplorazione multirobot all'interno di un ambiente sconosciuto sono la strategia di esplorazione ed il metodo di coordinamento. Vediamoli più nel dettaglio.

2.1.3.1 Strategie di esplorazione

Le strategie di esplorazione sono utilizzate per la decisione della posizione successiva verso cui i robot devono muoversi all'interno dell'ambiente parzialmente conosciuto [1]. È necessario confrontare sperimentalmente strategie di esplorazione differenti per poter decidere quale è migliore rispetto ad un'altra. In generale, non è possibile valutare una strategia di esplorazione rispetto ad un ottimo, perché di solito questo è sconosciuto. Attualmente, la valutazione sperimentale si basa per la maggior parte su un confronto relativo tra strategie. Solo alcuni lavori prendono in considerazione anche l'ottimo. In [28] gli autori presentano un metodo per calcolare un'approssimazione del percorso di esplorazione ottimo all'interno di un ambiente arbitrario. Per percorso di esplorazione ottimo viene inteso il percorso di esplorazione con lunghezza minore, in quanto nel lavoro è stato scelto come criterio di ottimalità la distanza percorsa.

La gran parte delle strategie di esplorazione riguardano il problema di costruzione della mappa dell'ambiente. I metodi di esplorazione, generalmente, preferiscono un approccio di scoperta incrementale dell'ambiente basato sul processo NBV (*Next Best View*), che ripete in modo greedy la selezione del miglior punto di osservazione da raggiungere all'interno dell'ambiente conosciuto [4, 13, 33, 37]. Solitamente, un approccio NBV considera un certo numero di locazioni candidate sulle frontiere e seleziona la migliore [38]. Le *frontiere* sono delle zone di confine tra l'ambiente conosciuto e l'ambiente sconosciuto ad un determinato istante di tempo. Ogni agente sceglie quale raggiungere tra le diverse frontiere, calcolando il costo necessario per arrivare alla singola frontiera ed il potenziale beneficio che ne verrebbe ricavato. L'obiettivo è quello di massimizzare l'utilità ottenuta una volta raggiunte queste posizioni. L'*utilità* è un valore utilizzato da un robot per valutare le locazioni candidate e selezionare la migliore. I criteri di valutazione possono essere differenti. Un esempio, può essere quello di calcolare la distanza tra la posizione in cui si trova la frontiera e la posizione in cui si trova il robot [42]. La maggior parte dei lavori combina diversi criteri utilizzando funzioni di utilità più complesse. Per esempio, in [33] il costo necessario a raggiungere una locazione candidata è legato in modo lineare ai suoi benefici. Un altro esempio è [13], in cui la distanza di una locazione candidata dal robot ed il potenziale guadagno di informazione in quella locazione sono legati secondo una funzione esponenziale.

Le strategie di esplorazione che riguardano la ricerca ed il soccorso, svolti in modo

autonomo da una squadra di robot, sono relativamente poche in confronto a quelle relative alla costruzione della mappa dell'ambiente. In [23], viene utilizzato come criterio di valutazione delle locazioni candidate il costo di spostamento per raggiungere una locazione, mentre l'utilità delle locazioni (calcolata secondo la vicinanza degli altri robot alla locazione di interesse) è utilizzata in eventuali situazioni di pareggio tra più membri della squadra per la scelta del robot. La strategia di esplorazione per ricerca e soccorso in [7] utilizza un formalismo basato sulle reti di Petri per sfruttare l'informazione a priori, che riguarda la distribuzione delle vittime all'interno dell'ambiente, in modo da migliorare il processo di ricerca. Recentemente sono stati sviluppati degli approcci diretti all'utilizzo di *semantic knowledge* dell'ambiente durante l'esplorazione, che si basano sull'assegnamento di un'etichetta semantica ad ogni zona (per esempio, “stanza” oppure “corridoio”). Nel lavoro [8] gli autori hanno proposto un sistema di esplorazione multirobot che sfrutta la semantic knowledge per portare i robot ad esplorare aree che sono considerate rilevanti secondo le informazioni che sono state fornite a priori da operatori umani. Questo sistema ha ottenuto prestazioni migliori durante l'esplorazione delle aree considerate rilevanti all'interno di un ambiente *indoor* rispetto a molti altri metodi presentati nello stato dell'arte.

2.1.3.2 Metodi di coordinamento

Un altro aspetto legato all'esplorazione dell'ambiente, da parte di una squadra di robot, è quello del coordinamento tra i robot. Un metodo di coordinamento è utilizzato per gestire le interazioni tra i robot e per allocare i compiti all'interno della squadra. I compiti che vengono allocati tra robot possono essere, per esempio, delle locazioni da raggiungere. Uno dei primi lavori nel campo dell'esplorazione multirobot è [42], in cui i robot esplorano in modo non coordinato le frontiere che si trovano più vicine a loro e integrano le loro mappe locali all'interno di una mappa globale dell'ambiente. Alcuni lavori [5, 6] (e in parte anche [11]) propongono un approccio interessante, in cui il metodo di coordinamento è integrato nella strategia di esplorazione. In particolare, il valore di utilità di una locazione candidata è ridotto a seconda del numero di robot che la conosce e che potrebbe raggiungerla. In questo modo, i robot sono spinti a raggiungere locazioni differenti disperdendosi il più possibile all'interno dell'ambiente. I

risultati sperimentali mostrano che, questo comportamento coordinato, fornisce delle prestazioni migliori rispetto a quello non coordinato. Questo è dovuto al fatto che, non essendo coordinati, può succedere che più robot raggiungano la stessa locazione candidata. I metodi di coordinamento, basati su meccanismi di mercato, sono stati studiati estensivamente. Per esempio, in [31] il coordinamento dei robot è realizzato da un esecutore centrale che, oltre a raccogliere le mappe locali e combinarle all'interno di una singola mappa globale, gestisce un'asta chiedendo offerte ai robot ed assegnando loro i compiti (per esempio, le locazioni da raggiungere) secondo le offerte ricevute. Le offerte contengono informazioni riguardo all'utilità potenziale assegnata alle coppie robot-locazione. In questo caso, le utilità sono calcolate come un guadagno potenziale di informazione a cui viene sottratto il costo di spostamento necessario per raggiungerle. In [14] viene proposto un metodo di coordinamento basato su asta, in cui non vengono assegnati solo i compiti, ma viene gestita anche la formazione di coalizioni tra i robot della squadra per raggiungere le locazioni candidate. Nel lavoro [2] gli autori hanno confrontato le prestazioni di diversi metodi di coordinamento all'interno di ambienti *indoor* utilizzando delle varianti del metodo di coordinamento presentato in [40].

2.2 Influenza della comunicazione sulle strategie di esplorazione multirobot

Il problema di esplorazione da parte di più robot, in presenza di una base station (*BS*) fissa a cui consegnare le informazioni raccolte durante l'esplorazione dell'ambiente, è stato affrontato da più autori in modi differenti. In questa sezione, vengono presentati i lavori più rilevanti organizzati in base a come il problema della comunicazione viene inteso.

Molti lavori sono stati sviluppati secondo l'approccio di Yamauchi [42] sull'esplorazione multirobot basata su frontiere, dove l'idea è di avere robot che si muovono verso regioni al confine tra aree libere conosciute e spazi non esplorati, senza alcun vincolo di comunicazione.

Un primo modo di intendere i vincoli di comunicazione è quello di mantenere tutti i robot *continuamente connessi* alla BS, o in modo diretto oppure in modo indiretto attraverso altri robot che si comportano da ripetitori di segnale (metodo *multihop*).

Questo può essere molto utile in situazioni in cui lo streaming di immagini è importante o dove la configurazione dei robot impone che possano essere teleoperati in modo remoto da un operatore durante l'esplorazione. Un esempio di applicazione è quello relativo alle missioni di ricerca e soccorso (s*earch and rescue*), in cui la squadra di robot è alla ricerca di vittime all'interno di un ambiente sconosciuto che ha subito un qualche tipo di catastrofe. In queste missioni, è fondamentale mantenere uno scambio di informazioni continuo tra la squadra di robot e la BS, in quanto i soccorritori devono intervenire tempestivamente appena viene localizzata una vittima dell'incidente. Lo stesso ragionamento vale per missioni in cui l'operatore controlla in modo remoto le azioni di un robot, come per esempio missioni di disinnesco di ordigni esplosivi, in cui gli artificieri vogliono avere uno streaming di immagini e dati in tempo reale da parte del robot, per poterlo controllare in modo remoto nelle migliori condizioni possibili e portare a termine la missione senza causare danni a persone e strutture. Il problema del mantenimento di una connessione continua tra la squadra di robot e la BS è stato studiato in [24] e in [29]. L'algoritmo proposto in [24] costruisce un albero di esplorazione in cui i robot sono organizzati in esploratori ed in stazioni di collegamento (*relay*): gli esploratori sono posizionati sulle foglie dell'albero, mentre le stazioni di collegamento corrispondono ai vertici interni dell'albero ed assicurano la connettività della BS (che è posizionata alla radice dell'albero) con gli esploratori (Figura 2.3). Questo lavoro è stato in seguito esteso per prendere in considerazione una BS in movimento [25]. In [29], invece, gli autori propongono un metodo di ricerca locale in cui l'utilità della configurazione della squadra è calcolata in termini di distanza dalla frontiera più vicina: una configurazione che non permette una piena connettività è altamente penalizzata e, quindi, non viene mai scelta dall'algoritmo.

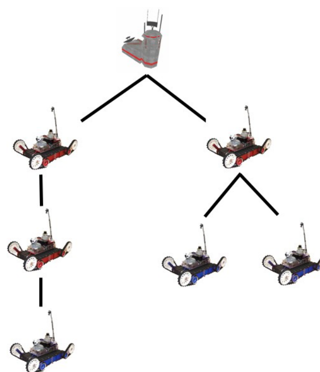


Figura 2.3: esempio di albero di comunicazione tra gli agenti [32].

Un altro modo in cui i vincoli di comunicazione possono essere intesi è per assicurare una connettività globale soltanto quando i robot raggiungono le posizioni di schieramento finali (*posizioni di deployment*). Questo è motivato dal fatto che, di solito, le nuove informazioni sono raccolte al raggiungimento delle posizioni obiettivo e, di conseguenza, i robot possono rimanere disconnessi tra loro, o con la BS, mentre si spostano verso queste posizioni. Questo approccio è utilizzato per applicazioni pratiche in cui non è necessario, o è troppo dispendioso in termini di larghezza di banda, ricevere informazioni in modo continuo da parte della squadra. In queste applicazioni non è possibile controllare in modo remoto i robot, in quanto può accadere che un robot sia disconnesso dalla BS e, di conseguenza, lo streaming di immagini non sia continuo. Quindi, la squadra di robot deve esplorare in modo autonomo l'ambiente sconosciuto (almeno parzialmente). In questo caso, la strategia di esplorazione adottata è spesso di tipo centralizzato. Una strategia di esplorazione di tipo centralizzato è caratterizzata dal fatto che un singolo agente (per esempio, la BS) governa il sistema, raccogliendo ed elaborando i dati ottenuti dagli altri robot della squadra e scegliendo, per esempio, le prossime locazioni all'interno dell'ambiente conosciuto che ogni robot deve raggiungere. In [16] gli autori studiano il problema di posizionamento di sensori mobili per massimizzare la copertura di un'area sconosciuta mantenendo ogni vertice connesso alla BS tramite vincoli di visibilità multihop. L'algoritmo procede posizionando sequenzialmente i vertici, dopo avere scelto le posizioni obiettivo migliori utilizzando delle semplici regole di selezione. Un lavoro più recente, legato al posizionamento di vertici di comunicazione, è stato presentato da Stump et al. [34]. In questo lavoro un insieme di agenti viene assunto essere già presente in un ambiente che sta esplorando e i due problemi affrontati sono (1) trovare un posizionamento dei vertici interni dell'albero (i robot relay) all'interno dell'ambiente, in modo che sia assicurata una connettività globale tra ogni agente e la BS (utilizzando un vincolo di mutua visibilità) e (2) dati un posizionamento attuale dei robot e nuove posizioni obiettivo che i robot devono raggiungere, trovare un riposizionamento che minimizza il tempo impiegato per lo spostamento dei robot. Due robot sono in mutua visibilità se, sul segmento che li unisce, non sono presenti ostacoli e se la loro distanza è inferiore ad una certa soglia. Il problema (1) è ridotto al calcolo di un albero di Steiner minimo, in cui le posizioni degli agenti fanno parte dell'insieme dei vertici terminali, mentre il problema (2) è risolto con

l'utilizzo di un algoritmo di programmazione dinamica (in generale subottimo). In Figura 2.4 è mostrato un esempio di applicazione di questo algoritmo. Infine, in [26] gli autori propongono un metodo per un'esplorazione multirobot che assicura, oltre ad una connettività totale dalle frontiere alla BS, una larghezza di banda sufficiente per la trasmissione dei dati sulla catena dei vertici relay. Questo è ottenuto dividendo il problema in tre sottoproblemi (piazzamento degli esploratori, piazzamento dei relay e generazione del percorso) che vengono risolti come una variazione di problemi di ottimizzazione combinatoria noti.

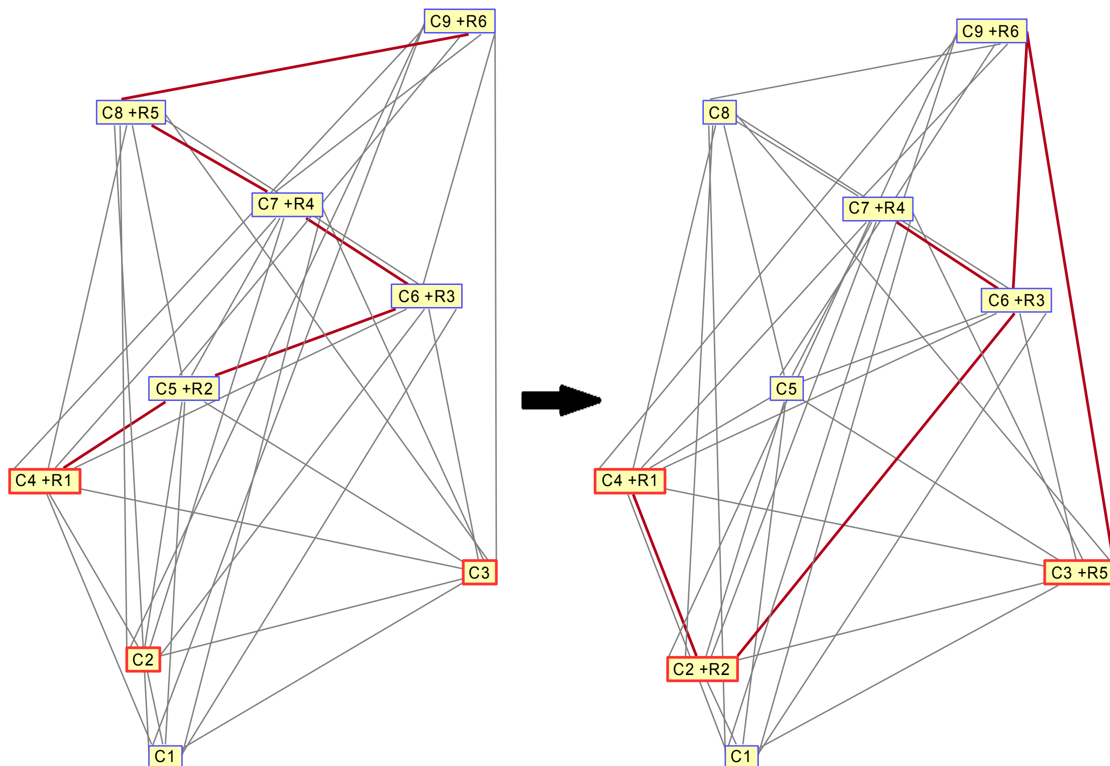


Figura 2.4: C_i indica la locazione *i* all'interno dell'ambiente; R_j indica il robot *j*; gli archi neri indicano le connessioni tra le locazioni dell'ambiente; gli archi rossi indicano la comunicazione tra i robot; le celle con contorno rosso indicano le posizioni che devono essere raggiunte dai robot. La figura di sinistra mostra lo schieramento iniziale di 6 robot ed il loro albero di comunicazione. La figura di destra mostra come cambiano lo schieramento dei robot e il loro albero di comunicazione dopo l'applicazione dell'algoritmo di Stump et al. [34].

Un terzo modo in cui i vincoli di comunicazione sono stati considerati è la *riconnessione periodica*. Ai robot è concesso esplorare molte regioni in autonomia, ma devono essere capaci di comunicare le loro scoperte alla BS con una certa regolarità. In

questo contesto, la strategia di esplorazione è spesso di tipo decentralizzato. Una strategia di esplorazione decentralizzata è basata sulla collaborazione degli agenti con i compagni di squadra che condividono tra loro, per esempio, le nuove informazioni dell'ambiente raccolte durante l'esplorazione. Ogni agente calcola in modo autonomo la prossima locazione da raggiungere elaborando i dati dell'ambiente in suo possesso. Il lavoro proposto in [15] considera uno scenario generale di una missione, in cui i robot devono recuperare la connettività globale con la BS dopo un intervallo di tempo fisso. Gli autori dimostrano la non approssimabilità del problema e propongono un algoritmo euristico basato sulla pianificazione dei percorsi dei robot a turno, scegliendo il miglior percorso da un insieme di alternative utilizzando una funzione di utilità, la quale può essere legata al guadagno di informazione (*information gain*) del percorso. In [10] e [32] gli autori considerano una connettività periodica come una condizione asincrona che, anche se desiderata, non è espressa nella forma di un vincolo di comunicazione esplicito. Infatti, è solo il risultato di un comportamento emergente dall'algoritmo che porta il robot a comunicare periodicamente con la BS. Nello specifico, [10] propone la strategia di esplorazione chiamata *Role-Based Exploration*, in cui ai robot è permesso esplorare l'ambiente senza prendere in considerazione i limiti di comunicazione, e i punti di incontro (*punti di rendezvous*) tra esploratori e relay permettono aggiornamenti asincroni della mappa dell'ambiente della BS. In [32], il comportamento dei robot è regolato da una funzione di utilità che considera la quantità di informazione che un robot non ha ancora consegnato alla BS e la quantità di informazione che il robot suppone che la BS conosca. Cambiando un parametro, il pianificatore è in grado di specificare strategie che variano da una esplorazione completamente *greedy*, senza alcun ritorno alla BS, fino ad una esplorazione che assicura il massimo aggiornamento delle informazioni alla BS. Un esempio di come il cambiamento del parametro influisce sull'esplorazione dell'ambiente è illustrato in Figura 2.5. L'aggiornamento delle informazioni, in modo asincrono, è considerato all'interno dei lavori [3] e [18]. Il primo propone un sistema di tipo *behaviour-based*, che è stato testato in scenari con un aumento dell'informazione preliminare conosciuta dell'ambiente. Il secondo, anche se non considera esplicitamente una BS fissa, è capace di raggiungere un'esplorazione completa di un ambiente sconosciuto con una architettura che si basa su un piccolo insieme di comportamenti e messaggi scambiati tra i robot. In entrambi i lavori, un

comportamento specifico è incaricato di recuperare la connettività con gli altri robot quando viene persa. Un sistema behaviour-based è composto da un insieme di comportamenti (*behaviours*). Ogni comportamento è un processo o una legge di controllo che porta a realizzare e/o mantenere degli specifici obiettivi. Per esempio, il comportamento “evita_ostacoli” mantiene l'obiettivo del robot di evitare le collisioni, mentre il comportamento “ritorna_BS” realizza l'obiettivo di raggiungere la BS [22].

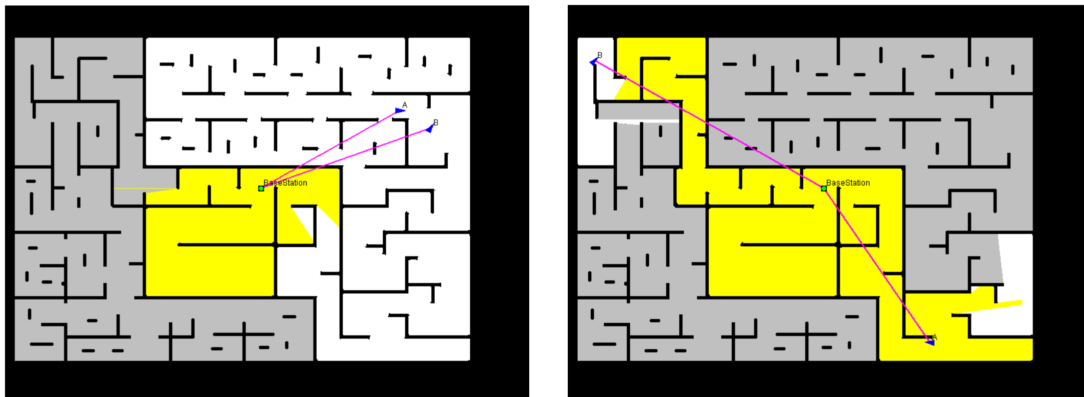


Figura 2.5: a sinistra è visibile una esplorazione quasi completamente greedy, in cui i robot tornano raramente alla BS per comunicare le nuove informazioni. A destra è rappresentata una esplorazione in cui i robot tornano molto più frequentemente alla BS. L'area gialla rappresenta la parte di ambiente conosciuta dalla BS; quella bianca rappresenta la parte di ambiente esplorata dai robot, ma non conosciuta dalla BS.

In conclusione, sono molte e diverse le strategie di esplorazione multirobot proposte in letteratura. Negli anni, sono stati portati a termine molti lavori di confronto del comportamento di una strategia al variare dell'ambiente di esplorazione, delle dimensioni della squadra di robot o di un parametro che caratterizza la strategia stessa [14, 32, 34, 42]. Sono stati portati a termine anche dei lavori di valutazione degli effetti di modelli di comunicazione diversi sulle prestazioni dell'esplorazione [39]. Ciò che non è ancora stato portato a termine, in modo completo, è un lavoro di confronto tra strategie di esplorazione che impiegano vincoli di comunicazione differenti. I vincoli di comunicazione, infatti, incidono sulle prestazioni della strategia di esplorazione e di conseguenza portano alla variazione di alcune metriche come, per esempio, il percorso effettuato dai robot per raggiungere le posizioni obiettivo, il tempo necessario per il calcolo delle nuove posizioni da raggiungere, il tempo di disconnessione di un robot dalla BS, ... Viene, quindi, spontaneo domandarsi *quanto* questi vincoli di

comunicazione incidano sulle prestazioni delle strategie di esplorazione.

L'obiettivo principale della tesi è proprio quello di capire meglio come i vincoli di comunicazione influiscono sulle prestazioni dell'esplorazione, proponendo un confronto sperimentale di alcune strategie di esplorazione che impiegano vincoli di comunicazione diversi. Abbiamo testato le strategie in ambienti simulati differenti e facendo variare alcuni parametri. Abbiamo raccolto i dati e fatto un confronto tra le diverse strategie evidenziandone punti di forza e di debolezza in modo da offrire un aiuto nello sviluppo futuro di strategie di esplorazione migliori.

Capitolo 3

Impostazione del problema di ricerca

Questo capitolo è diviso in due parti. Nella prima parte, forniamo una definizione formale dei due tipi di vincoli di comunicazione adottati dalle strategie di esplorazione che abbiamo confrontato. Nella seconda parte, invece, forniamo una descrizione delle diverse rappresentazioni dell'ambiente utilizzate nella tesi.

3.1 Definizione dei vincoli di comunicazione adottati

Come già detto, il tipo di comunicazione tra i robot della squadra influenza le prestazioni dell'esplorazione.
Diamo, quindi, una definizione dei due tipi di vincoli di comunicazione, chiamati *rigido* (hard) e *morbido* (soft), che sono adottati dalle diverse strategie di esplorazione che abbiamo confrontato.

3.1.1 Vincoli di comunicazione rigidi

Siccome la nuova informazione è ottenuta, solitamente, alle frontiere tra la parte di ambiente conosciuta e quella sconosciuta, è naturale estendere il concetto di esplorazione basata su frontiera [42] alla definizione di vincolo di comunicazione rigido con la BS.

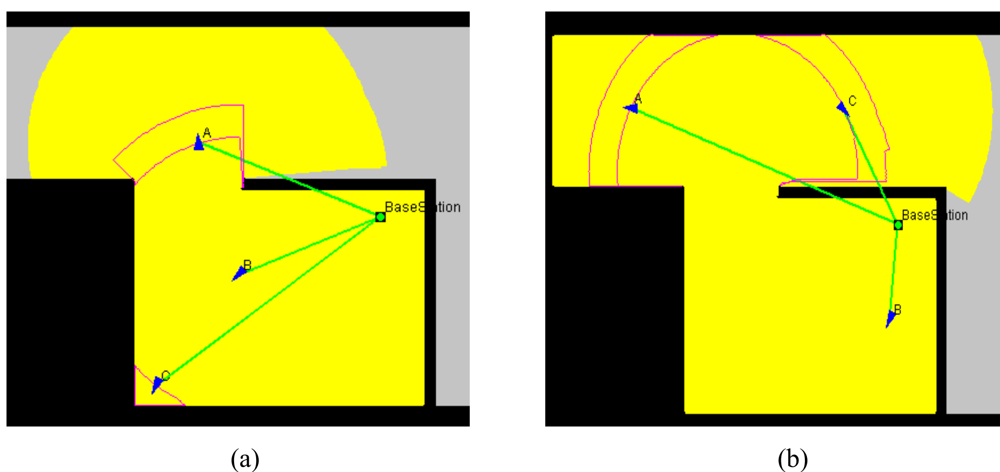
Definizione 1. *I vincoli di comunicazione sono detti rigidi se impongono che, quando un robot acquisisce qualche informazione in una locazione qualsiasi, deve essere in grado di trasmetterla alla BS (in modo diretto oppure attraverso un percorso multihop) da quella stessa locazione.*

In questo caso, la BS mantiene sempre la conoscenza più recente riguardo all'ambiente esplorato e può calcolare periodicamente percorsi globali per i robot.

Non abbiamo formulato esplicitamente il vincolo rigido più forte in modo tale che i robot durante l'esplorazione siano obbligati ad essere connessi alla BS in qualsiasi momento, per esempio anche quando si muovono da una locazione all'altra. Tuttavia, questo può essere visto come un caso speciale della nostra definizione.

Questo tipo di esplorazione è caratteristico delle strategie di tipo centralizzato, in cui la BS calcola lo schieramento finale che i robot devono raggiungere all'interno dell'ambiente conosciuto. All'arrivo nelle posizioni obiettivo, la squadra deve essere connessa alla BS. Tra le strategie di esplorazione che abbiamo testato, si comportano in questo modo quelle di Stump et al. [34] e di Rooker e Birk [29], i cui modelli vengono illustrati con maggiore dettaglio, rispettivamente, nelle sezioni 4.2 e 4.3.

In Figura 3.1. è mostrato un esempio di strategia di esplorazione che adotta un vincolo di comunicazione rigido.



(a) (b)

Figura 3.1: esempio di strategia di esplorazione caratterizzata da un vincolo di comunicazione rigido. I robot trasferiscono la nuova informazione alla BS dalla locazione raggiunta. In (a) e in (b) il robot A trasferisce la nuova informazione alla BS con un collegamento multihop (A → B → BS). Il robot C, invece, nel caso (a) trasferisce l'informazione alla BS in modo diretto, mentre nel caso (b) trasferisce l'informazione alla BS con un collegamento multihop (C → A → B → BS).

3.1.2 Vincoli di comunicazione morbidi

Definiamo i vincoli di comunicazione morbidi nel modo seguente:

Definizione 2. *I vincoli di comunicazione sono detti morbidi se la comunicazione tra la BS e i robot, nonostante sia una condizione desiderata, non ha bisogno di essere mantenuta su base regolare.*

Secondo questa definizione, un robot può esplorare più di una porzione di ambiente prima di trasferire l'informazione alla BS attraverso un collegamento diretto o tramite un relay. In altre parole, la comunicazione con la BS non è espressa nella forma di un vincolo reale, ma è il risultato di un comportamento emergente dalla strategia di esplorazione. Di solito, viene impostata una soglia di tempo o una quantità di area esplorata limite che, una volta superata, impone al robot di tornare alla BS per comunicare. Tra le strategie di esplorazione che abbiamo testato, questo comportamento viene adottato dalla strategia Utility, il cui modello è illustrato nel dettaglio nella Sezione 4.4.

In Figura 3.2 è mostrato un esempio di strategia di esplorazione che adotta un vincolo di comunicazione morbido.

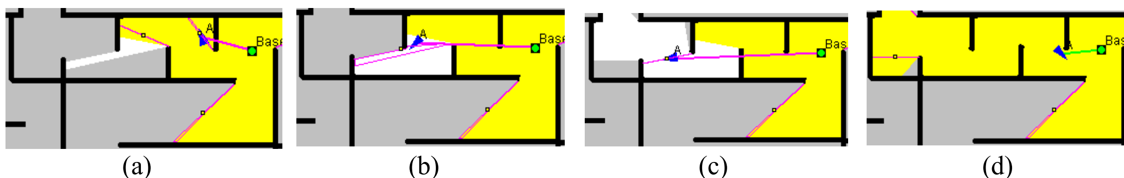


Figura 3.2: esempio di strategia di esplorazione caratterizzata da un vincolo di comunicazione morbido. Il robot A, infatti, esplora tre frontiere prima di tornare a comunicare le nuove informazioni raccolte alla BS.

3.2 Ambiente, rappresentazioni e loro relazioni

Come visto nel Capitolo 2, il problema relativo all'esplorazione di un ambiente sconosciuto e alla consegna delle nuove informazioni raccolte ad una base station fissa è stato formulato in modi differenti. Senza tenere conto di come sono stati espressi i vincoli di comunicazione, l'impostazione del problema generale che consideriamo nella tesi può essere formalizzato in termini dei seguenti elementi comuni:

1. un ambiente bidimensionale, continuo e connesso;

2. una base station *BS* fissa, che si trova all'interno dell'ambiente;
3. un insieme m di robot mobili equipaggiati di sensori che si muovono all'interno dell'ambiente. Da qualsiasi posizione, un robot può percepire l'ambiente che lo circonda ed aggiornare una mappa che tiene traccia della porzione di ambiente che ha scoperto. Assumiamo che la percezione avvenga a tempo discreto. Questo significa che, dato un percorso coperto da un robot, le percezioni sono eseguite solo su un sottoinsieme finito di punti di scansione lungo il percorso. Indichiamo con A l'insieme di tutti gli agenti composto dagli m robot e dalla BS;
4. un meccanismo di comunicazione, il quale permette ai robot e alla BS di comunicare tra loro direttamente, oppure in modo multihop. In questa tesi, assumiamo che la larghezza di banda di un collegamento di comunicazione sia sempre sufficiente per comunicare tutta l'informazione.

Vediamo più nel dettaglio quali sono le diverse rappresentazione dell'ambiente utilizzate nella tesi e come sono relazionate tra loro.

3.2.1 Rappresentazione a griglia

Una prima rappresentazione a griglia dell'ambiente è contenuta all'interno del simulatore. Le celle della griglia sono ordinate all'interno di un sistema di riferimento cartesiano ed ognuna è identificata univocamente da una posizione (x, y). L'origine del sistema si trova nell'angolo in alto a sinistra della griglia. Le ascisse crescono lungo la direzione destra, mentre le ordinate crescono spostandosi verso il basso.

Le celle della griglia possono assumere diversi valori. Possono appartenere ad ostacoli di dimensioni arbitrarie (valore *obstacle*) il cui insieme è indicato con Env_o oppure possono appartenere allo spazio libero (valore *free*) il cui insieme è indicato con $Env_f = Env \setminus Env_o$.

Possiamo, quindi, indicare l'insieme Env delle celle della griglia nel modo seguente:

$$Env = Env_f \cup Env_o$$

dove, ovviamente, $Env_f \cap Env_o = \emptyset$.

Questa rappresentazione dell'ambiente è accessibile soltanto al simulatore ed è utilizzata per operazioni di controllo della simulazione, ovvero:

- verificare la connessione tra due robot (per esempio, per permettere o meno lo

scambio di informazioni tra i robot);

- verificare che il prossimo passo scelto da un robot sia fattibile (per esempio, per evitare che il robot vada ad urtare un ostacolo oppure esca dalla mappa).

Questa rappresentazione è mostrata in Figura 3.3.

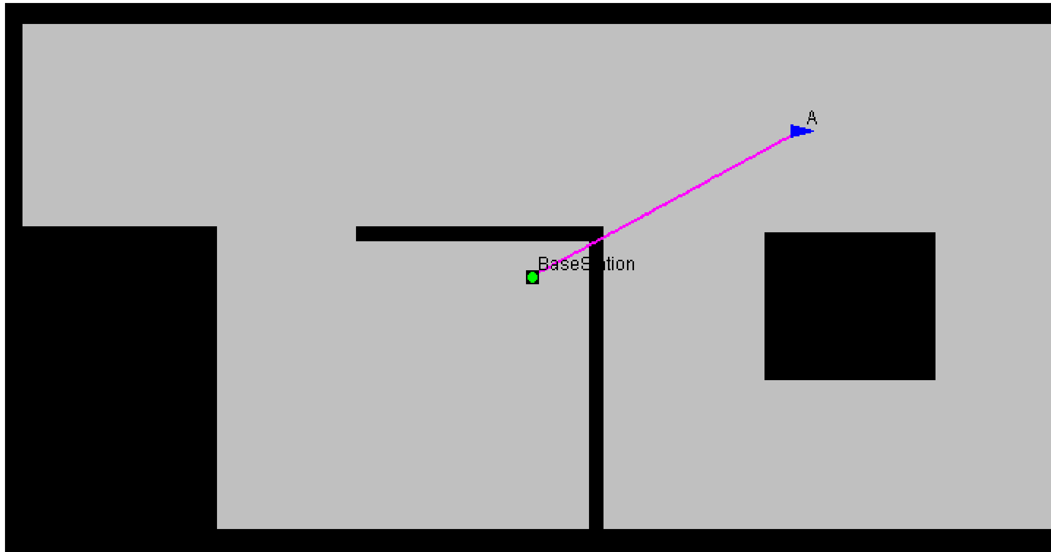


Figura 3.3: rappresentazione a griglia dell'ambiente interna al simulatore. Le celle grigie sono le celle libere (*free*) appartenenti all'insieme Env_f e quelle nere sono quelle occupate (*obstacle*) appartenenti all'insieme Env_o. La linea viola indica la disconnessione del robot A dalla BS (in quanto il meccanismo di comunicazione, in questo caso, è basato su linea di vista).

Un'altra rappresentazione a griglia dell'ambiente si trova all'interno del singolo agente (robot esploratori e BS). Anche in questo caso le celle della griglia sono ordinate all'interno di un sistema di riferimento cartesiano ed ogni cella è identificata univocamente dalle sue coordinate (x, y). Tuttavia, la singola cella appartenente a questa griglia può assumere valori differenti rispetto a quelli visti in precedenza. Chiamiamo questa rappresentazione dell'ambiente *Occ* (occupancy grid), per differenziarla da *Env*. Una cella all'interno di *Occ* può essere libera (*free*), occupata (*obstacle*) o sicura (*safe*). Questi insiemi sono rappresentati, rispettivamente, dai simboli Occ_f, Occ_o e Occ_s.
Indichiamo con Occ^t_r l'insieme delle celle dell'ambiente conosciute dal robot *r* all'istante di tempo *t* e lo esprimiamo come:

$$Occ^t_r = Occ^t_{f\,r} \cup Occ^t_{o\,r} \cup Occ^t_{s\,r}$$

dove, ovviamente, $Occ^t_{f\,r} \cap Occ^t_{o\,r} = \emptyset$, $Occ^t_{f\,r} \cap Occ^t_{s\,r} = \emptyset$ e $Occ^t_{s\,r} \cap Occ^t_{o\,r} = \emptyset$. Gli insiemi $Occ^t_{f\,r}$, $Occ^t_{o\,r}$ e $Occ^t_{s\,r}$ contengono rispettivamente le celle *free*, le celle *obstacle* e

le celle *safe* conosciute al generico istante di tempo *t* dal robot *r*.

Sia le celle *safe* che le celle *free* rappresentano lo spazio libero. La distinzione tra celle *safe* e celle *free* viene utilizzata dal robot per la scelta della prossima frontiera da esplorare. Maggiori dettagli sono riportati nella Sezione 5.1.1.

L'aggiornamento della rappresentazione *Occ* viene portato a termine dal simulatore utilizzando la rappresentazione *Env*. Le celle *safe* e *free*, all'interno di *Occ*, corrispondono alle celle *free* all'interno di *Env*. Le celle *obstacle*, all'interno di *Occ*, corrispondono alle celle *obstacle* all'interno di *Env*. La differenza principale tra una cella *safe* ed una cella *free* è dovuta alla distanza della cella dal robot durante la fase di percezione dell'ambiente. Il robot, infatti, possiede un raggio di percezione ed un raggio safe, chiamato *safe range* (che è inferiore al raggio di percezione). In generale, tutte le celle *free* all'interno di *Env*, che si trovano, durante la fase di percezione, ad una distanza dal robot inferiore al safe range, sono considerate di tipo *safe* in *Occ*, mentre tutte le celle *free* in *Env*, che si trovano ad una distanza maggiore del safe range e ad una distanza minore del raggio di percezione e, inoltre, sono appena state scoperte dal robot, sono considerate di tipo *free* in *Occ*. Maggiori dettagli riguardo l'aggiornamento della mappa sono riportati nella Sezione 5.1.3.

La Figura 3.4 mostra un esempio delle due rappresentazioni dell'ambiente *Env* e *Occ*.

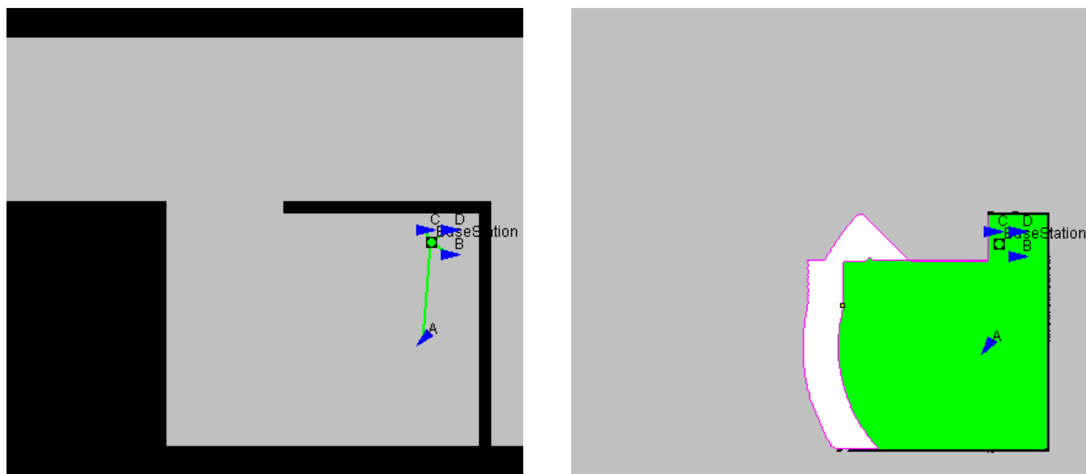


Figura 3.4: a sinistra è mostrata la rappresentazione dell'ambiente *Env*, a destra è mostrata la rappresentazione dell'ambiente *Occ* contenuta nel robot A. Si nota immediatamente che le celle libere in *Env* più vicine ad A assumono valore *safe* all'interno di Occ^t_A (area verde), mentre quelle più lontane assumono valore *free* (area bianca). Le celle *obstacle* in *Env*, assumono valore *obstacle* anche in Occ^t_A (aree nere).

Questa rappresentazione è utilizzata dai robot per diversi scopi, ovvero:

- calcolare e scegliere le frontiere da raggiungere. Vengono utilizzate le frontiere per la scelta della prossima posizione obiettivo che un robot deve raggiungere. Viene calcolato il costo necessario per raggiungere una frontiera ed il possibile beneficio che ne verrebbe ricavato all'arrivo. L'obiettivo è quello di massimizzare una funzione di *utilità*. Queste operazioni vengono descritte con maggiore dettaglio nel Capitolo 5;
- calcolare il percorso verso una posizione finale (quindi, in generale, per spostarsi all'interno dell'ambiente). Infatti, i robot utilizzano la distinzione delle celle *free* e *safe* (che sono celle libere) dalle celle *obstacle* (che sono celle occupate) per il movimento all'interno dell'ambiente, in modo tale da tracciare un percorso fattibile per raggiungere le posizioni obiettivo all'interno dell'area conosciuta senza, per esempio, scontrarsi con gli ostacoli presenti nell'ambiente;
- verificare la comunicazione tra due celle all'interno dell'ambiente conosciuto. Questa funzione può essere utilizzata dalla BS per calcolare uno schieramento finale dei robot tale per cui tutti gli agenti, al raggiungimento delle posizioni obiettivo, siano in comunicazione con la BS (in modo diretto o in modo multihop). Questo viene fatto nel caso in cui venga adottata una strategia di esplorazione centralizzata, in cui il vincolo di comunicazione è di tipo rigido.

A questa rappresentazione dell'ambiente possono accedere, in modo diretto, i singoli agenti ed, in modo indiretto, il simulatore. Il simulatore, infatti, può accedere a tutte le informazioni possedute da qualsiasi agente e, di conseguenza, anche alle mappe dell'ambiente (per esempio, per verificare se l'area conosciuta dalla BS ha superato una certa soglia che permetta di interrompere l'esplorazione).
Nella Figura 3.5 è mostrata questa rappresentazione dell'ambiente.

Questa rappresentazione è utilizzata, in generale, dalle strategie di esplorazione che adottano vincoli di comunicazione morbidi, le quali non necessitano di nessun'altra rappresentazione dell'ambiente per portare a termine l'esplorazione rispettando i vincoli di comunicazione.
Le strategie di esplorazione con vincoli di comunicazione rigidi, invece, adottano questa rappresentazione come fondamento per la costruzione di un'ulteriore rappresentazione

dell'ambiente utilizzata per rispettare i vincoli di comunicazione, come vediamo più nel dettaglio nella sezione successiva.

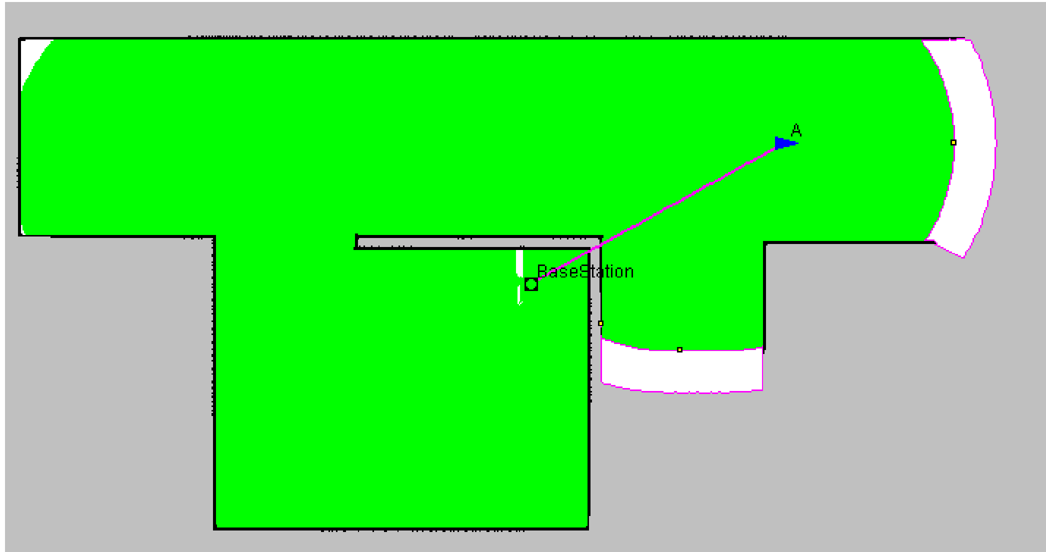


Fig 3.5: rappresentazione della mappa dell'ambiente *Occ* conosciuta dal robot A ad un generico istante di tempo t. L'area verde è la parte sicura di ambiente (*safe*) e l'area bianca è la parte libera di ambiente (*free*). I contorni neri sono i confini degli ostacoli (*obstacle*).

3.2.2 Rappresentazione topologica

La rappresentazione dell'ambiente che segue si trova su un altro livello rispetto a quella presentata nella sezione precedente. Questa rappresentazione viene utilizzata dalle strategie di esplorazione di tipo centralizzato, in cui è la BS che decide le posizioni obiettivo che i robot devono raggiungere.

Vengono prese in considerazione soltanto alcune celle della griglia *Occ*, appartenente alla BS, che sono considerate rappresentative. Questo viene fatto perché, per esempio, sarebbe troppo costoso calcolare le posizioni obiettivo da assegnare ai robot prendendo in considerazione tutte le celle conosciute appartenenti ad *Occ*. Di seguito, descriviamo più nel dettaglio come vengono scelte le locazioni rappresentative.

Rappresentiamo l'ambiente come un grafo $G = (V, E)$, dove V è l'insieme delle locazioni rappresentative dell'ambiente ed E è l'insieme delle connessioni fisiche tra i vertici del grafo. Dunque, se $(v_i, v_j) \in E$ (con $v_i, v_j \in V$), allora un robot può viaggiare direttamente dal vertice v_i al vertice v_j coprendo una distanza che chiamiamo $d(v_i, v_j)$. In altre parole,

se $(v_i, v_j) \in E$, allora i vertici v_i e v_j sono in linea di vista, cioè sul segmento che unisce i due vertici non sono presenti ostacoli. La verifica della visibilità tra due vertici $v_i, v_j \in V^t$ e, quindi, la verifica dell'esistenza dell'arco $(v_i, v_j) \in E^t$, viene eseguita sulla mappa dell'ambiente Occ^t_{BS} della BS.

In aggiunta a questo, consideriamo un insieme C di *collegamenti di comunicazione*. Questo insieme specifica quali vertici possono comunicare all'interno del grafo G. Più formalmente, se $(v_i, v_j) \in C$, allora un robot che si trova a v_i può comunicare con un robot che si trova a v_j, e viceversa. Assumiamo che il meccanismo di comunicazione sia tale che i collegamenti di comunicazione siano basati sulle connessioni fisiche tra i vertici del grafo. Più formalmente, $C \subseteq E$, dove C può essere calcolato da E. Il generico arco $(v_i, v_j) \in C^t$ viene calcolato nel modo seguente: se esiste l'arco $(v_i, v_j) \in E^t$ e i due vertici $v_i, v_j \in V^t$ si trovano ad una distanza inferiore ad una certa soglia all'interno della mappa Occ^t_{BS}, allora tra questi è presente il collegamento di comunicazione $(v_i, v_j) \in C^t$.

