GESTIONE PERCORSI LAST MILE DELIVERY

PDDL e ALGORITMI DI RICERCA

Bucella Giorgio – 731014

Pasolini Nicola – 730186

Rossi-Erba Riccardo – 731395

INDICE

Sommario

1. OBIETTIVO

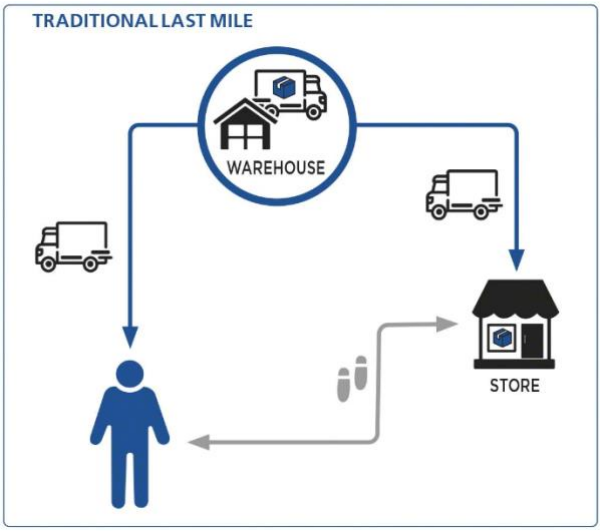
Lo scopo del progetto è quello di trovare il percorso più efficiente in ambito last-mile delivery, l'ultima porzione della consegna di un ordine con spedizione a domicilio, dall'ultimo magazzino al destinatario, avendo in input delle location in cui bisogna consegnare dei pacchi scelte da un elenco di n location note e collegate al magazzino.

Più in particolare il mondo in cui operano gli algoritmi utilizzati è notevolmente semplificato e simbolico, la mappa corrispondente è costituita da n nodi, che rappresentano il magazzino, dunque il punto iniziale e finale del percorso, e tutte le location raggiungibili dal magazzino stesso, e archi che collegano tali nodi, che rappresentano le strade che collegano le varie location ai quali è stata attribuita una funzione di costo che simula il tempo impiegato a percorrere ciascun arco.

Nell'implementazione PDDL è stata inoltre aggiunta l'azione di consegna vera e propria del pacco, anch'essa caratterizzata da un costo che ne descrive il tempo impiegato.

È stato scelto un modello di mondo in cui tutti gli stati sono raggiungibili, in quanto non risulta realistico per un reale planner di percorsi di questo genere prevedere location che non sono in alcun modo raggiungibili dal magazzino di riferimento.

Il termine location è da intendersi in senso generico, in quanto astrae un qualsiasi luogo di ritiro, che esso sia un'abitazione, una casella postale o un esercizio commerciale non fa alcuna differenza per il programma.



Schema essenziale last mile delivery

2. IMPLEMENTAZIONE PDDL

2.1 DOMINIO

2.1.1 TIPI

Per il livello di astrazione scelto è sufficiente definire un solo tipo, location, sottoclasse di Object.

Nel nostro caso location rappresenta un singolo nodo della mappa.

```
Show hierarchy
5   (:types
6     location - object
7   )
8
```

2.1.2 PREDICATI

```
(at ?loc - location)
```

Predicato usato per sapere ci troviamo o meno nella location passata come parametro.

```
(connected ?from ?to - location) 1👁
```

Stabilisce il collegamento fisico tra due località, con la possibilità di raggiungere direttamente la location ?to a partire dalla location ?from.

```
(delivered ?loc - location) 1👁 1☑
```

Indica se nella location a cui fa riferimento è stata effettuata o meno una consegna.

2.1.3 FUNZIONI

```
(distance ?from ?to - location) 1👁 - number
```

Permette di associare al concetto di location collegate un valore numerico che esprime il tempo di percorrenza del collegamento.

```
(total-cost) 1↗- number
```

Permette di attribuire un costo complessivo al percorso esplorato, tenendo conto sia del tempo di collegamento che di consegna.

