Spiegazione AI per il requisito di R-Map

Federico Stella - federico.stella4@studio.unibo.it

21 Luglio 2018

1 Prima analisi

Le righe che seguono sono presenti alla sezione 7.3 del LaTeX che abbiamo consegnato durante la discussione di Venerdì 20 Luglio 2018.

- **Reactiveness**: the robot should be reactive to stop commands even during its obstacle avoidance policy, the designer must take this into account.
- **Map creation, discovery walk and end check**: all these requirements can't be correctly tackled using the QActor language, as the abstraction gap is too high and we would spend more time on the infrastructure code than on the code that actually tries to satisfy the requirement. An approach from the field of Artificial Intelligence is also required in order to fulfill these requirements. In order to produce a map we need to split the continuum of the room into a discrete space, composed for example by little boxes of the size of the robot. We end up with a chessboard in which every box can contain an obstacle or not and be already cleaned or still dirty, and which we should cover completely with a path. The state of the robot comprehends the box the robot is in, and the directions the robot is facing. The optimal path is completely unfeasible given the computational power we have and the complexity of the problem of finding an Hamiltonian Circuit (NP-Complete, close to a factorial, completely unfeasible). For this reason it is already evident that an approximated approach is needed, a good one can be the Uniform Cost Search algorithm with a search for the closest reachable and dirty box, it is a designer's job to find a good solution to this.

2 Spiegazione completa

Riassumendo la prima spiegazione e completandola, possiamo dire che per affrontare il problema è necessario discretizzare il continuo bidimensionale della stanza, producendo un piano cartesiano discreto in cui una coppia di coordinate individua univocamente una cella, la quale può essere sporca o pulita, può contenere un ostacolo o meno, e può contenere il robot o meno. Non vogliamo impostare a priori una dimensione della stanza: l'unica conoscenza *a-priori* che abbiamo è il fatto che il robot

parta dall'angolo in alto a sinistra direzionato verso il basso e che vi sia un sonar posizionato in basso a destra, il cui output può essere letto.

Nota: una mossa del robot ha un costo. Non solo andare avanti ha un suo costo, ma anche girarsi a sinistra o a destra! Per questo motivo, a partire da un preciso stato, non è importante sapere se ci si può *muovere* indietro, verso sinistra o verso destra; è invece importante sapere che ci si può sempre *girare* su se stessi e sapere se per caso si può anche andare avanti.

Si farà riferimento ai progetti `it.unibo.exploremap` nel "workspace_sprint3" e `it.unibo.exploremap.extra` nel "workspace_sprint4"

Con queste premesse, il software è quindi diviso in parti: serve una rappresentazione della mappa (`RoomMap`), una rappresentazione dello stato attuale (`RobotState`), una rappresentazione delle azioni possibili (`RobotAction`), un modo per stabilire se il goal è stato raggiunto oppure no (`Functions.isGoalState(Object arg0)`), un modo per applicare l'azione a uno stato (`Functions.result(Object arg0, Action arg1)`), un modo per capire quali azioni sono disponibili in un certo stato, interrogando anche la mappa (`Functions.actions(Object arg0)`), e infine una funzione che calcoli il costo di un'azione su uno stato (`Functions.c(Object arg0, Action arg1, Object arg2)`).

2.1 RoomMap

La mappa della stanza è rappresentata come piano cartesiano di celle, per cui data una coppia di coordinate intere restituisce la cella. Ogni cella ha 3 caratteristiche con valori binari di verità/falsità: è un ostacolo, è sporca, è il robot. Ha senso mantenere attributi distinti per prevedere anche la possibilità che un ostacolo possa essere sporco, e quindi vi si possa, per esempio, cercare di passare ogni tanto per vedere se ora quell'ostacolo non c'è più.

2.2 RobotState

Lo stato del robot mantiene la coppia di coordinate della cella che lo contiene, e un enumerativo di direzione, che può essere verso l'altro, verso il basso, verso sinistra e verso destra. RobotState dovrà anche contenere i metodi per applicare correttamente i movimenti sulle proprie coordinate x e y.

2.3 RobotAction

Un'azione mantiene semplicemente indicazione di quale sia il suo significato: può essere un'azione di movimento in avanti o indietro, oppure un'azione di cambiamento di direzione verso destra o verso sinistra.

2.4 isGoalState()

Per capire se uno stato è il goal, è necessario prima definire come affrontare il problema. Come già detto nella prima sezione, ovvero quella che citava la relazione già

consegnata, è evidente che trovare il percorso ottimo all'interno di una mappa già nota sia computazionalmente impossibile. Dovendo poi anche costruire la mappa senza conoscenza a priori della dimensione della stessa, e della posizione e dimensione di eventuali ostacoli, si è deciso di approcciare il problema cercando, in ogni momento, la posizione della **casella ancora sporca più semplice da raggiungere**, dove la semplicità viene stabilita dal costo delle azioni di movimento.

Per questi motivi, una casella è definita "goal" se è pulita.