Il grafo G è inizialmente sconosciuto alla BS e deve essere scoperto in modo incrementale componendo le percezioni dei robot acquisite in istanti di tempo discreti e comunicate alla BS. Ad un generico istante di tempo t, indichiamo con G^t la porzione di grafo conosciuta dalla BS al tempo t. Questo grafo può essere definito come un sottografo di G, dove gli insiemi E^t e C^t sono versioni ristrette di E e C tali che solo i vertici esplorati all'istante t o prima, indicati con V^t, sono inclusi.

Indichiamo come *schieramento* della squadra (deployment) all'istante di tempo t, l'insieme $Q^t = \{b, q^t_1 ... q^t_m\}$, dove $b \in V^t$ indica la posizione fissa della BS e $q^t_i \in V^t$ indica la posizione (vertice) occupata dal robot i all'interno di G^t prima della fase di percezione dell'ambiente. Una volta che ogni robot ha raggiunto la propria posizione di schieramento specificata in Q^t, vengono acquisite le percezioni e, assumendo che ogni robot possa trasferire i dati percepiti alla BS, il grafo G^{t+1} è calcolato dalla BS. Questo grafo viene ottenuto unendo tutte le nuove percezioni compiute dai robot che occupano i vertici frontiera, ovvero quei vertici che rappresentano le locazioni in V^t ai confini tra l'ambiente conosciuto e l'ambiente sconosciuto. L'unione di queste percezioni è utilizzata dalla BS per la costruzione della mappa Occ^t_{BS} che la BS utilizza per calcolare le frontiere che dovranno essere esplorate successivamente. Per ogni frontiera f trovata dalla BS, viene generato il vertice v_f che rappresenta la locazione della frontiera. I vertici frontiera vengono aggiunti all'insieme dei vertici V^t del grafo ottenendo così

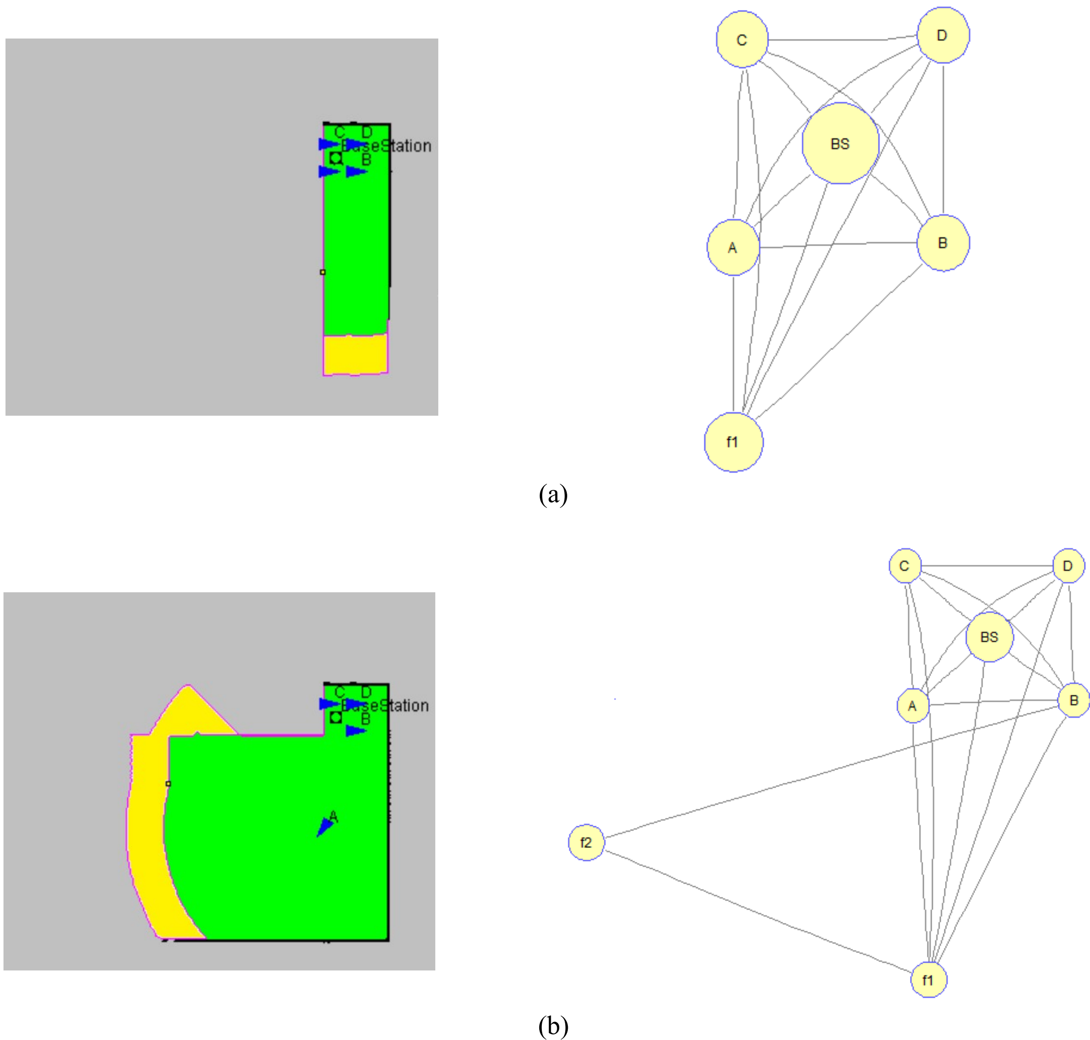
l'insieme V^{t+1}. Quindi, alcuni vertici presenti in V^{t+1} rappresentano locazioni di frontiere, mentre altri rappresentano locazioni interne all'ambiente già conosciuto dalla BS.

Per un vertice $v \in V^t$, indichiamo con $N(v)$, $\xi(v)$ e $\zeta(v)$, l'insieme di vertici fisicamente adiacenti a *v*, l'insieme di archi fisici di *E* incidenti a *v* in *G* e l'insieme di collegamenti di comunicazione di *C* incidenti a *v* in *G*, rispettivamente. Allora, possiamo esprimere le percezioni come la transizione da G^t a G^{t+1} con le seguenti Equazioni:

$$V^{t+1} = V^t \bigcup \cup_{a \in A} N(q_a^t); \quad E^{t+1} = E^t \bigcup \cup_{a \in A} \xi(q_a^t); \quad C^{t+1} = C^t \bigcup \cup_{a \in A} \zeta(q_a^t) \quad \text{(ii)}$$

Data questa rappresentazione dell'ambiente, i vincoli di comunicazione rigidi possono essere espressi come un requisito di *fattibilità* per qualsiasi schieramento della squadra. Ad ogni istante di tempo *t*, una volta che il grafo G^{t+1} è stato calcolato integrando in G^t le percezioni relative allo schieramento Q^t, gli schieramenti fattibili Q^{t+1} sono quelli per cui i sottografi di G^{t+1}, indotti da Q^{t+1}, sono connessi rispetto a C^{t+1}. In questi schieramenti Q^{t+1}, ogni robot può comunicare (direttamente o attraverso un percorso multihop) con la BS la cui posizione è fissa in *b* ad ogni schieramento.

In conclusione, il livello di rappresentazione topologico viene generato partendo dal livello di rappresentazione a griglia appartenente alla BS da cui vengono prese delle locazioni rappresentative ed aggiunte all'insieme dei vertici del grafo. La BS utilizza la rappresentazione topologica dell'ambiente per il calcolo dei nuovi schieramenti dei robot. Per quanto riguarda invece il calcolo delle nuove frontiere e la verifica della comunicazione, la BS utilizza la propria occupancy grid. In Figura 3.6 è mostrato un esempio di aggiornamento del grafo *G* e in Figura 3.7 sono mostrate le diverse rappresentazioni di ambiente.



(a)

(b)

Figura 3.6: esempio di aggiornamento del grafo G^t tramite la rappresentazione Occ^t_{BS}. I vertici del grafo, che rappresentano i punti significativi dell'ambiente sono chiamati con lettere che identificano le posizioni iniziali occupate dagli agenti (*A*, *B*, *C*, *D*, *BS*), mentre i vertici frontiera chiamati *f1* ed *f2* sono relativi alla prima frontiera scoperta e alla seconda frontiera scoperta. Nell'immagine (a), la BS individua la frontiera *f1* ed invia il robot A verso di essa. Al raggiungimento dello schieramento finale, la BS integra le nuove informazioni dell'ambiente scoperte da A all'interno della propria mappa ed individua un'altra frontiera *f2*, come si vede nella figura (b). Di conseguenza, viene aggiornato il grafo G^t presente in (a) con l'aggiunta del vertice frontiera *f2* all'insieme V^t e degli archi (*f2*, *B*) ed (*f2*, *f1*) agli insiemi E^t e C^t (come visto in (ii)), ottenendo così il grafo $G^{t+1} = (V^{t+1}, E^{t+1})$ mostrato in (b). Il vertice frontiera *f2*, nella figura (b), è connesso solo al vertice *B* e al vertice *f1* perché sono gli unici vertici con cui si trova in linea di vista attraverso l'area *conosciuta* dell'ambiente.

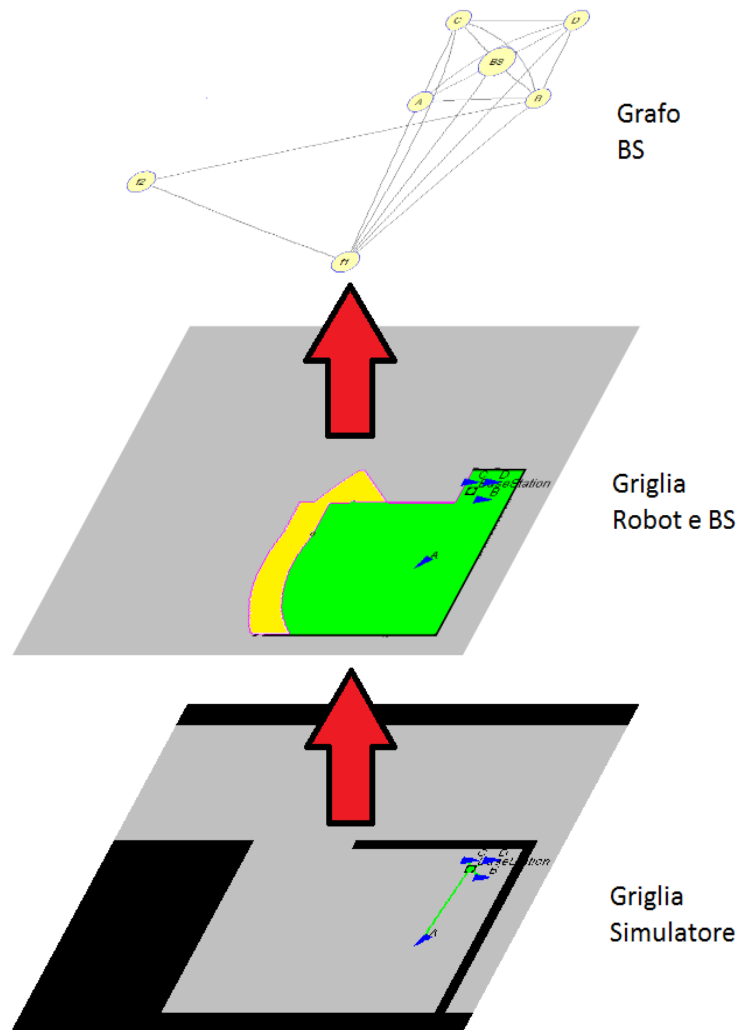


Figura 3.7: diversi livelli di rappresentazione dell'ambiente ad un generico istante di tempo *t*.

Capitolo 4

Descrizione dei metodi utilizzati

In questo capitolo, descriviamo i modelli appartenenti alle strategie di esplorazione testate basandoci sulle rappresentazioni dell'ambiente descritte nel Capitolo 3.

4.1 Programmazione lineare intera

Data la rappresentazione topologica dell'ambiente (Sezione 3.2.2), utilizzata dalle strategie di esplorazione che adottano un vincolo di comunicazione rigido, è naturale interpretare il problema dell'esplorazione di un ambiente sconosciuto in modo centralizzato, assumendo che l'esplorazione sia dettata da un processo di pianificazione che risiede all'interno della BS.

Da questo punto di vista, la BS svolge il compito di calcolare lo schieramento Q^{t+1} della squadra di robot, dati il grafo G^{t+1} costruito dall'unione delle percezioni comuni e lo schieramento attuale Q^t. Nel fare questo, possiamo definire le misure di costi e guadagni che un nuovo schieramento può introdurre. I costi possono essere definiti senza incertezza, essendo legati alla transizione dal vecchio schieramento a quello nuovo. Possiamo definire, semplicemente, il costo di un nuovo schieramento come la distanza totale percorsa dai robot per raggiungere le posizioni che sono state loro assegnate nello schieramento Q^{t+1} a partire dalle posizioni che attualmente occupano nello schieramento Q^t. I guadagni, invece, possono essere soltanto stimati poiché dipendono dalla quantità di nuova informazione che può essere scoperta con una percezione comune al raggiungimento dello schieramento Q^{t+1}. Come fatto solitamente per le strategie di

esplorazione standard, assumiamo di avere una qualche stima del guadagno di informazione ottenibile da una percezione compiuta ad un vertice v e la indichiamo con $g(v)$. Ovviamente, per i vertici che non sono frontiere, avremo $g(v) = 0$.

Tuttavia, come abbiamo già discusso, il vincolo di comunicazione rigido richiede, in generale, di assegnare dei robot (relay) ad alcuni dei vertici che non portano informazione, in modo da connettere tutti i robot alla BS e rispettare i requisiti di connessione. Assumendo di cercare una soluzione ottima rispetto ad un dato compromesso tra costi e guadagni, possiamo formulare questo problema come un problema di *programmazione lineare intera* (*PLI*).

Per questo tipo di formulazione, adottiamo due insiemi di variabili di decisione binarie da cui possiamo derivare la soluzione del problema di pianificazione Q^{t+1} dopo che il grafo G^{t+1} è stato calcolato:

- y_{av} per un robot $a \in A$ e un vertice $v \in V^{t+1}$, che assume un valore uguale a *1* se e solo se $q_a^{t+1} = v$, dove q_a^{t+1} indica il vertice occupato dal robot a all'interno del grafo G^{t+1};
- x_c per un collegamento di comunicazione $c = (v_i, v_j) \in C^{t+1}$, che assume un valore uguale a *1* se e solo se, per qualche $a_i, a_j \in A$, vale che $q_{ai}^{t+1} = v_i$ e $q_{aj}^{t+1} = v_j$.

Indichiamo con α il parametro usato per regolare il compromesso tra costi e guadagni e con $\delta(S)$ il taglio indotto sul grafo G^{t+1} dall'insieme di vertici $S \subseteq V^{t+1}$ quando si considerano i collegamenti di comunicazione C^{t+1}.

Vediamo il modello PLI:

$$maximize \sum_{a \in A} \sum_{v \in V} g(v) y_{av} - \alpha \sum_{a \in A} \sum_{v \in V} d(q_a^t, v) y_{av} \qquad (1)$$

soggetto ai seguenti vincoli :

$$y_{BS,b} = 1 \qquad (2)$$

$$\sum_{a \in A} y_{av} \leq 1 \qquad \forall v \in V \qquad (3)$$

$$\sum_{v \in V} y_{av} = 1 \qquad \forall a \in A \qquad (4)$$

$$\sum_{c \in C} x_c \leq |A| - 1 \qquad (5)$$

$$\sum_{c \in \delta(S)} x_c \geq \sum_{a \in A} y_{av} \qquad \forall S \subseteq V \backslash b, \; S \neq \emptyset, \forall v \in S \qquad (6)$$

La funzione obiettivo (1) è divisa in due parti. Il primo termine massimizza il guadagno di informazione totale che i robot possono ricevere da una percezione comune, mentre il

secondo termine si riferisce al costo della distanza totale percorsa; il vincolo (2) tiene la BS fissa alla sua posizione b; il vincolo (3) assicura che ogni vertice $v \in V$ sia occupato da al massimo un robot $a \in A$; il vincolo (4) forza ogni robot a schierarsi in esattamente un vertice $v \in V$; il vincolo (5) assicura che al massimo $|A|$-1 collegamenti di comunicazione siano costruiti tra i nodi del grafo; infine, il vincolo (6) forza il fatto che, se un robot è schierato in un particolare vertice $v \in V$, allora deve esistere una sequenza connessa di collegamenti di comunicazione che portano alla BS (facciamo notare che, cercare un albero connesso è sufficiente per assicurare la connettività globale).

Il parametro α può essere scelto in modo che, il costo massimo speso da un robot per spostarsi, non sia mai più grande del minimo guadagno di informazione raggiungibile. In questo modo, assicuriamo un ordine di preferenza per i due obiettivi: il guadagno di informazione ha una priorità più alta rispetto al costo di spostamento. Per ottenere questa proprietà adottiamo il seguente valore per α:

$$\alpha = \frac{\min_{v \in V}\{g(v)\} - \varepsilon}{m \cdot \max_{v \in V, a \in A}\{d(q_a^t, v)\}}$$

dove ε è una costante sufficientemente piccola.

4.2 Stump

Questo metodo è basato sul lavoro recente presentato da Stump et al. in [34]. La ragione di questa scelta è la similarità del problema indirizzato da questo lavoro con quello che abbiamo definito nella sezione precedente.

Il metodo proposto utilizza la *programmazione dinamica* per calcolare uno schieramento della squadra di robot, all'interno dell'ambiente conosciuto, provando a minimizzare la distanza totale percorsa e, allo stesso tempo, garantire la connessione tra la BS e le locazioni obiettivo scelte. Le locazioni obiettivo vengono assegnate ad un insieme di robot esploratori. L'algoritmo riceve in ingresso:

- un grafo $G = (V, C)$, dove i vertici $v \in V$ rappresentano possibili locazioni dell'ambiente che i robot possono occupare ed i collegamenti di comunicazione $c = (v_i, v_j) \in C$ rappresentano la possibilità di trasmettere i dati tra le locazioni v_i, $v_j \in V$. In particolare, è possibile trasmettere i dati da una locazione v_i ad una locazione v_j (con v_i, $v_j \in V$) se e solo se le due locazioni sono in linea di vista e la

loro distanza è inferiore ad una certa soglia;

- lo schieramento della squadra attuale Q^t e la topologia di comunicazione attuale C^t. In questo caso, la topologia di comunicazione non è intesa come l'insieme di collegamenti di comunicazione disponibili, ma come l'albero minimale di collegamenti che è realmente utilizzato per trasmettere i dati;
- assegnamenti robot-frontiera per un sottoinsieme di robot.

Nel caso in cui venga trovata una soluzione, il metodo restituisce in uscita un nuovo schieramento Q^{t+1} dei robot, dove le locazioni dei robot preassegnati alle frontiere sono fisse e le posizioni degli altri robot sono calcolate in modo che venga ottenuta una configurazione connessa con la BS ed i costi di spostamento siano minimizzati. Nel caso in cui, invece, non venga trovata una soluzione, allora non viene restituito alcuno schieramento Q^{t+1}.

Per trovare il nuovo schieramento Q^{t+1}, l'algoritmo costruisce una topologia di albero di comunicazione dove la BS è la radice ed i robot preassegnati alle frontiere sono le foglie dell'albero. Invece, i robot relay sono assegnati a vertici intermedi dell'albero (Figura 4.1). I costi di spostamento e di comunicazione degli assegnamenti sono calcolati secondo l'Equazione:

$$Q(V^t, C^t, Q^t, C^{t+1}, Q^{t+1}) = \sum_{a \in A} d(q_a^t, q_a^{t+1}) + \mu\Big(\sum_{(v_i, v_j) \in C^{t+1}} w(q_a^t, q_a^{t+1}) \Big) \qquad \text{(i)}$$

dove C^{t+1} è la nuova topologia di comunicazione che deve essere utilizzata per lo scambio di messaggi tra il robot e la BS, Q^{t+1} è il nuovo schieramento della squadra nell'ambiente, $d(v_i, v_j)$ è la distanza tra un vertice v_i e un vertice v_j, $w(v_i, v_j)$ rappresenta i costi di comunicazione sostenuti dai robot quando vengono inviati i dati dal vertice v_i al vertice v_j utilizzando la topologia di comunicazione C^{t+1}. Infine, μ è un parametro che rappresenta un compromesso tra schieramenti in cui i robot si muovono il meno possibile (piccoli valori di μ) e schieramenti in cui i robot terminano il più vicino possibile tra loro (grandi valori di μ). Questo è dovuto al fatto che, al crescere del valore di μ, cresce l'importanza del termine che rappresenta il costo di comunicazione, e viceversa. L'assegnamento che minimizza la somma di questi costi, se esiste, è scelto come nuovo schieramento Q^{t+1}.

Il costo di comunicazione tra due vertici v_i, $v_j \in V$ è dato da:

$$w(v_i, v_j) = \begin{cases} d(v_i, v_j) + R_{BS} & se (v_i, v_j) \in C, \\ \infty & altrimenti. \end{cases}$$

dove R_{BS} rappresenta la dimensione del raggio di comunicazione della BS. Quindi, se due vertici sono in linea di vista e la loro distanza è inferiore ad una certa soglia, allora il costo di comunicazione ha valore finito ed è uguale alla distanza tra i due vertici a meno di una costante. Altrimenti, ha valore infinito in quanto i due vertici non possono comunicare.

Il metodo [34] appena descritto non può essere adottato in questa forma per il nostro modello di esplorazione, in quanto assume che l'ambiente in cui si muovono i robot sia completamente conosciuto fin dal primo istante di tempo e che siano già presenti degli assegnamenti robot-locazione. Il nostro modello di esplorazione, invece, assume che l'ambiente sia inizialmente sconosciuto e debba essere scoperto in modo incrementale dalla squadra di robot. Di conseguenza, non consideriamo alcun preassegnamento dei robot alle frontiere, ma, invece, lo includiamo nella soluzione che cerchiamo.

A questo fine, adattiamo il metodo proposto in [34] nel modo seguente:

(a) ad ogni istante di tempo discreto t, i robot percepiscono l'ambiente e la BS unisce le loro mappe ottenendo una rappresentazione comune dell'ambiente. Successivamente, la BS trova le frontiere all'interno della rappresentazione comune, genera i vertici frontiera e li aggiunge all'insieme dei vertici V^t. A questo punto, assegna ad ogni robot $a \in A$ un vertice frontiera $v_f \in V^{t+1}$. Un vertice frontiera può essere assegnato al massimo ad un robot per aumentare il più possibile la dispersione dei robot all'interno dell'ambiente e massimizzare il guadagno di nuova informazione. Gli assegnamenti sono determinati con un semplice algoritmo greedy che massimizza l'utilità del vertice frontiera v_f selezionato per un robot a. La funzione di utilità è definita in modo simile a ciò che è stato fatto in [32], ovvero $u(v_f) / d^2(q_a^t, v_f)$, dove $u(v_f)$ è il valore di utilità associato al vertice frontiera v_f e $d(q_a^t, v_f)$ è la distanza tra la posizione occupata al tempo t del robot a, cioè $q_a^t \in Q^t$, ed il vertice frontiera v_f;

(b) viene eseguito a questo punto l'algoritmo in [34] e, se questo assegnamento comune ammette una topologia di comunicazione connessa C^{t+1} quando tutti i robot sono assegnati alle frontiere, allora viene adottata. Altrimenti,

l'assegnamento robot-frontiera con la minor utilità viene rimosso e l'algoritmo in [34] viene eseguito nuovamente. Il processo continua finché non viene trovato uno schieramento con una topologia di comunicazione connessa. Se non viene trovato alcuno schieramento connesso, allora significa che non è possibile continuare oltre ed il processo di esplorazione termina.

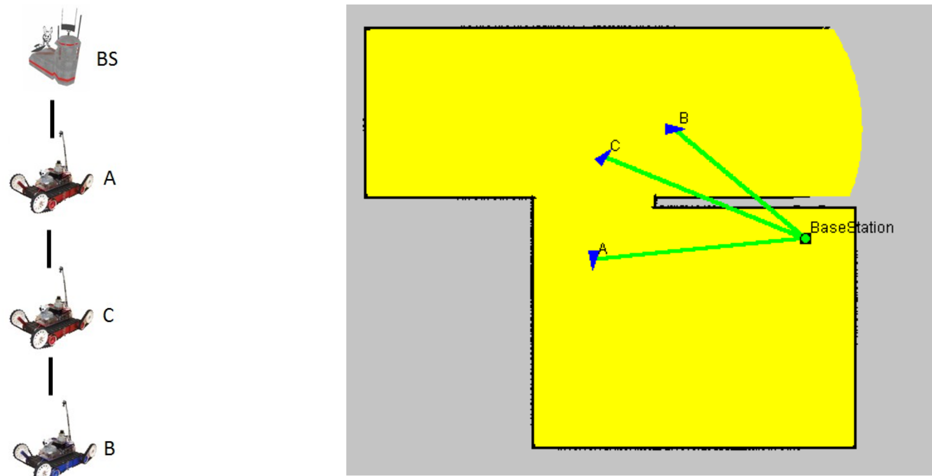


Figura 4.1: a sinistra è mostrato l'albero di comunicazione tra i robot e la BS. Il robot B, che è stato assegnato ad una frontiera, si trova sulla foglia dell'albero, mentre i robot A e C, interni all'albero di comunicazione, sono i relay che si occupano di fare comunicare la foglia con la radice (BS). A destra è mostrata l'applicazione di questo grafo all'interno della rappresentazione a griglia dell'ambiente.

4.3 Birk

Il metodo presentato in [29] è basato su un approccio di *ricerca locale* all'interno della rappresentazione a griglia dell'ambiente vista nella Sezione 3.2.1. Abbiamo assunto che ogni cella i della griglia corrisponda ad un nodo $vi \in V$ del grafo $G = (V, E)$ e le adiacenze tra due celle i e j della griglia (ovvero, tra due vertici v_i, $v_j \in V$) corrispondano ad adiacenze fisiche (ovvero, $(v_i, v_j) \in E$) del grafo G. Il metodo lavora secondo i seguenti passi. Dato lo schieramento attuale della squadra Q^t:

(a) un insieme di schieramenti di squadra candidati Q_1^{t+1}, Q_2^{t+1}, ... Q_k^{t+1} è generato casualmente da Q^t in questo modo: per ogni Q_i^{t+1} $(i \leq 1 \leq k)$, vengono selezionate m locazioni casuali (dove m è il numero di robot) in modo che ogni locazione selezionata o è occupata da un robot in Q^t, oppure è adiacente ad una locazione che ospita un robot in Q^t (per esempio, un robot posizionato in un vertice v o

rimane in *v*, oppure si muove in uno dei vertici adiacenti *N(v)*, come mostrato in Figura 4.2);

(b) per ogni schieramento generato viene calcolata l'utilità $U(Q_i^{t+1})$, e quello che massimizza *U()* è scelto come prossimo schieramento; poi il processo ricomincia dal passo precedente.

Dopo alcuni esperimenti preliminari, abbiamo modificato leggermente la funzione di utilità utilizzata in [29] per migliorare la qualità della soluzione ottenuta. La funzione di utilità che abbiamo adottato è definita, con un piccolo abuso di notazione, come:

$$U(Q_a^{t+1}) = \sum_{a \in A} U(q_a^{t+1})$$

dove:

$$U(q_a^{t+1}) = \begin{cases} -M & \text{se } q_a^{t+1} \text{ non è fattibile,} \\ u(f^*_{q_a^{t+1}})/d^2(q_a^{t+1}, f^*_{q_a^{t+1}}) & \text{altrimenti.} \end{cases} \qquad \text{(ii)}$$

Il ragionamento è il seguente: una locazione assegnata q_a^{t+1} non è *fattibile* se provoca una collisione con un ostacolo oppure se provoca una perdita di connessione con la BS. In questo caso, viene applicata una grande penalizzazione *-M*. Altrimenti, l'utilità è dipendente da $f^*_{q_a^{t+1}}$, ovvero dalla locazione della frontiera più vicina a q_a^{t+1} (*u()* e *d()* hanno lo stesso significato visto nelle sezioni precedenti).

Questo metodo soffre di problemi di minimo locale. In questa tesi sono stati affrontati nel modo seguente: se viene generato per un certo numero di volte consecutive lo stesso schieramento, allora la frontiera con utilità minore viene rimossa dall'insieme delle frontiere disponibili e l'algoritmo si ripete.

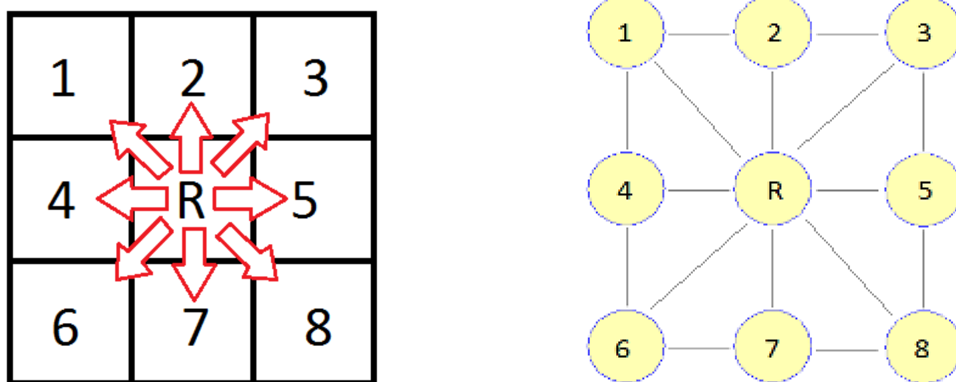


Figura 4.2: il robot *R* può rimanere fermo o muoversi lungo le 8 direzioni (rappresentate con delle frecce) all'intero della griglia a sinistra. A destra è presente la rappresentazione della griglia come un grafo in cui i vertici sono le celle della griglia e gli archi sono le adiacenze fisiche tra le celle.

4.4 Utility

Questo è il metodo di esplorazione decentralizzato presentato in [32] ed è relativo al vincolo di comunicazione morbido.

La rappresentazione dell'ambiente utilizzata da questa strategia non si trova sullo stesso livello di quella utilizzata per le strategie precedenti. Infatti, non viene utilizzato alcun grafo contenuto nella BS per il calcolo delle posizioni obiettivo che i robot devono raggiungere. Viene utilizzata la rappresentazione dell'ambiente a griglia presente all'interno di ogni agente (robot esploratori e BS) descritta nella Sezione 3.2.1.

All'inizio dell'esplorazione, il pianificatore della missione imposta un parametro di soglia $r \in [0, 1)$ il quale rappresenta il compromesso tra un comportamento di esplorazione greedy ($r = 0$), in cui i robot non tornano a comunicare con la BS fin quando non hanno scoperto totalmente l'ambiente, e un comportamento in cui i robot tornano frequentemente alla BS per condividere le nuove informazioni ($r \to 1$). Di conseguenza, il vincolo di comunicazione non è espresso esplicitamente, ma è soltanto il risultato di un comportamento emergente dall'algoritmo. In Figura 4.3 è mostrata la differenza tra un'esplorazione quasi completamente greedy ed un'esplorazione in cui i robot tornano frequentemente alla BS.

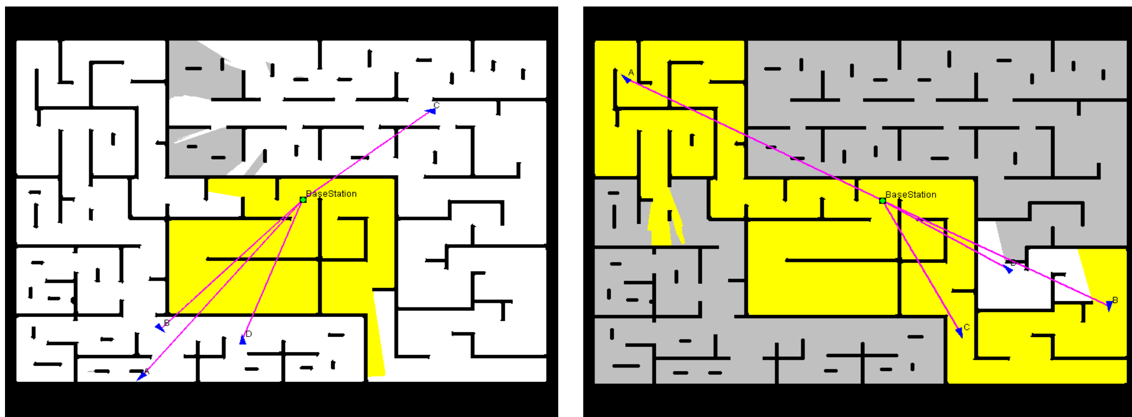


Figura 4.3: a sinistra è mostrato un esempio di esplorazione quasi completamente greedy (r = 0.1), a destra è mostrato un esempio di esplorazione in cui i robot tornano molto frequentemente a comunicare con la BS (r = 0.9). Le aree bianche sono le zone di ambiente conosciute solo dai robot esploratori, mentre quelle gialle sono le zone conosciute anche dalla BS. Le aree grigie sono le zone che non sono ancora state esplorate. Notiamo come con alti valori di r, la BS possiede quasi tutta l'informazione conosciuta dai robot.

Seguendo la stessa notazione di [32], chiamiamo $InfBase_i$ l'informazione dell'ambiente che il robot $i \in A$ crede che la BS possegga. Questo valore può essere ottenuto o tramite una connessione diretta con la BS, oppure scambiando messaggi con un robot che ha comunicato con la BS più recentemente di i. Definiamo $InfNew_i$ come la nuova informazione dell'ambiente che possiede il robot i, ovvero l'informazione conosciuta dal robot i che non è stata consegnata alla BS o ad un altro robot.
Se due agenti i e j si incontrano, e j è più vicino alla BS rispetto ad i, i valori $InfNew_i$ e di $InfNew_j$ vengono aggiornati nel modo seguente:

$$InfNew_i = \emptyset$$

$$InfNew_j = InfNew_i \cup InfNew_j$$

ovvero, il robot più lontano dalla BS (in questo caso i) cede tutta la sua nuova informazione al robot che si trova più vicino alla BS (in questo caso j), in modo che, in caso di ritorno alla BS, il robot più vicino abbia una distanza minore da percorrere e, di conseguenza, impieghi meno tempo per consegnare la nuova informazione. Un altro motivo per cui questo viene fatto è ridurre la quantità di informazione ridondante consegnata alla BS e diminuire, quindi, il tempo che la BS dedica al controllo dell'informazione ricevuta.
I robot, dopo la fase di percezione dell'ambiente, calcolano un valore che è confrontato con r. Un robot i decide di tornare alla BS se:

$$\frac{|InfBase_i|}{|InfNew_i| + |InfBase_i|} < r \qquad \text{(iii)}$$

dove il valore assoluto indica l'area della regione di ambiente, che viene misurata come il numero delle celle che rappresentano l'area libera (ovvero, il numero di celle di tipo *free* e *safe*).
Quando un robot non deve tornare in una posizione in cui è in grado di comunicare con la BS, sceglie in modo greedy una frontiera da esplorare secondo la funzione di utilità $u(v_f)\ /\ d^2(q_a^t, v_f)$. Il processo di scelta di una nuova frontiera viene fatto o al raggiungimento della frontiera che il robot ha come obiettivo, oppure dopo un certo numero di passi (che viene impostato come limite prima dell'inizio dell'esplorazione dal pianificatore della missione). Questo numero di passi limite è utile in situazioni che portano il robot a volere cambiare opportunisticamente la frontiera obiettivo. Per esempio, un robot entra in comunicazione con un compagno di squadra che ha già

esplorato la zona di ambiente in cui è presente la frontiera che ha come obiettivo, oppure, durante lo spostamento, trova una frontiera con un valore di utilità maggiore.

Capitolo 5

Architettura del sistema

In questo capitolo, descriviamo le classi, i package e l'implementazione delle funzioni principali necessarie per portare a termine l'esplorazione dell'ambiente all'interno del simulatore MRESim. Inoltre, descriviamo l'implementazione delle strategie di esplorazione che abbiamo confrontato.

5.1 MRESim

In questa tesi, abbiamo utilizzato il simulatore MRESim[1] [9] per confrontare le strategie di esplorazione i cui modelli sono stati illustrati nel Capitolo 4. MRESim simula la percezione, la comunicazione ed i movimenti di più agenti che esplorano un ambiente in 2D. Il simulatore adotta una rappresentazione dell'ambiente secondo una griglia in cui le celle possono essere libere (*free*) oppure occupate da ostacoli (*obstacle*), come visto nella Sezione 3.2.1. Inoltre, assume localizzazione, locomozione e percezione dell'ambiente perfette. Nel mondo reale, queste assunzioni non valgono. Tuttavia, ci permettono di avere una buona idea riguardo ai punti di forza e di debolezza che caratterizzano le strategie di esplorazione testate.

MRESim utilizza il linguaggio di programmazione Java ed include l'implementazione di diverse strategie di esplorazione, tutte di tipo decentralizzato, che permettono ad una squadra di robot di dimensione arbitraria di esplorare un ambiente scelto dal pianificatore della missione. Queste strategie sono incluse all'interno del package

[1]MRESim può essere scaricato gratuitamente dal sito https://github.com/v-spirin/MRESim

exploration. Abbiamo implementato all'interno del simulatore le strategie di esplorazione *IlpExploration*, *StumpExploration* e *BirkExploration*, i cui modelli sono stati descritti, rispettivamente, nelle sezioni 4.1, 4.2 e 4.3.

Nel repository[2] del simulatore sono presenti ambienti di tipologie diverse: uffici, labirinti, spazi esterni, casuali, … Gli ambienti possono essere caricati in MRESim e sono rappresentati come immagini bitmap in bianco e nero di dimensioni 800x600 pixel, in cui le zone nere rappresentano gli ostacoli, mentre le zone bianche rappresentano aree libere in cui i robot possono muoversi. Ogni pixel dell'immagine viene convertito in una cella all'interno della griglia *Env* presentata nella Sezione 3.2.1.

Nel simulatore sono implementati anche i seguenti modelli di comunicazione inclusi nel package *communication*: static circle (due robot comunicano se si trovano l'uno all'interno del raggio di comunicazione dell'altro), line-of-sight (due robot comunicano se sul segmento che li unisce non sono presenti ostacoli e se la loro distanza è inferiore ad una certa soglia) e propagation model (viene preso in considerazione lo spessore degli ostacoli per simulare la degradazione del segnale). Negli esperimenti condotti in questa tesi abbiamo utilizzato il modello di comunicazione line-of-sight, implementato all'interno della classe *DirectLine*. L'utilizzo di questo modello è giustificato dalla rappresentazione topologica dell'ambiente (Sezione 3.2.2) in cui la comunicazione tra due vertici è possibile se e solo se sono in linea di vista tra loro.

Tramite l'interfaccia grafica (Figura 5.1), è possibile cambiare le preferenze di esplorazione. L'interfaccia del simulatore fornisce anche diverse informazioni a runtime riguardanti l'esplorazione in corso, come, per esempio, la percentuale di ambiente conosciuto dalla BS o il tempo medio di esecuzione di un ciclo di simulazione. Le fasi che compongono un ciclo di simulazione sono descritte nella Sezione 5.1.2. Possono essere visualizzate altre informazioni riguardanti l'ambiente, come per esempio le frontiere conosciute ad un determinato istante di tempo, il percorso che sta seguendo un robot, … Maggiori dettagli riguardanti l'interfaccia grafica e l'utilizzo generale del simulatore sono riportati nell'*Appendice A: Manuale d'installazione e d'uso.*

[2]Le immagini degli ambienti sono contenute all'interno della cartella *MRESim-master\environments*.

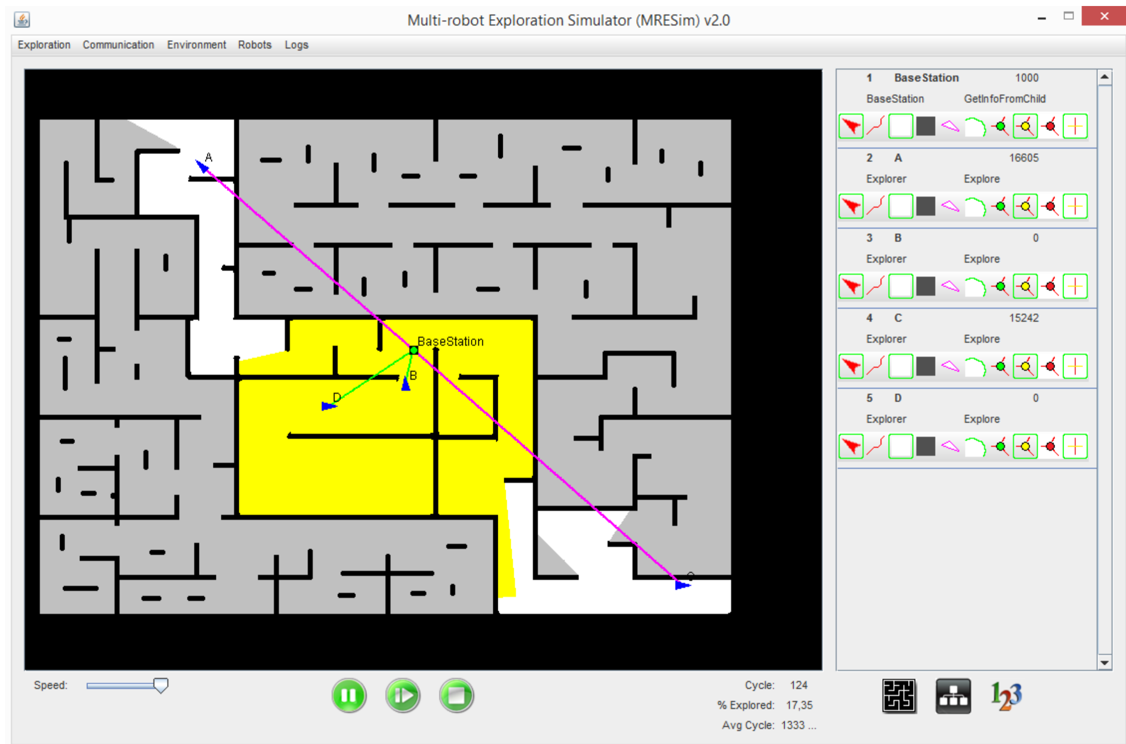


Figura 5.1: interfaccia grafica di MRESim durante l'esplorazione. A destra sono presenti alcune informazioni relative alla squadra di robot. Al centro è visibile l'ambiente di esplorazione con le posizioni occupate dai robot ad un determinato ciclo. Le aree bianche indicano le zone di ambiente conosciute dai robot, quelle gialle indicano le zone di ambiente conosciute dalla BS, quelle grigie indicano le zone di ambiente non ancora esplorate, quelle nere indicano gli ostacoli. In basso sono presenti alcune informazioni relative alla simulazione in corso.

5.1.1 Package e classi principali

In questa sezione, vediamo più nel dettaglio quali sono i package[3] e le classi principali del simulatore e delle strategie di esplorazione implementate in questa tesi.

In Figura 5.2 è mostrato il package *agents* che contiene le classi che rappresentano gli agenti all'interno del simulatore. Tutte le classi implementano l'interfaccia *Agent*.

La classe principale è *BasicAgent* che contiene informazioni di base dell'agente, come la posizione (x e y) e lo stato in cui si trova (*state*). Il valore di *state* rappresenta lo stato di esplorazione in cui si trova il robot ed è utilizzato da alcune strategie di esplorazione (per esempio, *UtilityExploration*) per il calcolo del prossimo passo (come descritto nella Sezione 5.5). Da *BasicAgent*, le classi *RealAgent* e *TeammateAgent* ereditano le operazioni e gli attributi.

[3]I package sono contenuti all'interno della cartella *MRESim-master\src*.

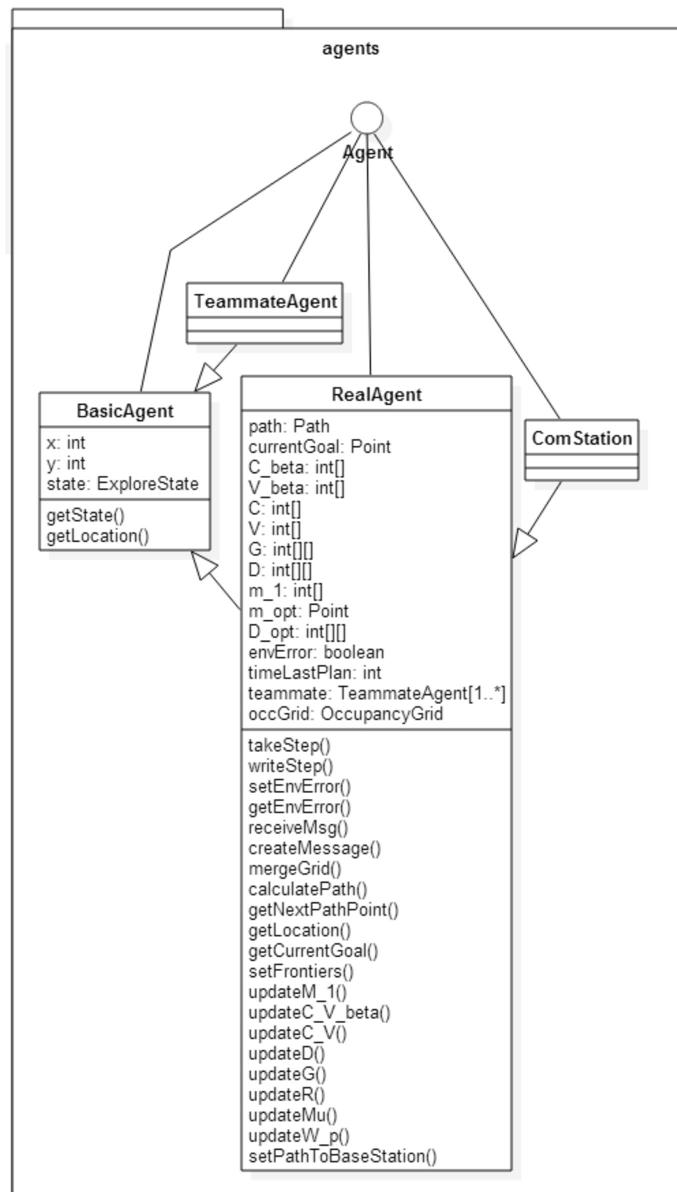


Figura 5.2: package agents e gerarchia delle classi che rappresentano gli agenti.