2.1.4 AZIONI

```
(:action move
  :parameters (?from ?to - location )
  :precondition (and
                  (at ?from )
                  (connected ?from ?to)
  )
  :effect (and
            (not (at ?from))
            (at ?to)
            (increase (total-cost) (distance ?from ?to))
          )
)
```

Move: permette il movimento tra due location direttamente collegate.

Parametri: location di partenza e di destinazione.

Condizioni iniziali: è necessario essere posizionati nella location di partenza, la destinazione e la partenza devono essere collegate.

Effetto: la posizione corrente diventa la location di arrivo, non si è più in quella di partenza e il costo totale del percorso subisce un incremento pari al costo del collegamento effettuato.

```
(:action deliver
  :parameters (?loc - location)
  :precondition (and (at ?loc) (not (delivered ?loc)))
  :effect (and
            (delivered ?loc)
            (increase (total-cost) 1)
          )
)
```

Permette di consegnare un pacco in una determinata location.

Parametro: la location oggetto della consegna.

Condizioni iniziali: essere nella location obiettivo e non aver ancora consegnato un pacco in tale località.

Effetto: il pacco viene consegnato e il costo totale del percorso incrementa di uno.

Il fatto che le consegne abbiano tutte lo stesso costo è una semplificazione del modello, il fatto che esso sia uno è perché è nettamente inferiore a qualsiasi costo di collegamento.

2.2 PROBLEMA

2.2.1 OGGETTI DEL PROBLEMA

```
(:objects
  A B C D E F G H I J K L M N O P Q R S T U V W - location
)
```

Le n istanze del tipo location, in questo caso W è stato scelto come nodo che rappresenta il magazzino.

2.2.2 CONDIZIONI INIZIALI DEL PROBLEMA

```
(at W)
(= (total-cost) 0)
```

La location in cui si deve partire in ogni istanza del dominio è il magazzino e il costo totale iniziale è impostato a 0.

2.2.3 MAPPA DEL MONDO

```
(connected W A)
(connected A W)
(connected W B)
(connected B W)
```

Si costruiscono i collegamenti scelti, nell'esempio sono 39 ma in figura sono riportati solo i primi 4, in maniera bidirezionale, altrimenti non sarebbe sempre possibile tornare al magazzino o effettuare tutte le consegne. Ogni nodo ha almeno un collegamento bidirezionale.

```
(= (distance W A) 11)
(= (distance A W) 11)
(= (distance W B) 21)
(= (distance B W) 21)
```

In seguito si attribuisce a ciascun collegamento il relativo costo, che non cambia a seconda del verso di percorrenza del collegamento bidirezionale.

2.2.4 OBIETTIVO DEL PROBLEMA

```
(:goal (and
            (at W)
            (delivered E)
            (delivered P)
            (delivered H)
        )
)
```

L'obiettivo del programma è quello di trovarsi nella location del magazzino e aver compiuto tutte le consegne dichiarate nell'obiettivo, inoltre deve trovare il percorso che gli permetta di soddisfare queste richieste al minor costo possibile.

La scelta di associare alla consegna un costo è stata fatta anche per assicurarci che durante il percorso non vengano fatte consegne "superflue", ovvero non specificate negli obiettivi del problema.

```
(:metric minimize (total-cost))
```

2.3 SOLVER

Avendo utilizzato funzioni numeriche, è necessario l'utilizzo di un solver che abbia il supporto per i fluents e la scelta è ricaduta su ENHSP, testato in modalità ottima e subottima.

Script ottimo: Java-jar enhsp-20.jar-o domain.pddl-f problem.pddl-planner opt-hmax

Questa configurazione ottimale utilizza A* con euristica numerica hmax

Abbiamo poi testato il parametro-sat che generalmente utilizza algoritmi Greedy Best First Search.

3. ALGORITMI DI SEARCH IMPLEMENTATI ATTRAVERSO PYTHON

Per risolvere il problema è stato possibile utilizzare anche un'altra metodologia di risoluzione: gli algoritmi di ricerca.