Inoltre, un ulteriore requisito è quello di terminare la pulizia andando sul sonar2. Per questo, una volta che tutta la mappa sarà esplorata e pulita, il goal diventerà la cella contenente il sonar2, ovvero quella più in basso a destra nella mappa. La ricerca nello spazio degli stati produrrà il percorso più breve per raggiungerla.

2.5 result()

L'applicazione di un'azione a uno stato è semplice: produce un nuovo RobotState con le nuove coordinate/la nuova direzione.

2.6 actions()

Anche trovare le azioni possibili da una certa posizione è piuttosto semplice: è sempre possibile girarsi a sinistra e a destra, è sempre NON possibile andare indietro (perché un robot, allo stato attuale, non vede gli ostacoli dietro di sé, ma è molto facile da modificare in caso un robot possa farlo), e può essere possibile o meno andare avanti. Interrogando la mappa si capisce facilmente se è possibile o no: se c'è un ostacolo è impossibile, se è pulito è possibile ma non è goal, se è sporco è possibile, e se è una cella ancora sconosciuta è possibile.

Notare che questa sezione in particolare poteva essere affrontata in maniera molto diversa, per esempio dicendo che le azioni possibili a partire da uno stato possono essere andare avanti, andare a sinistra (ovvero girarsi a sinistra + andare avanti), ecc. Il modo scelto da me, però, permette di considerare costose anche le azioni di rotazione del robot, cosa decisamente realistica, ed è comunque flessibile, tramite la funzione di costo, a situazioni in cui la rotazione non sia costosa o in cui ci si possa anche muovere in direzioni diverse da quella frontale senza costi aggiuntivi.

2.7 c()

Il calcolo del costo è estremamente semplice: si è deciso che il costo sia identico sia per azioni di movimento sia per azioni di rotazione. È molto semplice cambiare questo comportamento.

2.8 Ricerca

Per trovare i percorsi si è scelta una strategia di ricerca nello spazio degli stati, in particolare una strategia completa e ottimale senza euristica detta **Ricerca a Costo Uniforme** (Uniform-Cost Search). Nel nostro caso particolare in cui ogni

azione ha lo stesso costo, essa prende il nome di **Ricerca in Ampiezza**, meglio nota come Breadth-First Search. Il comportamento previsto sarà evidentemente quello di trovare, in ogni stato, la cella sporca più vicina, trovando sempre il percorso ottimale per andarvici. Vista la palese struttura a grafo del problema, e non ad albero, si è deciso di optare per la versione **Graph-Search** della Ricerca in Ampiezza: in questo modo l'efficienza aumenta incredibilmente e, con essa, diminuisce il tempo di calcolo e si evitano degenerazioni cicliche. Le prestazioni di una ricerca a grafo rispetto a una ad albero sono incomparabilmente superiori.

Per quanto riguarda i problemi di reattività, bisogna distinguere tra 3 possibili interruzioni: sonar, collisione, stop.

- **Sonar**: dando una dimensione non troppo ampia alle caselle, i dati in arrivo dai sonar sono importanti soltanto quando il robot è fermo, per capire se si è raggiunto il muro in basso e/o il muro a destra. L'assunzione di avere caselle sufficientemente piccole è un'assunzione non limitante per la risoluzione del problema, ed anzi più le caselle sono piccole e più aumenta l'accuratezza della mappa generata.

- **Collisione**: i dati delle collisioni sono, viceversa, importanti soltanto mentre il robot si sta muovendo in avanti o indietro, in tutti gli altri casi non possono esserci collisioni. Considerato questo e considerato il fatto che un movimento del robot richiede un certo tempo, è possibile passare quel tempo in attesa sul canale di ricezione dei dati delle collisioni al posto di sprecarlo. In questo modo, non appena arriva una collisione, è possibile fermarsi e agire di conseguenza: dapprima si aspetta un po' per vedere se l'ostacolo si sposta e poi, se non si è spostato, si ricalcola una nuova strategia.

- **Stop**: il segnale di stop, invece, può arrivare in qualunque momento. Dovendo capire quanto sia importante il ritardo per l'utente, si è stabilito che una frazione di secondo, o poco più, non impattano minimamente sull'esperienza d'uso. Considerato questo e considerato il fatto che la ricerca è ad albero, sappiamo che un'iterazione di ricerca è molto breve su qualunque CPU anche domestica, perciò è tranquillamente possibile controllare la presenza di segnali di stop ogni volta che finisce il ciclo di ricerca e ogni volta che si sta per attuare un'azione.

3 Note finali

La funzione goal è sostituibile in qualunque momento nell'algoritmo di AI, perciò è facile impostare come goal "andare al sonar2" quando la pulizia termina. Allo stesso modo è stata anche introdotta l'apposita funzione per tornare al sonar1 ed eventualmente ricominciare la pulizia con mappa nota.

Il progetto finale che mette insieme l'intero backend e l'AI è `it.unibo.iss.finaltask.design3` nel "workspace_sprint4".