La classe *RealAgent* è quella che contiene informazioni dell'agente durante l'esplorazione, come per esempio la mappa conosciuta (*occGrid*), il percorso che sta seguendo (*path*), … Abbiamo esteso la classe *RealAgent* aggiungendo i parametri ed i metodi necessari al funzionamento delle strategie di esplorazione testate. Alcuni di questi parametri sono, per esempio, i vettori necessari al funzionamento della strategia di esplorazione *StumpExploration* (Sezione 5.3), come *C*, *V*, *m_1*, … Alcuni di questi metodi sono, per esempio, le funzioni che permettono di aggiornare i parametri necessari al funzionamento della strategia di esplorazione *StumpExploration*, come

updateC_V(), updateM_1(), ...
La classe *TeammateAgent* è utilizzata per la rappresentazione di un agente all'interno di un compagno di squadra. In altre parole, l'agente *r*, rappresentato dalla classe *RealAgent*, memorizza le informazioni relative all'agente *t*, suo compagno di squadra, all'interno della variabile *teammate* di tipo *TeammateAgent*.
La classe *ComStation* rappresenta la BS ed eredita attributi ed operazioni dalla classe *RealAgent*.
In Figura 5.3 è rappresentato il package *exploration* che contiene tutte le classi e le operazioni necessarie per lo svolgimento dell'esplorazione. Le classi *RandomWalk*, *FrontierExploration,* *UtilityExploration*, *IlpExploration*, *StumpExploration* e *BirkExploration* rappresentano l'implementazione di alcune strategie di esplorazione. Le prime tre erano già presenti all'interno del simulatore. Abbiamo implementato le ultime tre e le loro implementazioni vengono presentate nel dettaglio nelle sezioni successive.
La classe *Utility* rappresenta l'utilità assegnata ad una coppia robot-frontiera. Questa classe contiene l'ID del robot, la frontiera *frontier* ed il valore di utilità *utility* assegnato alla coppia robot-frontiera. La funzione di utilità, all'interno di MRESim, è definita nel modo seguente: $U(r, f) = A(f) / C(r, f)$, dove $U(r, f)$ rappresenta il valore di utilità della frontiera f per il robot r, $A(f)$ rappresenta il guadagno di informazione potenziale ottenibile al raggiungimento della frontiera f e $C(r, f)$ rappresenta il costo che deve affrontare il robot r per raggiungere la frontiera f. Una frontiera in sé non fornisce informazioni riguardo al potenziale guadagno di informazione ricavabile dall'esplorazione dell'area presente al di là della frontiera stessa. Per risolvere questo problema, viene utilizzato il concetto di *poligono della frontiera*. Il poligono della frontiera è il poligono costruito tra le celle a confine dello spazio *free* e quelle a confine dello spazio *safe*. Dunque, la distinzione tra celle *free* e celle *safe* all'interno della griglia *Occ* (introdotta nella Sezione 3.2.1) è utilizzata dal robot per il tracciamento del poligono della frontiera. L'area del poligono è utilizzata come stima del guadagno di informazione ottenibile al raggiungimento della frontiera. Infatti, le frontiere che hanno un poligono grande hanno più probabilità di fornire maggiori informazioni rispetto a quelle che hanno un poligono piccolo. Il costo associato ad una coppia robot-frontiera è dato dalla distanza tra la posizione del robot e la posizione della frontiera. La posizione della frontiera viene identificata da un punto appartenente al poligono della frontiera

chiamato *centro della frontiera*.

All'interno della classe *SimulationFramework* sono presenti tutti i metodi che permettono di portare a termine la simulazione con le impostazioni che sono state scelte dall'utente (per esempio, modello di comunicazione e strategia di esplorazione) contenute nel parametro *SimConfig*. In *SimulationFramework* sono presenti, inoltre, l'array degli agenti *agents* e l'ambiente *env*, rappresentato dalla classe *Environment*.

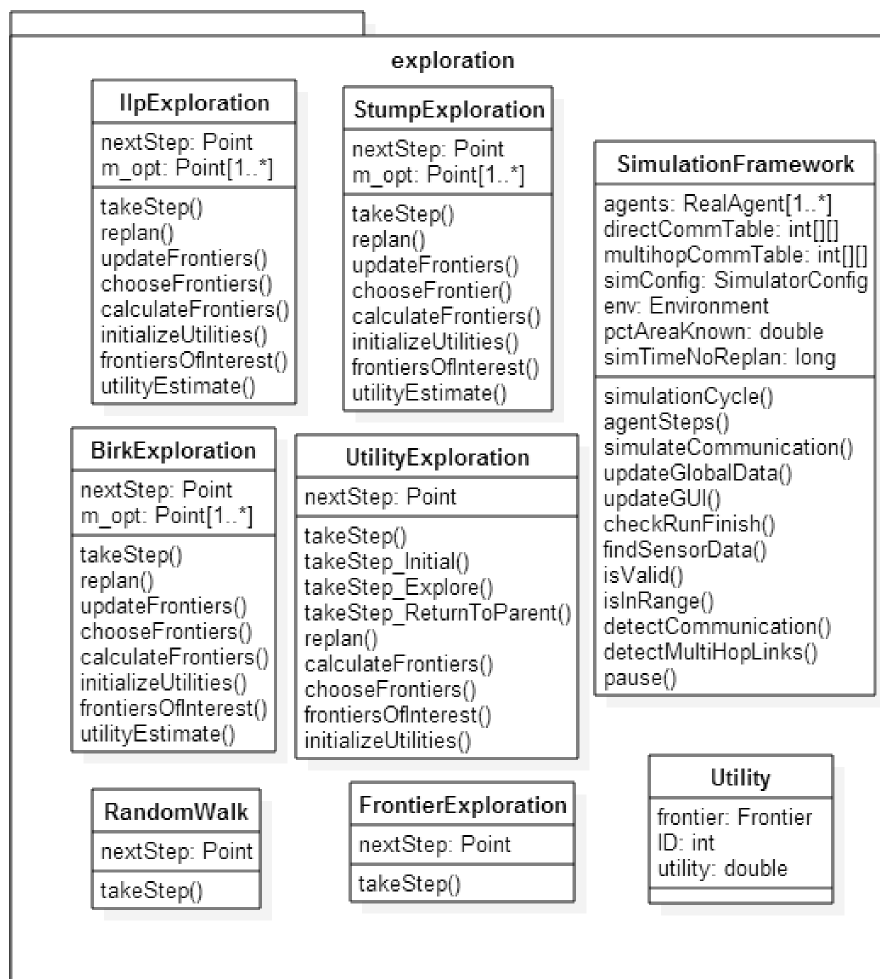


Figura 5.3: package exploration contenente la classe SimulationFramework e le strategie di esplorazione implementate nel simulatore.

In Figura 5.4 sono rappresentati altri package ed altre classi utilizzati per la simulazione. Di seguito sono illustrati con maggiore dettaglio.

Il package *config* contiene le impostazioni scelte dall'utente prima dell'inizio della simulazione. Queste impostazioni si trovano all'interno della classe *SimulatorConfig* e della classe *Constants* e sono, per esempio, l'algoritmo di esplorazione *expAlgorithm*

(rappresentato come un valore *exptype* di tipo *enum*), il modello di comunicazione *commModel* (rappresentato come un valore *commtype* di tipo *enum*), il numero di cicli che devono trascorrere prima che il robot possa ricalcolare il percorso verso una posizione obiettivo *REPLAN_INTERVAL* ed il valore *infoRatio* obiettivo, chiamato *TARGET_INFO_RATIO*, utilizzato dai robot all'interno della strategia di esplorazione *UtilityExploration* per decidere quando tornare alla BS (come visto nella Sezione 4.4).

Il package *communication* contiene l'implementazione dei modelli di comunicazione presentati nella sezione precedente. Inoltre, è presente la classe *DataMessage* che rappresenta i messaggi che vengono scambiati tra gli agenti durante la fase di comunicazione.

Il package *environment* contiene la rappresentazione dell'ambiente utilizzata dal simulatore (rappresentata dalla classe *Environment*) e la rappresentazione dell'ambiente utilizzata dal singolo agente (rappresentata dalla classe *OccupancyGrid*). Queste sono le due rappresentazioni a griglia presentate nella Sezione 3.2.1.

Il package *path* contiene la classe *Path* che rappresenta il percorso che il robot deve seguire per andare dalla posizione in cui si trova fino ad una posizione obiettivo.

Il package *gui* contiene la classe *MainGUI* che rappresenta l'interfaccia grafica del simulatore.

Abbiamo aggiunto il package *decisore* che contiene gli algoritmi utilizzati per il calcolo delle locazioni obiettivo che i robot devono raggiungere, ovvero la classe *Ilp*, che rappresenta il modello presentato nella Sezione 4.1 e la classe *Stump*, che rappresenta l'adattamento all'algoritmo [34] presentato nella Sezione 4.2.

In Figura 5.5 sono rappresentate le relazioni tra le classi più importanti appartenenti ai diversi package. *SimulationFramework* contiene l'ambiente *Environment*, le preferenze di simulazione *SimulatorConfig*, l'interfaccia grafica *MainGUI*, il vettore dei robot *RealAgent* e la BS *ComStation*. *SimulatorConfig* contiene il modello di comunicazione *commModel* e la strategia di esplorazione *expAlgorithm.* I robot e la BS contengono una propria mappa dell'ambiente *OccupancyGrid* e le informazioni relative ai propri compagni di squadra *TeammateAgent*. Il singolo robot contiene anche il percorso *Path* che deve seguire.

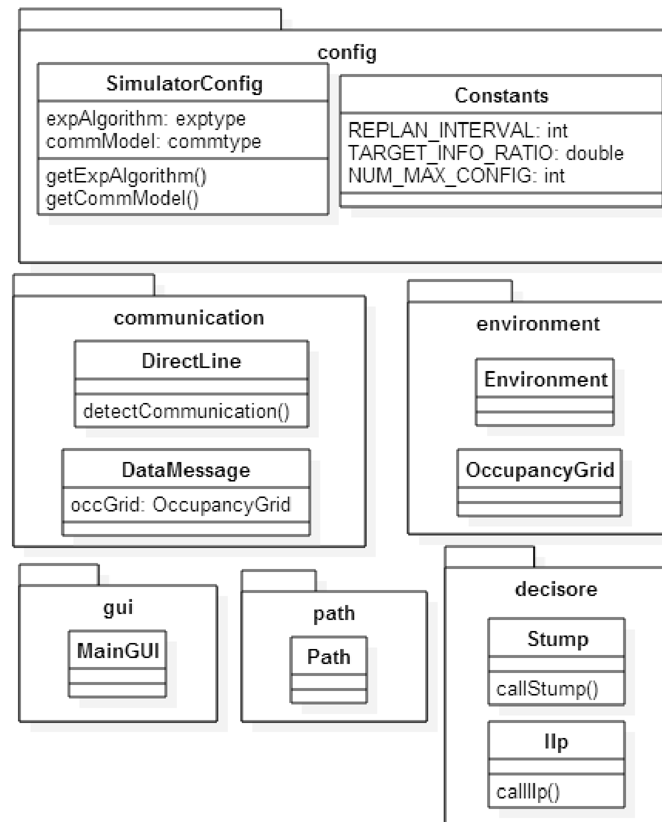


Figura 5.4: rappresentazione di package e di classi utilizzati per la simulazione.

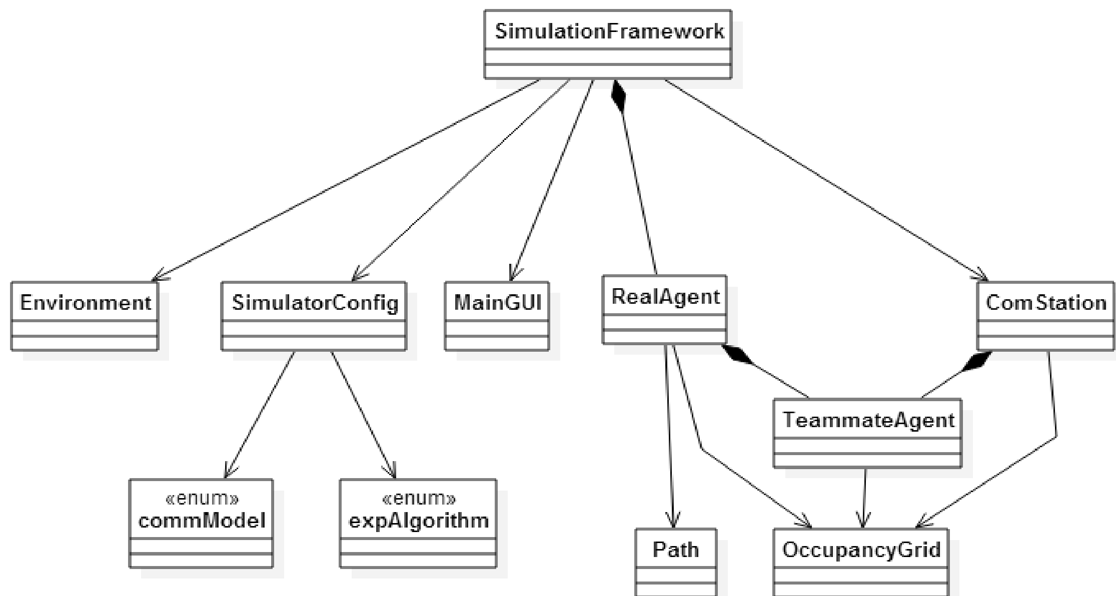


Figura 5.5: rappresentazione delle relazioni tra le classi più importanti.

5.1.2 Ciclo di simulazione

La simulazione viene eseguita all'interno di un unico thread. Le fasi che compongono l'esplorazione, viste nella Sezione 2.1, sono implementate in modo sequenziale all'interno di *simulationCycle()*. Questo metodo è incaricato di simulare l'esplorazione dell'ambiente da parte dei robot ed aggiornare l'interfaccia grafica del simulatore. È contenuto all'interno della classe *SimulationFramework*. Possiamo schematizzare il metodo con lo pseudocodice seguente:

```
void simulationCycle() {
        // Simulazione movimenti e percezione delle informazioni
        agentSteps();
        // Simulazione della comunicazione e aggiornamento informazioni interne ai robot
        simulateCommunication();
        // Aggiornamento dati rilevanti interni al simulatore
        updateGlobalData();
        // Aggiornamento interfaccia grafica del simulatore
        updateGUI();
        // Controllo termine esplorazione
        checkRunFinish();
}
```

All'inizio di ogni ciclo, ciascun robot sceglie la prossima posizione in cui muoversi ed effettua lo spostamento. Successivamente, percepisce l'ambiente che lo circonda tramite i sensori ed aggiorna la propria mappa dell'ambiente. Queste due operazioni sono svolte all'interno del metodo *agentSteps()*. Dopodiché, ogni robot scambia messaggi con i compagni di squadra che si trovano all'interno del proprio raggio di comunicazione ed aggiorna le informazioni che riguardano i compagni di squadra e l'ambiente. Queste operazioni vengono svolte all'interno del metodo *simulateCommunication()*. In seguito, il simulatore aggiorna le informazioni relative alla simulazione tramite il metodo *updateGlobalData()* (per esempio, numero di cicli trascorsi, percentuale di area conosciuta dalla BS, tempo medio di esecuzione di un ciclo, …). All'interno del metodo *updateGUI()* il simulatore si occupa di aggiornare l'interfaccia grafica di MRESim, mostrando all'utente come si muovono i robot all'interno dell'ambiente, le aree libere e gli ostacoli, come comunicano tra loro i robot, … Infine, il simulatore controlla se è possibile terminare l'esplorazione tramite il metodo *checkRunFinish()*.

Descriviamo di seguito i metodi più importanti per lo svolgimento dell'esplorazione: *agentSteps()*, *simulateCommunication()* e *checkRunFinish()*.

5.1.3 Movimento degli agenti e percezione dell'ambiente

Come già detto, *agentSteps()* è il metodo che si occupa di far muovere i robot di un passo, di far loro percepire l'ambiente che li circonda e di aggiornare la loro mappa. Possiamo schematizzare il metodo con lo pseudocodice seguente:

```
void agentSteps() {
    foreach ri ∈ agents do
        if ri != BS then
            nextStep = ri.takeStep(simConfig);
            if isValid(nextStep) then
                sensorData = findSensorData(ri, nextStep);
                ri.writeStep(nextStep, sensorData);
            else
                ri.setEnvError(true)
            end
        else
            if simConfig.getExpAlgorithm() == StumpExploration then
                m_opt = StumpExploration.replan(ri);
            elseif simConfig.getExpAlgorithm() == BirkExploration then
                m_opt = BirkExploration.replan(ri);
            elseif simConfig.getExpAlgorithm() == IlpExploration then
                m_opt = IlpExploration.replan(ri);
            end
        end
    end
}
```

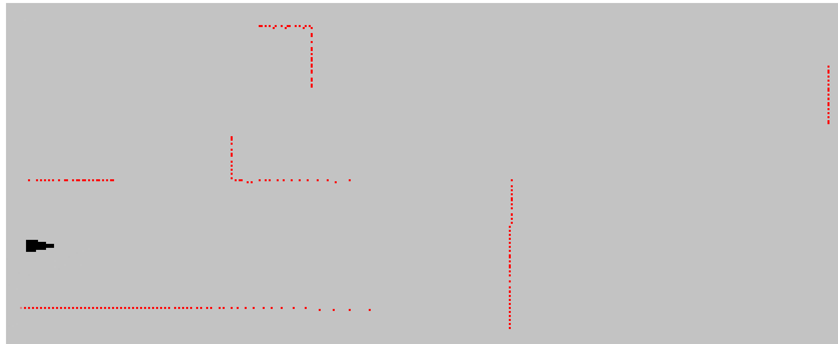
All'interno del metodo è presente un ciclo *for*. Questo significa che gli agenti vengono presi in considerazione in modo sequenziale. Come prima cosa, il simulatore controlla se l'agente *ri* è un robot esploratore oppure se è la BS.

Se *ri* è un robot esploratore, allora viene chiamato il metodo *takeStep()* appartenente alla classe *RealAgent*. Il metodo *takeStep()* si occupa di calcolare il prossimo passo che il robot deve eseguire (entriamo nel dettaglio di questo metodo alla fine della sezione). Una volta scelto il prossimo passo, il metodo *isValid()* si occupa di verificare che la posizione scelta sia valida (per esempio, verifica che non sia una posizione occupata da un ostacolo). Nel caso in cui non sia valida, viene impostato a *true* un parametro di errore *envError* all'interno del robot (vedremo nelle sezioni successive cosa provoca). Invece, nel caso in cui la posizione scelta sia valida, vengono percepiti i dati dell'ambiente tramite il metodo *findSensorData()*. Questo metodo si occupa di simulare la percezione dell'ambiente tramite i sensori montati sul robot. I dati di percezione sono generati utilizzando il tracciamento di raggi all'interno della mappa dell'ambiente *env* del simulatore, partendo dalla posizione occupata dal robot ed arrivando ad una distanza

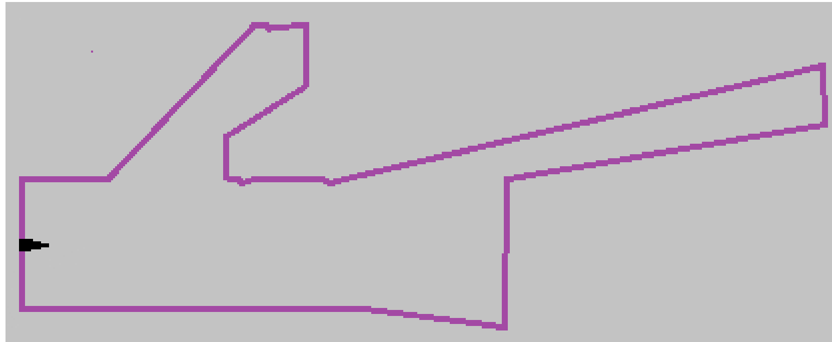
massima data dal raggio di percezione dei sensori. I raggi sono tracciati ad intervalli regolari di 1° in un campo di visione del robot di 180°. Di conseguenza, viene generato un array con 181 misurazioni e passato al robot tramite il metodo *writeStep()*. Questo metodo memorizza i dati percepiti ed il passo appena compiuto all'interno del robot ed aggiorna la sua mappa *occGrid*, appartenente alla classe *OccupancyGrid*.

Se, invece, l'agente *ri* è la BS e, inoltre, la strategia di esplorazione (contenuta all'interno del parametro *simConfig* del *SimulationFramework* ed appartenente alla classe *SimulatorConfiguration*) è *IlpExploration, StumpExploration* oppure *BirkExploration*, allora viene chiamato il metodo *replan()* relativo alla strategia di esplorazione utilizzata. Questo metodo si occupa di far calcolare alla BS lo schieramento *m_opt* che i robot esploratori devono raggiungere all'interno dell'ambiente conosciuto dalla BS stessa e, per questo motivo, è utilizzato soltanto dalle strategie di esplorazione di tipo centralizzato. Nelle strategie di esplorazione di tipo decentralizzato, infatti, la BS non deve svolgere questo compito, in quanto i robot decidono autonomamente le posizioni obiettivo che devono raggiungere. Del metodo *replan()* parleremo in modo approfondito nella Sezione 5.2.2.

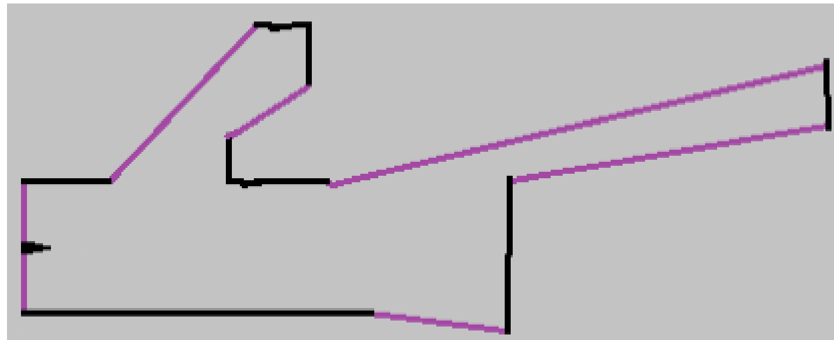
Il metodo *writeStep()* utilizza il parametro *sensorData* per trovare i vertici del poligono, che rappresenta il nuovo spazio libero percepito, tramite il metodo *findRadialPolygon()*. All'interno del simulatore, il poligono è chiamato *newFreeSpace* ed è rappresentato come un vettore di punti che corrispondono ai sui vertici. In Figura 5.6(a) sono rappresentati i vertici di questo poligono e in Figura 5.6(b) è rappresentato graficamente il poligono ottenuto unendo i vertici consecutivi presenti nel vettore *newFreeSpace*. Successivamente, il metodo *updateObstacles()* aggiorna il valore delle celle all'interno della mappa *occGrid* del robot nel modo seguente: se la distanza tra la cella relativa al vertice in posizione *i* nel vettore *newFreeSpace* e la cella relativa al vertice in posizione *i+1* nel vettore *newFreeSpace* è inferiore ad una certa soglia, allora viene assegnato il valore *obstacle* a queste due celle e a tutte le celle che si trovano sul segmento che le unisce. Altrimenti, non viene assegnato alcun valore. In Figura 5.6(c) è mostrato questo aggiornamento. Dopodiché, il metodo *updateFreeAndSafeSpace()* assegna il valore *safe* o il valore *free* a tutte le celle interne al poligono, che non sono state etichettate come ostacoli, a seconda della loro distanza dal robot (Figura 5.6(d)).



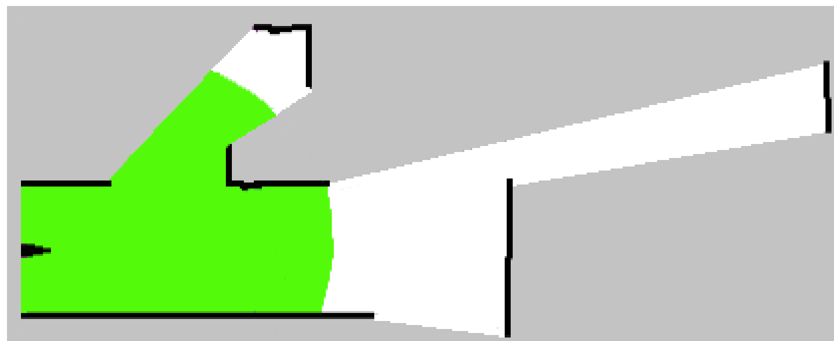
(a) Celle interessate dalla percezione che formano i vertici del poligono di spazio libero.



(b) Rappresentazione del poligono di spazio libero.



(c) Alle celle che si trovano tra vertici consecutivi abbastanza vicini viene assegnato il valore obstacle (aree nere).



(d) A tutte le altre celle vengono assegnati i valori safe (aree verdi) o free (aree bianche) a seconda della distanza dal robot.

Figura 5.6: rappresentazione grafica di un esempio di funzionamento del metodo writeStep(). Il robot è rappresentato come un triangolo.

Il metodo *updateObstacles()* è il responsabile degli errori di integrazione delle nuove informazioni all'interno della mappa dell'ambiente del robot. Questi errori provocano problemi con il calcolo delle frontiere. Infatti, alcune frontiere vengono trovate all'interno di muri (Figura 5.7) e questo porta a spostamenti del robot non validi.

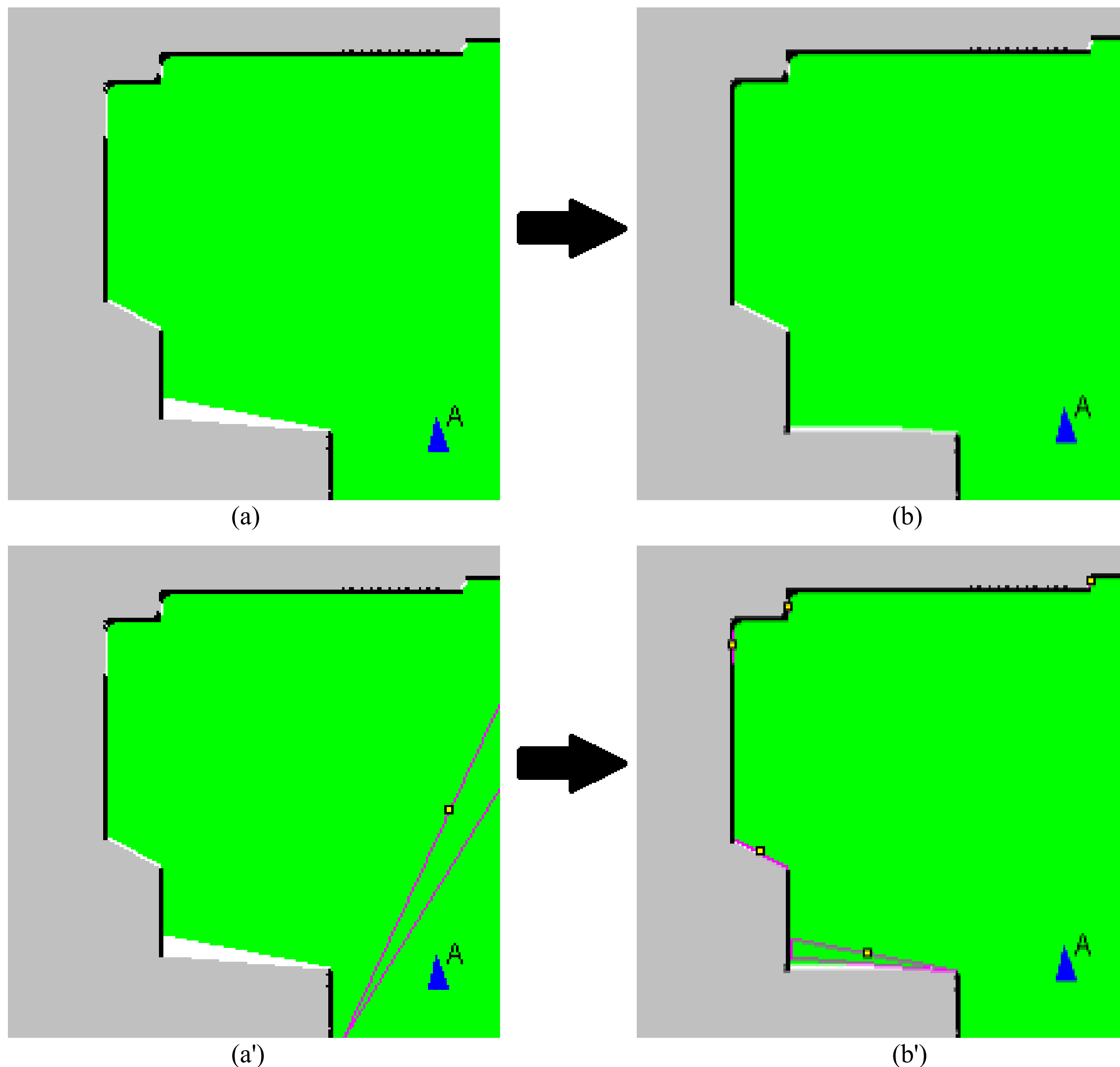


Figura 5.6: rappresentazione degli errori di integrazione. Le immagini (a') e (b') rappresentano le stesse situazioni presenti, rispettivamente, nelle immagini (a) e (b), con la differenza che vengono mostrate le frontiere conosciute dal robot. Nel passaggio da (a) a (b) (e, quindi, da (a') a (b')) il robot calcola le nuove frontiere, fa un passo verso la frontiera scelta e percepisce l'ambiente. Notiamo che, nelle immagini (a) e (a'), in alto a sinistra, alcune celle sono di tipo *free*. Quando il robot calcola le nuove frontiere da raggiungere, individua una frontiera nell'area occupata da queste celle. Dopo essersi spostato ed avere percepito l'ambiente, il robot aggiorna la mappa assegnando a queste celle il valore *obstacle* (immagini (b) e (b')). A questo punto, la frontiera si trova all'interno del muro. Una volta raggiunta dal robot, provoca uno spostamento non valido ed il conseguente assegnamento del valore *true* al parametro *envError*.

Approfondiamo il funzionamento del metodo *RealAgent.takeStep()*. Possiamo schematizzare il metodo con lo pseudocodice seguente:

```
Point takeStep(SimulatorConfig simConfig) {
        switch(simConfig.getExpAlgorithm())
                case StumpExploration:
                        nextStep = StumpExploration.takeStep(this);
                        break;
                case BirkExploration:
                        nextStep = BirkExploration.takeStep(this)
                        break;
                case UtilityExploration:
                        nextStep = UtilityExploration.takeStep(this);
                        break;
                case genericExpAlgorithm:
                        nextStep = genericExpAlgorithm.takeStep(this);
                        break;
        end
        return nextStep;
}
```

I nomi delle strategie di esplorazione si trovano all'interno del parametro *expAlgorithm* appartenente alla classe *SimulatorConfig*. Per una generica strategia di esplorazione, identificata dalla classe *genericExpAlgorithm* che si trova all'interno del package *exploration*, *RealAgent.takeStep()* chiama il metodo *genericExpAlgorithm.takeStep()* che calcola la prossima posizione *nextStep* che il robot deve raggiungere all'interno della mappa *occGrid* dell'ambiente conosciuto. I metodi *takeStep()*, relativi alle diverse strategie utilizzate per le simulazioni, sono descritti nelle sezioni 5.2.1 e 5.5.1.

5.1.4 Simulazione della comunicazione

Vediamo ora come viene eseguita la comunicazione tra gli agenti. Il metodo *simulateCommunication()* si occupa di far scambiare i messaggi tra gli agenti. Possiamo schematizzarlo con lo pseudocodice seguente:

```
void simulateCommunication() {
        detectCommunication()
        foreach ri ∈ agents do
                foreach rj ∈ R and i != j do
                        if isInRange(ri, rj) then
                                ri.receiveMsg(rj.createMessage())
                                rj.receiveMsg(ri.createMessage())
                        end
                end
```

```
        end
}
```

Il metodo *detectCommunication()* si occupa di costruire una matrice di comunicazione che viene utilizzata dal metodo *isInRange()* per verificare se due agenti sono in comunicazione. Tramite il metodo *RealAgent.receiveMsg()* vengono scambiati i messaggi tra i robot comunicanti e vengono aggiornate le informazioni al loro interno. I messaggi contengono informazioni riguardo al compagno di squadra r_i (per esempio, posizione, stato, …) e all'ambiente conosciuto dal compagno (per esempio, *occGrid*) e sono generati dal metodo *RealAgent.createMessage()*.
Possiamo schematizzare *detectCommunication()* con lo pseudocodice seguente:

```
void detectCommunication() {
        switch(simConfig.getCommModel())
                case DirectLine:
                        directCommTable = DirectLine.detectCommunication(env, agent);
                        break;
                case genericCommModel:
                        directCommTable = genericCommModel.detectCommunication(env, agent);
                        break;
        end
        multihopCommTable = detectMultiHopLinks(directCommTable);
}
```

I nomi dei modelli di comunicazione si trovano all'interno del parametro *commModel* appartenente alla classe *SimulatorConfig*. A seconda del modello di comunicazione adottato, rappresentato in generale dalla classe *genericCommModel* all'interno del package *communication*, il metodo *genericCommModel.detectCommunication()* si occupa di costruire la matrice di comunicazione *directCommTable*. Il generico elemento m_{ij} della matrice (e, di conseguenza, anche l'elemento m_{ji}, essendo la matrice simmetrica) è uguale ad *1* se gli agenti r_i ed r_j sono in comunicazione diretta, altrimenti è uguale a *0*. Nel caso del modello di comunicazione *DirectLine*, per esempio, due robot comunicano direttamente se non vi sono ostacoli sul segmento che li unisce e se la loro distanza è inferiore ad una certa soglia. Infine, il metodo *detectMultiHopLinks()* si occupa di costruire la matrice di comunicazione *multihopCommTable* che rappresenta il grafo di comunicazione multihop fra gli agenti all'interno dell'ambiente.

Possiamo schematizzare *detectMultiHopLinks()* con lo pseudocodice seguente:

```
int[][] detectMultiHopLinks(int[][] directCommTable) {
        for (int i=0; i<commTable.length; i++) do
                for (int j=0; j<commTable[0].length; j++) do
                        if commTable[i][j] == 1 or commTable[j][i] == 1 then
                                for (int k=0; k<commTable.length; k++) do
                                        if commTable[k][j] == 1 or commTable[j][k] == 1 then
                                                commTable[k][i] = 1;
                                                commTable[i][k] = 1;
                                        end
                                end
                        end
                end
        end
        return directCommTable;
}
```

Il metodo aggiorna la matrice di comunicazione diretta *directCommTable*. In generale, se gli agenti r_i ed r_j comunicano e gli agenti r_j ed r_k comunicano, allora anche gli agenti r_i ed r_k comunicano. Di conseguenza, viene assegnato il valore *1* agli elementi m_{ik} ed m_{ki} della matrice *directCommTable*. Alla fine di questo procedimento, viene restituita la matrice che rappresenta il grafo di comunicazione multihop.
Il metodo *RealAgent.receiveMsg()* può essere schematizzato con lo pseudocodice seguente:

```
void receiveMsg(DataMessage msg) {
        updateTeammateInfo(msg.getInfoTeammate());
        mergeGrid(msg.occGrid);
}
```

L'agente aggiorna le informazioni relative al suo compagno *teammate* con le nuove informazioni ricevute all'interno del messaggio *msg* (per esempio, posizione, stato, mappa dell'ambiente, ...) tramite degli assegnamenti che indichiamo con il metodo *updateTeammateInfo()*. Successivamente, l'agente aggiorna la propria mappa dell'ambiente unendola a quella del compagno tramite il metodo *mergeGrid()*. Se il *teammate* con cui il robot sta comunicando è la BS, allora il robot riceve, all'interno del messaggio, la posizione obiettivo che deve raggiungere. Questo vale soltanto per le strategie di tipo centralizzato *IlpExploration*, *StumpExploration* e *BirkExploration* in cui è la BS che assegna le posizioni obiettivo ai singoli robot.

5.1.5 Controllo della terminazione

Questo metodo si occupa di verificare se il criterio di terminazione dell'esplorazione è soddisfatto. Possiamo schematizzarlo con lo pseudocodice seguente:

```
if simTimeNoReplan >= timeLimit or pctAreaKnown >= pctAreaTreshold or m_opt == null then
        pause();
end
```

L'esplorazione viene terminata nel caso in cui una delle seguenti condizioni sia soddisfatta:

- il tempo di simulazione supera una certa soglia impostata dall'utente;
- la percentuale di area dell'ambiente conosciuto dalla BS supera una certa soglia impostata dall'utente;
- non è stato possibile trovare uno schieramento connesso dei robot all'interno della mappa dell'ambiente della BS (vale soltanto per le stategie *IlpExploration*, *StumpExploration* oppure *BirkExploration*).

5.2 Ilp Exploration

La strategia di esplorazione, di cui abbiamo descritto il modello nella Sezione 4.1, è contenuta all'interno della classe *IlpExploration* del package *exploration*.
In questa classe sono presenti il metodo *takeStep()*, che si occupa dei movimenti dei robot esploratori, ed il metodo *replan()*, che viene utilizzato dalla BS per calcolare i nuovi schieramenti dei robot all'interno della mappa dell'ambiente conosciuto.
Descriviamo più nel dettaglio l'implementazione di questa strategia di esplorazione.

5.2.1 IlpExploration.takeStep()

Per lo spostamento di un robot esploratore, viene utilizzato il metodo *IlpExploration.takeStep()*. Possiamo schematizzarlo con lo pseudocodice seguente:

```
Point takeStep(RealAgent agent) {
        if agent.getEnvError() == true then
                nextStep = RandomWalk.takeStep(agent);
```

```
                agent.setEnvError(false);
        elseif agent.getTimeSinceLastPlan() > Constants.REPLAN_INTERVAL then
                agent.calculatePath(agent.getLocation(), agent.getCurrentGoal());
                agent.setTimeSinceLastPlan(0);
                nextStep = agent.getNextPathPoint();
        else
                nextStep = agent.getNextPathPoint();
        end
        agent.timeSinceLastPlan++;
        return nextStep;
}
```

Il metodo, come prima cosa, controlla che non ci sia stato alcun errore al passo compiuto precedentemente dal robot *agent*. Se l'errore è presente (può succedere quando un robot vuole spostarsi in una posizione occupata da un ostacolo), allora *agent* fa un passo casuale, tramite la strategia di esplorazione *RandomWalk*, e viene impostato il valore *false* alla variabile di errore *envError*. Altrimenti, il metodo controlla quanti cicli sono passati dall'ultima volta che ha calcolato il percorso verso la frontiera. Se il numero di cicli è oltre una certa soglia, che viene impostata dall'utente prima dell'inizio dell'esplorazione ed è contenuta in *REPLAN_INTERVAL* all'interno della classe *Constants*, allora il robot ricalcola il percorso verso la posizione che vuole raggiungere tramite il metodo *RealAgent.calculatePath()*. Dopodiché, aggiorna *timeSinceLastPlan*, che indica il numero di cicli passati dall'ultimo ricalcolo del percorso, e restituisce come prossimo passo *nextStep* il primo elemento contenuto nel percorso appena calcolato. Invece, se non è ancora il momento di ricalcolare il percorso, viene scelto come prossimo passo il primo elemento contenuto all'interno del percorso *path* utilizzando il metodo *getNextPathPoint()*. Infine, viene aggiornato il parametro *timeSinceLastPlan* e viene restituita la prossima posizione *nextStep* che il robot deve raggiungere all'interno dell'ambiente.

Ricalcolare il percorso ogni *REPLAN_INTERVAL* può essere utile, per esempio, per migliorare il percorso calcolato in precedenza alla luce delle nuove informazioni scoperte riguardo all'ambiente.

La strategia di esplorazione *RandomWalk* si occupa di calcolare in modo casuale, tramite il metodo *RandomWalk.takeStep()*, la prossima posizione che il robot deve raggiungere. Per fare questo, il robot viene ruotato in modo casuale di +45°, +22.5°, 0°, -22.5° oppure -45° rispetto alla direzione verso cui è rivolto in quel momento. Una volta effettuata la rotazione viene calcolata la posizione da raggiungere traslando il robot

lungo la nuova direzione verso cui è rivolto di una distanza pari alla lunghezza massima del passo. Il valore della lunghezza massima del passo è contenuto nel parametro *Constants.DEFAULT_SPEED*. Se è una posizione accettabile (per esempio, il robot non si scontra con un ostacolo), allora viene restituita quella posizione. Altrimenti, viene ripetuto il procedimento.

5.2.2 IlpExploration.replan()

La BS utilizza il metodo *replan()* contenuto all'interno della classe *IlpExploration* che possiamo schematizzare con lo pseudocodice seguente:

```
Array<Point> replan(RealAgent BS) {
	if needNewGoals == true then
		updateFrontiers(BS);
		m_opt = chooseFrontier(BS);
		return m_opt;
	else
		return BS.getM_opt();
	end
}
```

La BS controlla se è arrivato il momento di calcolare il nuovo schieramento dei robot. È necessario calcolare un nuovo schieramento della squadra quando tutti i robot hanno raggiunto le loro posizioni obiettivo. Se questa condizione è soddisfatta (*needNewGoals* è *true*) allora la BS calcola le frontiere presenti nell'ambiente conosciuto fino a quel momento tramite il metodo *updateFrontiers()*. Dopodiché, utilizza il metodo *chooseFrontier()* per calcolare lo schieramento *m_opt* dei robot all'interno dell'ambiente. Lo schieramento è rappresentato come un array, in cui sono contenute le posizioni obiettivo che i robot devono raggiungere. Se, invece, i robot non hanno ancora raggiunto le posizioni che hanno come obiettivo, allora il metodo restituisce lo schieramento attuale.
Più nel dettaglio, il metodo *updateFrontiers()* salva all'interno della BS le frontiere interessanti presenti nell'ambiente. Possiamo schematizzarlo con lo pseudocodice seguente:

```
void updateFrontiers(RealAgent BS) {
	calculateFrontiers(BS);
```

```
        newFrontiers = frontiersOfInterest(BS.frontiers, BS.occGrid);
        BS.setFrontiers(newFrontiers);
}
```

Tramite il metodo *calculateFrontiers()* la BS trova tutte le frontiere esistenti all'interno della propria mappa dell'ambiente. Successivamente, vengono selezionate le frontiere appena trovate utilizzando il metodo *frontiersOfInterest()*. Questa selezione consiste nel rimuovere, per esempio, le frontiere troppo piccole (l'utente può scegliere la dimensione minima di una frontiera attraverso il parametro C*onstants.MIN_FRONTIER_SIZE*). Infine, vengono salvate all'interno della BS le frontiere che hanno superato la selezione.

5.2.3 IlpExploration.chooseFrontier()

Il metodo *chooseFrontier()* contiene la logica dell'algoritmo che permette di calcolare un nuovo schieramento dei robot all'interno dell'ambiente. Vediamo più nel dettaglio il funzionamento di questo metodo che può essere schematizzato con lo pseudocodice seguente:

```
Array<Point> chooseFrontier(RealAgent BS) {
        BS.updateF_FA();
        BS.updateG();
        BS.updateM_1;
        BS.updateW_p();
        m_opt = Ilp.callIlp(BS);
        return m_opt;
}
```

Tutti i metodi che vengono descritti di seguito si trovano all'interno della classe *RealAgent* ed hanno la funzione di generare i parametri[4] che vengono utilizzati dal metodo *callIlp()*, contenuto nella classe *Ilp* all'interno del package *decisore*.

Il metodo *updateF_FA()* si occupa di generare il vettore *F* contenente gli ID delle locazioni corrispondenti alle frontiere conosciute in quel momento ed il vettore *FA* contenente le dimensioni di queste frontiere. Il primo elemento del vettore *FA* corrisponde all'area della frontiera il cui ID si trova in prima posizione nel vettore *F*, il secondo elemento del vettore *FA* corrisponde all'area della frontiera il cui ID si trova in

[4]Tutti i parametri necessari al funzionamento di *callIlp()* vengono salvati all'interno di file di testo nella cartella del simulatore *MRESim-master*. Il generico parametro *parametro* è contenuto nel file chiamato *parametro.txt*.

seconda posizione nel vettore *F*, … L'ID della locazione *l* di una frontiera non è altro che la posizione di *l* all'interno della lista delle locazioni *locList* contenuta nella BS. In *locList* sono contenute le locazioni rappresentate dai vertici dell'insieme V^{t+1} presentato nella Sezione 4.1.

Il metodo *updateG()* si occupa di generare la matrice di comunicazione *G*. Questa matrice rappresenta il grafo di comunicazione tra le locazioni dell'ambiente e corrisponde al grafo G^{t+1} presentato nella Sezione 4.1. Il generico elemento g_{ij} della matrice *G* assume valore *1* se e solo se le locazioni *i* e *j*, appartenenti a *locList*, possono comunicare tra loro. Altrimenti, assume valore *0*.

Il metodo *updateM_1()* si occupa di generare il vettore contenente gli ID delle locazioni occupate, ad un certo istante di tempo, dai robot. Il primo elemento del vettore *m_1* corrisponde all'ID della locazione occupata dalla BS, il secondo elemento corrisponde all'ID della locazione occupata dal primo robot esploratore, … Il vettore *m_1* contiene gli ID delle locazioni presenti nello schieramento Q^t presentato nella Sezione 4.1.

Il metodo *updateW_p()* si occupa di generare la matrice che rappresenta i costi di spostamento da una locazione ad un'altra. I costi di spostamento non sono altro che le distanze euclidee tra le diverse locazioni appartenenti a *locList*. Ogni elemento della matrice è calcolato utilizzando la funzione $d(v_i, v_j)$ presentata nella Sezione 4.1 che calcola la distanza tra il vertice v_i e il vertice v_j.

5.2.4 Ilp.callIlp()

Questo metodo si trova all'interno della classe *Ilp* del package *decisore*. Possiamo schematizzarlo con lo pseudocodice seguente:

```
Array<Point> callIlp() {
        m_opt, D_opt, costMin = exec(ilp_model.py);
                if costMin == Infinity then
                        // Schieramento non trovato
                        return null;
                else
                        // Schieramento trovato
                        BS.setM_opt(m_opt);
                        return m_opt;
                end
        end
}
```

In altre parole, l'algoritmo è contenuto in un file esterno *ilp_model.py* che abbiamo scritto in Python ed il problema di programmazione lineare intera viene risolto utilizzando *gurobi* come ottimizzatore. Il file *ilp_model.py* viene chiamato da *callIlp* e legge i parametri presentati nella sezione precedente. Dopo l'esecuzione, *ilp_model.py* restituisce due valori: *m_opt* e *costMin*, che rappresentano rispettivamente il vettore delle posizioni degli agenti all'interno dello schieramento trovato (ovvero, Q^{t+1} presentato nella Sezione 4.1) ed il costo totale dello schieramento trovato. Nel caso in cui non venga trovato alcuno schieramento, che permetta ai robot di essere in comunicazione con la BS, la variabile *costMin* contiene il valore *infinito* ed il metodo *callIlp()* restituisce il valore *null*. Questo significa che l'esplorazione è terminata, perché non è stato possibile trovare uno schieramento connesso, all'interno dell'ambiente esplorato, che permetta di visitare almeno una frontiera. Se, invece, lo schieramento viene trovato, allora il metodo restituisce lo schieramento *m_opt*.