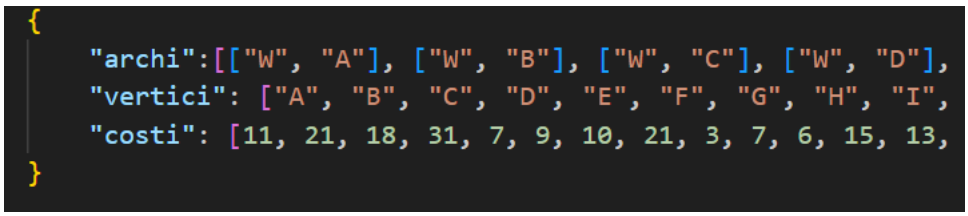
La prima cosa da fare è stata la formulazione dell'obiettivo, per andare a organizzare il comportamento e definire le azioni da considerare. Successivamente abbiamo dovuto formulare il problema definendo il modello della parte di mondo di nostro interesse.

Gli algoritmi utilizzati non si basano sull'euristica, come accade invece con quelli utilizzati dal solver PDDL, scritti in codice Python.

3.1 MODELLIZZAZIONE

La mappa del mondo è descritta da un file JSON contenente un solo oggetto descritto da tre proprietà: archi, vertici e costi, questa modalità è stata scelta per poter salvare le informazioni riguardo ad una mappa, per avere dei dati persistenti.
La mappa vera e propria è poi rappresentata da un grafo, non orientato, dove i vertici rappresentano le n location del mondo, gli archi i collegamenti tra esse e i costi sono relativi agli archi e rappresentano la distanza fra due nodi.

Gli archi non sono caratterizzati da una direzione specifica, sono percorribili partendo da uno qualsiasi dei due vertici che li descrivono. Se fra due nodi non è presente un arco allora i due nodi non sono direttamente collegati.



esempio di mappa

Il mondo è un oggetto di tipo World, che ha come attributi:

- Una mappa, un oggetto di tipo Mappa (definito nel file mappa.py)
- Un insieme di stati, gli stati del mondo.

La mappa a sua volta ha due attributi:

- Il grafo, un oggetto Graph della libreria esterna igraph, una raccolta di librerie per la creazione e la manipolazione di grafici
- Un oggetto GraphManager, di utility, che ha i compiti di creare il grafo a partire dal file JSON rappresentante la mappa, salvare in un'immagine la rappresentazione visiva del grafo. GraphManager può anche creare delle mappe casuali o delle mappe quadrate con costi casuali. Le informazioni per la creazione della mappa vengono lette dal file di configurazione: "world_type" : "random" per la mappa casuale e "size" è il numero di location, : "world_type" : "square" per la mappa quadrata e "size" per avere size*size location. Nell'impostazione attuale le mappe funzionano fino a 26 location, dato che ogni location è definita da una lettera maiuscola.

Le informazioni sul problema sono definite in un file di configurazione ("config.JSON"), dove sono specificati il goal e lo stato iniziale.

Il problema viene modellizzato nel file last-mile.py, in cui si definiscono gli stati del mondo e le azioni che può compiere l'agente. Più precisamente il problema è stato definito come un oggetto LastMileProblem che ha come attributi lo stato iniziale, una serie di stati obiettivo, un mondo e un insieme di azioni.

Il problema è quindi definito come segue:

- Lo spazio degli stati è un attributo del mondo, una lista di stati, dove ogni stato è identificato da una stringa
- Lo stato iniziale è un attributo del problema, viene letto dal file di configurazione
- Il goal è un attributo del problema, definito come una lista di stati, viene letto dal file di configurazione
- Le azioni sono un attributo del problema, vengono definite automaticamente a partire dal mondo. Un'azione è un oggetto di tipo Action, che ha un nome e un costo. L'azione di interesse per il problema è l'azione di movimento da uno stato all'altro e questa è definita come un oggetto Move, sottoclasse di Action, che ha in più gli attributi stato di partenza e stato di arrivo. L'unica azione permessa all'agente è quella di muoversi tra un nodo e un altro attraverso gli archi che li collegano, non tutti i nodi sono collegati mutuamente quindi non tutte le coppie di nodi formano un arco.
  Il metodo get_actions(state) del problema ritorna una lista di azioni disponibili in uno stato.
- Il modello di transizione è definito all'interno dell'azione Move, con l'attributo stato di destinazione
- La funzione di costo è definita all'interno dell'azione Action, con l'attributo costo