5.3 Stump Exploration

La strategia di esplorazione, di cui abbiamo descritto il modello nella Sezione 4.2, è contenuta all'interno della classe *StumpExploration* del package *exploration*.
Il metodo *StumpExploration.takeStep()* è incaricato di far muovere i robot esploratori. La struttura e le funzionalità di questo metodo sono identiche a quelle del metodo *IlpExploration.takeStep()* descritte nella Sezione 5.2.1. Sono identiche anche le strutture del metodo *StumpExploration.replan()* che viene chiamato dalla BS e del metodo *StumpExploration.updateFrontiers()*, interno a *StumpExploration.replan()*, rispetto a quelle dei metodi *IlpExploration.replan()* e *IlpExploration.updateFrontiers()* che sono stati presentati nella Sezione 5.2.2. Ciò che cambia, invece, è il metodo *StumpExploration.chooseFrontier()* che viene presentato di seguito.

5.3.1 StumpExploration.chooseFrontier()

Il metodo *chooseFrontier()* contiene la logica dell'algoritmo che permette di calcolare un nuovo schieramento dei robot nell'ambiente. Vediamo più nel dettaglio il funzionamento di questo metodo, che possiamo schematizzare con lo pseudocodice

seguente:

```
Array<Point> chooseFrontier(RealAgent BS) {
        utilities = initializeUtilities(BS, BS.frontiers);
        BS.updateC_V_beta(utilities);
        BS.updateD();
        BS.updateC_V();
        BS.updateR();
        BS.updateMu();
        BS.updateG();
        BS.updateM_1;
        BS.updateW_p();
        m_opt = Stump.callStump(BS);
        return m_opt;
}
```

Il metodo *initializeUtilities()* genera tutti i valori di utilità *utilities* per le coppie robot-frontiera. Il valore di utilità è direttamente proporzionale alla dimensione della frontiera ed è inversamente proporzionale alla distanza del robot dalla frontiera.

Tutti i metodi che vengono descritti di seguito si trovano all'interno della classe *RealAgent* e sono incaricati di generare i parametri[5] che vengono utilizzati dal metodo *callStump()*, contenuto nella classe *Stump* all'interno del package *decisore*.

I valori di utilità *utilities* sono passati come parametro di ingresso al metodo *updateC_V_beta()* che si occupa di generare i vettori *C_beta* e *V_beta*. Questi vettori rappresentano, rispettivamente, le frontiere che devono essere raggiunte all'interno dell'ambiente ed i robot che devono raggiungerle. Quindi, rappresentano gli assegnamenti robot-frontiera. Il metodo *updateC_V_beta()* è descritto con maggiore dettaglio al termine di questa sezione.

Il metodo *updateD()* si occupa di generare la configurazione di comunicazione degli agenti, ovvero l'albero di comunicazione attuale. La matrice D rappresenta la topologia di comunicazione C^t presentata nella Sezione 4.2 nell'Equazione *(i)*.

Il metodo *updateC_V()* si occupa di generare il vettore C, contenente gli ID delle locazioni conosciute, e di generare il vettore V, contenente gli ID di tutti i robot esploratori. Al vettore C vengono aggiunti gli ID delle locazioni che corrispondono ai centri delle frontiere appena scoperte dai robot. Nel vettore C sono contenuti gli ID delle locazioni rappresentate dai vertici appartenenti all'insieme V^t presentato nella

[5]Tutti i parametri necessari al funzionamento di *callStump()* vengono salvati all'interno di file di testo nella cartella del simulatore *MRESim-master*. Il generico parametro *parametro* è contenuto nel file chiamato *parametro.txt*.

Sezione 4.2. L'ID di una locazione *l* non è altro che la posizione di *l* all'interno della lista delle locazioni *locList* contenuta nella BS. L'ID di un agente, invece, è rappresentato dal valore del parametro *id* contenuto nella classe *RealAgent*.

I metodi *updateR()* e *updateMu()* si occupano di generare il parametro *R*, che indica la distanza massima entro cui due vertici possono comunicare ed è utilizzato per la costruzione della matrice di comunicazione *G*, ed il parametro *Mu*, che rappresenta il peso che viene assegnato ai costi di comunicazione tra robot e corrisponde al parametro μ presentato nella Sezione 4.2.

Il metodo *updateG()* è stato presentato nella Sezione 5.2.3. La matrice di comunicazione *G* rappresenta il grafo di comunicazione $G = (V, C)$ presentato nella Sezione 4.2.

Il metodo *updateM_1()* è stato presentato nella Sezione 5.2.3. Il vettore *m_1* contiene gli ID delle locazioni presenti nello schieramento Q^t visto nella Sezione 4.2.

Il metodo *updateW_p()* è stato presentato nella Sezione 5.2.3.

Il metodo *updateC_V_beta()* è utilizzato per la generazione degli assegnamenti robot-frontiera e rappresenta l'implementazione della fase *(a)* illustrata nella Sezione 4.2. Può essere schematizzato con lo pseudocodice seguente:

```
void updateC_V_beta(Array<Utility> utilities) {
        // Ricerca delle utilità con valore maggiore
        while !allAssignmentsDone do
                maxUtility = takeMax(utilities);
                if notIn(maxUtility.getIDs(), bestUtilites) and notIn(maxUtility.getFrontiers(), bestUtilities) then
                        bestUtilities.add(maxUtility);
                end
                check(allAssignmentsDone);
        end
        // Aggiornamento dei vettori C_beta e V_beta
        foreach utility ∈ bestUtilities do
                C_beta.add(utility.getFrontier());
                V_beta.add(bestUtilities.getID());
        end
}
```

Nel metodo vengono ricercati all'interno del vettore *utilites*, che rappresenta i valori di utilità di tutti i possibili assegnamenti robot-frontiera, gli assegnamenti con valore di utilità maggiore. L'assegnamento *maxUtility*, che ha valore di utilità massimo, viene inserito all'interno del vettore *bestUtilites*. Un assegnamento u_i viene inserito all'interno del vettore *bestUtilites* se e solo se questo vettore non contiene un assegnamento u_j che si riferisce alla stessa frontiera di u_i oppure allo stesso robot di u_i. Questo per evitare che

un robot sia assegnato a più frontiere o che una frontiera sia assegnata a più robot. Il processo di assegnamento si interrompe quando ogni robot è assegnato ad una frontiera, oppure quando non ci sono più frontiere disponibili da assegnare ai robot. Al termine di questo processo, vengono inseriti gli ID delle locazioni corrispondenti ai centri delle frontiere assegnate e gli ID degli agenti che devono raggiungere queste frontiere, all'interno dei vettori *C_beta* e *V_beta*, rispettivamente. La locazione il cui ID si trova alla posizione *i*-esima del vettore *C_beta* deve essere raggiunta dal robot il cui ID è stato inserito alla posizione *i*-esima del vettore *V_beta*. In prima posizione all'interno di *C_beta* e di *V_beta* sono presenti, rispettivamente, l'ID della locazione occupata dalla BS e l'ID della BS. A parte questo primo assegnamento della BS alla sua locazione fissa, gli altri assegnamenti robot-frontiera sono inseriti in ordine di valore di utilità decrescente. Affinché l'algoritmo implementato in *callStump* funzioni è necessario che la dimensione di questi vettori sia almeno due. In altre parole, deve esistere almeno un assegnamento robot-frontiera, oltre a quello BS-locazione fissa, affinché l'algoritmo possa calcolare il nuovo schieramento all'interno della mappa dell'ambiente.

5.3.2 Stump.callStump()

Vediamo ora il metodo *callStump()* che rappresenta l'implementazione della fase *(b)* descritta nella Sezione 4.2. Questo metodo si trova all'interno della classe *Stump* del package *decisore*. Possiamo schematizzarlo con lo pseudocodice seguente:

```
Array<Point> callStump() {
        while C_beta.size > 1 do
                m_opt, D_opt, costMin = exec(stump.exe);
                if costMin == Infinity then
                        // Schieramento non trovato
                        C_beta.removeLast();
                         V_beta.removeLast();
                else
                        // Schieramento trovato
                        BS.setM_opt(m_opt);
                        BS.setD_opt(D_opt);
                        return m_opt;
                end
        end
        // Fine del processo di esplorazione
        return null;
}
```

In altre parole, l'algoritmo di Stump [34] è contenuto in un file eseguibile esterno *stump.exe* che abbiamo scritto in Matlab. Questo eseguibile viene chiamato da *callStump* e legge i parametri presentati nella sezione precedente. Dopo l'esecuzione, *stump.exe* restituisce tre valori: *m_opt*, *D_opt* e *costMin*, che rappresentano rispettivamente lo schieramento trovato (ovvero, Q^{t+1} presentato nella Sezione 4.2), la matrice di adiacenza corrispondente all'albero di comunicazione dei robot nello schieramento trovato (ovvero, C^{t+1} presentato nella Sezione 4.2) ed il costo totale dello schieramento trovato. Nel caso in cui non venga trovato alcuno schieramento, che permetta ai robot di essere in comunicazione con la BS, la variabile *costMin* contiene il valore *infinito*. Di conseguenza, viene eliminato l'assegnamento robot-frontiera meno promettente, cioè quello con utilità minore. In particolare, vengono eliminati l'ID della locazione della frontiera presente nell'ultima posizione all'interno del vettore *C_beta* e l'ID del rispettivo robot all'interno di *V_beta*. A questo punto, il procedimento si ripete con i vettori *C_beta* e *V_beta* aggiornati. Finché non viene trovato uno schieramento fattibile, vengono rimossi gli ultimi elementi dei vettori *C_beta* e *V_beta*. Quando il vettore *C_beta* (e, di conseguenza, anche il vettore *V_beta*) contiene soltanto un elemento, cioè la posizione occupata dalla BS, il procedimento si interrompe ed il metodo *callStump()* restituisce il valore *null*. Questo significa che l'esplorazione è terminata perché non è stato possibile trovare uno schieramento connesso all'interno della mappa dell'ambiente della BS che permetta di visitare almeno una frontiera. Se, invece, lo schieramento viene trovato, allora il metodo restituisce lo schieramento *m_opt*.

5.4 Birk Exploration

La strategia di esplorazione, di cui abbiamo descritto il modello nella Sezione 4.3, è contenuta all'interno della classe *BirkExploration* del package *exploration*.

Il metodo *BirkExploration.takeStep()* è il metodo incaricato di fare muovere i robot esploratori. La struttura e le funzionalità di questo metodo sono identiche a quelle del metodo *IlpExploration.takeStep()* descritte nella Sezione 5.2.1. Sono identiche anche le strutture del metodo *BirkExploration.replan()* che viene chiamato dalla BS e del metodo *BirkExploration.updateFrontiers()*, interno a *BirkExploration.replan()*, rispetto a quelle

dei metodi *IlpExploration.replan()* e *IlpExploration.updateFrontiers()* che sono stati presentati nella Sezione 5.2.2. Ciò che cambia, invece, è il metodo *BirkExploration.chooseFrontier()* che viene presentato di seguito.

5.4.1 BirkExploration.chooseFrontier()

Questo metodo viene chiamato all'interno del metodo *BirkExploration.replan()* e calcola lo schieramento finale dei robot. Infatti, contiene la logica dell'algoritmo di esplorazione *Birk Exploration* [29]. Possiamo schematizzarlo con lo pseudocodice seguente:

```
Array<Point> chooseFrontier(RealAgent BS) {
    // Generazione della popolazione
    foreach i < Constants.NUM_MAX_CONFIG do
        new populationConfig, utilities, populationUtility;
        // Generazione delle mosse casuali
        foreach t ∈ BS.getTeammates() do
            populationConfig.add(generateRandomMove(t));
        end
        // Calcola l'utilità di ogni membro della squadra
        foreach t ∈ BS.getTeammates() do
            position_t = populationConfig.get(t);
            if DirectLinePossible(position_t, BS.location) then
                utility_t = -bigValue;
            else
                utility_t = utilityEstimate(position_t, t.getBestFrontier());
            end
            utilities.add(utility_t);
        end
        // Calcola l'utilità totale della popolazione
        populationUtility = sum(utilities);
        // Salva soltanto la configurazione con utilità maggiore
        if bestConfig.getUtility() < populationUtility then
            bestConfig.setPopulation(populationConfig);
            bestConfig.setUtility(populationUtility);
        end
    end
    checkDeployment(bestConfig.getPopulation());
    BS.setM_opt(bestConfig.getPopulation());
    return bestConfig.getPopulation;
}
```

Vengono generate mosse di spostamento casuali per ogni robot all'interno della griglia dell'ambiente, formando così una popolazione *populationConfig* rappresentata come un vettore di locazioni dell'ambiente (ovvero, lo schieramento *i*-esimo Q_i^{t+1} visto nella Sezione 4.3). Il robot può spostarsi in una cella adiacente oppure rimanere fermo. Ogni elemento all'interno di questo vettore rappresenta la posizione obiettivo che un robot

deve raggiungere. Queste prime operazioni rappresentano l'implementazione della fase *(a)* presentata nella Sezione 4.3. Dopodiché, viene calcolata l'utilità *utility_t* di ogni posizione *position_t* all'interno di *populationConfig* nel modo seguente. Se il robot *t*, raggiunta la posizione *position_t* che gli è stata assegnata, non comunica con la BS, allora all'utilità *utility_t* viene associato il valore di penalità *-bigValue*. Se, invece, *t* comunica con la BS, allora all'utilità *utility_t* viene associata l'utilità calcolata rispetto alla frontiera più vicina tramite il metodo *utilityEstimate()*. Questo modo di assegnare i valori di utilità non è altro che l'implementazione della funzione di utilità presentata nella Sezione 4.3 nell'Equazione *(ii)*. Le utilità *utility_t* vengono inserite all'interno del vettore *utilities*. Alla fine di questo procedimento viene confrontata l'utilità totale *populationUtility* della popolazione attuale *populationConfig* con l'utilità totale della migliore popolazione *bestConfiguration* trovata fino a quel momento. L'utilità totale della popolazione è calcolata come la somma delle singole utilità appartenenti al vettore *utilities*. Nel caso in cui, l'utilità totale della popolazione migliore *bestConfiguration* sia minore rispetto a quella della popolazione attuale *populationConfig*, allora *bestConfiguration* viene sostituita da *populationConfig*. Le operazioni appena descritte rappresentano l'implementazione della fase *(b)* presentata nella Sezione 4.3.

Questo procedimento viene ripetuto *NUM_MAX_CONFIG* volte. Dopodiché, viene controllato il migliore schieramento trovato *populationConfig* tramite il metodo *checkDeployment()*. Questo metodo verifica che lo schieramento trovato sia diverso da quello del ciclo precedente. Nel caso sia uguale, viene incrementato un contatore. Quando questo contatore supera la soglia *Constants.TIMEOUT_FRONTIERS*, il metodo inserisce nella lista *blacklist* la frontiera dello schieramento *bestConfig* che ha valore di utilità positivo minore. Se una frontiera *f* si trova all'interno della lista *blacklist*, allora, a qualsiasi assegnamento di un robot alla frontiera *f*, *utilityEstimate()* assegna il valore di utilità *-bigValue*. Questo è il modo in cui abbiamo affrontato i problemi di minimo locale che abbiamo presentato nella Sezione 4.3. Infine, viene salvata all'interno della BS la popolazione che rappresenta lo schieramento migliore *bestConfig*.

5.5 Utility Exploration

La strategia di esplorazione, di cui abbiamo descritto il modello nella Sezione 4.4, è contenuta all'interno della classe *UtilityExploration* del package *exploration*. .

5.5.1 UtilityExploration.takeStep()

Il metodo che permette lo spostamento dei robot è, anche in questo caso, il metodo *takeStep()*. Questo metodo è contenuto all'interno della classe *UtilityExploration*. Possiamo schematizzarlo con lo pseudocodice seguente:

```
Point takeStep(RealAgent agent) {
        if agent.getEnvError() == true then
                nextStep = RandomWalk.takeStep(agent);
                agent.setEnvError(false);
                return nextStep;
        end
        switch(agent.getState())
                case Initial:
                        nextStep = takeStep_Initial(agent);
                        break;
                case Explore:
                        nextStep = takeStep_Explore(agent);
                        break;
                case ReturnToParent:
                        nextStep = takeStep_ReturnToParent(agent);
                        break;
        end
        return nextStep;
}
```

Anche in questo caso, come prima cosa, viene controllato che non ci sia stato alcun errore al passo precedente (è presente un errore quando un robot vuole spostarsi in una posizione occupata da un ostacolo). Nel caso l'errore ci sia stato, il robot *agent* compie un passo casuale, tramite il metodo *RandomWalk.takeStep()*, e termina l'esecuzione restituendo il valore della posizione *nextStep* scelta casualmente. Altrimenti, il prossimo passo dipende dallo stato *state* in cui si trova il robot.

Lo stato *Initial* è lo stato che il robot assume nel primo ciclo di simulazione, quando non ha ancora percepito l'ambiente che lo circonda.

Lo stato *Explore* conduce il robot ad esplorare l'ambiente, facendolo muovere verso la frontiera che ha selezionato ad un istante di tempo precedente, oppure portandolo a

scegliere un'altra frontiera da raggiungere tra quelle che presenti all'interno dell'ambiente conosciuto.

Lo stato *ReturnToParent* conduce il robot a tornare alla BS per comunicarle la nuova informazione scoperta riguardo all'ambiente.

Vediamo più nel dettaglio i tre metodi di movimento relativi ai diversi stati in cui può trovarsi il robot: *takeStep_Initial()*, *takeStep_Explore()* e *takeStep_ReturnToParent()*.

5.5.2 UtilityExploration.takeStep_Initial()

Il metodo *takeStep_Initial()* è utilizzato per lo spostamento del robot *agent* al primo ciclo di simulazione. Possiamo schematizzarlo con lo pseudocodice seguente:

```
Point takeStep_Initial(RealAgent agent) {
        nextStep = new Point(agent.getLocation().x + 1, agent.getLocation().y);
        agent.setState(Explore);
        return nextStep;
}
```

Come prossimo passo *nextStep* viene scelta la cella adiacente a quella occupata dal robot *agent*. Inoltre, viene cambiato lo stato *state* di *agent* in *Explore*. In questo modo il prossimo passo viene calcolato dal metodo *takeStep_Explore()*.

5.5.3 UtilityExploration.takeStep_Explore()

Il metodo *takeStep_Explore()* può essere schematizzato con lo pseudocodice seguente:

```
Point takeStep_Explore(RealAgent agent) {
        infoRatio = agent.getInfoKnownBS() / (agent.getInfoKnownBS() + agent.getInfoNew());
        if infoRatio < Constants.TARGET_INFO_RATIO then
                // Torna alla BS a comunicare
                agent.setState(ReturnToParent);
                agent.setPathToBaseStation();
                nextStep = agent.getNextPathPoint();
                return nextStep;
        end
        nextStep = FrontierExploration.takeStep(agent);
        return nextStep;
}
```

Viene calcolato il parametro *infoRatio* e confrontato con il parametro

TARGET_INFO_RATIO, come visto nella *(iii)* presentata nella Sezione 4.4, dove i parametri, interni al robot *agent*, chiamati *infoKnownBS* e *infoNew* corrispondono, rispettivamente, a *infBase* e *infNew* e *TARGET_INFO_RATIO* corrisponde al parametro di soglia *r*. La quantità di informazione viene misurata in termini di numero di celle. Se *infoRatio* è inferiore a *TARGET_INFO_RATIO*, allora significa che *agent* ha raccolto abbastanza informazione e deve tornare a comunicare con la BS. In questo caso cambia il suo stato *state* in *ReturnToParent* e calcola il percorso per tornare alla BS. Altrimenti, significa che non ha raccolto ancora abbastanza informazione e continua l'esplorazione. Il metodo che gestisce l'esplorazione dell'ambiente è *takeStep()* contenuto all'interno della strategia di esplorazione basata su frontiera chiamata *FrontierExploration* presente all'interno del package *exploration*.
Entriamo nei particolari di *FrontierExploration.takeStep()* che possiamo schematizzare con lo pseudocodice seguente:

```
Point takeStep(RealAgent agent) {
        if agent.getEnvError() == true then
                nextStep = RandomWalk.takeStep(agent);
                agent.setEnvError(false);
        elseif agent.getTimeSinceLastPlan() < Constants.REPLAN_INTERVAL and agent.getPath() != null then
                // Non è ancora il momento di calcolare un nuovo path
                nextStep = agent.getNextPathPoint();
        elseif agent.getTimeSinceLastPlan() > Constants.REPLAN_INTERVAL then
                // È il momento di calcolare un nuovo path
                nextStep = replan(agent);
        else
                // Non sono rimasti punti all'interno del path
                nextStep = replan(agent);
        end
        agent.timeSinceLastPlan++;
        return nextStep;
}
```

Innanzitutto, viene controllato che il robot non abbia avuto problemi durante il passo precedente (può succedere quando un robot vuole spostarsi in una posizione occupata da un ostacolo). Se li ha avuti, allora viene fatto un passo casuale. Altrimenti, viene controllato se il robot *agent* dispone di un percorso da seguire e se non è ancora arrivato il momento di calcolarne uno nuovo. Se è così, allora viene selezionata come posizione da raggiungere *nextStep* il prossimo passo del percorso conosciuto tramite il metodo *getNextPathPoint()*. Se, invece, è arrivato il momento di calcolare un nuovo percorso, ovvero sono passati *REPLAN_INTERVAL* cicli dall'ultimo ricalcolo del percorso, viene

chiamato il metodo *replan()* che si trova all'interno di *FrontierExploration*. Questo metodo viene chiamato anche nel caso in cui il robot *agent* ha raggiunto la frontiera obiettivo e di conseguenza non ha un percorso da seguire. Infine, *takeStep()* restituisce il prossimo punto da raggiungere *nextStep* all'interno dell'ambiente.
Vediamo più nel dettaglio il metodo *FrontierExploration.replan()* che possiamo schematizzare con lo pseudocodice seguente:

```
Point replan(RealAgent agent) {
        calculateFrontiers(agent);
        chooseFrontier(agent);
        agent.setTimeSinceLastPlan(0);
        nextStep = agent.getNextPathPoint();
        return nextStep;
}
```

Come prima cosa, vengono calcolate le frontiere all'interno dell'ambiente conosciuto dal robot *agent* tramite il metodo *calculateFrontiers()*. Dopodiché, viene chiamato il metodo *chooseFrontier()* che si occupa di scegliere la frontiera migliore da raggiungere e di calcolare il percorso verso quella frontiera. Infine, viene impostato a *0* il numero di cicli passati dall'ultimo calcolo del percorso e viene restituito il prossimo punto *nextStep* da raggiungere che corrisponde al primo punto del percorso calcolato.
Il metodo *FrontierExploration.chooseFrontier()* può essere schematizzato con lo pseudocodice seguente:

```
void chooseFrontier(RealAgent agent) {
        frontiers = frontiersOfInterest(agent.frontiers, agent.occGrid);
        utilities = initializeUtilities(agent, frontiers);
        frontier = bestFrontier(utilities);
        agent.calculatePath(agent.getLocation(), frontier.getCentre());
}
```

Il metodo *frontiersOfInterest()* si occupa di selezionare le frontiere migliori *frontiers* tra quelle trovate precedentemente (per esempio, esclude le frontiere troppo piccole). Dopodiché, il metodo *initializeUtilites()* calcola le utilità *utilities* relative alle frontiere *frontiers* per il robot *agent*. Il valore di utilità è direttamente proporzionale alla dimensione della frontiera ed è inversamente proporzionale alla distanza del robot dalla frontiera. Infine, con una sequenza di operazioni, che riassumiamo per semplicità utilizzando il metodo *bestFrontier()*, viene scelta la frontiera *frontier* con valore di utilità maggiore e viene calcolato il percorso del robot *agent* verso la frontiera *frontier.*

5.5.4 UtilityExploration.takeStep_returnToParent()

Il metodo *takeStep_ReturnToParent()* può essere schematizzato con lo pseudocodice seguente:

```
Point takeStep_ReturnToParent(RealAgent agent) {
        if agent.isInRange(BS) then
                // Ha comunicato con BS, torna ad esplorare
                agent.setState(Explore);
                return takeStep_Explore(agent);
        end
        infoRatio = agent.getInfoKnownBS() / (agent.getInfoKnownBS() + agent.getInfoNew());
        if infoRatio > Constants.TARGET_INFO_RATIO then
                // La nuova informazione non è abbastanza, torna ad esplorare
                agent.setState(Explore);
                return takeStep_Explore(agent);
        end
        nextStep = agent.getNextPathPoint();
        return nextStep;
}
```

Se il robot *agent* si trova all'interno del raggio di comunicazione della BS, allora vuol dire che ha comunicato con la BS durante la fase di comunicazione *simulateCommunication()* nel ciclo di simulazione precedente. Di conseguenza, *agent* può tornare allo stato *Explore* ed esplorare l'ambiente utilizzando il metodo *takeStep_Explore()*. Altrimenti, viene calcolato il valore di *infoRatio*. Se questo valore si trova al di sopra della soglia *TARGET_INFO_RATIO*, il robot riprende ad esplorare l'ambiente. Questo può succedere quando un robot incontra un altro robot che ha comunicato più recentemente con la BS. Se nessuna di queste condizioni è soddisfatta, allora *agent* continua a muoversi verso la BS seguendo il percorso che aveva costruito all'interno di *takeStep_Explore()* prima di cambiare stato.

Capitolo 6

Realizzazioni sperimentali e valutazione

In questo capitolo, entriamo nel dettaglio delle attività sperimentali svolte utilizzando le diverse strategie di esplorazione i cui modelli sono stati presentati nel Capitolo 4 e le cui implementazioni sono state descritte nel Capitolo 5.
Il capitolo è diviso in due parti. Nella prima parte, descriviamo in che modo abbiamo impostato le simulazioni. Nella seconda parte, mostriamo i grafici costruiti dai dati raccolti e commentiamo i risultati ottenuti.

6.1 Impostazione dell'attività sperimentale

Per svolgere test replicabili sotto condizioni controllate, abbiamo utilizzato il simulatore MRESim [9], la cui implementazione è stata descritta nel dettaglio all'interno del Capitolo 5. Il simulatore è già stato utilizzato per testare le strategie in [6] e in [16]. Inoltre, include già i metodi per la comunicazione, la navigazione ed il mapping. In questo modo ci siamo dovuti concentrare soltanto sull'implementazione delle strategie di esplorazione.
Il singolo robot all'interno del simulatore è una piattaforma robotica differenziale, come un P3AT, equipaggiato con un telemetro laser (per esempio, un SICK LMS200), con un raggio di percezione massimo di 150 pixel, un FOV di 180° e una risoluzione angolare di 1°. Abbiamo assunto locomozione, localizzazione e percezione dei dati perfette.

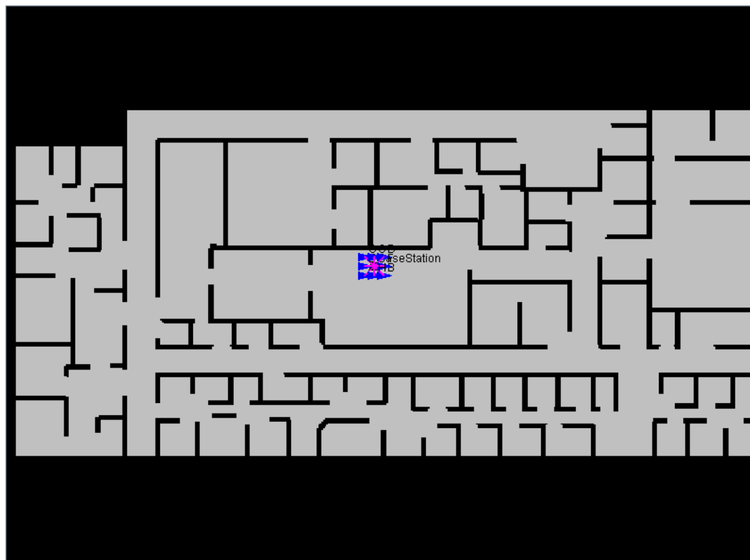
Nonostante queste assunzioni trovino difficilmente riscontro nella pratica, i risultati danno un'idea della bontà delle strategie. La comunicazione è basata su un modello a linea di vista limitato (raggio di 200 pixel). Facciamo notare che i raggi di percezione e di comunicazione possono corrispondere realisticamente a 7,5 metri e 10 metri rispettivamente, considerando un pixel come 5 centimetri.

Le strategie di esplorazione, presentate nei capitoli precedenti, sono state valutate in tre ambienti dove i robot hanno iniziato l'esplorazione da locazioni fisse (Figura 6.1). L'ambiente *office* è parte dell'ambiente *vasche_library_floor1* del repository Radish [8] e rappresenta un ambiente indoor realistico. Gli ambienti *open* e *maze* sono presenti all'interno del repository di MRESim. Il primo rappresenta una ambiente outdoor ed è caratterizzato dalla presenza di grandi spazi liberi. Il secondo rappresenta un ambiente labirintico ed è caratterizzato dalla presenza di molti ostacoli. Le loro dimensioni sono di circa 800 per 600 pixel (risultando così circa 40 per 30 metri). Abbiamo considerato squadre di 2, 4, 6 e 8 robot che si muovono alla velocità di 5 pixel (25 centimetri) per passo all'interno di ogni ambiente. Abbiamo definito un setting sperimentale come: un ambiente (*office*, *open* o *maze*), un numero di robot (2, 4, 6 o 8) e una strategia di esplorazione (*StumpExploration* che chiamiamo semplicemente *Stump*, *BirkExploration* che chiamiamo semplicemente *Birk* o *UtilityExploration* che chiamiamo semplicemente *Utility*; per l'ultima abbiamo considerato i valori di r 0.1, 0.5 e 0.9). Per ogni setting sperimentale abbiamo eseguito 5 esperimenti, poiché potevano capitare situazioni in cui il robot non riusciva a trovare alcun percorso fattibile (a causa degli errori di approssimazione durante l'integrazione dei dati sensoriali all'interno della mappa a griglia del robot presentati nella Sezione 5.1.3) e recuperava la situazione facendo un movimento casuale. Abbiamo valutato le prestazioni delle strategie misurando ogni 30 secondi la distanza percorsa (la media sui robot), il tempo in cui i robot sono rimasti disconnessi dalla BS (la media sui robot), la percentuale di area scoperta dai robot e conosciuta dalla BS ed il tempo necessario al calcolo delle prossime locazioni obiettivo da raggiungere (che chiamiamo *tempo di replan*). Questi dati sono stati raccolti utilizzando due timer differenti: *timer_1* ha misurato il tempo di esplorazione effettivo e *timer_2* ha misurato il tempo di esplorazione a cui è stato sottratto il tempo di replan. Abbiamo imposto la terminazione delle simulazioni al raggiungimento dei 20 minuti di timer_2. Questo ha portato ad avere simulazioni con un tempo misurato da timer_1 oltre

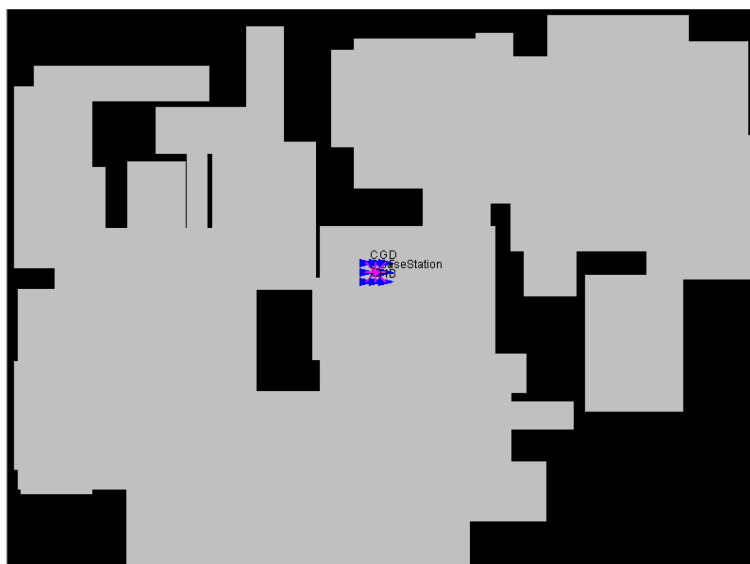
i 20 minuti, soprattutto a causa dell'alta complessità delle strategie *Stump* e *Birk*.

Facciamo notare che il tempo impiegato dalle simulazioni dipende fortemente dalla macchina su cui vengono eseguiti gli esperimenti (noi abbiamo utilizzato una macchina su cui è installato Windows 8.1 con una CPU 2.60 Ghz i5-4200U e 4GB di RAM), ma questa metrica può fornire una misura approssimativa sulla differenza tra gli algoritmi in termini di tempo di calcolo.

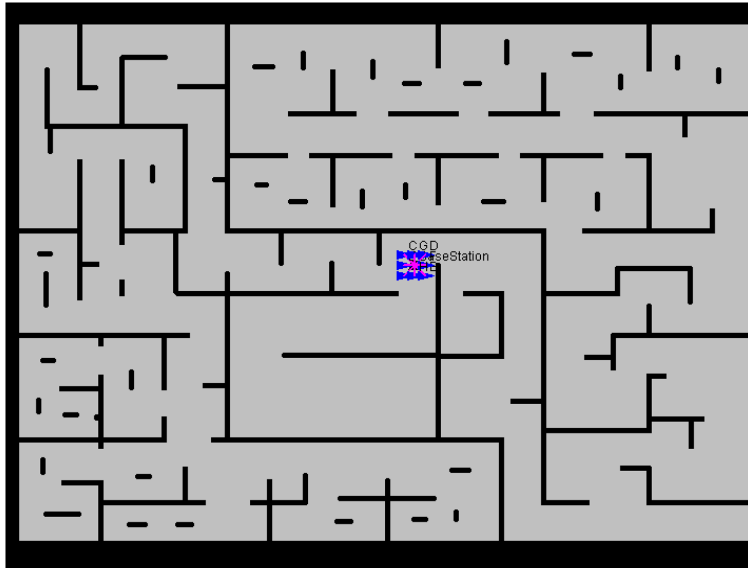
Inoltre, abbiamo eseguito alcuni esperimenti con la strategia IlpExploration (che chiamiamo *PLI*) e facciamo alcuni commenti comparandola alla strategia *Stump*.



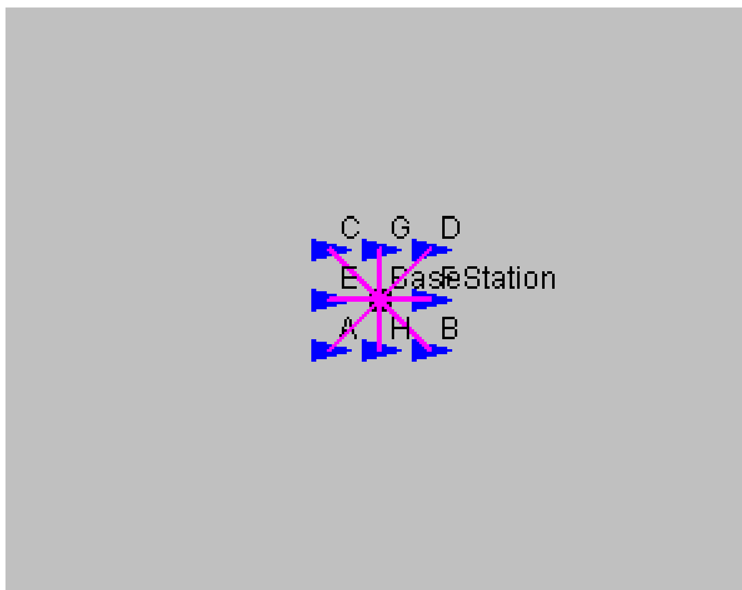
(a) Office



(b) Open



(c) Maze



(d) Posizioni iniziali

Figura 6.1: ambienti di test e disposizione iniziale degli agenti (BS: in posizione centrale; squadra di 2 robot: triangoli in basso a sinistra e a destra; squadra di 4 robot: aggiunta dei due triangoli in alto a sinistra e in alto a destra; squadra di 6 robot: aggiunta dei due triangoli a sinistra e a destra; squadra di 8 robot: aggiunta dei due triangoli in alto al centro e in basso al centro).

6.2 Risultati ottenuti

Nelle sezioni successive esponiamo i risultati ottenuti negli ambienti *office*, *open* e *maze* in cui abbiamo eseguito le simulazioni.

6.2.1 Office

La Figura 6.2 e la Figura 6.3 mostrano i grafici dei dati aggregati relativi a 20 minuti di esplorazione nell'ambiente *office*, rispettivamente, considerando e non considerando il tempo di replan.

Osserviamo che la distanza percorsa e l'area conosciuta dalla BS sono relativamente bassi per le strategie *Birk* e *Stump* rispetto alla strategia *Utility* (per qualsiasi valore del parametro *r* utilizzato). Questo può essere spiegato dal fatto che le prime due strategie assicurano dei vincoli di comunicazione rigidi, mentre l'ultima adotta solo un vincolo morbido. Anche per quanto riguarda il tempo di disconnessione, i valori sono relativamente bassi per le strategie *Birk* e *Stump* rispetto alla strategia *Utility*. Il motivo è legato, ancora una volta, al vincolo di comunicazione che caratterizza le strategie di esplorazione. Per quanto riguarda la strategia *Birk*, il vincolo di comunicazione rigido impone che i robot siano connessi con la BS in qualsiasi istante di tempo e che il tempo di disconnessione sia nullo. Per la strategia *Stump*, invece, il vincolo di comunicazione rigido impone che i robot siano connessi con la BS all'arrivo nelle posizioni obiettivo. Per quanto riguarda la strategia *Utility*, invece, i robot si muovono nell'ambiente senza essere connessi con la BS per la maggior parte del tempo. Dopo che un robot è arrivato alla BS ed ha condiviso le nuove informazioni, torna subito ad esplorare l'ambiente. Per questo motivo, i tempi di disconnessione sono molto alti. Osserviamo, inoltre, che i tempi di replan sono relativamente alti per le strategie *Birk* e *Stump*, rispetto a quelli della strategia *Utility* (per qualsiasi valore di *r*). Questo è dovuto al fatto che calcolare uno schieramento in modo centralizzato, dovendo rispettare un vincolo di comunicazione rigido, richiede molto più tempo rispetto a calcolare in modo decentralizzato la prossima frontiera da raggiungere senza curarsi di mantenere la connessione con la BS. Facciamo notare che considerando 2 e 4 robot, il processo di esplorazione della strategia *Stump* è terminato prima dello scadere dei 20 minuti in quanto non è stata trovata alcuna configurazione che soddisfacesse i vincoli di

comunicazione rigidi.

Confrontando le strategie che assicurano il vincolo rigido, la strategia *Stump* si comporta meglio di *Birk* in termini di area esplorata. Per esempio, nel caso di 6 robot (considerando il tempo di replan), questa differenza è statisticamente significativa (p-value < 10^{-9}) secondo una analisi ANOVA con un livello di significatività p-value < 0.05 [14]. La debolezza della strategia *Birk* sta principalmente nel fatto che può trovarsi in qualche minimo locale durante il calcolo del prossimo schieramento e impiega del tempo per uscirne. Un'altra debolezza è dovuta al fatto che la strategia stessa non è stata progettata per essere centralizzata, il che porta ad alcune situazioni in cui i robot non sono sparsi sulle frontiere. Tuttavia, la strategia *Birk* garantisce una comunicazione continua, mentre la strategia *Stump* assicura connettività soltanto una volta che i robot hanno raggiunto le locazioni obiettivo e non durante lo spostamento ed ha, di conseguenza, tempi di disconnessione dalla BS maggiori. Per quanto riguarda la distanza percorsa, in generale, la strategia *Birk* garantisce una distanza percorsa minore rispetto a *Stump*. Questo può essere motivato con il fatto che, ogni passo compiuto da un robot che utilizza la strategia *Stump* ha una lunghezza di 5 pixel, mentre ogni passo eseguito da un robot che utilizza la strategia *Birk* ha una lunghezza di 1 pixel (spostamento sulle celle adiacenti). Per quanto riguarda il tempo di replan, quando la strategia *Stump* riesce a concludere l'esplorazione senza terminare prima dei 20 minuti, notiamo che ha un tempo di replan più alto rispetto a *Birk*.

Confrontando la strategia *Stump* con alcuni esperimenti che abbiamo condotto con la strategia *PLI* (abbiamo eseguito simulazioni con 4 e 8 robot), abbiamo notato che la strategia *PLI* non scala bene con la dimensione del grafo di comunicazione, in quanto riesce ad ottenere risultati solo fino a 180 secondi (senza considerare il tempo di replan), prima che sia occupata troppa memoria. Il grafo precedente al momento in cui è stato fermato il calcolo ha circa 24 vertici. La strategia *PLI* ottiene risultati migliori, però, rispetto alla strategia *Stump* a parità di istante di tempo. Per esempio, con 4 robot, a 180 secondi, mentre la distanza percorsa è circa la stessa (284.3 pixel per *PLI* contro 297.7 pixel per *Stump*), la percentuale di area esplorata e conosciuta dalla BS è maggiore per *PLI* (25.3% contro 19.0%). Tuttavia, il tempo di replan è molto più alto per *PLI* (720.9 secondi contro circa 110 secondi).

Guardiamo ora come cambiano le prestazioni al variare del parametro *r* per la strategia

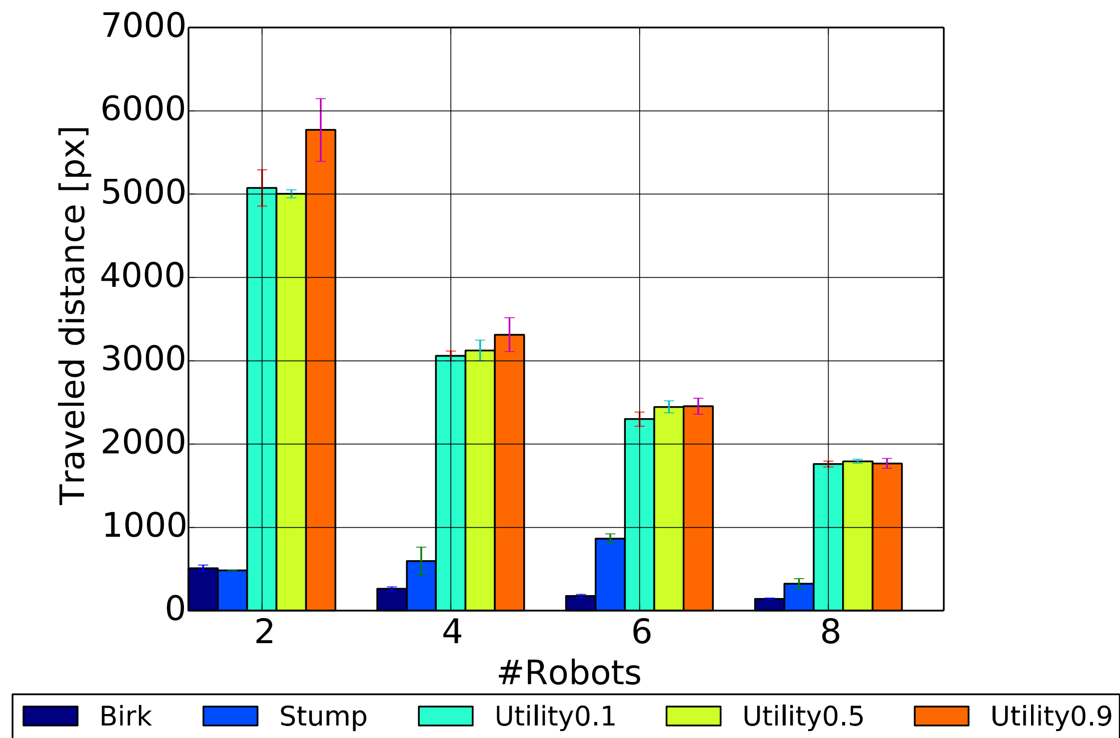
Utility. Maggiore è il valore di *r*, maggiore è l'area esplorata e conosciuta dalla BS (nella gran parte dei casi, questa differenza non è statisticamente significativa (per esempio, p-value = 0.09, con 4 robot, *Utility0.1* e *Utility0.9*)). Questo può essere giustificato dal fatto che, con bassi valori di *r*, i robot tornano meno frequentemente alla BS e non sono in grado di condividere le nuove informazioni entro i 20 minuti di simulazione. La distanza percorsa, invece, aumenta leggermente con *r* e la differenza è, a volte, statisticamente significativa, specialmente da $r = 0.1$ a $r = 0.9$ (per esempio, p-value = 0.03, con 4 robot, *Utility0.1* e *Utility0.9*). La ragione per cui questo succede è che, con alti valori di *r*, i robot devono tornare più frequentemente alla BS per condividere la nuova informazione dell'ambiente. Nella maggior parte dei casi, c'è una piccola diminuzione del tempo in cui i robot non comunicano con la BS con *r* più grandi. Tuttavia, questa differenza non è statisticamente significativa (per esempio, p-value = 0.3, con 8 robot, per *Utility0.1* e *Utility0.9*). Questo può essere giustificato dal fatto che, una volta che i robot comunicano la nuova informazione, tornano ad esplorare l'ambiente riattraversando e raggiungendo aree dell'ambiente dove la comunicazione con la BS non è possibile.

L'aumento del numero di robot non fornisce un vantaggio significativo in termini di area esplorata in questo ambiente (per esempio, p-value = 0.3, con 2 e 6 robot, *Utility0.9*), ma lo fa in termini di distanza percorsa e tempo di disconnessione, i quali diminuiscono, e la differenza è statisticamente significativa (per esempio, con p-value $< 10^{-9}$ e p-value = 0.005, rispettivamente, nello stesso setting sperimentale dell'ultimo esempio). Per quanto riguarda la distanza percorsa, questo miglioramento può essere spiegato dal fatto che ogni robot esplora zone dell'ambiente diverse, con il risultato che la distanza percorsa è minore. Per quanto riguarda il tempo di disconnessione, questo miglioramento può essere dovuto al fatto che alcuni robot possono spargersi per l'ambiente e fare da relay per comunicare le nuove informazioni raccolte alla BS. È interessante notare che l'area esplorata e conosciuta dalla BS decresca leggermente da 6 a 8 robot, perché i robot devono far fronte ad un numero maggiore di collisioni tra loro.

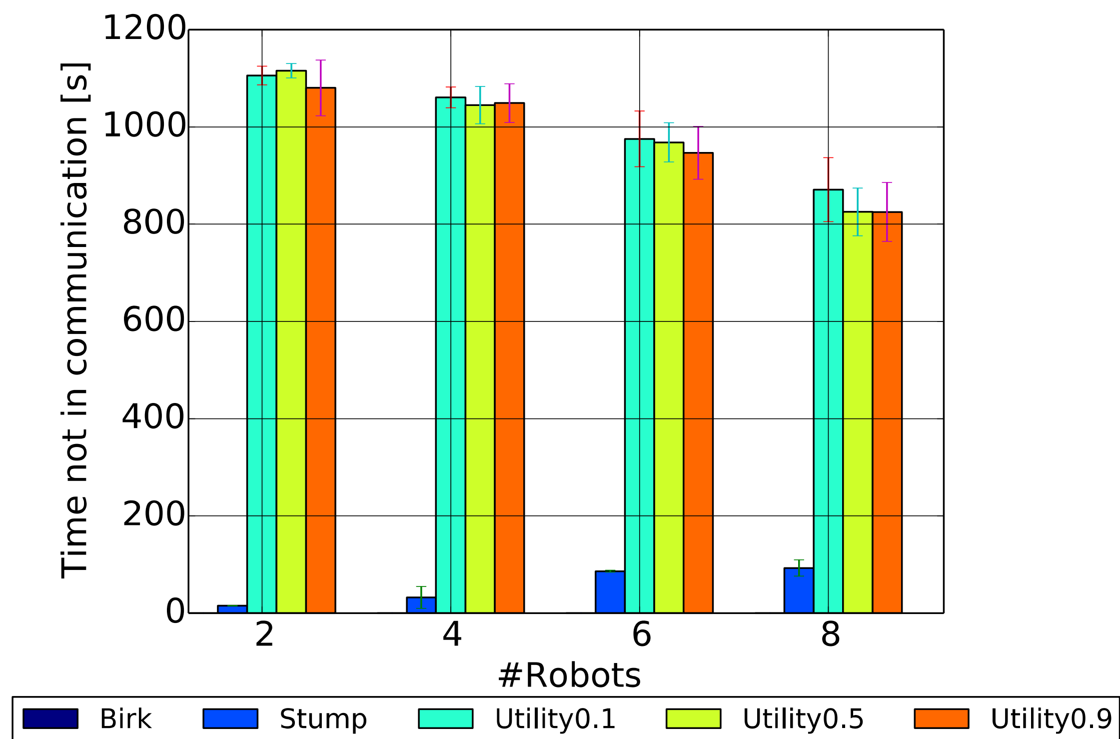
Osservando il tempo di replan, si nota che la strategia *Utility* non richiede lunghi tempi per calcolare le prossime frontiere da raggiungere (per qualsiasi numero di robot) e i suoi valori sono più o meno gli stessi sia includendo sia non includendo il tempo di replan nei 20 minuti limite. Le strategie *Birk* e *Stump* hanno alti tempi di replan rispetto

a *Utility*. La strategia *Birk* impiega molto tempo a calcolare i valori dell'utilità di ogni configurazione della popolazione. Inoltre il tempo di replan cresce leggermente con più robot, ma rimane più o meno costante per 4, 6 e 8 robot. Tuttavia, le prestazioni in termini di area esplorata rimangono più o meno le stesse, a causa delle debolezze presentate sopra. Un ragionamento simile può essere fatto per la strategia *Stump*, anche se c'è un piccolo miglioramento in termini di area esplorata. Inoltre, la strategia *Stump* non sembra scalare bene quando il numero di robot cresce.

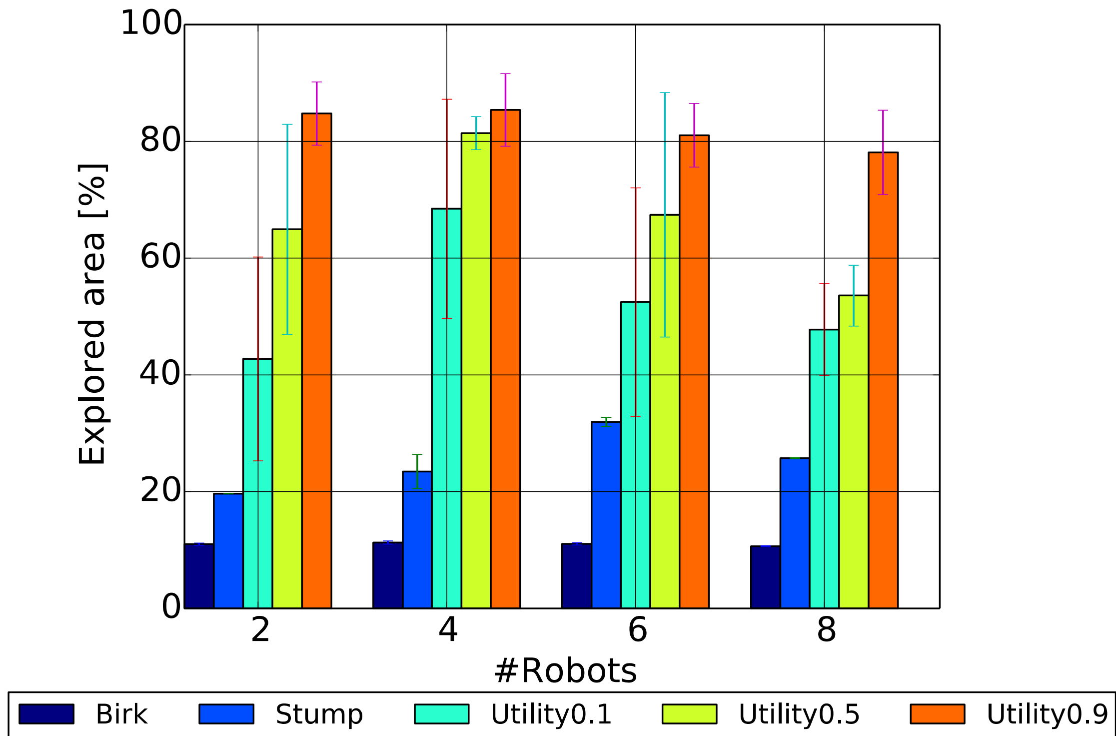
La Figura 6.4 e la Figura 6.5 mostrano l'andamento della distanza percorsa, il tempo in cui i robot non comunicano con la BS, la percentuale di area esplorata e conosciuta dalla BS ed il tempo di replan sui 20 minuti di intervallo di tempo con una squadra di 6 robot all'interno dell'ambiente *office*, rispettivamente, considerando e non considerando il tempo di replan. È interessante notare che, osservando i grafici in Figura 6.5, la strategia *Stump* raggiunge la strategia *Utility* con $r = 0.1$ in termini di area esplorata a circa 1000 secondi (entrambe le strategie hanno consegnato alla BS il 40% dell'area totale dell'ambiente), ma la strategia *Stump* ha valori di tempo di disconnessione (150 secondi contro 800 secondi circa) e di distanza percorsa (1300 pixel contro 2000 pixel circa) minori.



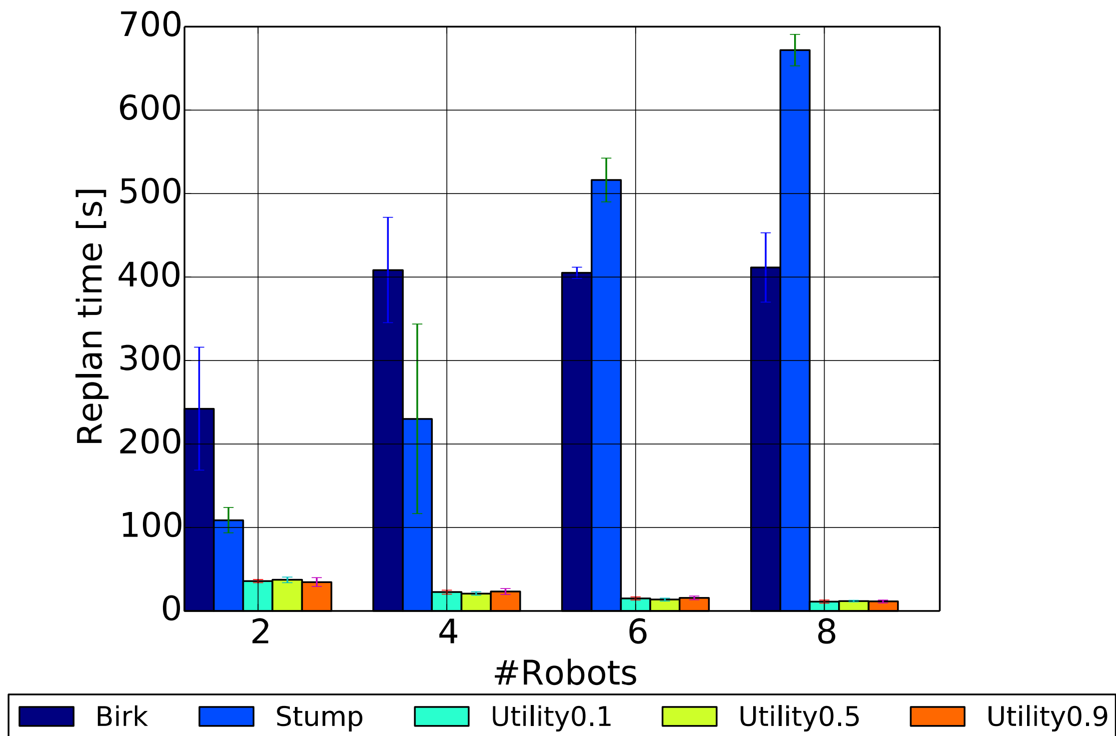
(a) Distanza percorsa



(b) Tempo di disconnessione

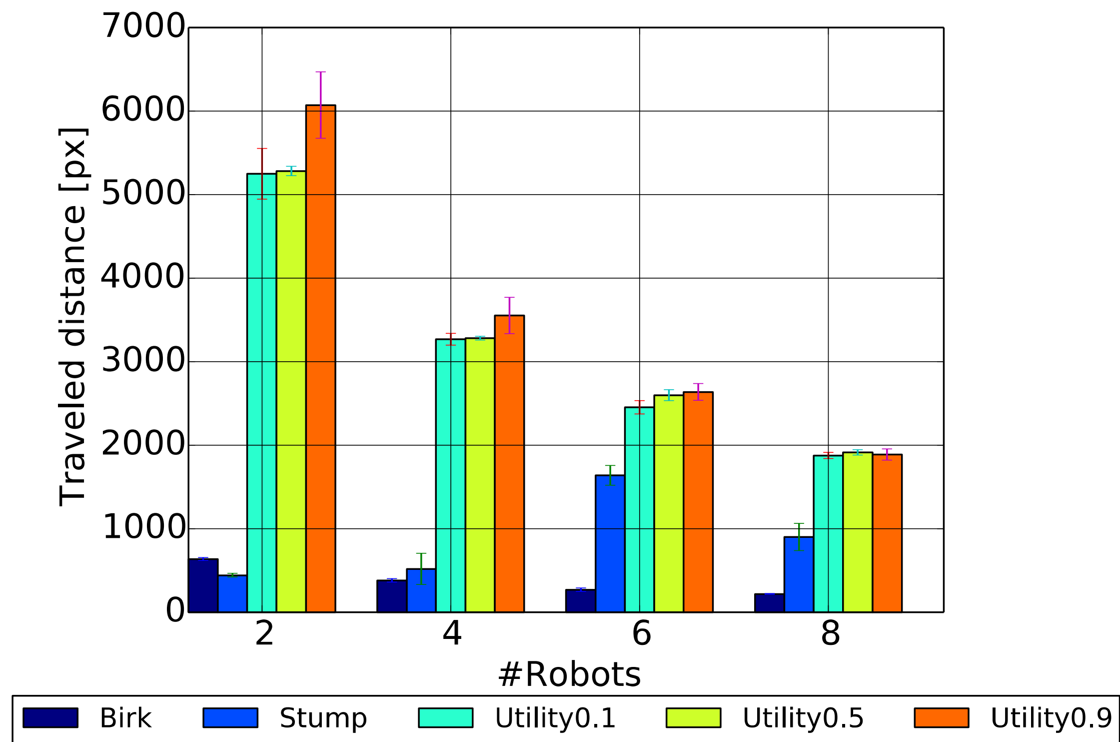


(c) Area esplorata

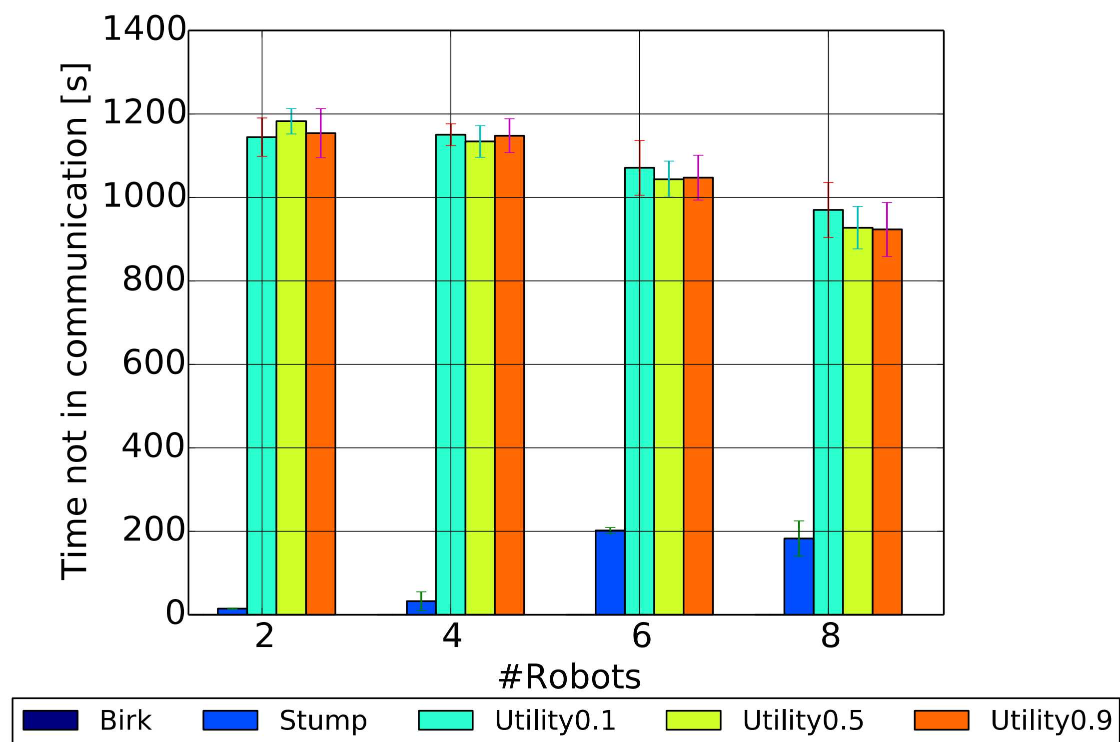


(d) Tempo di replan

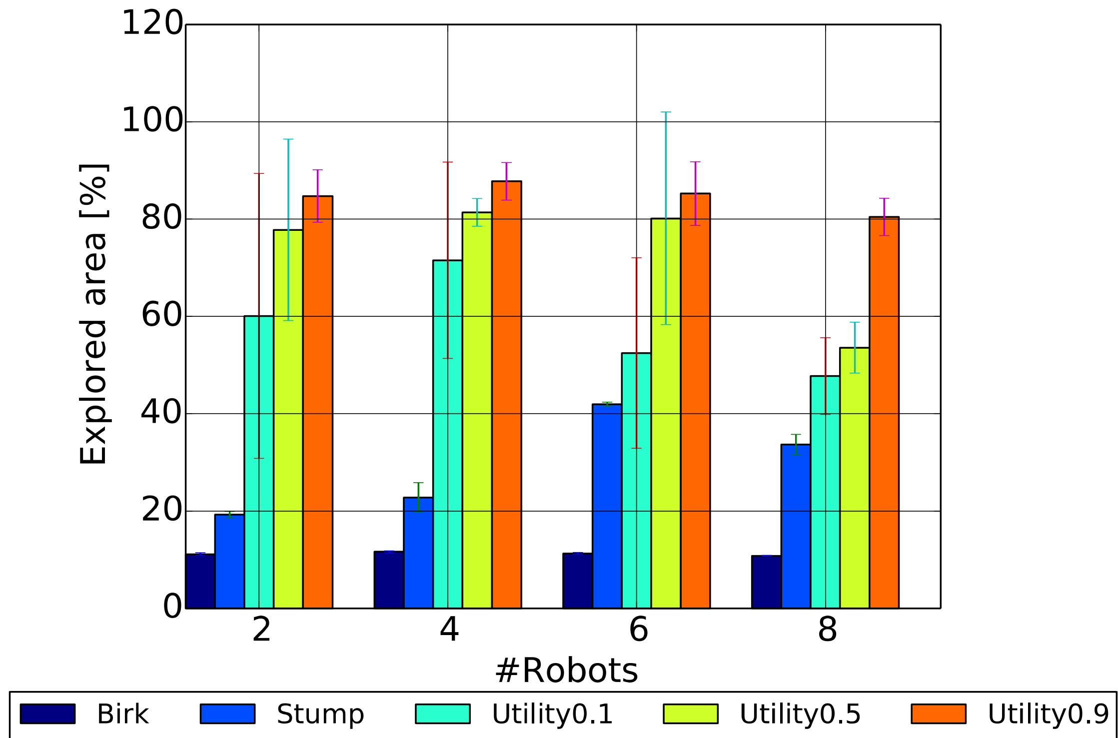
Figura 6.2: risultati (con media e deviazione standard) per l'ambiente office, dopo 20 minuti di esplorazione (considerando il tempo di replan).



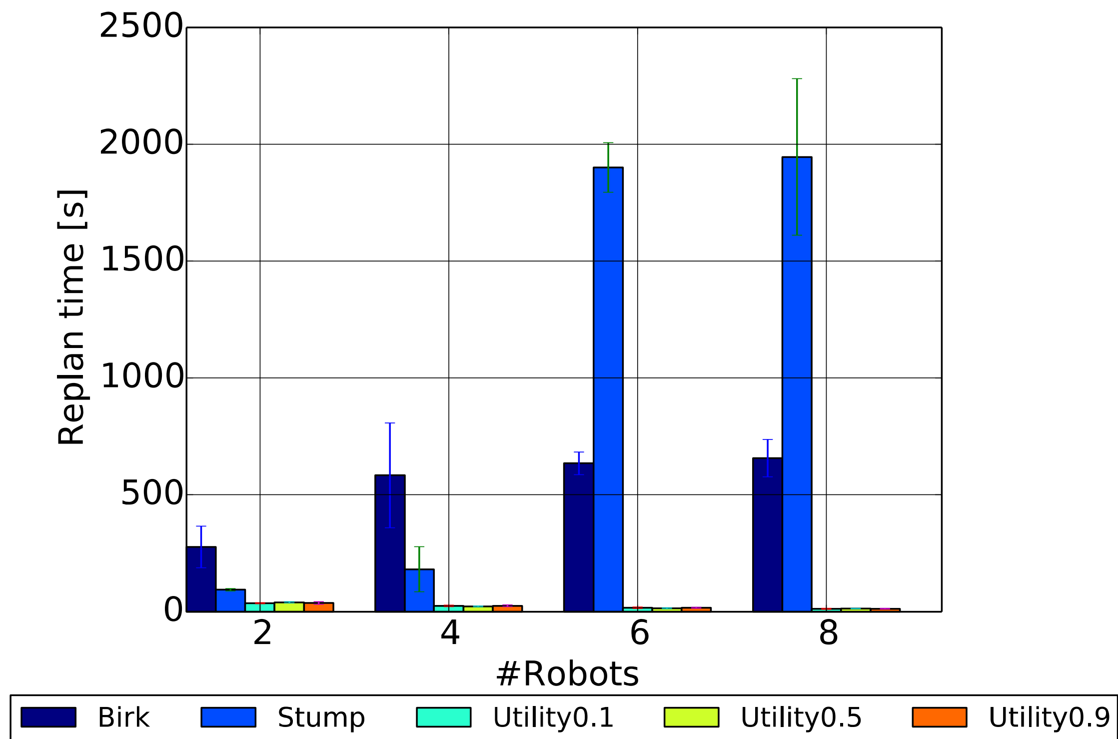
(a) Distanza percorsa



(b) Tempo di disconnessione

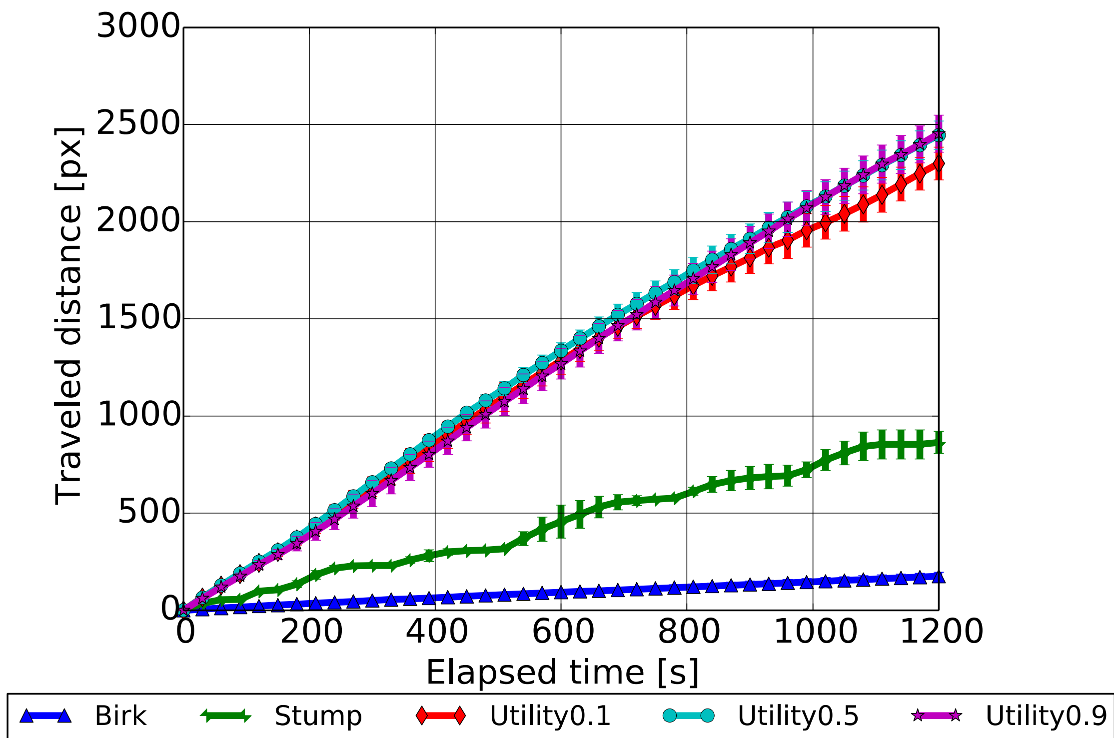


(c) Area esplorata

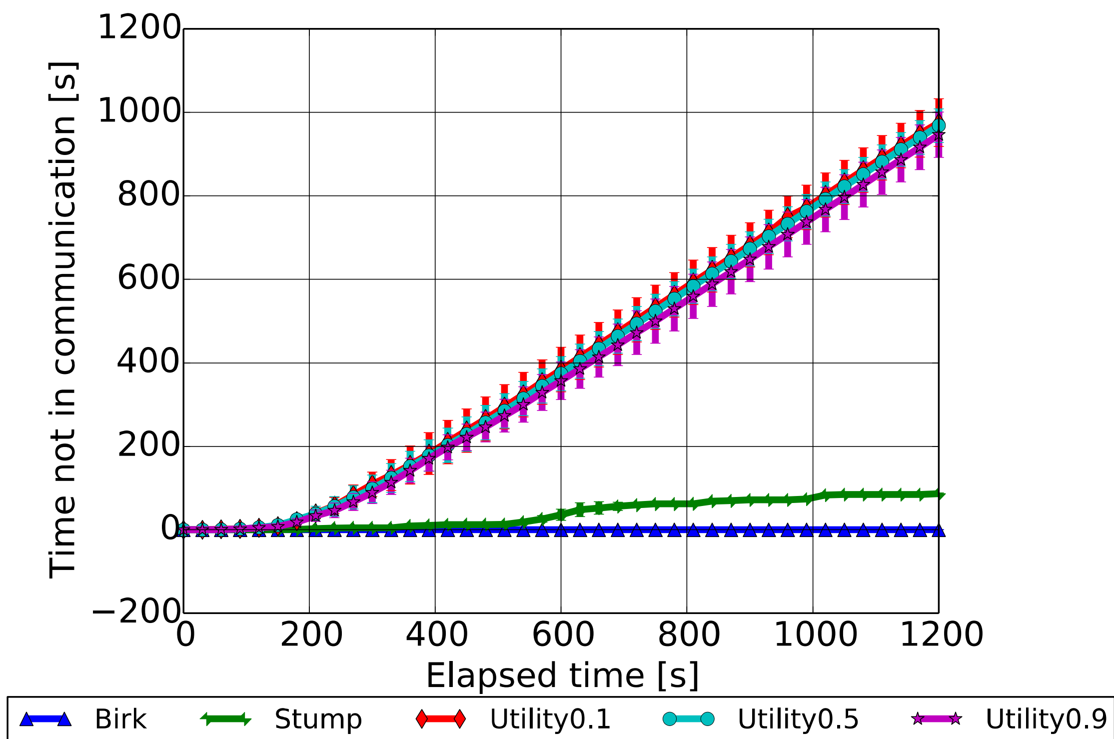


(d) Tempo di replan

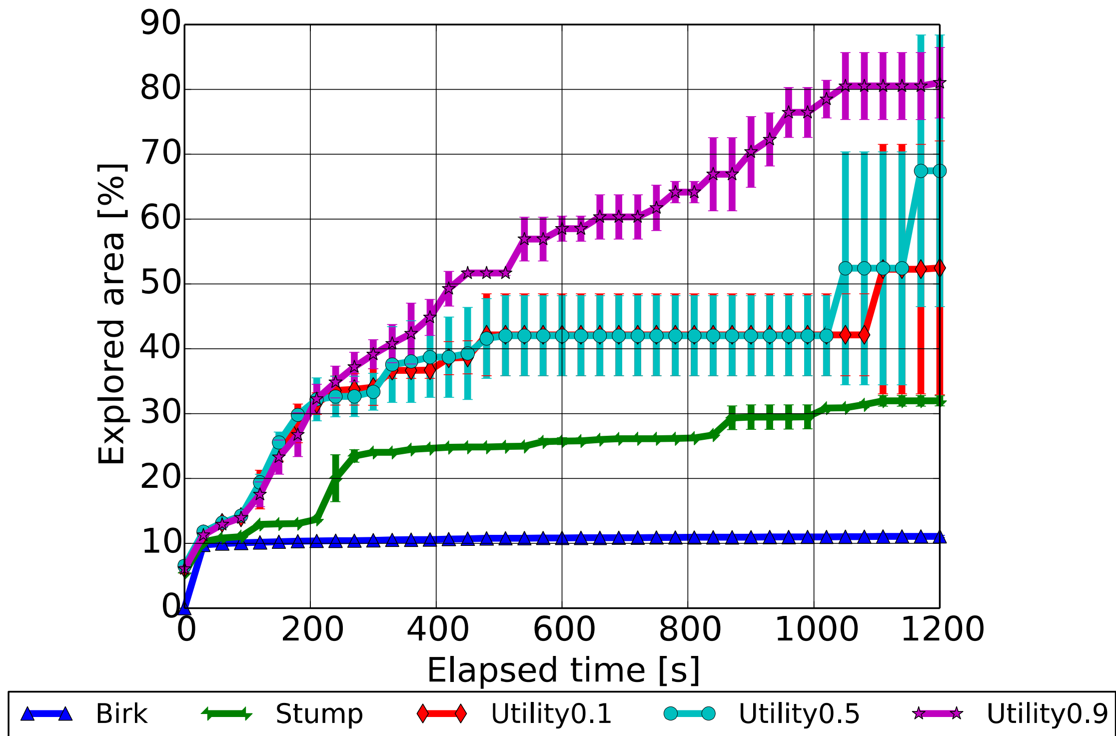
Figura 6.3: risultati (con media e deviazione standard) per l'ambiente office, dopo 20 minuti di esplorazione (non considerando il tempo di replan).



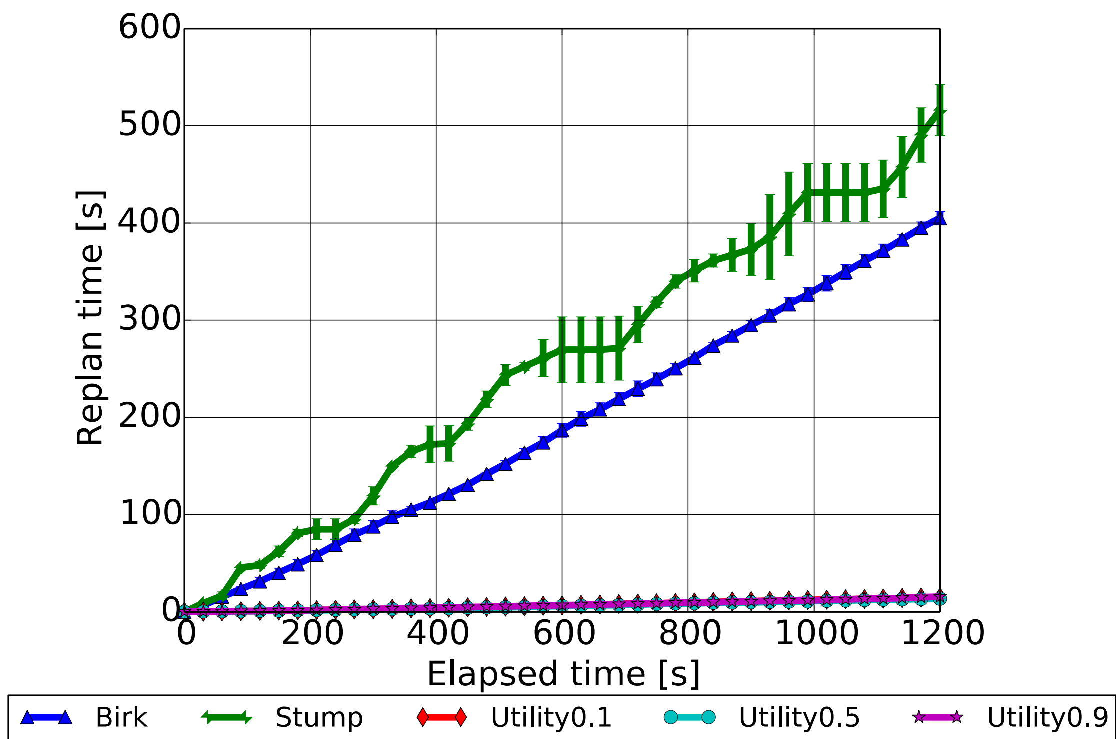
(a) Distanza percorsa



(b) Tempo di disconnessione

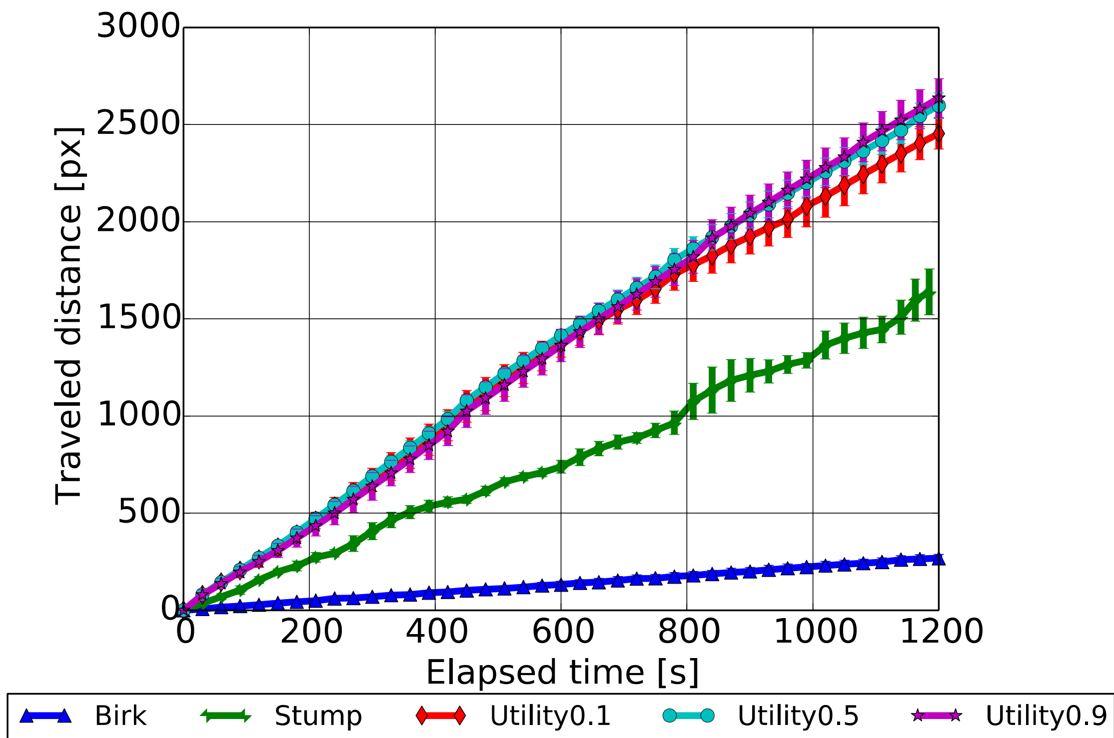


(c) Area esplorata

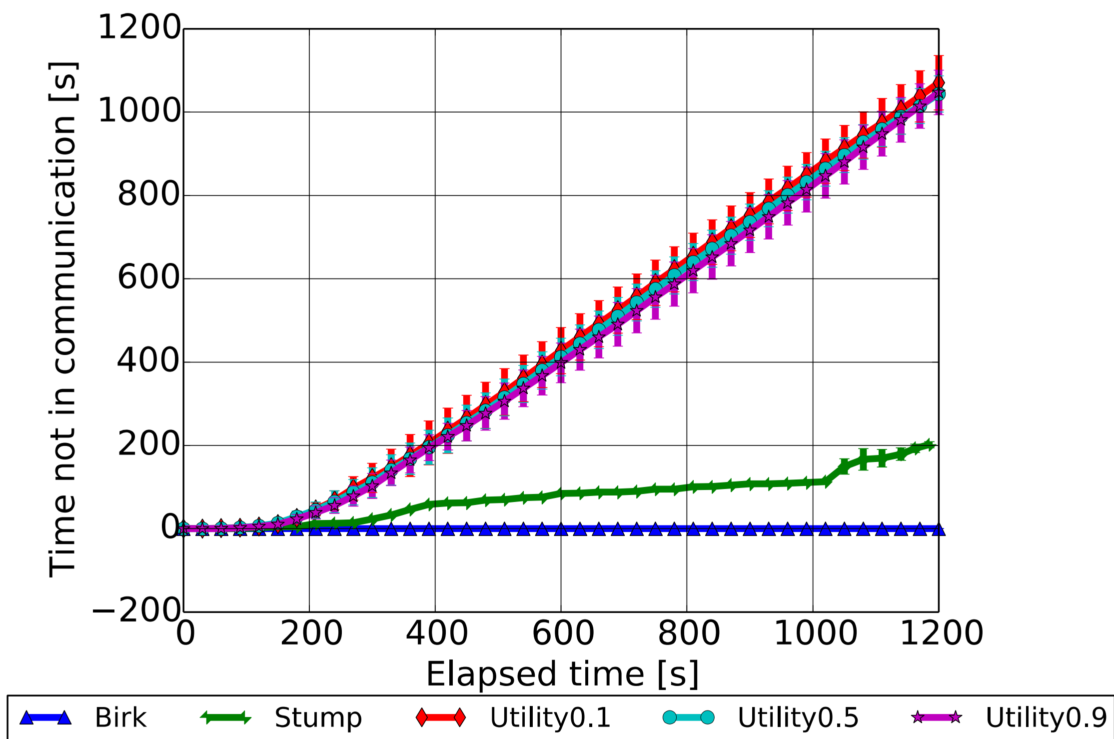


(d) Tempo di replan

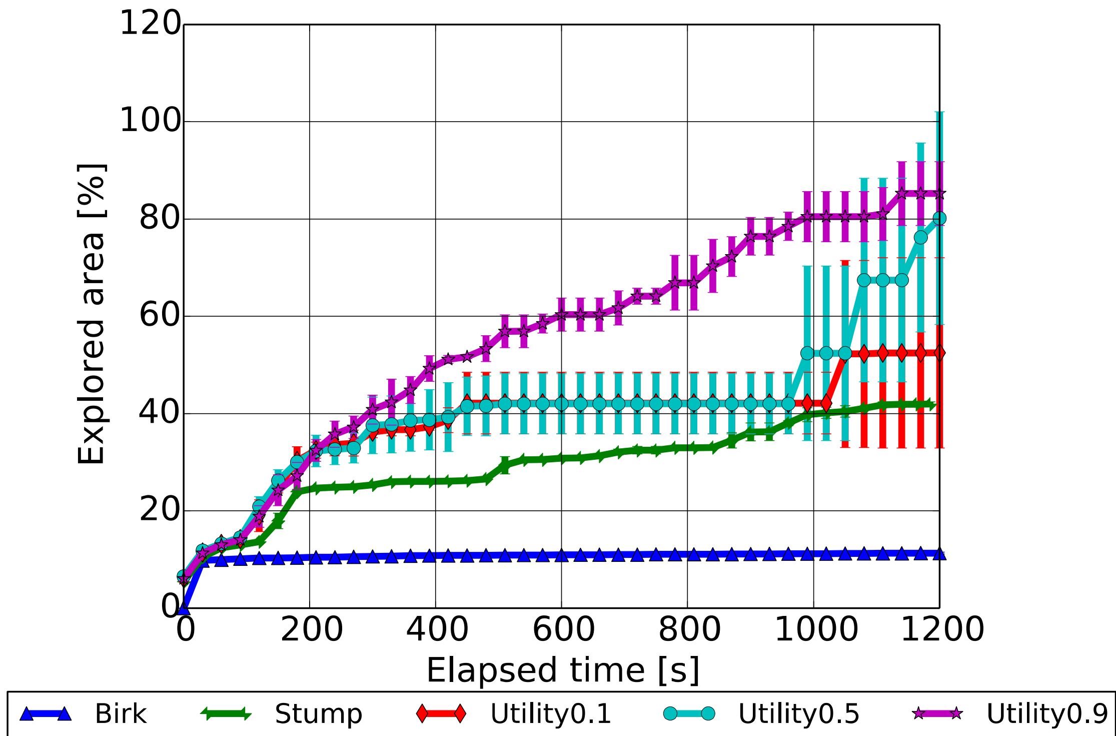
Figura 6.4: risultati (media e deviazione standard) per l'ambiente office, su 20 minuti di esplorazione, con 6 robot (considerando il tempo di replan).



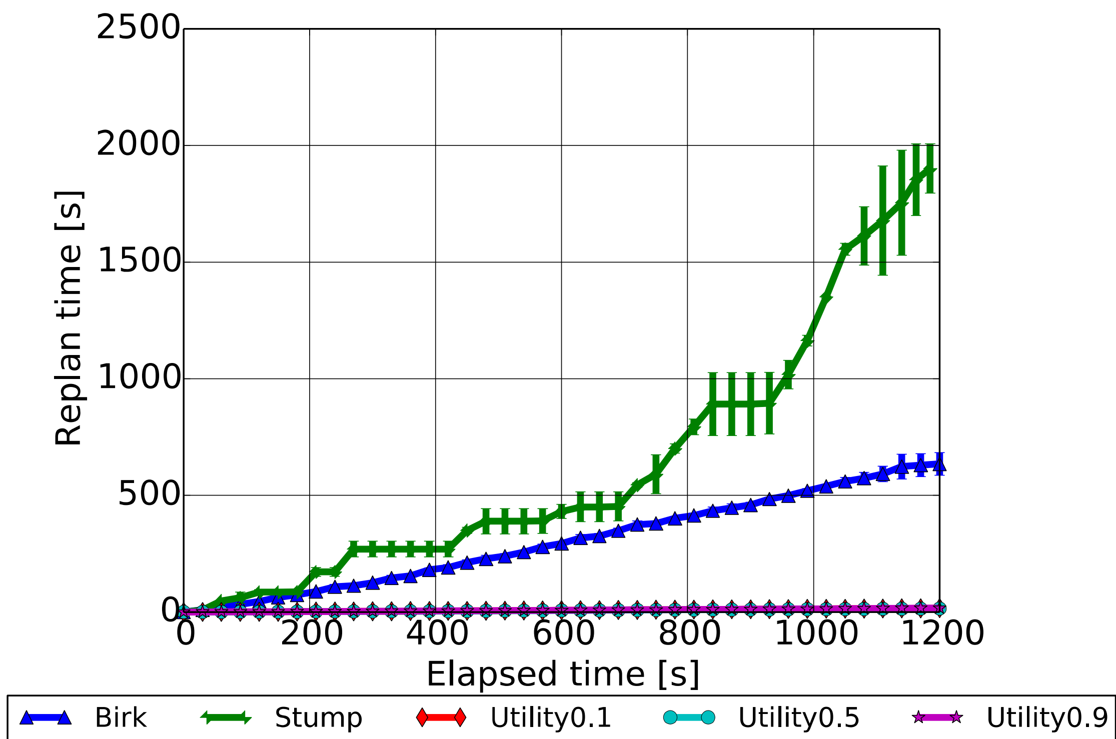
(a) Distanza percorsa



(b) Tempo di disconnessione



(c) Area esplorata



(d) Tempo di replan

Figura 6.5: risultati (media e deviazione standard) per l'ambiente office, su 20 minuti di esplorazione, con 6 robot (non considerando il tempo di replan).

6.2.2 Open

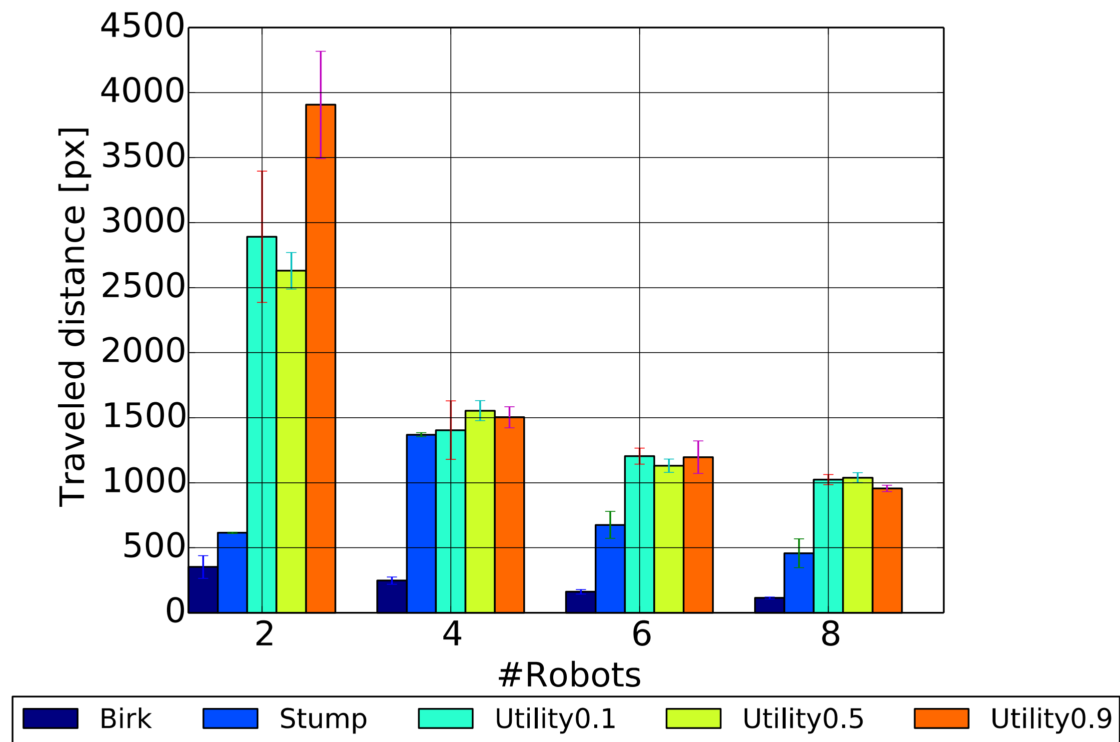
La Figura 6.6 e la Figura 6.7 mostrano i grafici dei dati aggregati relativi a 20 minuti di esplorazione nell'ambiente *open*, rispettivamente, considerando e non considerando il tempo di replan.

L'ambiente sembra più semplice da esplorare, infatti la strategia *Utility* è in grado di esplorare il 100% dell'area dell'ambiente (per qualsiasi valore di *r*). Inoltre, la strategia *Stump* raggiunge risultati abbastanza buoni in termini di area esplorata (per esempio, vicino al 100% nel caso di 4 e 6 robot senza considerare il tempo di replan). Con la strategia *Birk*, l'area esplorata aumenta in confronto a quella ottenuta in *office*, ma rimane intorno al 20%. Questo miglioramento può essere dovuto al fatto che i robot sono in grado di percepire porzioni maggiori di area dell'ambiente con una sola percezione. Il tempo di replan delle strategie *Birk* e *Stump* rimane alto rispetto a quello della strategia *Utility*. In questo caso, però, il tempo di replan per la strategia *Stump* è minore rispetto a quello della strategia *Birk*. La strategia *Birk* impiega circa lo stesso tempo speso in *office* per la generazione delle diverse popolazioni e per il calcolo della loro utilità. La strategia *Stump* ha tempi di replan molto più bassi rispetto a quelli di *office* (soprattutto per 6 e 8 robot). A causa dei grandi spazi, vengono trovate meno frontiere in *open*, rispetto all'ambiente precedente, ed il grafo ha dimensioni minori. Questo permette una computazione più rapida dello schieramento finale. La strategia *Stump*, con l'aumento del numero dei robot, sembra scalare meglio in questo ambiente rispetto ad *office* e, infatti, passando da 6 ad 8 robot, il tempo di replan diminuisce. La presenza di minori ostacoli e di grandi spazi all'interno dell'ambiente si riflette anche sulla distanza percorsa e sul tempo in cui i robot non sono in comunicazione con la BS. Infatti, in generale, la distanza percorsa dai robot all'interno dell'ambiente *open* è minore rispetto a quella percorsa all'interno dell'ambiente *office*, così come il tempo di disconnessione dalla BS. Questo è dovuto al fatto che, in *open*, i robot comunicano tra loro e con la BS molto più frequentemente rispetto all'ambiente precedente, grazie alla presenza di minori ostacoli, ed evitano di esplorare zone che sono già conosciute da altri robot.