Lo stato iniziale è una posizione della mappa e l'obiettivo è trovare un cammino che parta dalla location iniziale (che si presume essere il magazzino merci), torni alla location iniziale, e passi attraverso una serie di location specificate. Una soluzione è pertanto un percorso che soddisfa queste richieste.

Nell'implementazione dei search algorithms non è previsto il concetto di consegna, l'azione di delivery (presente nel PDDL) è implicita nel momento in cui si passa per la prima volta in una location target, in quanto il modello permette che l'agente transizioni più volte nello stesso stato.
A differenza della modellazione attraverso PDDL, gli stati sono atomici, senza attributi interni.

3.2 ALGORITMI UTILIZZATI

Sfruttando la nozione di ereditarietà è stato scritto il file search_algorithms.py con le caratteristiche comuni a tutti gli algoritmi implementati, che sono Best First Search, Breadth First Search, Deep First Search Limited e Iterative Search.

Nella classe SearchAlgorithm sono stati definiti i metodi comuni a tutti gli algoritmi:

- solve(), implementazione effettiva dell'algoritmo, è implementato dalle sottoclassi di SearchAlgortihm
- extractSolution(nodo_goal), dato un nodo goal restituisce la soluzione, che varia a seconda dell'algoritmo, per raggiungere tale nodo, sotto forma di oggetto Solution
- isGoal(nodo), definisce se lo stato corrente corrisponde a uno stato obiettivo
- camminoAzioni(nodo), dato un nodo ritorna l'insieme delle azioni che hanno permesso all'agente di raggiungere tale nodo a partire dallo stato iniziale, usato per evitare loop
- espandi(nodo) espande la frontiera del nodo, ritornando una lista di nodi, il cui stato è prossimo a quello di nodo, e il cui costo è il numero di azioni eseguite per raggiungere il nodo espanso

I metodi isGoal(), extractSolution() e camminoAzioni() sono comuni in tutti gli algoritmi, il metodo espandi() definito in SearchAlgorithm è l'implementazione del metodo utilizzato dalla maggior parte degli algoritmi.

Un nodo è un oggetto della classe Nodo, che ha come attributi:

- lo stato in cui si trova l'agente, espresso come una stringa
- il nodo padre, espresso tramite un altro nodo
- l'azione che ha portato a quel nodo dal nodo padre
- il costo per raggiungere il nodo, espresso come intero

3.2.1 BEST FIRST SEARCH

Ritorna il percorso ottimo valutato con una funzione numerica che assegna a ciascun arco un valore per poter scegliere i nodi da espandere in modo tale da avere un percorso finale a costo minimo.

La frontiera viene gestita con una coda di tipo PriorityQueue, in cui si assegnano diversi livelli di priorità agli oggetti, i primi sono quelli con livello più basso. La priorità viene definita all'interno dell'oggetto Nodo, in cui è stato definito il metodo __lt__(self, other) specificando che un nodo ha una priorità pari al suo costo.

Il metodo espandi(nodo_A) espande la frontiera del nodo_A, ritornando una lista di nodi, il cui stato è prossimo a quello di nodo_A, e il cui costo è il costo di nodo_A sommato al costo dell'azione che porta al nuovo nodo.

Un nodo viene aggiunto alla frontiera se l'azione che porta al nuovo nodo non è ancora stata percorsa, in questo modo non si può effettuare la stessa azione due volte.

È un algoritmo completo perché visita tutti i cammini in modo sistematico, non essendo permesso effettuare cammini ciclici.

3.2.2 BREADTH FISRT SEARCH

Ritorna la soluzione con il minor numero di azioni eseguite, quindi con il minor numero di strade percorse, ma non è detto che sia la soluzione ottima dal punto di vista del costo.