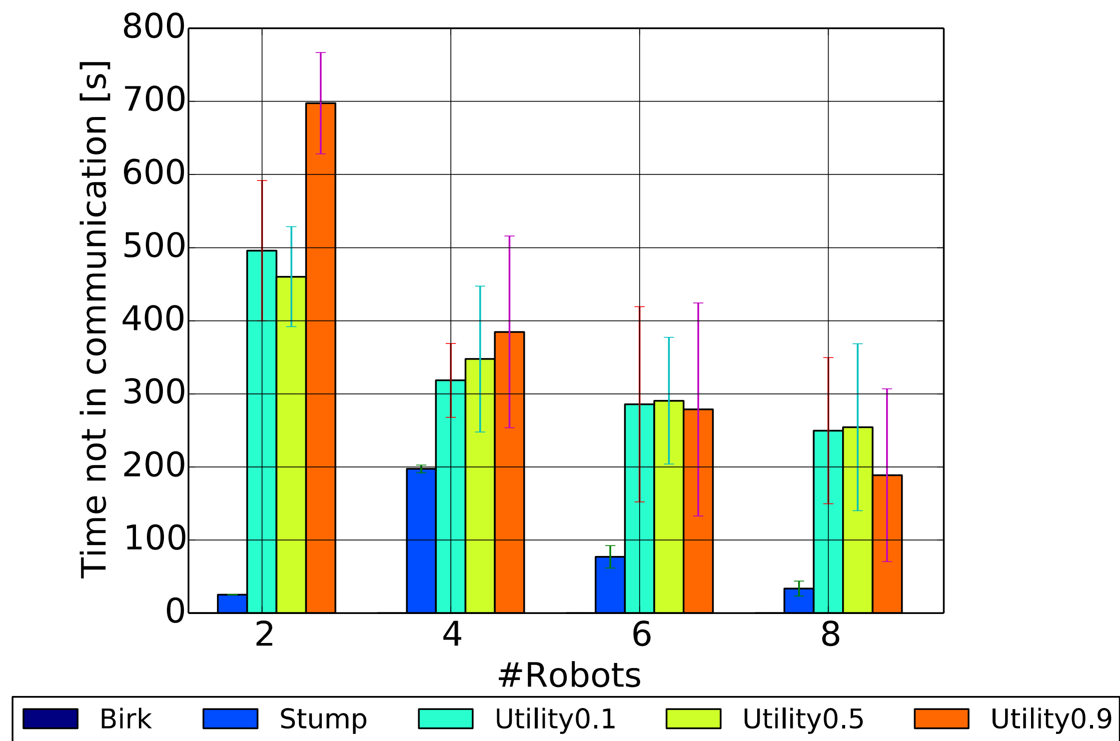
Confrontata con la strategia *PLI* (abbiamo eseguito esperimenti con 4 e 8 robot), la strategia *Stump* si comporta in modo simile (per esempio, per entrambi le strategie dopo 150 secondi con 4 robot la percentuale di area esplorata e conosciuta dalla BS è circa il

50% e la distanza percorsa è circa 200 pixel).

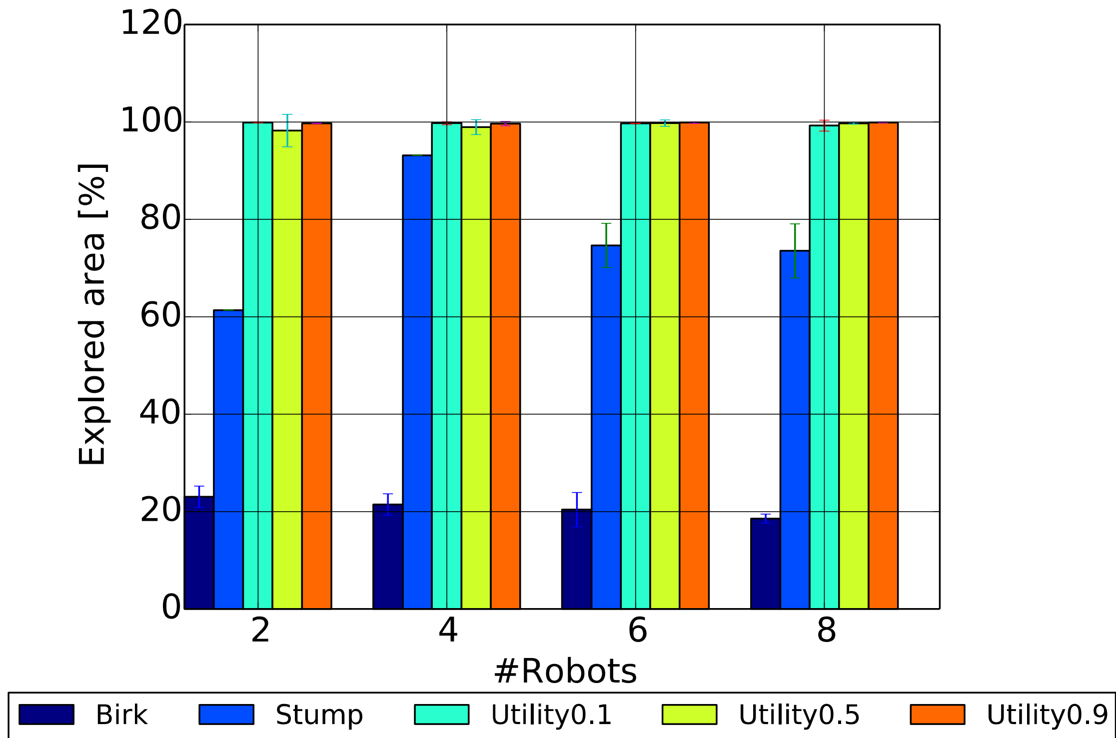
La Figura 6.8 e la Figura 6.9 mostrano l'andamento della distanza percorsa, il tempo in cui i robot non comunicano con la BS, la percentuale di area esplorata e conosciuta dalla BS ed il tempo di replan sui 20 minuti di intervallo di tempo con una squadra di 6 robot all'interno dell'ambiente *open*, rispettivamente, considerando e non considerando il tempo di replan. Possiamo notare che, per quanto riguarda i grafici in Figura 6.8, i valori di area conosciuta dalla BS nei primi 160 secondi di esplorazione con la strategia *Stump* sono molto simili a quelli ottenuti con la strategia *Utility*. Lo stesso vale per il tempo di disconnessione. Invece, i valori di distanza percorsa dai robot, nello stesso intervallo di tempo, sono minori per la strategia *Stump*. Per quanto riguarda i grafici in Figura 6.9, possiamo notare che la strategia *Stump* raggiunge valori finali molto simili a quelli di *Utility* in termini di area conosciuta dalla BS e di distanza percorsa, ma con un tempo di disconnessione minore. Per i primi 150 secondi, inoltre, la strategia *Stump* ha valori di area conosciuta maggiori rispetto a quelli della strategia U*tility* ed i robot hanno percorso una distanza minore.



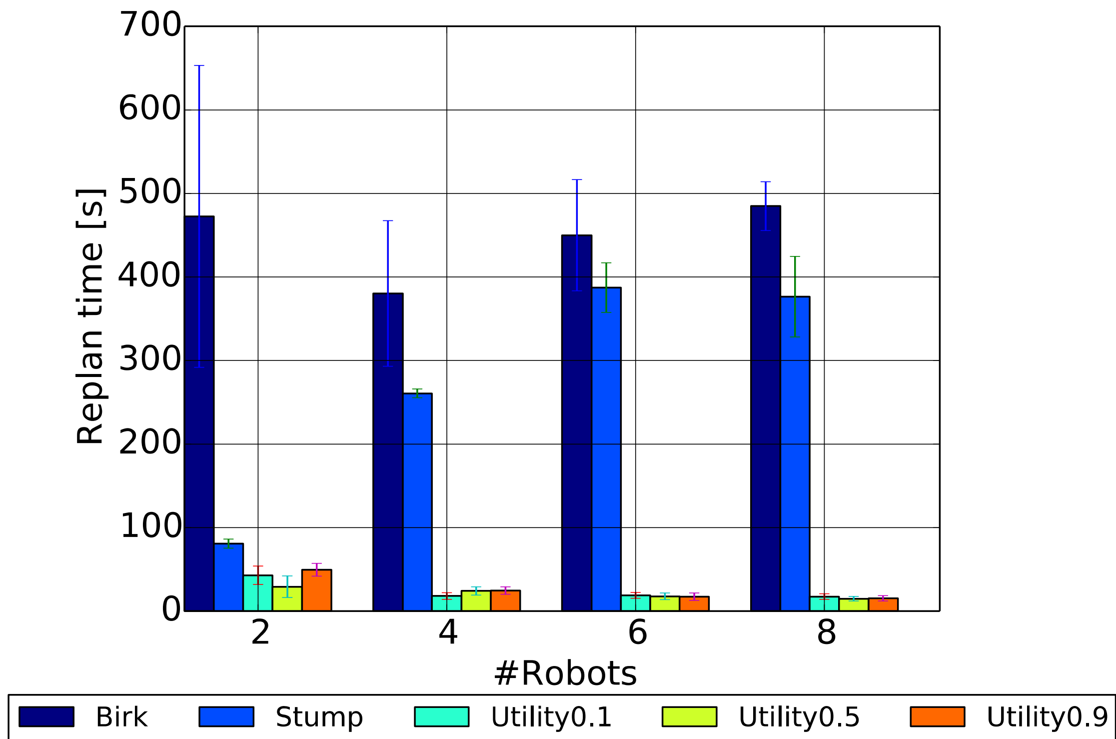
(a) Distanza percorsa



(b) Tempo di disconnessione

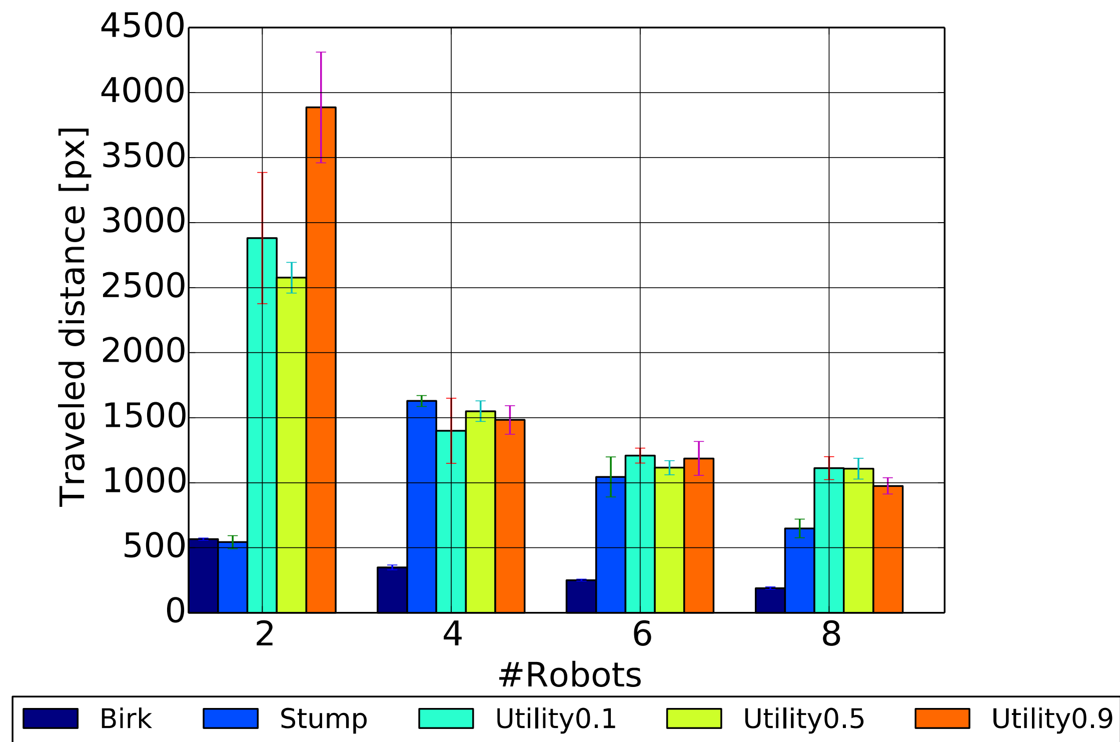


(c) Area esplorata

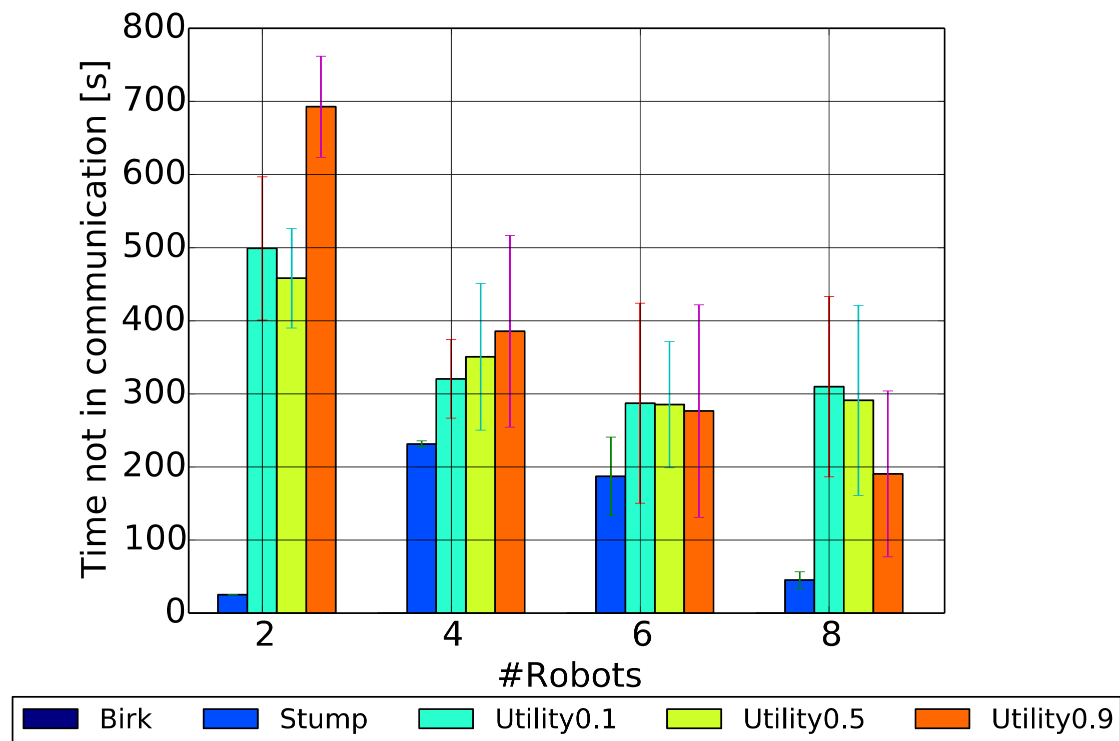


(d) Tempo di replan

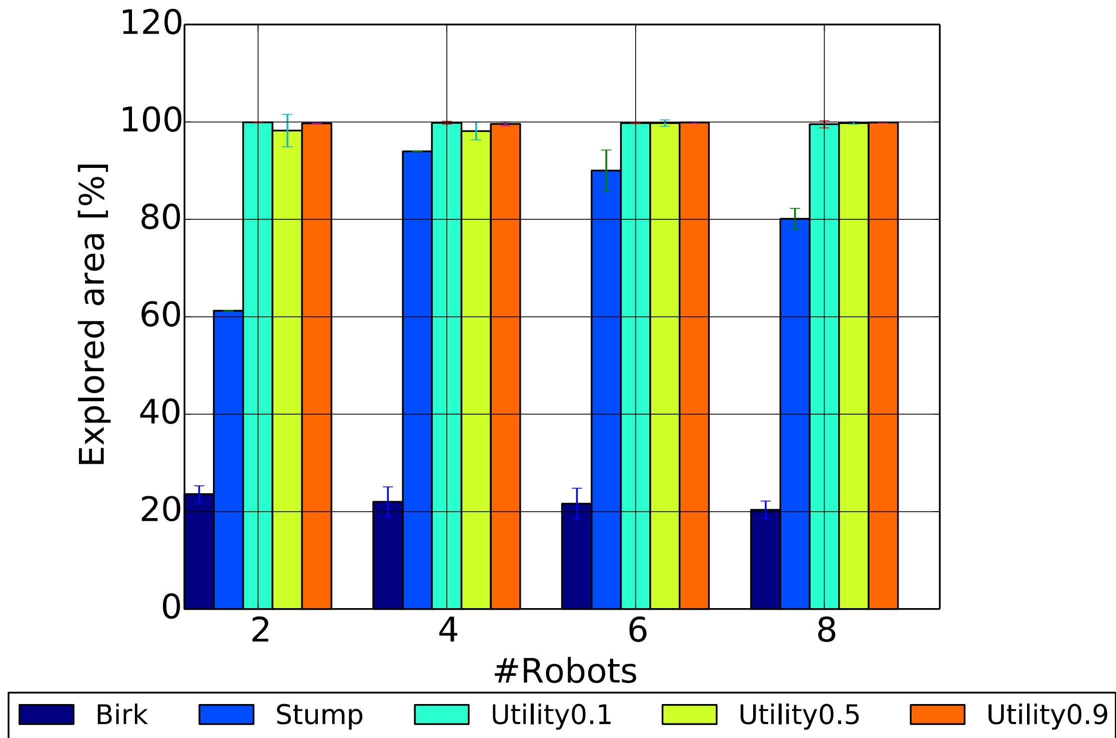
Figura 6.6: risultati (media e deviazione standard) per l'ambiente open, dopo 20 minuti di esplorazione (considerando il tempo di replan).



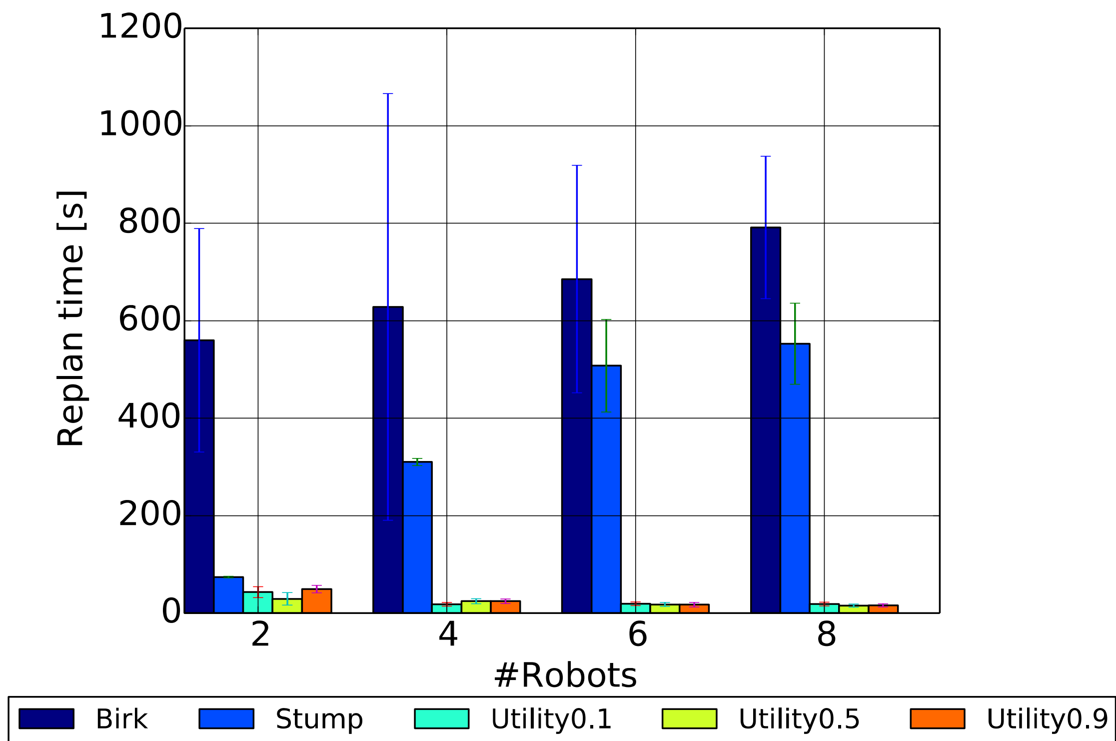
(a) Distanza percorsa



(b) Tempo di disconnessione

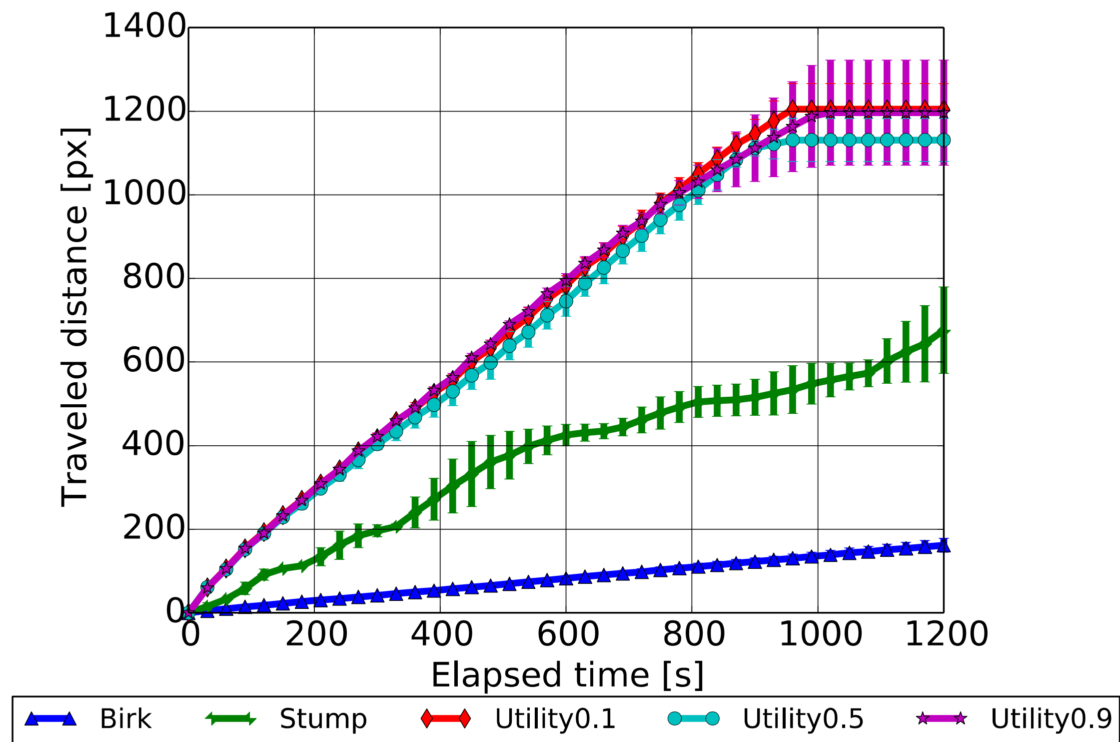


(c) Area esplorata

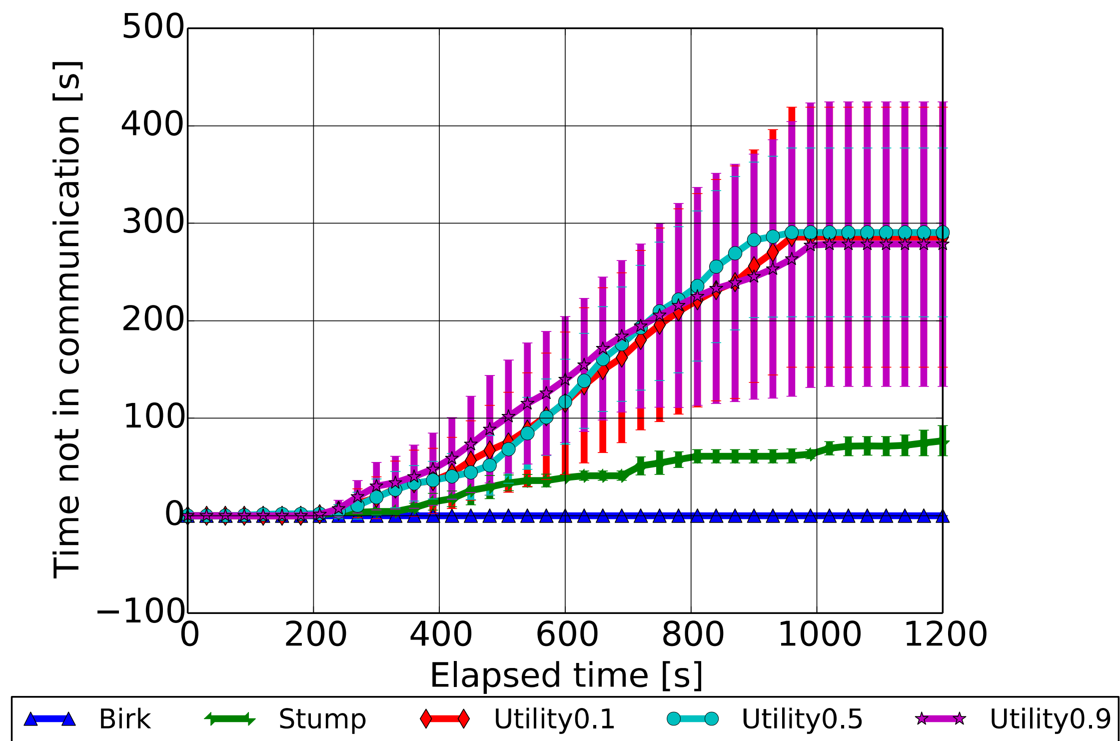


(d) Tempo di replan

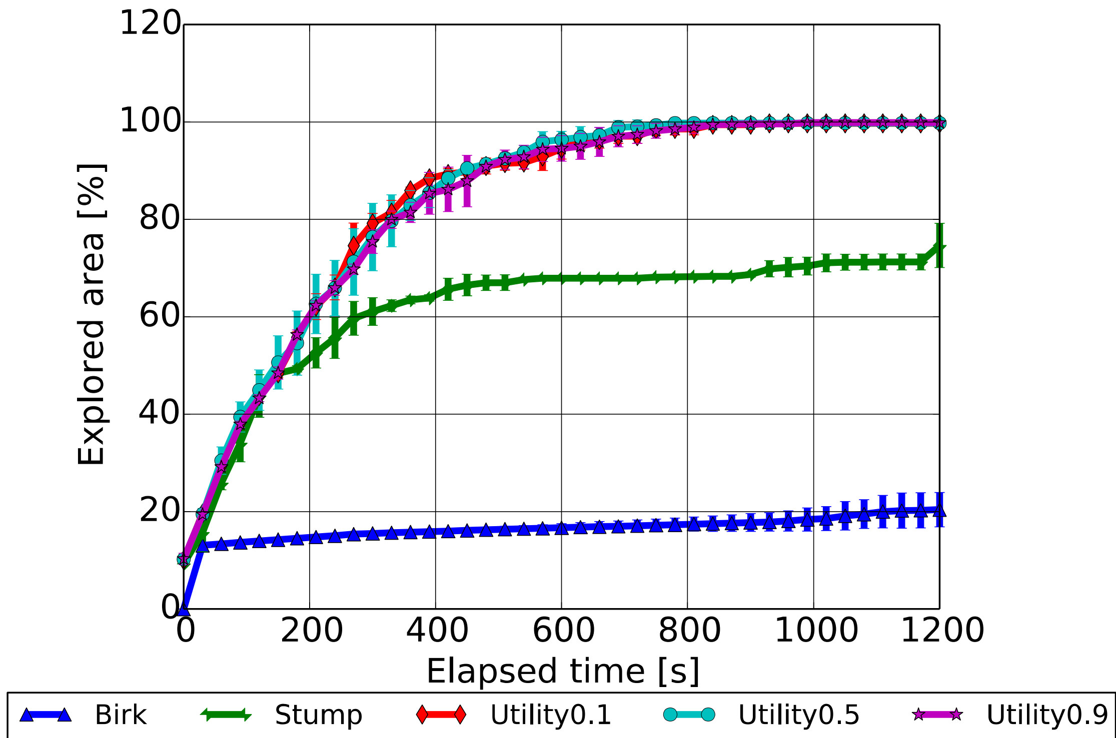
Figura 6.7: risultati (media e deviazione standard) per l'ambiente open, dopo 20 minuti di esplorazione (non considerando il tempo di replan).



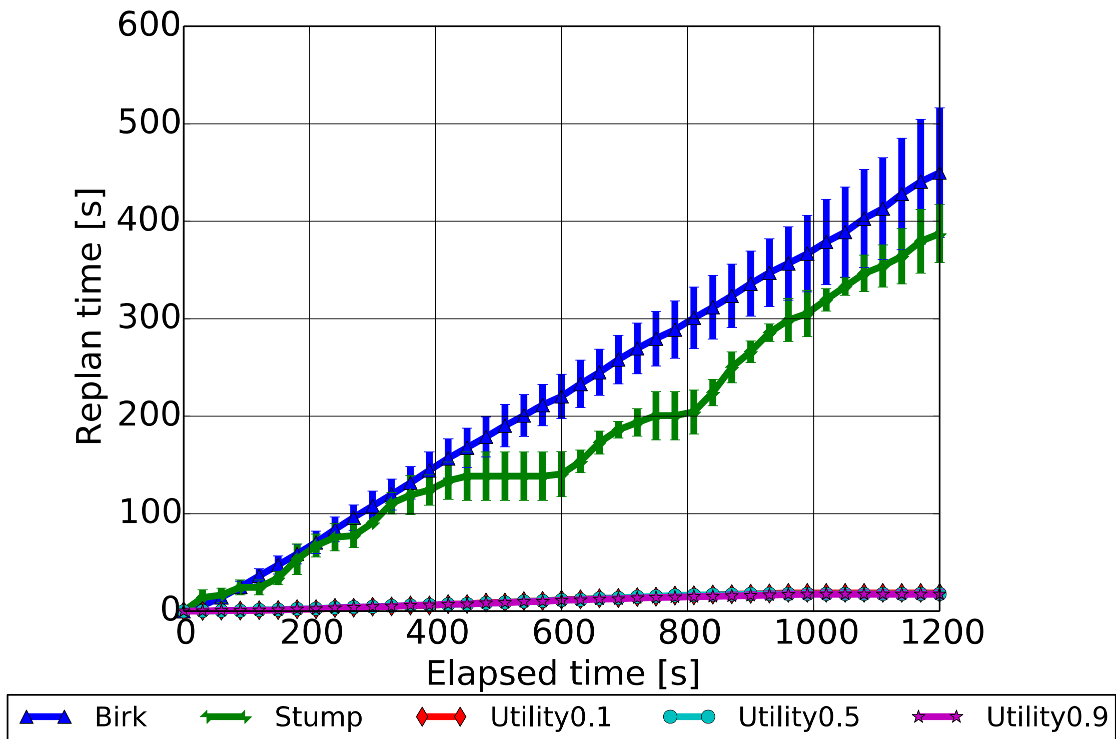
(a) Distanza percorsa



(b) Tempo di disconnessione

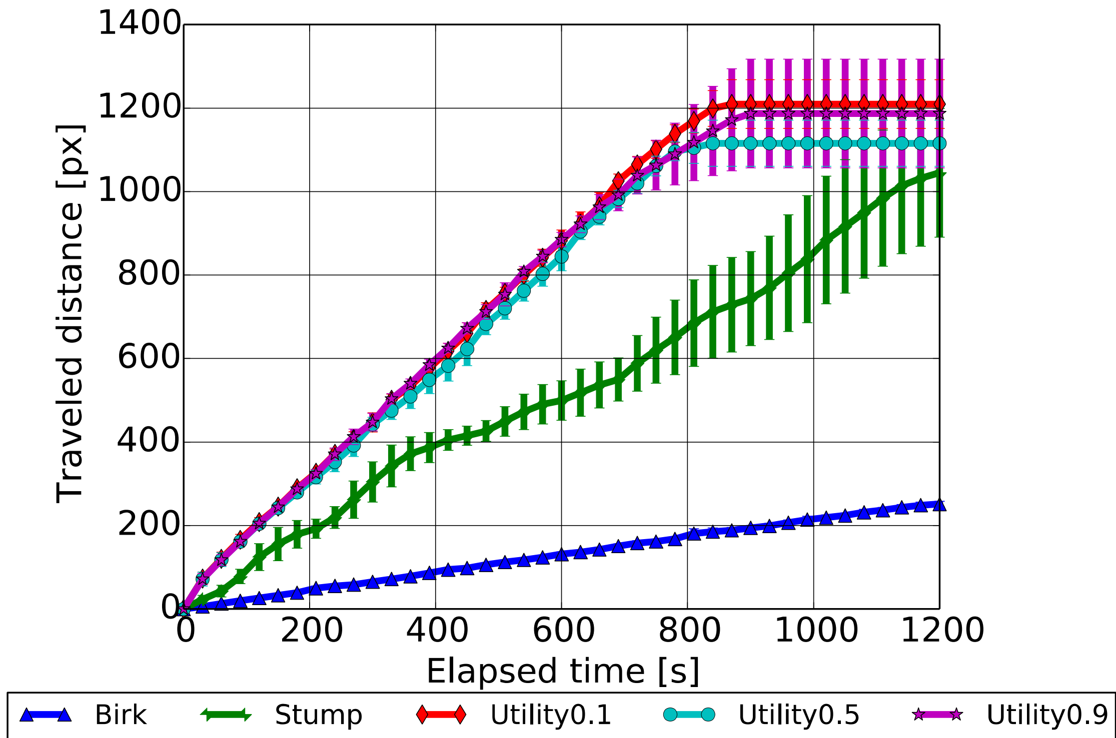


(c) Area esplorata

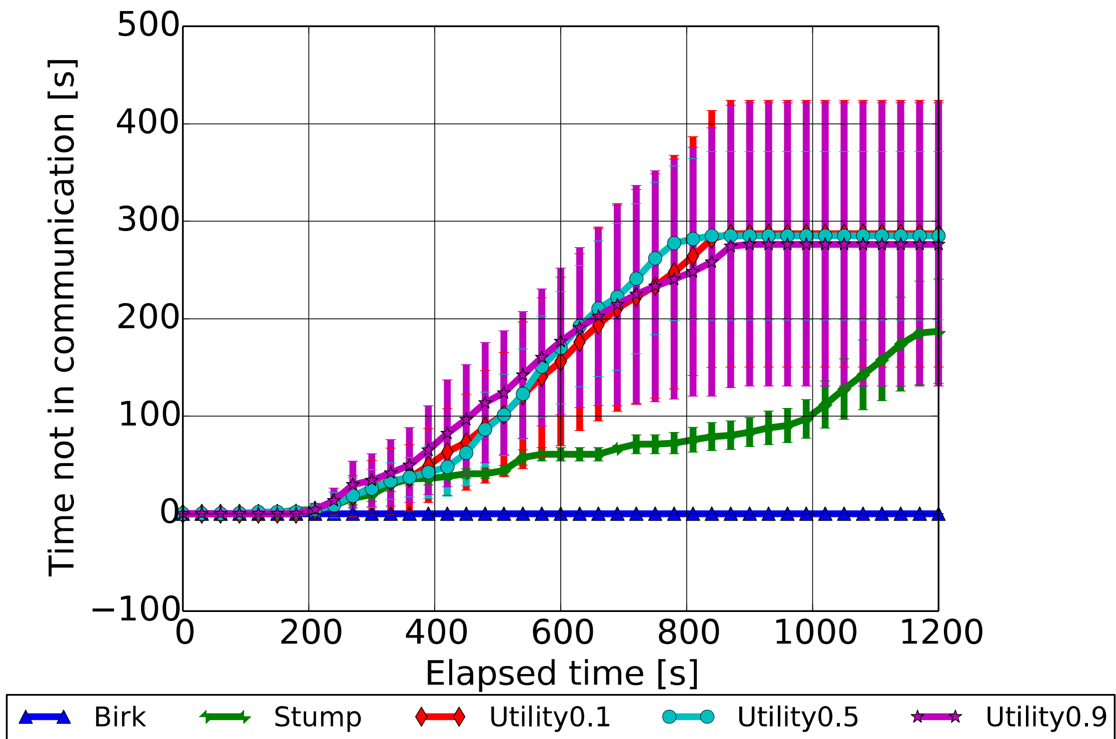


(d) Tempo di replan

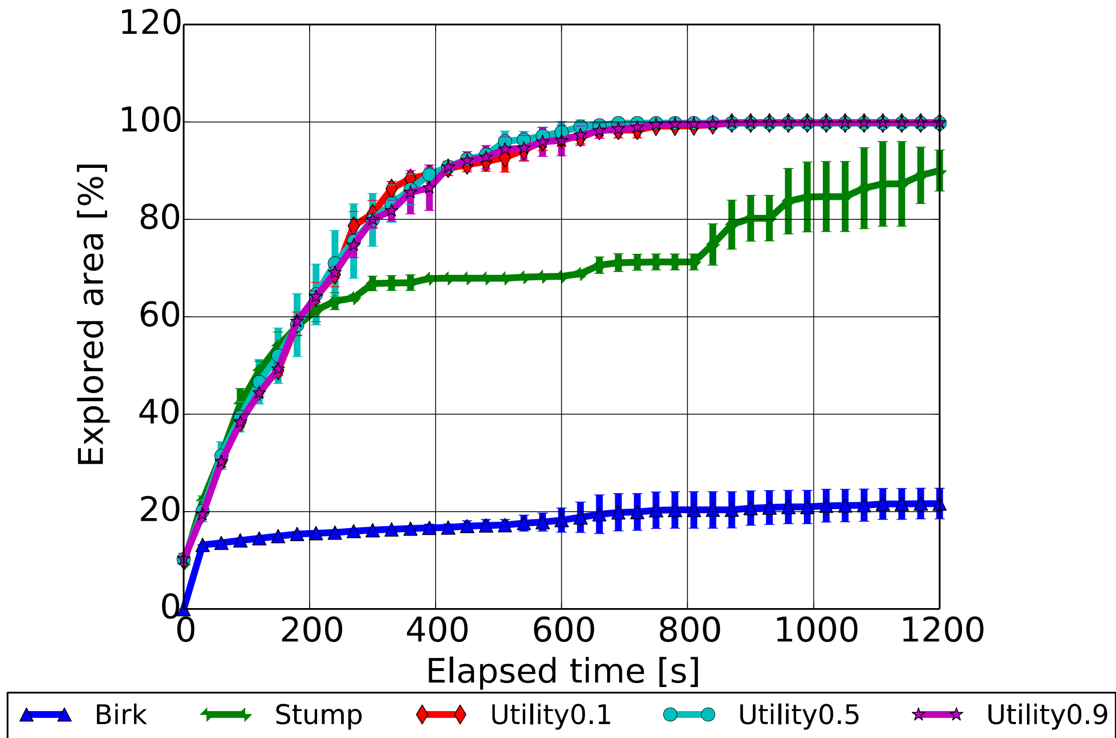
Figura 6.8: risultati (media e deviazione standard) per l'ambiente open, su 20 minuti di esplorazione, con 6 robot (considerando il tempo di replan).



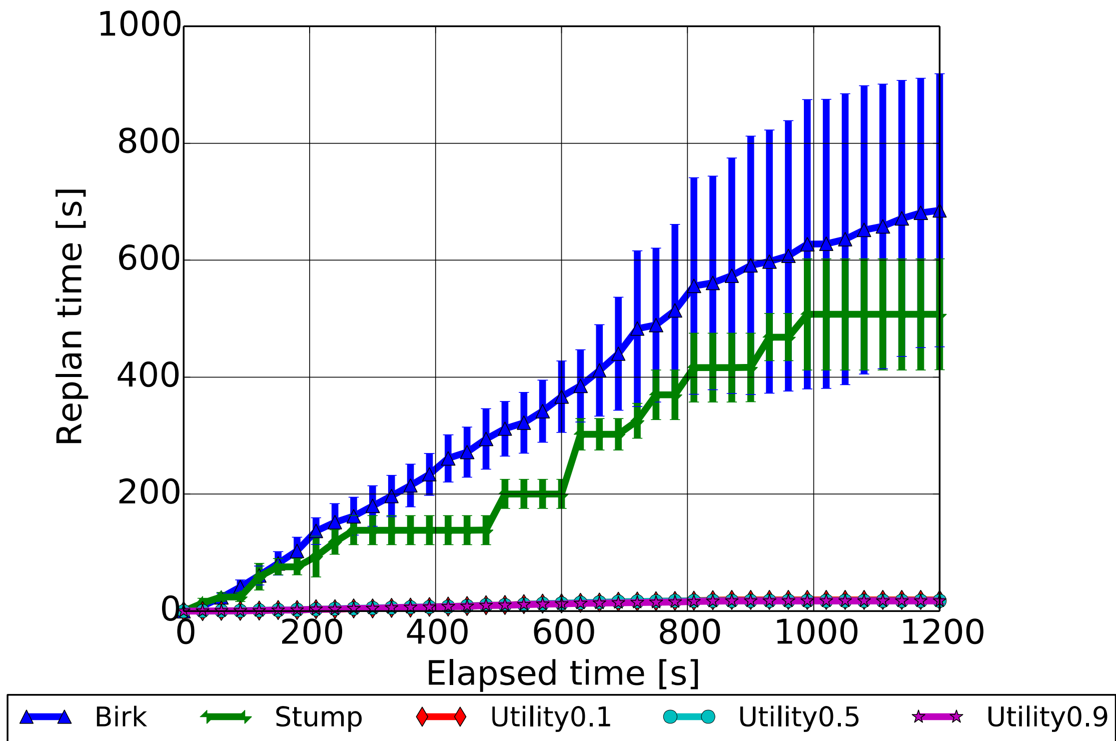
(a) Distanza percorsa



(b) Tempo di disconnessione



(c) Area esplorata



(d) Tempo di replan

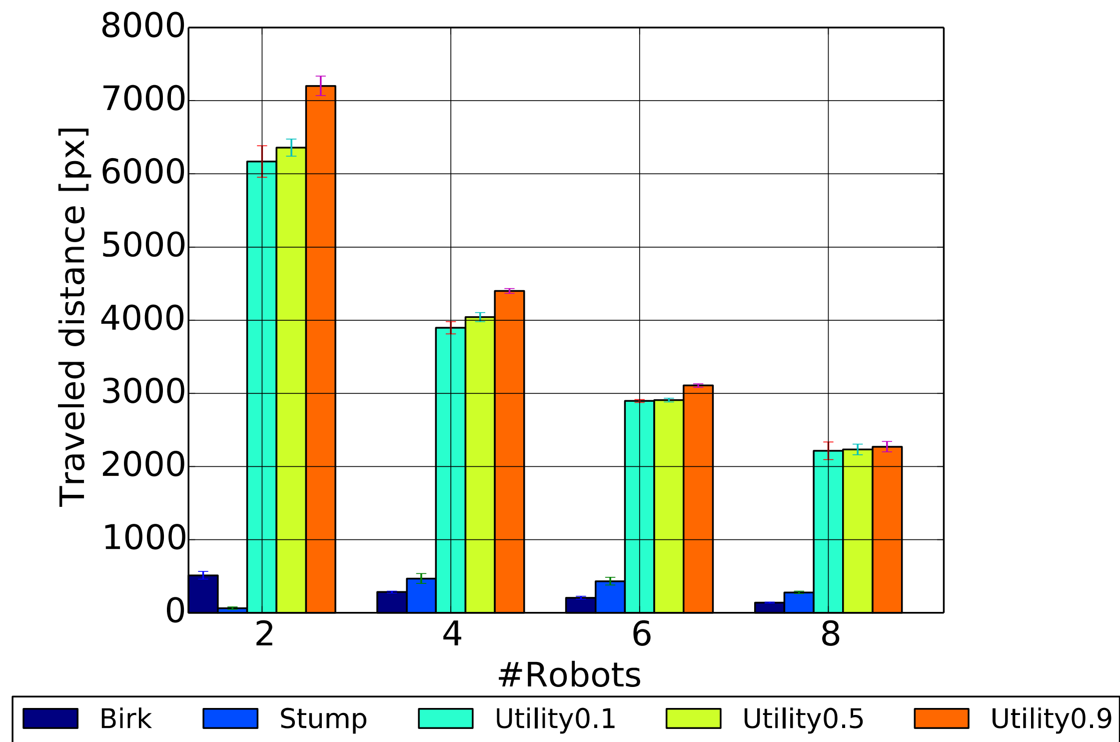
Figura 6.9: risultati (media e deviazione standard) per l'ambiente open, su 20 minuti di esplorazione, con 6 robot (non considerando il tempo di replan).

6.2.3 Maze

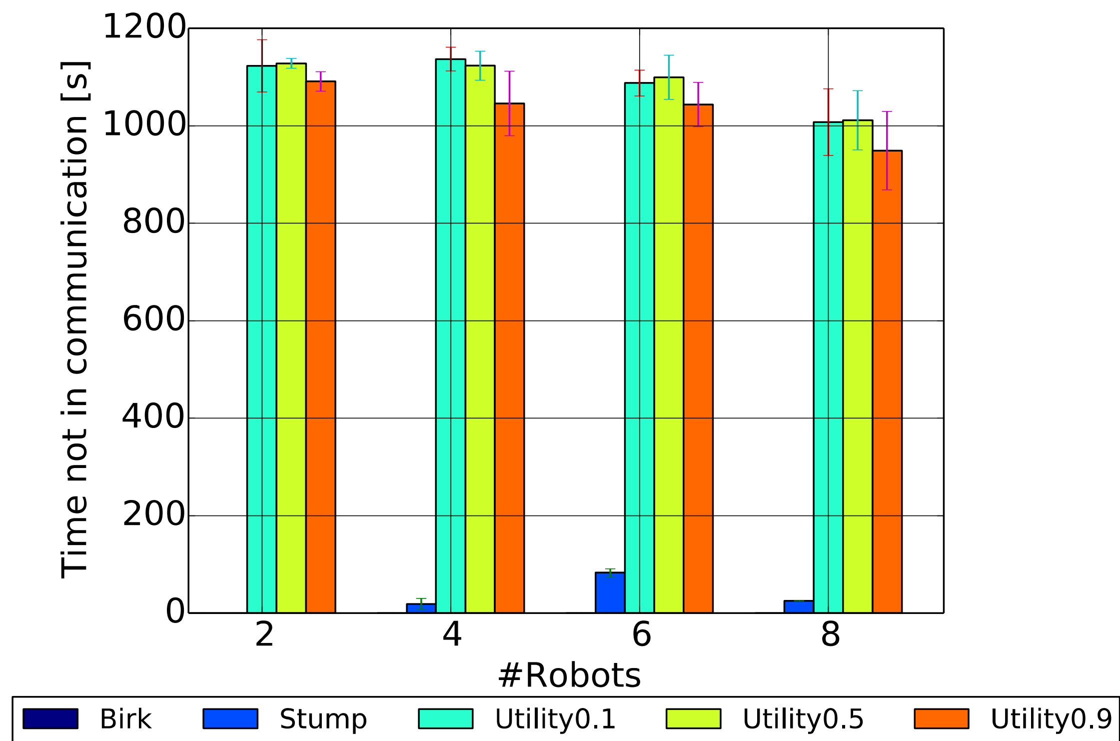
La Figura 6.10 e la Figura 6.11 mostrano i grafici dei dati aggregati relativi a 20 minuti di esplorazione nell'ambiente *maze*, rispettivamente, considerando e non considerando il tempo di replan.

Confrontando le prestazioni delle diverse strategie, possono essere fatte osservazioni simili a quelle dell'ambiente *office*. Una differenza può essere osservata per la strategia *Utility* e r = 0.9 nel caso di 2 e 4 robot, dove l'area esplorata e conosciuta dalla BS è minore rispetto a quella ottenuta con valori di r differenti. La ragione per cui questo accade è che *maze* ha una struttura più complessa rispetto agli altri due ambienti e, quindi, un robot impiega molto tempo a spostarsi da una locazione all'altra. Con alti valori di r i robot devono tornare più frequentemente alla BS e spendono più tempo ad attraversare aree dell'ambiente che hanno già visitato. Per quanto riguarda la strategia *Utility* con valore di r = 0.1, la struttura complessa porta ad esplorazioni in cui l'area conosciuta dalla BS è al di sotto del 20% (4 robot). Questo è dovuto al fatto che i robot non riescono a tornare alla BS entro il limite di 20 minuti per consegnare l'informazione conosciuta. Osserviamo che aumentare il numero di robot porta ad un peggioramento delle prestazioni in termini di area esplorata per la strategia *Utility*. Questo può essere causato, ancora una volta, dalla struttura dell'ambiente *maze*, per cui i robot devono evitare collisioni tra loro. Inoltre, in questo caso, nessuna strategia riesce ad esplorare il 100% dell'ambiente. Sempre a causa della complessità dell'ambiente, la strategia *Stump* con 2 e 4 robot ha terminato l'esplorazione prima dei 20 minuti, in quanto non sono state trovate delle configurazioni connesse.

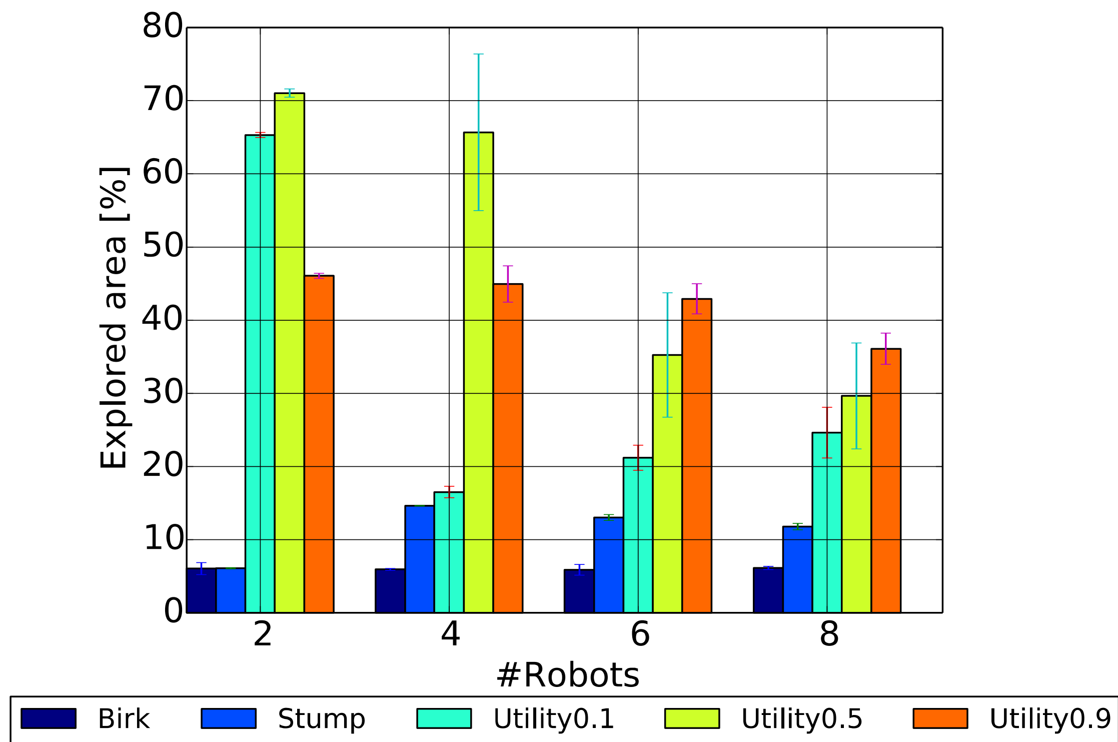
La Figura 6.12 e la Figura 6.13 mostrano l'andamento della distanza percorsa, il tempo in cui i robot non comunicano con la BS, la percentuale di area esplorata e conosciuta dalla BS e il tempo di replan sui 20 minuti di intervallo di tempo con una squadra di 6 robot all'interno dell'ambiente *maze*, rispettivamente, considerando e non considerando il tempo di replan. Nel grafico in Figura 6.13 è interessante osservare che la strategia *Stump* con 6 robot raggiunge una percentuale di area conosciuta dalla BS molto simile a quella della strategia *Utility0.1*, ma con un tempo di disconnessione ed una distanza percorsa molto più bassi.



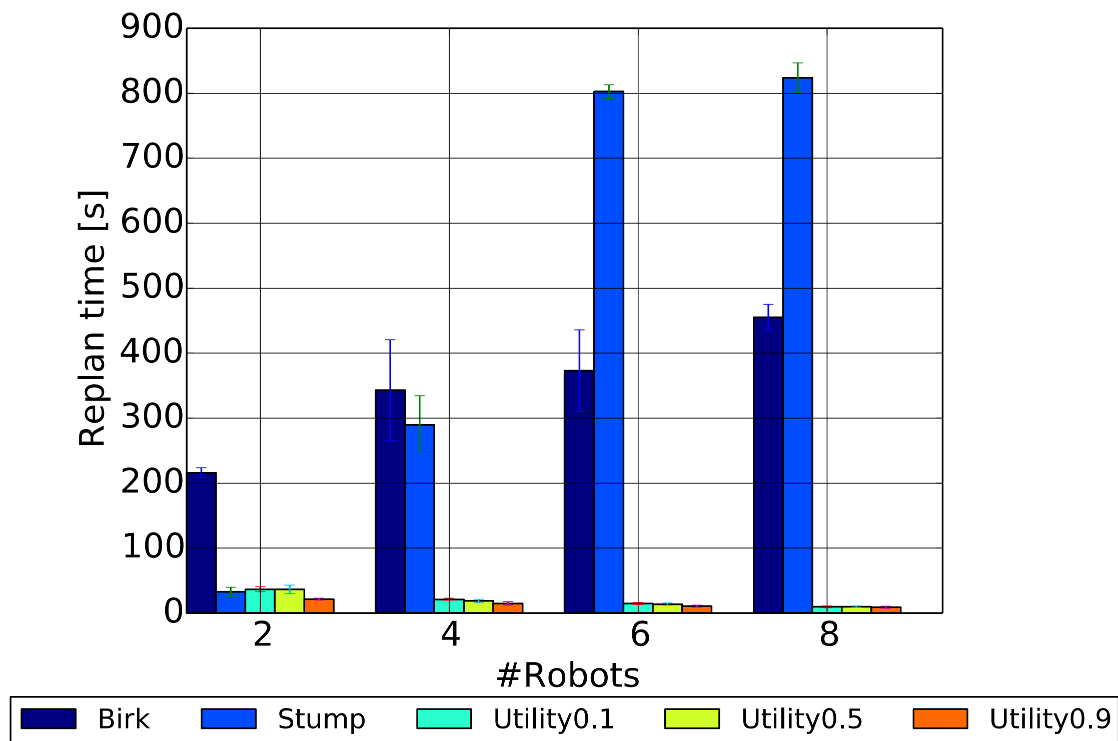
(a) Distanza percorsa



(b) Tempo di disconnessione

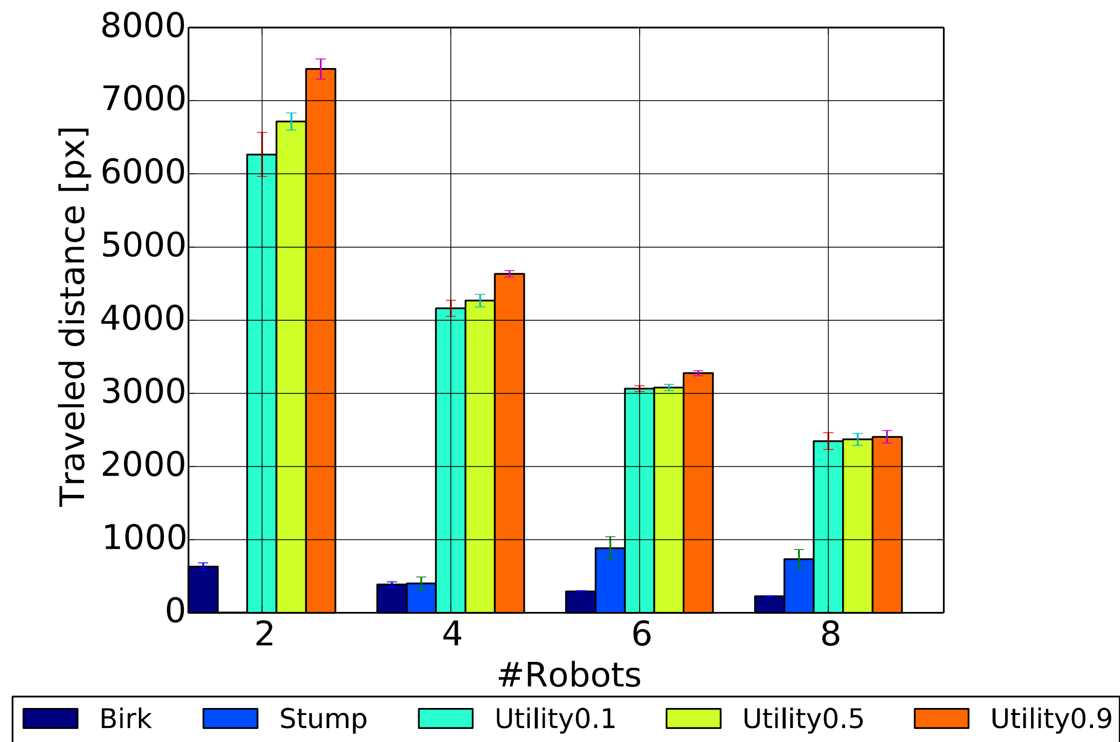


(c) Area esplorata

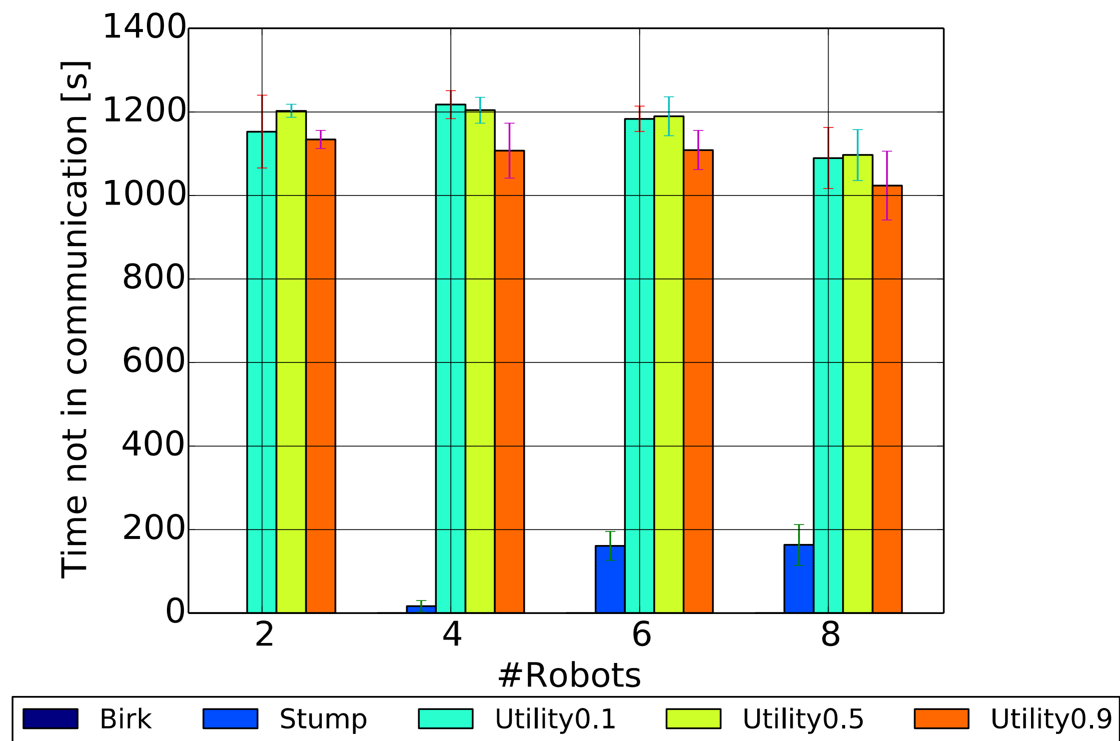


(d) Tempo di replan

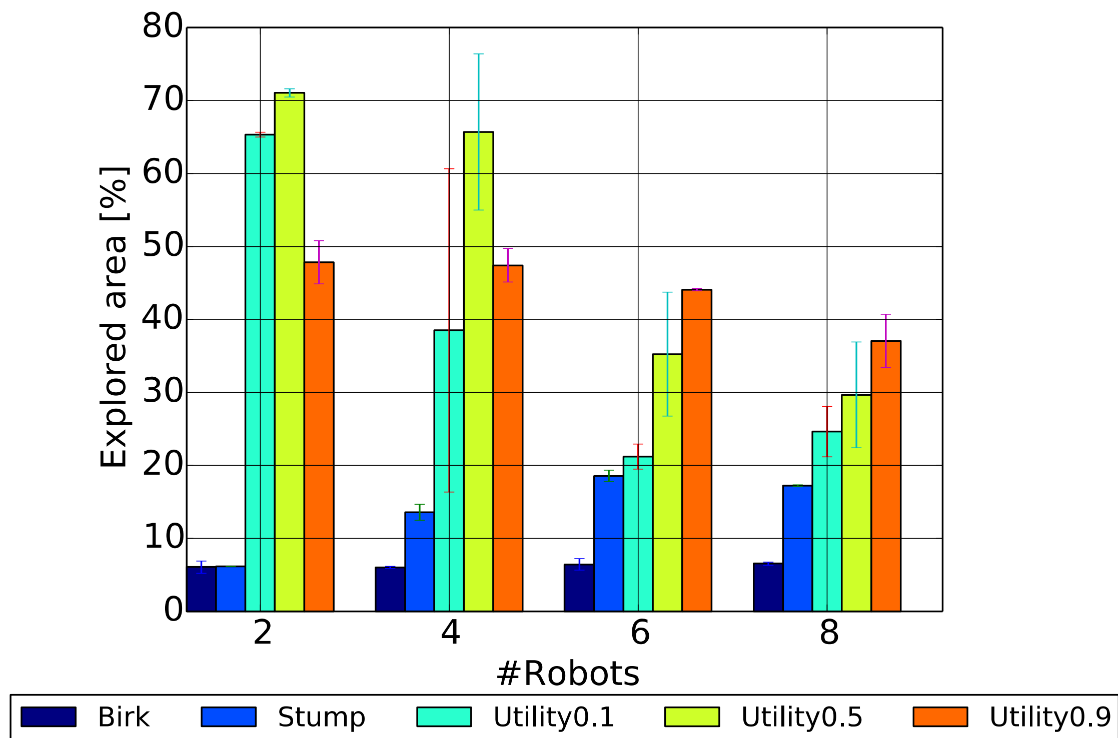
Figura 6.10: risultati (media e deviazione standard) per l'ambiente maze, dopo 20 minuti di esplorazione (considerando il tempo di replan)



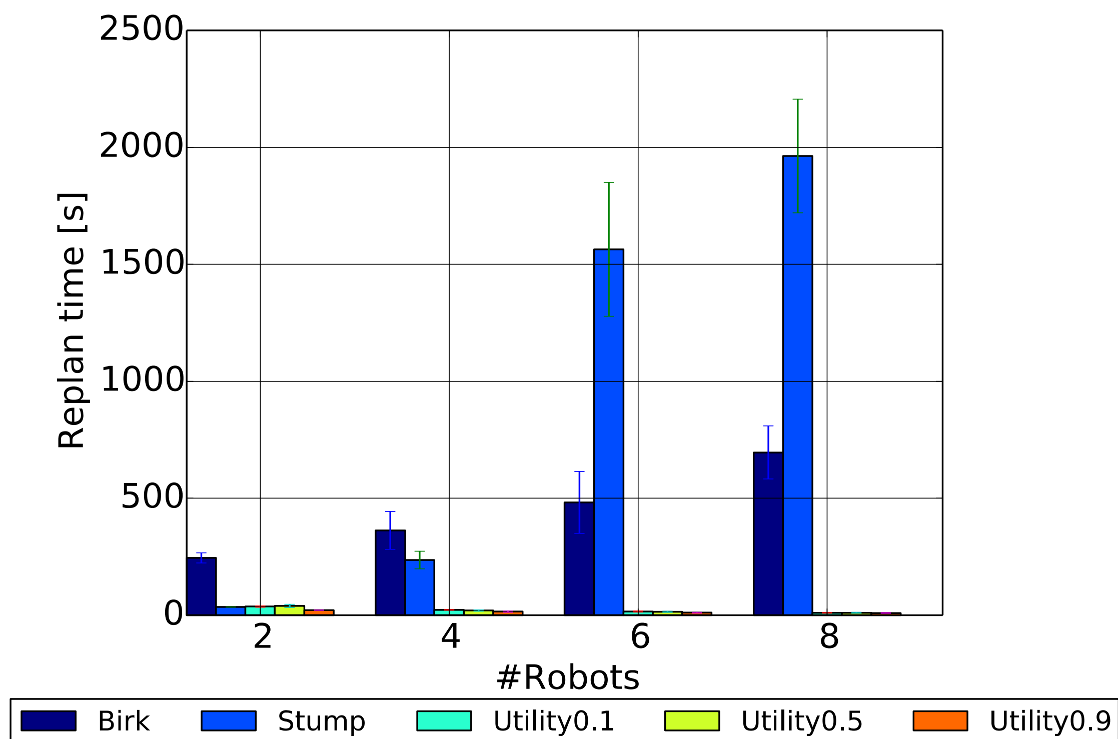
(a) Distanza percorsa



(b) Tempo di disconnessione

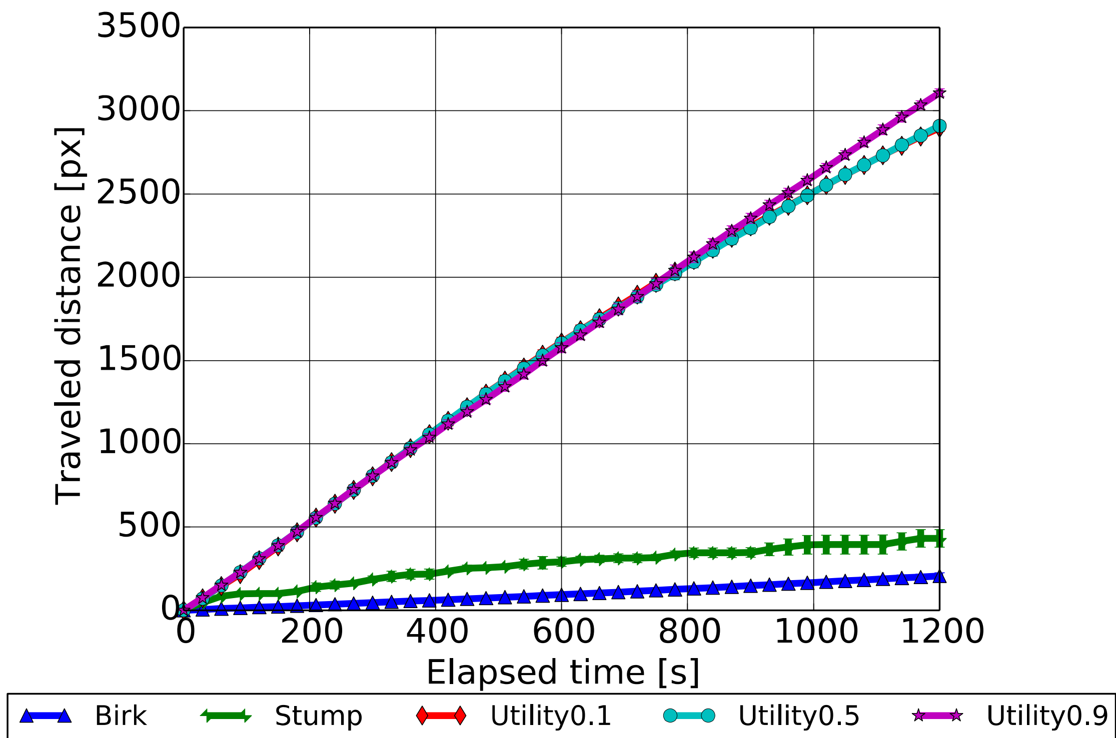


(c) Area esplorata

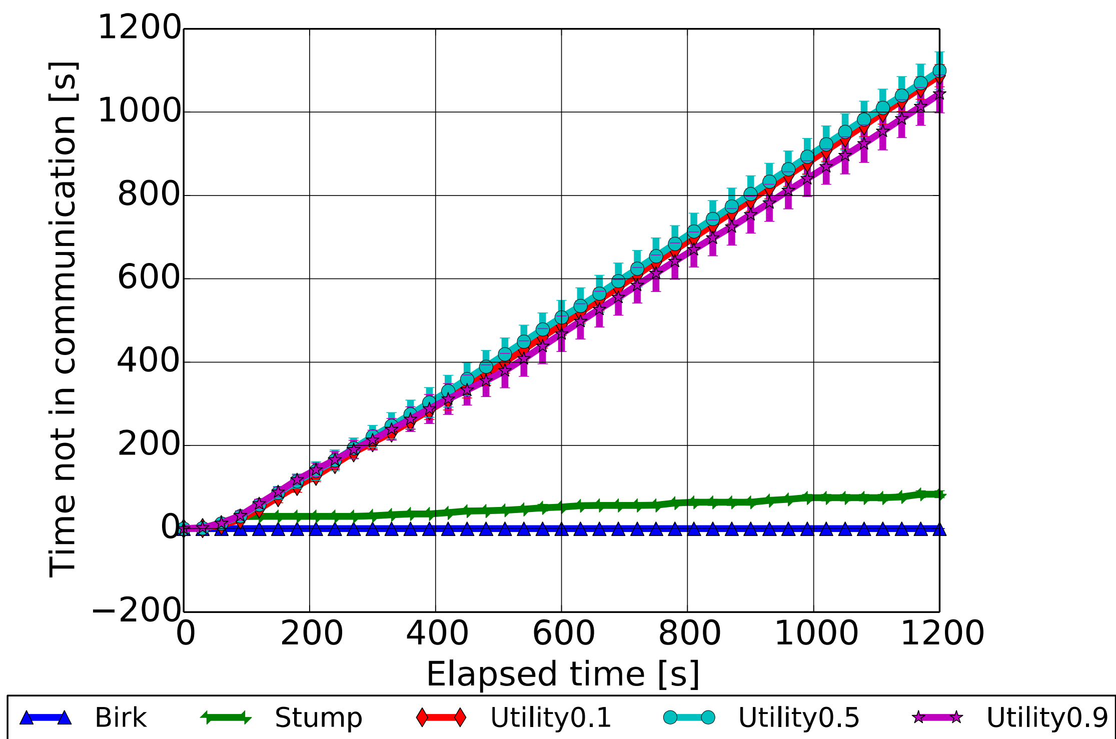


(d) Tempo di replan

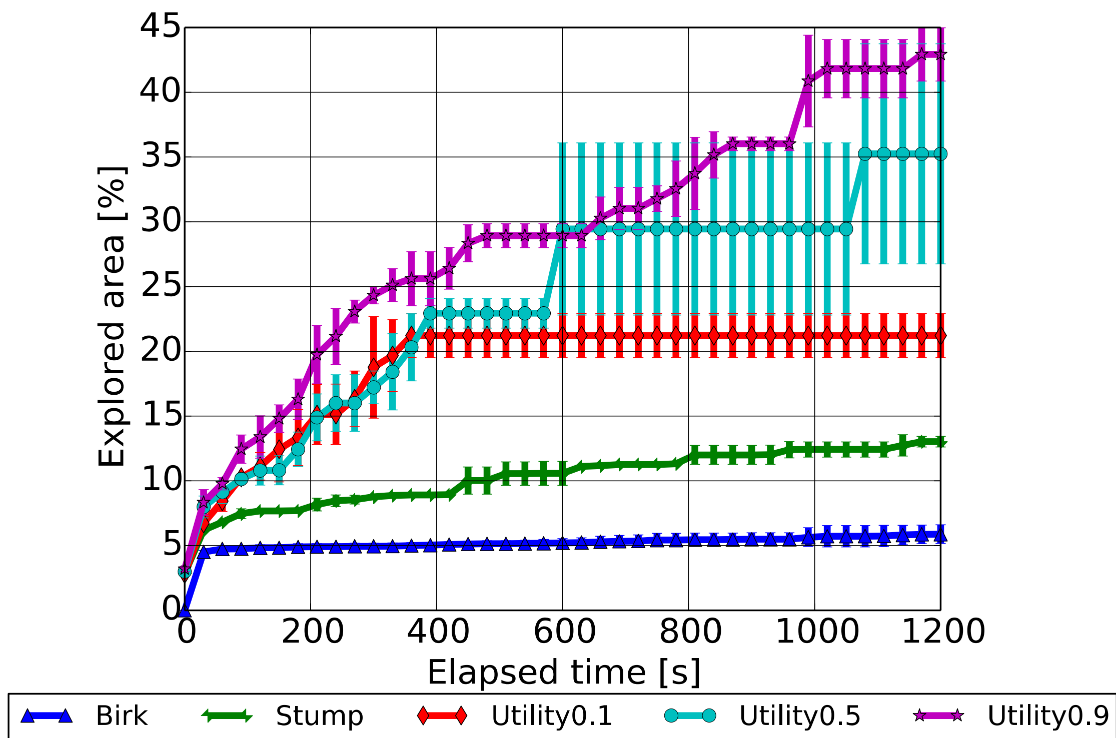
Figura 6.11: risultati (media e deviazione standard) per l'ambiente maze, dopo 20 minuti di esplorazione (non considerando il tempo di replan)



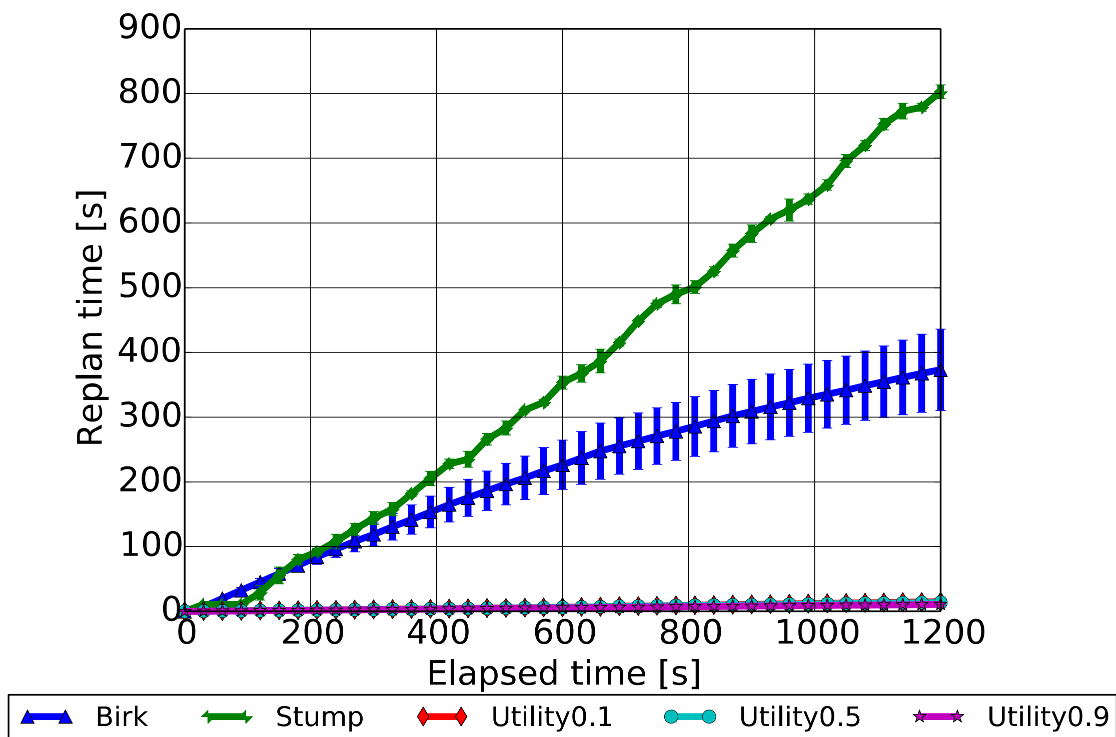
(a) Distanza percorsa



(b) Tempo di disconnessione

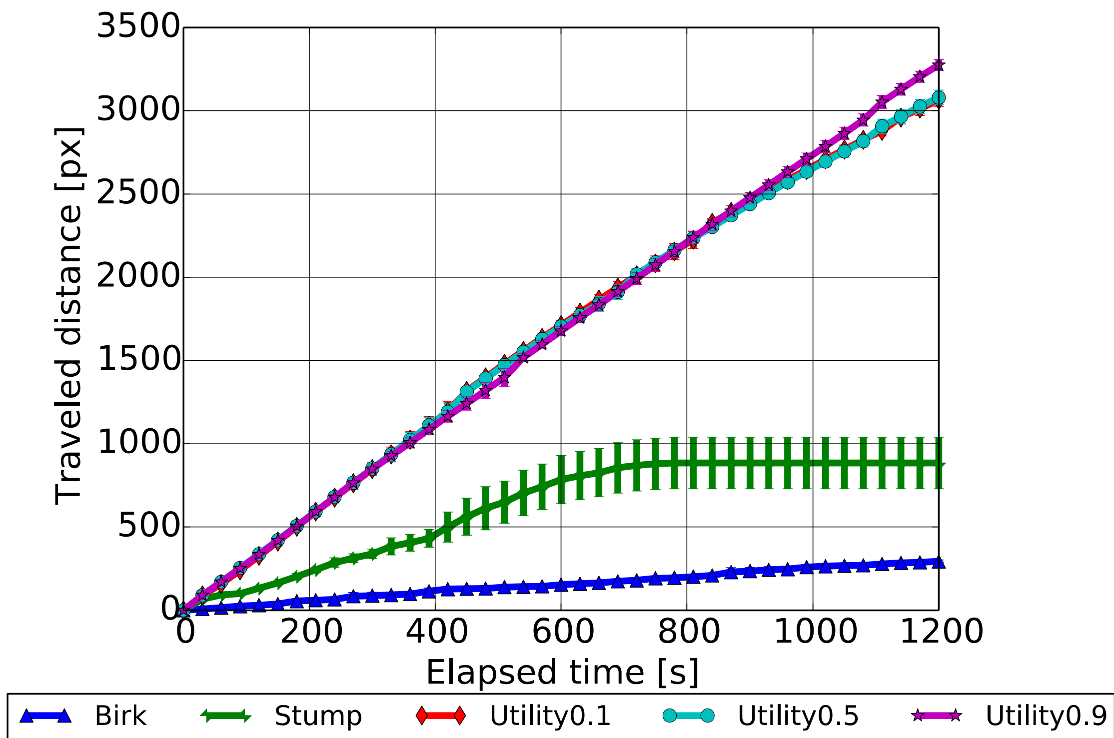


(c) Area esplorata

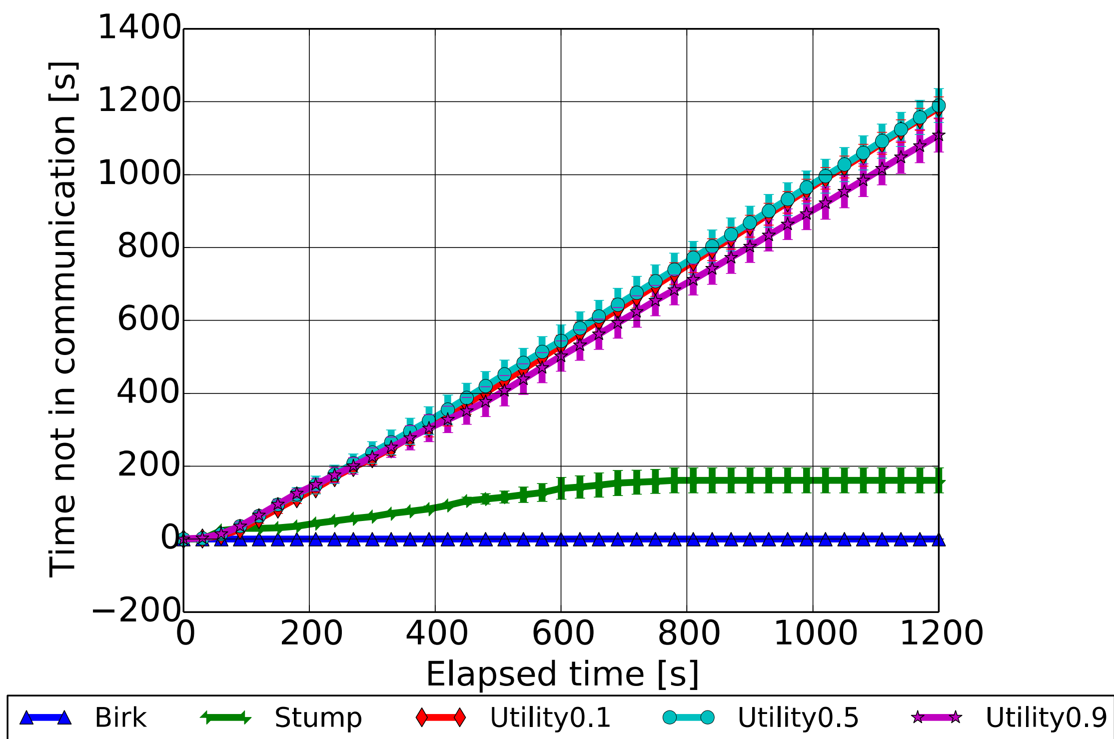


(d) Tempo di replan

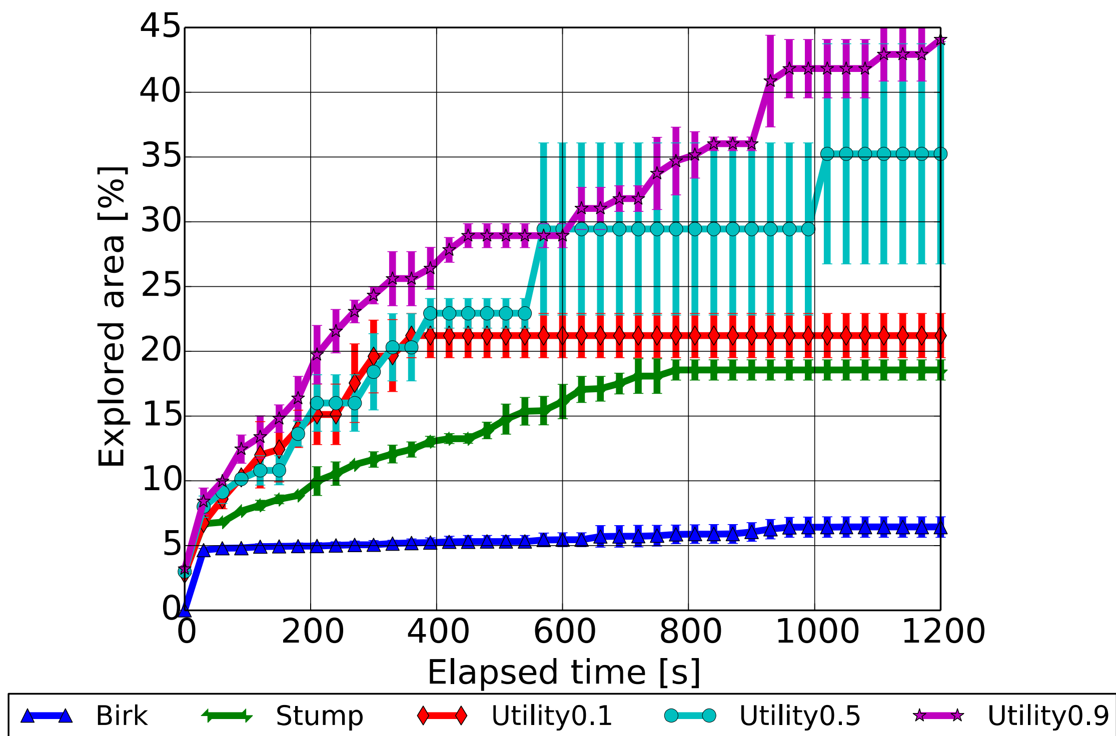
Figura 6.12: risultati (media e deviazione standard) per l'ambiente maze, su 20 minuti di esplorazione, con 6 robot (considerando il tempo di replan).



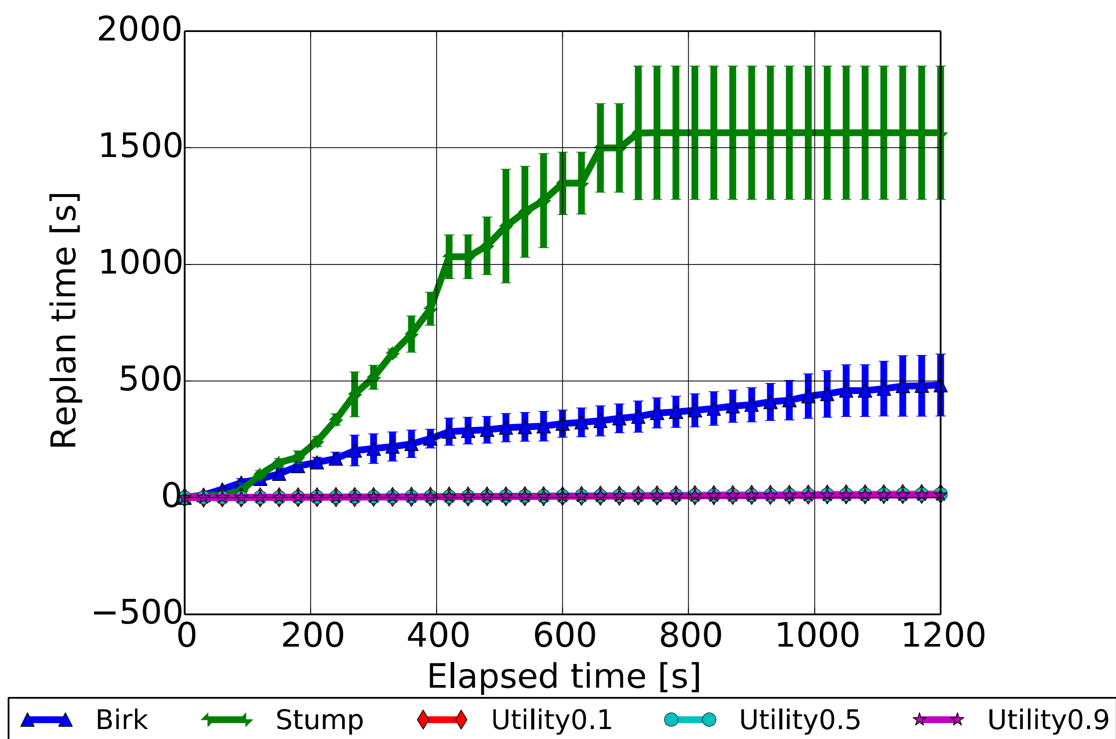
(a) Distanza percorsa



(b) Tempo di disconnessione



(c) Area esplorata



(d) Tempo di replan

Figura 6.13: risultati (media e deviazione standard) per l'ambiente maze, su 20 minuti di esplorazione, con 6 robot (non considerando il tempo di replan).

6.2.4 Sommario dei risultati

Riassumiamo di seguito le considerazioni più importanti che sono emerse dall'attività di simulazione:

- più è rigido il vincolo di comunicazione, minori sono, a parità di tempo, l'area esplorata e consegnata alla BS ed il tempo di disconnessione dei robot dalla BS;
- osservando la Figura 6.14, notiamo che, utilizzando strategie che adottano vincoli di comunicazione rigidi, la BS, al termine dell'esplorazione, possiede tutta la conoscenza che i robot hanno dell'ambiente, mentre, utilizzando strategie che adottano vincoli di comunicazione morbidi, molte aree dell'ambiente esplorato dai robot non sono conosciute dalla BS al termine dell'esplorazione;
- minore è il numero di ostacoli all'interno dell'ambiente, minori sono il tempo di disconnessione e la distanza percorsa, e, inoltre, maggiore è l'area esplorata e conosciuta dalla BS;
- per quanto riguarda la strategia di esplorazione *Utility*, l'aumento delle dimensioni della squadra di robot provoca una diminuzione del tempo di disconnessione dalla BS (in quanto alcuni robot si comportano come dei relay e consegnano le informazioni alla BS più frequentemente) e una diminuzione dell'area esplorata e conosciuta dalla BS (in quanto i robot devono far fronte ad un maggior numero di collisioni tra loro);
- le strategie di esplorazione *Birk* e *Stump* hanno alti tempi di replan (in quanto sono di tipo centralizzato e sono caratterizzate da vincoli di comunicazione rigidi), mentre *Utility* ha bassi tempi di replan (in quanto è di tipo decentralizzato ed è caratterizzata da un vincolo di comunicazione morbido);
- la strategia *Birk* impiega molto tempo per il calcolo delle popolazioni e del loro valore di utilità;
- in ambienti in cui sono presenti molti ostacoli, la strategia *Stump* non scala bene. Infatti, è caratterizzata da alti tempi di replan per un numero elevato di robot;
- la strategia *Stump*, all'interno dell'ambiente *open* e senza considerare il tempo di replan, si comporta in modo simile, in termini di area esplorata, alla strategia *Utility* ed ha un tempo di disconnessione dalla BS minore;
- non è stato possibile portare a termine alcuna simulazione con la strategia *PLI* a causa dell'elevata occupazione di memoria sulla macchina.

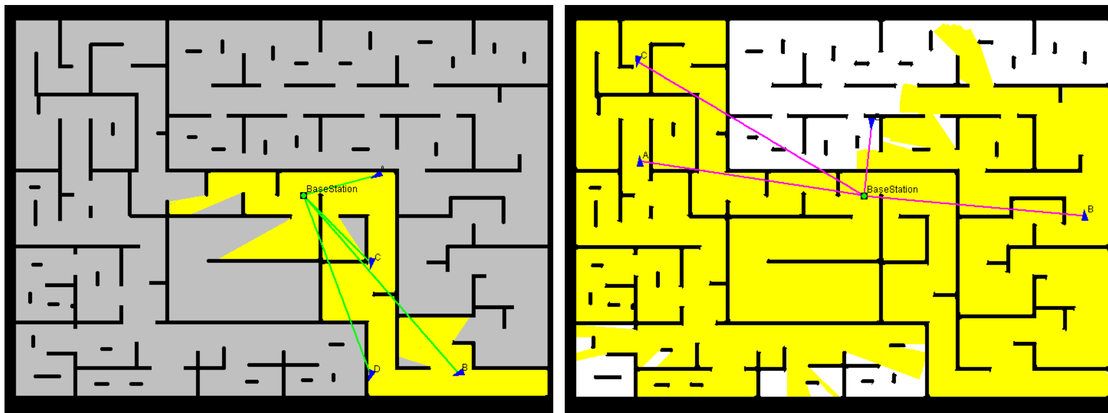


Figura 6.14: screenshot dell'ambiente maze alla fine dell'esplorazione. A sinistra è stata utilizzata la strategia *Stump*, mentre a destra è stata utilizzata la strategia *Utility*. L'area gialla è quella conosciuta dalla BS, mentre quella bianca è conosciuta solo dai robot.

Capitolo 7

Conclusioni e sviluppi futuri

L'esplorazione di un ambiente inizialmente sconosciuto è un compito fondamentale in molte applicazioni della robotica mobile autonoma, come quelle di ricerca e soccorso di vittime di ambienti disastrati. In letteratura, molti lavori propongono strategie di esplorazione che permettono ad una squadra di robot di scoprire l'ambiente in modo incrementale. Gli effetti di una comunicazione realistica tra i robot sono stati considerati da alcuni di questi lavori, che includono i vincoli di comunicazione all'interno della progettazione delle strategie di esplorazione. Tuttavia, il lavoro di confronto tra strategie di esplorazione, con vincoli di comunicazione differenti, non è ancora stato completamente portato a termine. In questa tesi, abbiamo presentato un confronto sperimentale quantitativo tra alcune strategie di esplorazione che considerano diversi vincoli di comunicazione.

Innanzitutto, abbiamo considerato un problema di esplorazione in cui i robot devono comunicare con una base station (BS) fissa e abbiamo fornito una possibile tassonomia per i vincoli di comunicazione che caratterizzano una strategia di esplorazione multirobot, dividendoli in vincoli di comunicazione rigidi e vincoli di comunicazione morbidi (Capitolo 3).