La frontiera è gestita mediante una coda FIFO, quindi un oggetto Queue.

Il metodo espandi(nodo) è quello ereditato dalla classe SearchAlgorithm.

È un algoritmo completo perché visita tutti i cammini in modo sistematico.

3.2.3 DEEP FIRST SEARCH LIMITED

Espande prima i nodi a profondità maggiore, fino ad arrivare ad un limite prestabilito.

Per gestire la frontiera utilizza una coda LIFO, quindi un oggetto LifoQueue. In tal modo i primi nodi espansi sono quelli più distanti, più "*profondi*", dal nodo iniziale. Inoltre è necessario impostare un parametro di profondità l che sarà il limite della ricerca dell'algoritmo, i nodi a una distanza maggiore di l archi dal nodo iniziale non vengono considerati.

Non completo perché se il parametro profondità non è correttamente scelto la soluzione è indeterminata.

3.2.4 ITERATIVE SEARCH

Itera l'algoritmo Deep First Search Limited variando il parametro profondità a partire da 1 fino a che per un valore i finito del parametro produce una soluzione determinata, oppure ritorna un fallimento.

Completo perché restituisce sempre la prima soluzione che incontra, risolve il problema del Deep First Search Limited Algorithm andando a trovare un valore accettabile di profondità, andando a provarle tutte in maniera crescente.

3.3 ESECUZIONE

3.3.1 CLASSI DI UTILITY

Sono state definite anche delle classi di utility per rendere più versatile l'applicazione.

La classe Solver è caratterizzata da:

- solver_type, un parametro che indica il tipo di algoritmo con il quale si vuole risolvere il problema
- prob, il problema che si vuole risolvere, è un oggetto LastMileProblem
- folder, il percorso di una cartella nella quale si vuole salvare i risultati

All'interno della classe Solver è definita la classe SolverSelector, che ha il compito di restituire il giusto solver basato su solver_type. SolverSelector ha un unico attributo, un dizionario che ha come chiave il nome dell'algoritmo di ricerca, e come valore una sua istanza. È stata realizzata per permettere di aggiungere facilmente nuovi algoritmi all'applicazione.
Per aggiungere nuovi algoritmi di ricerca e permettere la risoluzione del problema senza modificare troppo codice bisogna aggiungere il nuovo algoritmo al dizionario di questa classe.

La classe Solver implementa il metodo solution_description(solution, elapsed_time) che ritorna una stringa con delle informazioni sulla soluzione.

La classe Solver implementa anche il metodo solve() che è il metodo che invoca SearchAlgorithm.solve() per trovare una soluzione al problema, calcola il tempo necessario per l'esecuzione e invoca il metodo solution_description andando a salvare le informazioni riguardanti la soluzione in un file di testo.

La classe FileWriter è una classe che ha come attributi la cartella all'interno della quale scrivere dei file di testo. Il metodo write_file(to_write) scrive all'interno del file "results.txt" la stringa to_write.

3.3.2 ESECUZIONE DELL'APPLICAZIONE

I tipi di solver con cui ottenere una soluzione sono specificati nel file di configurazione alla voce "solvers". I possibili solver attualmente sono: "DLFS" (deep limited first search), "IterativeSearch", "BFS" (breadth first search), "BestFirstSearch".

Il metodo run(), cuore dell'applicazione, ottiene le informazioni dal file di configurazione, istanzia il problema e per ogni solver specificato istanzia un oggetto Solver per ottenere una soluzione con l'algoritmo specificato.

I risultati sono salvati in una cartella creata appositamente dall'applicazione.

I file salvati saranno:

- una rappresentazione della mappa
- una rappresentazione degli stati visitati dalla soluzione con un determinato algoritmo
- un file di testo con un resoconto sullo stato iniziale, sul goal e sulla descrizione delle soluzioni ottenute con gli algoritmi selezionati

Per l'esecuzione dovrà essere installata la libreria igraph, usata per la gestione dei grafi.