Successivamente, abbiamo presentato una formulazione per il problema di esplorazione con vincoli di comunicazione rigidi all'interno della Sezione 4.1 (formulazione PLI), che permette di trovare una soluzione ottima al problema di allocare i robot alle locazioni interessanti dell'ambiente mantenendoli connessi alla BS.

Dopodiché, abbiamo confrontato, in modo sperimentale, tre strategie che i robot

possono impiegare nella costruzione della mappa di un ambiente sconosciuto caratterizzate da vincoli di comunicazione differenti. La strategia di esplorazione *Birk* [29] impiega un vincolo di comunicazione rigido, che impone una connessione continua tra i robot e la BS. La strategia di esplorazione *Stump* [34] impiega un vincolo di comunicazione rigido, che impone la connessione tra i robot e la BS soltanto al raggiungimento delle posizioni obiettivo. La strategia di esplorazione *Utility* [32] impiega un vincolo di comunicazione morbido, che impone un ritorno periodico dei robot alla BS.

Gli esperimenti sono stati svolti utilizzando il simulatore MRESim [9], che include i metodi per la comunicazione, la navigazione ed il mapping. In questo modo, ci siamo potuti concentrare soltanto sull'implementazione delle strategie. L'analisi sperimentale ha portato a qualche intuizione sui punti di forza e di debolezza dei diversi approcci. In generale, la strategia *Utility* ha ottenuto alti valori di area esplorata e bassi valori di tempo di replan. Tuttavia, ha ottenuto alti valori di tempo di disconnessione dalla BS e di distanza percorsa. Le strategie *Birk* e *Stump* hanno ottenuto alti valori di tempo di replan e bassi valori di area esplorata, ma anche bassi valori di tempo di disconnessione dalla BS e di distanza percorsa. Quindi, con la nostra analisi, abbiamo confermato che, a parità di tempo, più è rigido il vincolo di comunicazione, minore è l'area esplorata e conosciuta dalla BS, ma è anche minore il tempo di disconnessione. Questo suggerisce che bisogna trovare un compromesso tra la connettività e la copertura dell'ambiente a seconda della specifica applicazione. Sono stati ottenuti risultanti interessanti per l'ambiente *open*, caratterizzato dalla presenza di grandi spazi e di pochi ostacoli, dove la strategia *Stump* (con 4 e 6 robot) si è comportata in modo simile alla strategia *Utility*, in termini di area conosciuta dalla BS e di distanza percorsa, ottenendo, inoltre, valori minori in termini di tempo di disconnessione dalla BS. L'ambiente *maze*, al contrario di *open*, è un ambiente caratterizzato dalla presenza di molti ostacoli che impediscono la comunicazione tra i robot e la BS. A causa di questa struttura complessa, i robot, per potere consegnare la nuova informazione alla BS, impiegano molto tempo attraversando aree di ambiente che hanno già visitato. Facendo un confronto tra l'ambiente *maze* e l'ambiente *open* si può notare, a parità di dimensione della squadra di robot, una diminuzione della quantità di area esplorata e conosciuta dalla BS per tutte le strategie di esplorazione testate.

I risultati presentati in questa tesi costituiscono solo un primo passo verso la valutazione di strategie di esplorazione diverse caratterizzate da vincoli di comunicazione differenti. Per esempio, ulteriori ambienti e strategie hanno bisogno di essere valutati per trarre conclusioni più solide, in modo da poter essere utili per la progettazione di migliori strategie di esplorazione.

In aggiunta, l'implementazione delle strategie presentate può essere ottimizzata. Per esempio, il metodo *Stump*, ogni volta che viene chiamato, ricalcola completamente la matrice che rappresenta il grafo di comunicazione invece di, semplicemente, aggiornarla con le nuove locazioni scoperte.

Inoltre, alcuni aspetti importanti, inclusi gli effetti di incertezza e di estensione agli ambienti outdoor, non sono stati considerati in questo lavoro e sarebbe interessante conseguire ulteriori indagini.

Bibliografia

[1] Amigoni, F.: Experimental evaluation of some exploration strategies for mobile robots. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp. 2818-2823 (2008)

[2] Amigoni, F., Basilico, N., Quattrini Li, A.: How Much Worth Is Coordination of Mobile Robots for Exploration in Search and Rescue. In: RoboCup 2012: Robot Soccer World Cup XVI, pp. 106-117 (2012)

[3] Arkin, R., Diaz, J.: Line-of-sight constrained exploration for reactive multiagent robotic teams. In: Proceedings of Advanced Motion Control, pp. 455-461 (2002)

[4] Basilico, N., and Amigoni, F.: Exploration strategies based on multi-criteria decision making for searching environments in rescue operations. Autonomous Robots 31(4), pp. 401–417 (2011)

[5] Burgard, W., Fox, D., Moors, M., Simmons, R., Thrun, S.: Collaborative multirobot exploration. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp. 476–481 (2000)

[6] Burgard, W., Moors, M., Schneider, F.: Coordinated multi-robot exploration. In: IEEE Transactions of Robotics 21(3), pp. 376–378 (2005)

[7] Calisi, D., Farinelli, A., Iocchi, L., Nardi, D.: Multi-objective exploration and search for autonomous rescue robots. Journal of Field Robotics 24(8-9), pp. 763–777 (2007)

[8] Cipolleschi, R., Giusto, M., Quattrini Li, A., Amigoni, F.: Semantically-informed coordinated multirobot exploration of relevant areas in search and rescue settings. In: Proceedings of the European Conference on Mobile Robots, pp. 216-221 (2013)

[9] de Hoog, J., Cameron, S., Visser, A.: Role-based autonomous multi-robot exploration. In: Proceedings of the International Conference on Advanced Cognitive Technologies and Applications, pp. 482-487 (2009)

[10] de Hoog, J., Cameron, S., Visser, A.: Autonomous multi-robot exploration in communication-limited environments. In: Proceedings of the Conference Towards Autonomous Robotic Systems, pp. 68-75 (2010)

[11] Fox, D., Ko, J., Konolige, K., Limketkai, B., Schulz, D., Stewart, B.: Distributed multirobot exploration and mapping. In: Proceedings of the IEEE 94(7), pp. 1325–1339 (2006)

[12] Gini, G., Caglioti, V.: Robotica. Zanichelli (2003)

[13] Gonzáles-Baños, H., Latombe, J.C.: Navigation strategies for exploring indoor environments. In: International Journal of Robotics Research 21(10-11), pp. 829–848 (2002)

[14] Hawley, J., Butler, Z.: Hierarchical distributed task allocation for multi-robot exploration. In: Proceedings of Distributed Autonomous Robotic Systems, pp. 445–458 (2010)

[15] Hollinger, G., Singh, S.: Multirobot coordination with periodic connectivity: Theory and experiments. In: IEEE Transactions of Robotics 28(4), pp. 967-973 (2012)

[16] Howard, A., Mataric, M., Sukhatme, G.: An incremental self-deployment algorithm for mobile sensor networks. In: Autonomous Robots 13(2), pp. 113-126 (2002)

[17] Howard, A., Roy, N.: The robotics data set repository (Radish). http://radish.sourceforge.net/ (2003)

[18] Jensen, E., Nunes, E., Gini, M.: Communication-restricted exploration for robot teams. In: Multiagent Interaction without Prior Coordination Workshop at Advancement of Artificial Intelligence (2014)

[19] Julia, M., Gil, A., Reinoso, O.: A comparison of path planning strategies for autonomous exploration and mapping of unknown environments. In: Autonomous Robots 33(4), pp. 427-444 (2012)

[20] Ko, J., Stewart, B., Fox, D., Konolige, K., Limketkai, B.: A practical, decision-theoretic approach to multi-robot mapping and exploration. In: Proceedings of the International Conference on Intelligent Robots and Systems, pp. 3232–3238 (2003)

[21] Lopez-Sanchez, M., Esteva, F., Lopez de Mantaras, R., Sierra, C., Amat, J.: Map generation by cooperative low-cost robots in structured unknown environments. In: Autonomous Robots 5, pp. 53–61 (1998)

[22] Mataric´, M. J.: Behavior-Based Robotics. In: MIT Encyclopedia of Cognitive Sciences, Robert A. Wilson and Frank C. Keil, eds., MIT Press (1999)

[23] Marjovi, A., Nunes, J., Marques, L., de Almeida, A.: Multi-robot exploration and fire searching. In: Proceedings of the International Conference on Intelligent Robots and Systems, pp. 1929–1934 (2009)

[24] Mukhija, P., Krishna, K., Krishna, V.: A two phase recursive tree propagation based multi-robotic exploration framework with fixed base station constraint. In: Proceedings of the International Conference on Intelligent Robots and Systems, pp. 4806-4811 (2010)

[25] Pandey, R., Singh, A., Krishna, K.: Multi-robot exploration with communication requirement to a moving base station. In: Proceedings of the Conference on Automation Science and Engineering, pp. 823-828 (2012)

[26] Pei, Y., Mutka, M., Xi, N.: Connectivity and bandwidth-aware real-time exploration in mobile robot networks. In: Wireless Communications and Mobile Computing 13(9), pp. 847-863 (2013)

[27] Pestman, W.: Mathematical Statistics: an Introduction. de Gruyter (1998)

[28] Quattrini Li, A., Amigoni, F., Basilico N.: Searching for Optimal Off-Line Exploration Paths in Grid Environments for a Robot with Limited Visibility. In: Proceedings of Advancement of Artificial Intelligence. (2012)

[29] Rooker, M., Birk, A.: Multi-robot exploration under the constraints of wireless networking. In: Control Engineering Practice 15(4), pp. 435-445 (2007)

[30] Sariel, S., Balch, T.: Real time auction based allocation of tasks for multi-robot exploration problem in dynamic environments. In: Proceedings of Advancement of Artificial Intelligence Workshop on Integrating Planning and Scheduling, pp. 27–33 (2005)

[31] Simmons, R., Apfelbaum, D., Burgard, W., Fox, D., Moors, M., Thrun, S., Younes, H.: Coordination for multi-robot exploration and mapping. In: Proceedings of Advancement of Artificial Intelligence, pp. 852–858 (2000)

[32] Spirin, V., Cameron, S., de Hoog, J.: Time preference for information in multiagent exploration with limited communication. In: Proceedings of the Conference Towards Autonomous Robotic Systems, pp. 34-45 (2013)

[33] Stachniss, C., Burgard, W.: Exploring unknown environments with mobile robots using coverage maps. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1127–1134 (2003)

[34] Stump, E., Michal, N., Kumar, V., Isler, V.: Visibility-based deployment of robot formations for communication maintenance. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp. 4498-4505 (2011)

[35] Tadokoro, S.: Rescue Robotics. Springer (2010)

[36] Thrun, S.: Robotic mapping: A survey. In: Exploring Artifcial Intelligence in the New Millenium, pp. 1-35. Morgan Kaufmann (2002)

[37] Tovar, B., Munoz, L., Murrieta-Cid, R., Alencastre, M., Monroy, R., Hutchinson, S.: Planning exploration strategies for simultaneous localization and mapping. In: Robotics and Autonomous Systems 54(4), pp. 314–331 (2006)

[38] Tovey, C., Koenig, S.: Improved analysis of greedy mapping. In: Proceedings of the International Conference on Intelligent Robots and Systems, pp. 3251–3257 (2003)

[39] Tuna, G., Gulez, K., Gungor, V. C.: The Effects of Exploration Strategies and Communication Models on the Performance of Cooperative Exploration. In: Ad Hoc Networks (Elsevier) (2013)

[40] Visser, A., Slamet, B.: Including communication success in the estimation of information gain for multi-robot exploration. In: Proceedings of Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks, pp. 680–687 (2008)

[41] Wurm, K., Stachniss, C., Burgard, W.: Coordinated multi-robot exploration using a segmentation of the environment. In: Proceedings of the International Conference on Intelligent Robots and Systems, pp. 1160-1165 (2008)

[42] Yamauchi, B.: Frontier-based exploration using multiple robots. In: Proceedings of the Conference on Autonomous Agents, pp. 47-53 (1998)

Appendice A

Manuale d'installazione e d'uso

Di seguito sono illustrati i passi necessari per installare e per utilizzare il simulatore MRESim [9].
L'archivio *MRESim-master.zip* contiene il codice del simulatore MRESim che abbiamo esteso come visto nel Capitolo 5. Per compilare ed eseguire il codice è possibile utilizzare un IDE come *Eclipse*[1].

Per importare il simulatore all'interno di *Eclipse*, bisogna creare un nuovo progetto all'interno del workspace di *Eclipse*. Per fare questo, bisogna cliccare su *File → New → Java Project*. Nella finestra che si apre (Figura A.1) bisogna inserire il nome del progetto *MRESim-master* e cliccare su *Finish*.
Dopodiché, bisogna importare in *Eclipse* il progetto presente all'interno dell'archivio *MRESim-master.zip*. Per fare questo, bisogna cliccare su *File → Import*. Nella finestra che si apre (Figura A.2) bisogna selezionare *Archive File* e cliccare *Next*. Ora bisogna selezionare l'archivio da importare (*MRESim-master.zip*) e la cartella di destinazione in cui importare l'archivio (*MRESim-master* creata precedentemente nel workspace di *Eclipse*) come mostrato in Figura A.3. A questo punto cliccare *Finish*.
Con questi passi abbiamo importato il codice del simulatore MRESim all'interno di *Eclipse* (Figura A.4).

[1] https://www.eclipse.org/downloads

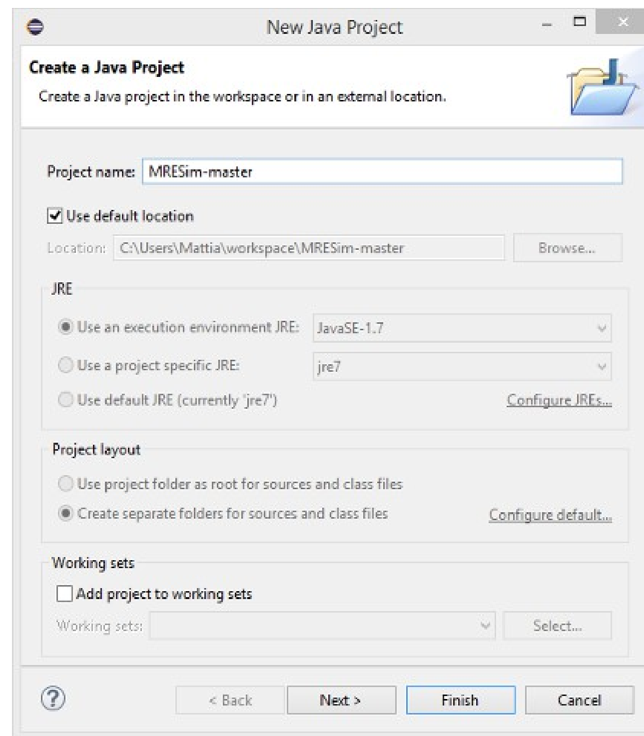


Figura A.1: finestra di creazione di un nuovo progetto Java.

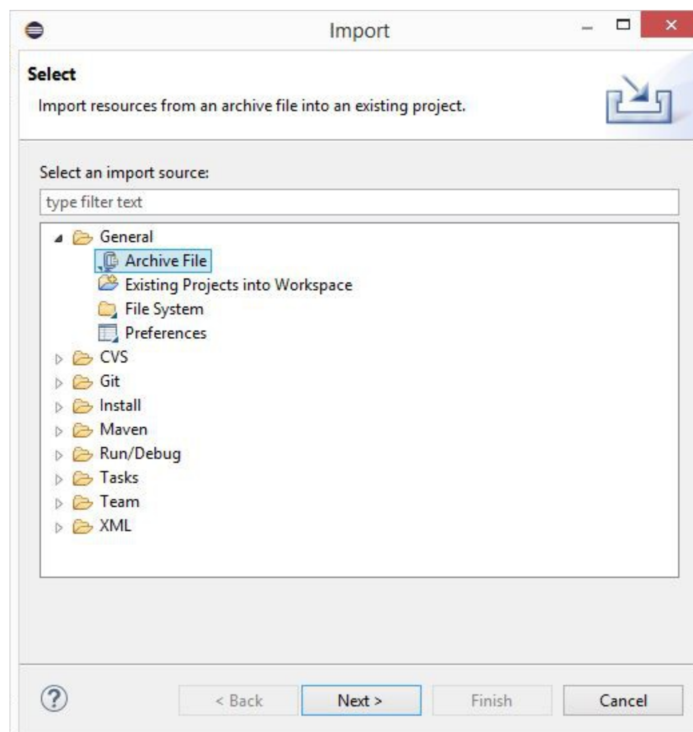


Figura A.2: finestra di importazione di un progetto all'interno di Eclipse.

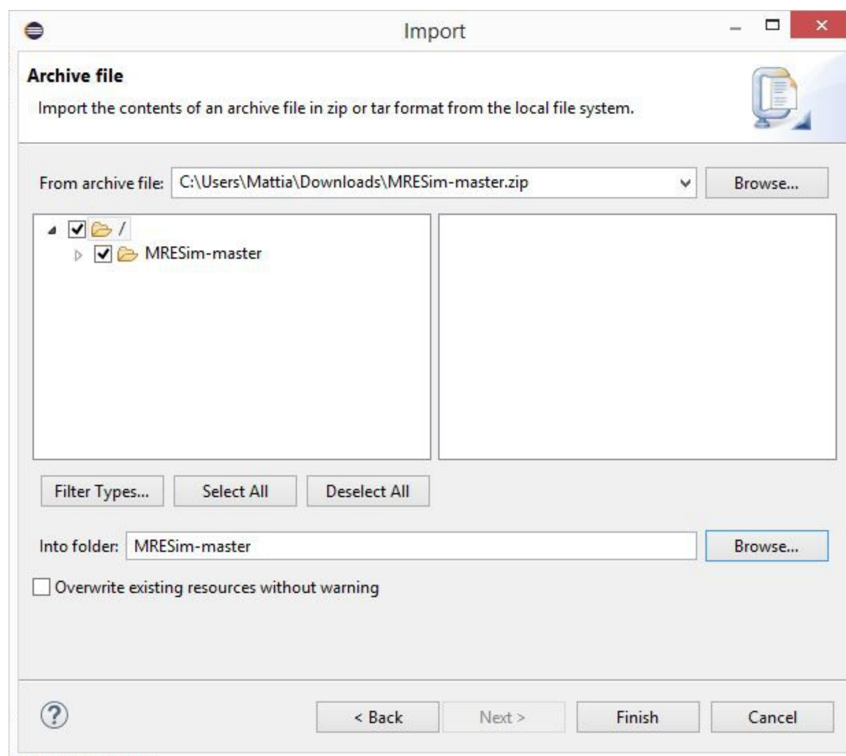


Figura A.3: finestra di importazione dell'archivio contenente il simulatore.

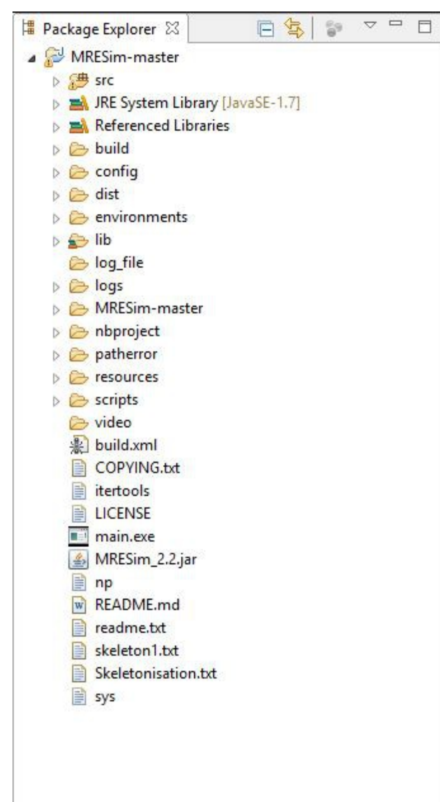


Figura A.4: simulatore importato in Eclipse.

Se dovesse essere presente qualche errore all'interno del progetto, bisogna importare manualmente le librerie utilizzate da MRESim nel modo seguente: *Project* → *Properties* → *Java Build Path* → *addJars*… e selezionare le librerie necessarie dalla finestra *JAR Selection* (Figura A.5).

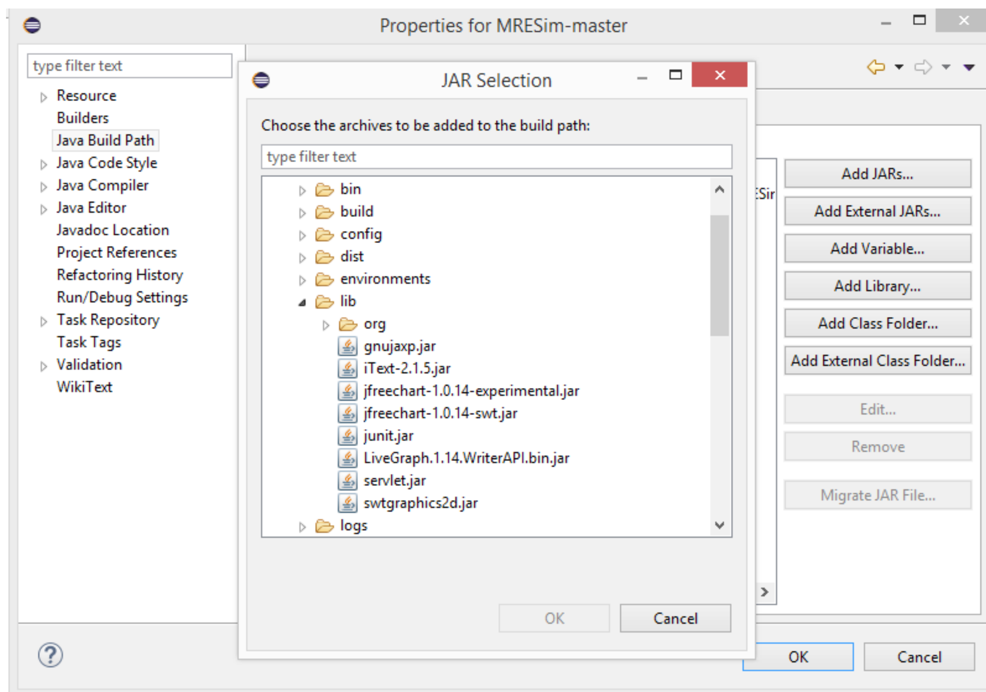


Figura A.5: finestra di importazione delle librerie.

Una volta eseguite queste istruzioni è possibile compilare il codice ed avviare il simulatore. Per farlo bisogna cliccare con il tasto destro sul file *MainGUI.java* (nel package *gui* all'interno della cartella *src*) e selezionare *Run As* → *Java Application* (Figura A.6).

A questo punto il codice viene compilato ed eseguito e viene mostrata l'interfaccia grafica del simulatore (Figura A.7).

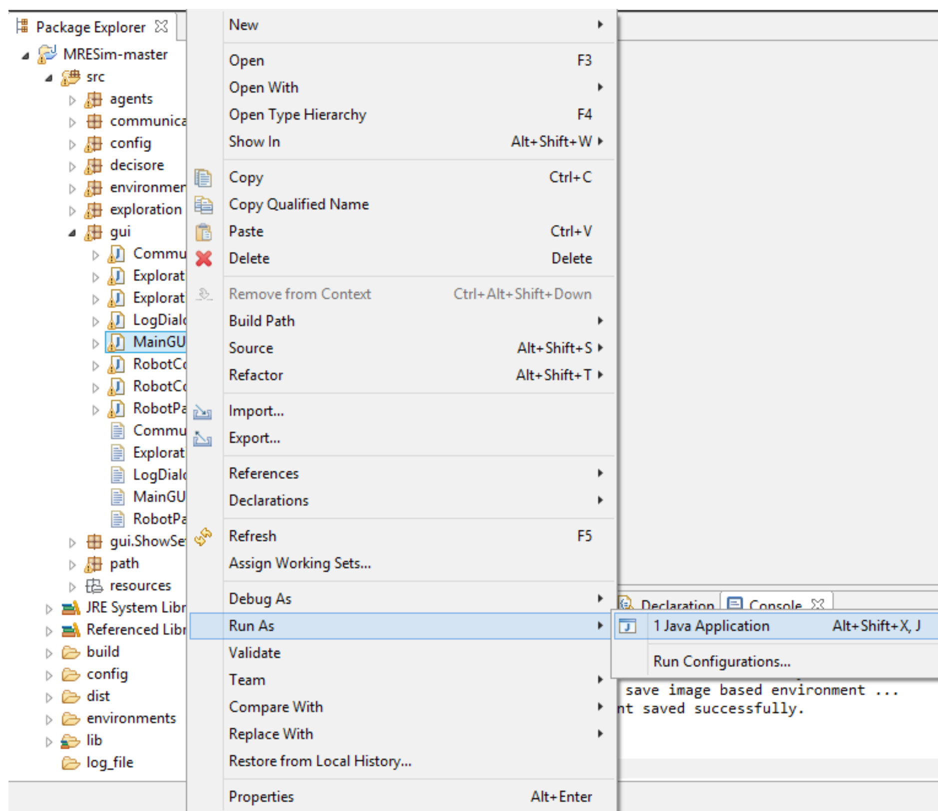


Figura A.6: compilazione ed esecuzione del codice del simulatore.

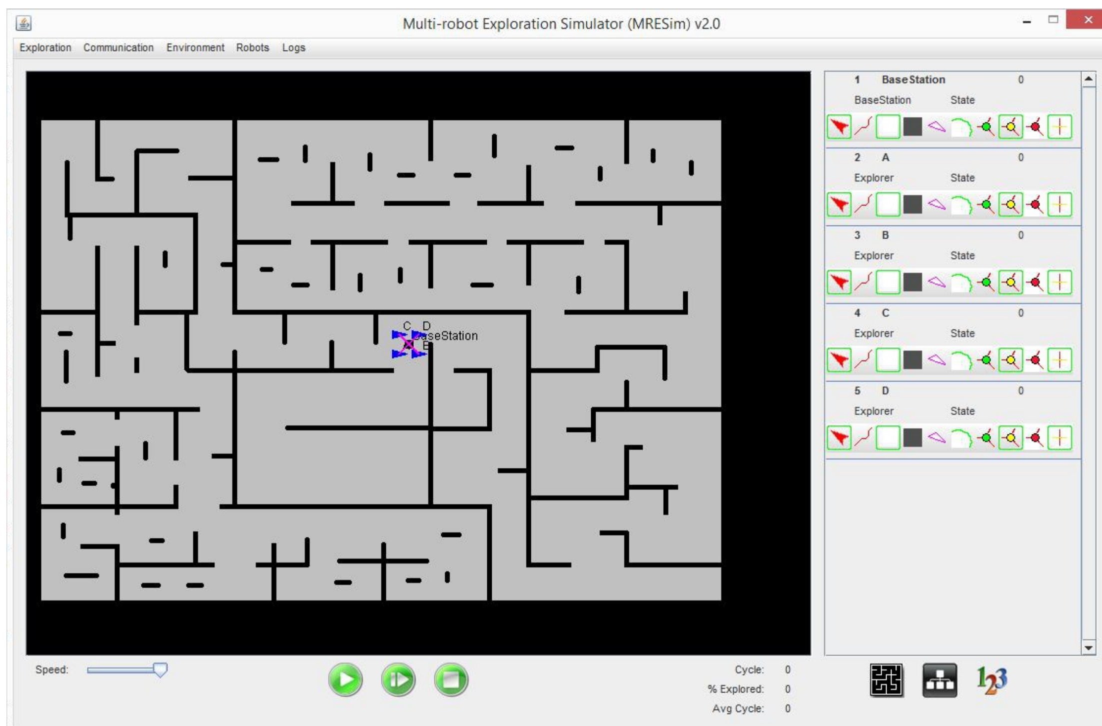


Figura A.7: interfaccia grafica di MRESim.

All'interno dell'interfaccia grafica sono presenti diverse sezioni che permettono di personalizzare l'esplorazione. Vediamole più nel dettaglio.

Nella parte alta dell'interfaccia sono presenti alcuni pulsanti:

- *Exploration*: permette di selezionare una strategia di esplorazione, tra quelle implementate nel simulatore, che deve essere adottata dalla squadra di robot (Figura A.8). Per selezionare le strategie *IlpExploration*, *StumpExploration* oppure *BirkExploration* bisogna aprire il file *lastSimulatorConfig.txt*[2] e bisogna inserire nella prima riga il nome della strategia di esplorazione;
- *Communication*: permette di selezionare una strategia di comunicazione, tra quelle implementate nel simulatore, che deve essere adottata dai robot (Figura A.9);
- *Environment*: permette di selezionare un'immagine bitmap che rappresenta l'ambiente da esplorare (Figura A.10);
- *Robots*: permette di cambiare le caratteristiche della squadra di robot. È possibile aggiungere e rimuovere i robot tramite i pulsanti *Add* e *Remove* ed è possibile salvare e caricare una particolare configurazione iniziale dei robot tramite i pulsanti *Save* e *Load*. È possibile modificare per ogni robot il nome, la posizione di partenza, il raggio di percezione, il raggio di comunicazione, la durata della batteria, il ruolo ed impostare gli ID del genitore (a cui consegnare le nuove informazioni) e del figlio (da cui ricevere le nuove informazioni) nel caso sia necessaria una gerarchia di comunicazione (Figura A.11);
- *Logs*: permette di scegliere se raccogliere informazioni riguardanti l'esplorazione memorizzando, ad ogni ciclo di esplorazione, le posizioni dei robot, alcuni dati riguardanti la simulazione (per esempio, la durata media di un ciclo o l'area conosciuta dalla BS) oppure gli screenshot dell'ambiente (Figura A.11).

[2]Il file si trova nella cartella MRESim-master\config.

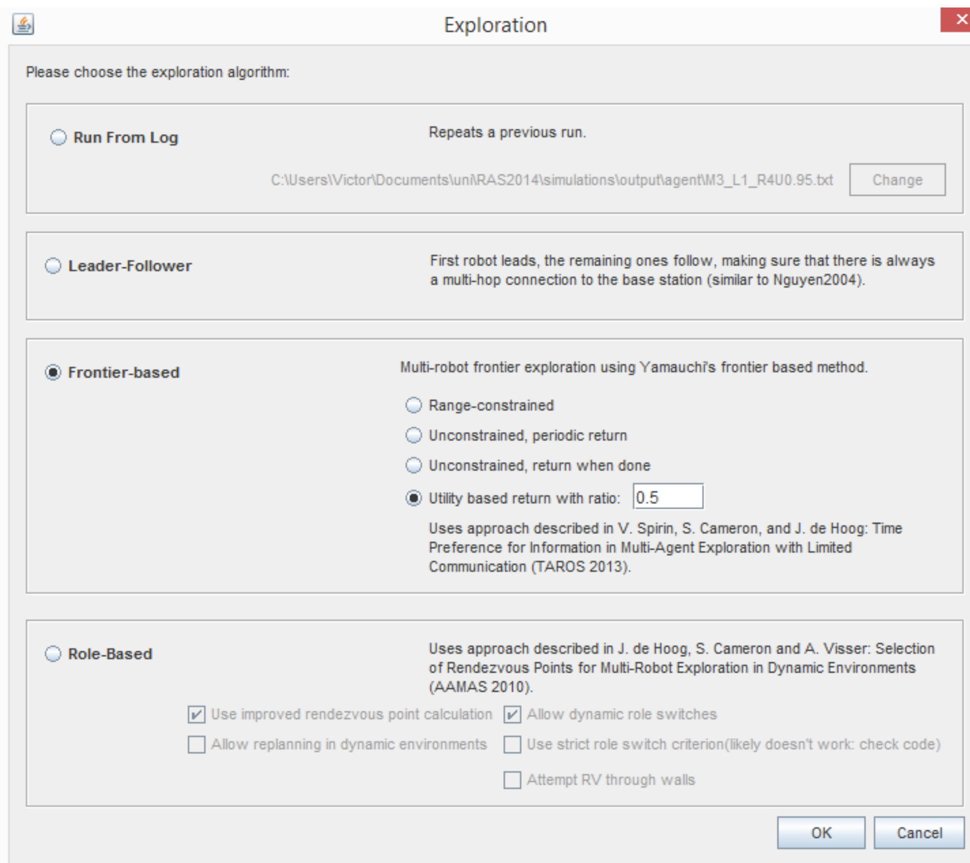


Figura A.8: finestra da cui è possibile selezionare la strategia di esplorazione.

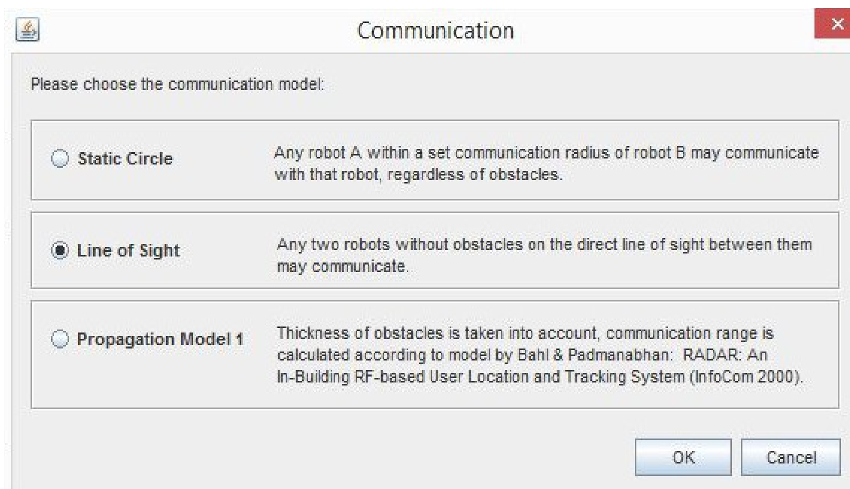


Figura A.9: finestra da cui è possibile selezionare il modello di comunicazione.

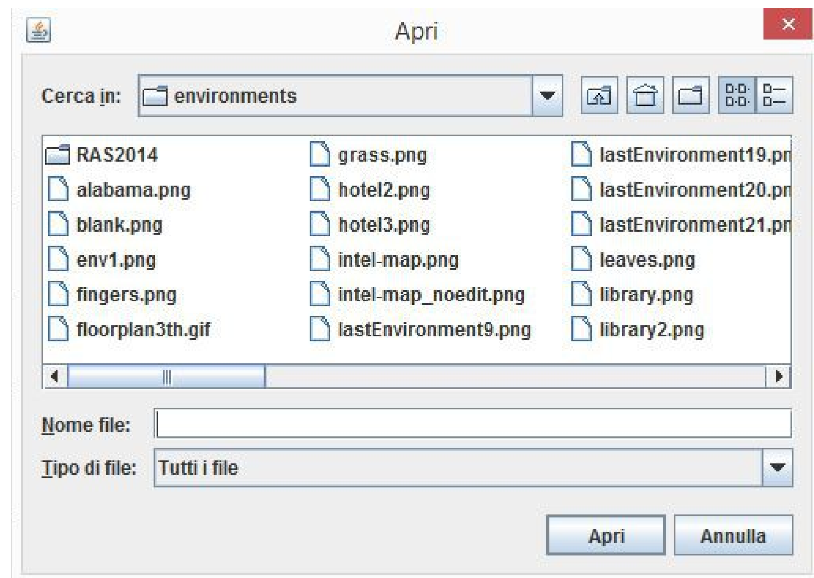


Figura A.10: finestra da cui è possibile selezionare l'ambiente di esplorazione.

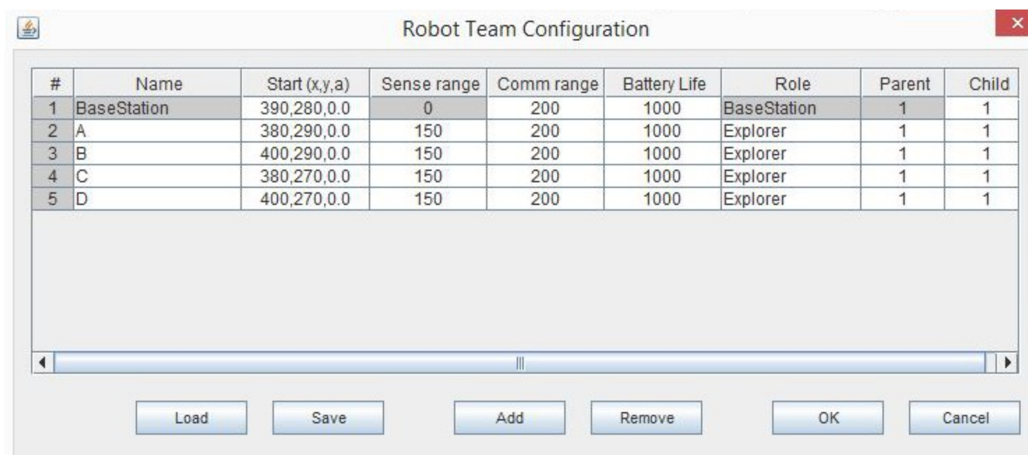


Figura A.11: finestra da cui è possibile selezionare le impostazioni iniziali dei robot.

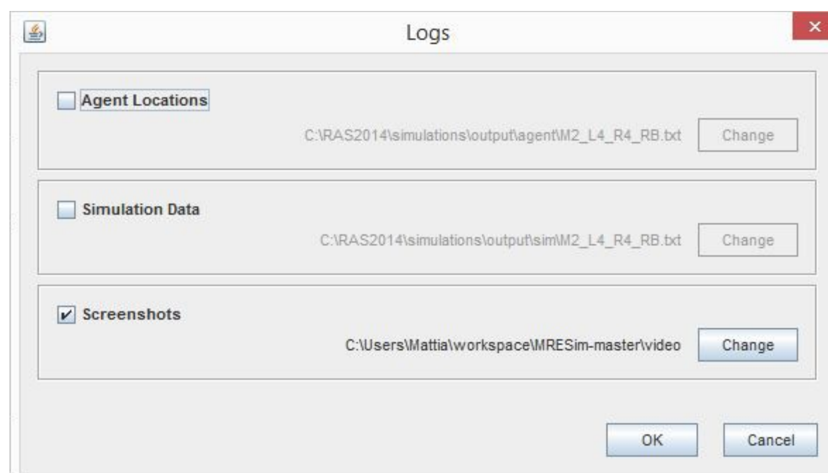


Figura A.12: finestra da cui è possibile scegliere se raccogliere le informazioni dell'esplorazione o meno.

Nella parte centrale dell'interfaccia si trova la mappa dell'ambiente (Figura A.13) in cui, inizialmente, sono rappresentati in nero gli ostacoli e in grigio le aree libere sconosciute in cui i robot possono muoversi. I robot sono rappresentati come dei triangoli blu e la BS è rappresentata come un cerchio. Questa mappa viene aggiornata ad ogni ciclo di simulazione mostrando all'utente come si comportano i robot all'interno dell'ambiente.

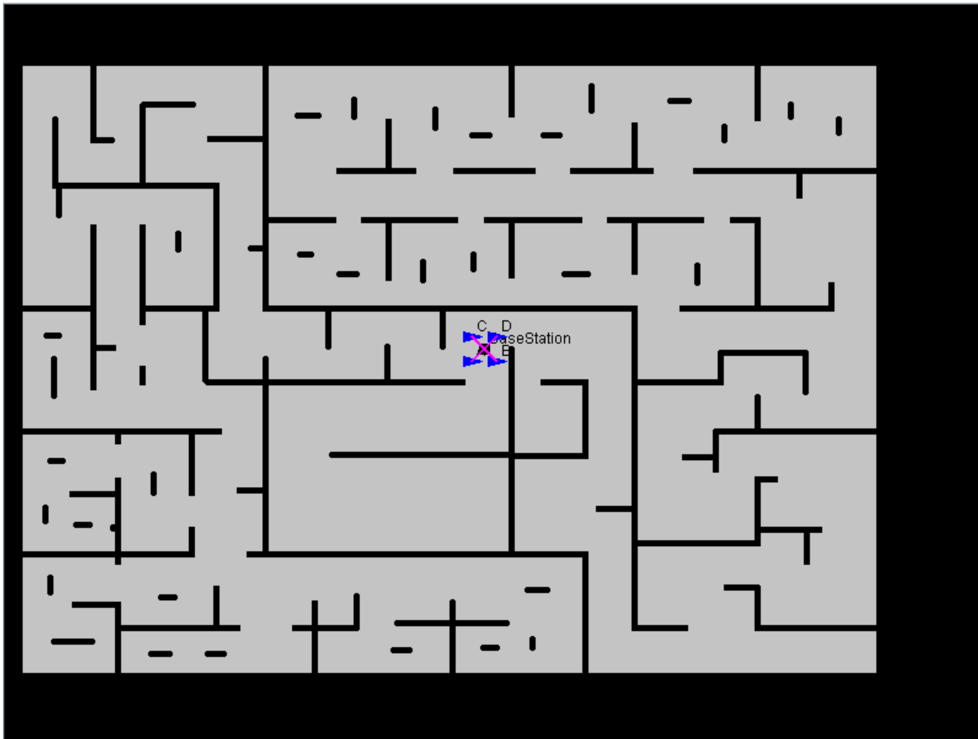


Figura A.13: mappa dell'ambiente.

Nella parte destra dell'interfaccia sono presenti le informazioni relative alla squadra di robot (Figura A.14). Per ogni robot sono mostrati: ID, nome, numero di celle conosciute non ancora consegnate alla BS o ad un altro robot, ruolo e stato.

Inoltre, sono presenti alcuni pulsanti (Figura A.15) che permettono di modificare a runtime alcune informazioni che vengono mostrate nell'immagine dell'ambiente. Nel dettaglio:

1. permette di mostrare o nascondere il triangolo che rappresenta un robot;
2. permette di mostrare o nascondere il percorso che un robot sta seguendo;
3. permette di mostrare o nascondere le celle *free* conosciute da un robot;
4. permette di mostrare o nascondere le celle *safe* conosciute da un robot;
5. permette di mostrare o nascondere le frontiere conosciute da un robot;

6. permette di mostrare o nascondere gli archi di comunicazione tra robot comunicanti;
7. permette di mostrare o nascondere lo skeleton della mappa dell'ambiente conosciuto dal robot;
8. permette di mostrare o nascondere i punti di rendezvouz che un robot ha concordato con altri robot;
9. permette di mostrare o nascondere i punti conosciuti da un robot in cui è possibile comunicare attraverso gli ostacoli;
10. permette di mostrare o nascondere i potenziali punti di rendezvouz conosciuti da un robot per comunicare attraverso gli ostacoli con i compagni di squadra.

Nella parte bassa dell'interfaccia sono presenti altri pulsanti (Figura A.16). Nel dettaglio:

1. permette di regolare di velocità di movimento dei robot;
2. permette di fare iniziare (o mettere in pausa) la simulazione;
3. permette di fare muovere i robot di un solo passo;
4. permette di resettare l'esplorazione portandola allo stato iniziale;
5. mostra all'utente alcune informazioni riguardanti l'esplorazione: numero del ciclo di simulazione, percentuale di area conosciuta dalla BS e tempo medio di esecuzione di un ciclo di simulazione;
6. permette di scegliere tra mostrare soltanto le parti di ambiente effettivamente esplorate e conosciute dai robot o mostrare la totalità dell'ambiente (comprese le zone non ancora esplorate);
7. permette di mostrare o nascondere la gerarchia di comunicazione tra i robot.

Una volta selezionate le preferenze è possibile avviare la simulazione tramite il pulsante numero *2*. A questo punto i robot iniziano ad esplorare l'ambiente.

Sulla *Console* di *Eclipse* (Figura A.17) vengono riportate ad ogni ciclo di simulazione le informazioni che riguardano i robot.

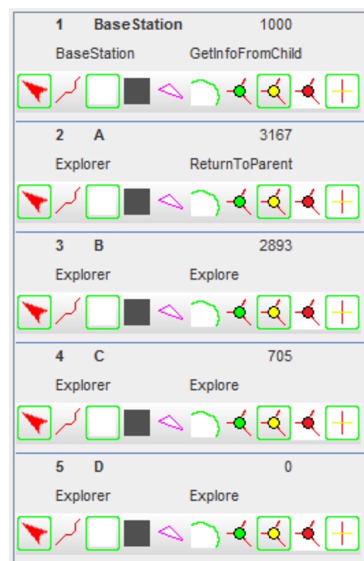


Figura A.14: informazioni relative alla squadra di robot durante l'esplorazione.

Figura A.15: pulsanti presenti nella parte destra dell'interfaccia grafica.

Figura A.16: pulsanti presenti nella parte bassa dell'interfaccia grafica.

```
Problems  @ Javadoc  Declaration  Console
MainGUI [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (11/mar/2015 13:30:05)
[Simulator] ************* CYCLE 2 *****************

[A]  distance left: 5.0
Agent [A]  continuing on path.
[A] takeStep took 0ms.
[A]  takeStep Explore, took 0ms.
[A] Taking step complete, moving from java.awt.Point[x=385,y=295] to java.awt.Point[x=385,y=299], took 0ms.
[A]  distance to next path point: 4.0
[A]  distance left: 1.0
[A] Taking step complete, moving from java.awt.Point[x=385,y=299] to java.awt.Point[x=382,y=302], took 0ms.
[A]  distance to next path point: 4.242640687119285
[A]  exceeded speed. Distance left: 1.0, dist to next path point: 4.242640687119285
[A]  speed corrected. Now is: 1.4142135623730951
[A] Agent cycle complete, took 29ms.
[B]  distance left: 5.0
[B] Path to base computation took 0ms.
Agent [B]  continuing on path.
[B] takeStep took 0ms.
[B]  takeStep Explore, took 0ms.
[B] Taking step complete, moving from java.awt.Point[x=405,y=285] to java.awt.Point[x=405,y=281], took 0ms.
[B]  distance to next path point: 4.0
[B]  distance left: 1.0
[B] Taking step complete, moving from java.awt.Point[x=405,y=281] to java.awt.Point[x=405,y=278], took 0ms.
[B]  distance to next path point: 3.0
[B]  exceeded speed. Distance left: 1.0, dist to next path point: 3.0
[B]  speed corrected. Now is: 1.0
[B] Agent cycle complete, took 11ms.
[Simulator] agentSteps took 40ms.

[Simulator] simulateCommunication took 67ms.

[Simulator] updateGlobalData took 151ms.

[Simulator] Cycle complete, took 682ms.
```

Figura A.17: esempio di informazioni riportate a runtime mostrate nella console durante la simulazione.