Autonomous Software Agents

Project Report

AGVZZ Group

Francesco Vaccari [239927]
francesco.vaccari@studenti.unitn.it

Simone Compri [239953]
simone.compri@studenti.unitn.it

Contents

1 Implementation of BDI

2 Single-Agent Architecture

2.1 Beliefs Sets and Beliefs Revision

Most of the belief sets implemented can be considered knowledge, facts that the agent knows certainly are true. Only with regard to parcels and agents, some information that might not be true is stored; this allows the agent to have some memory which is fundamental for planning. The belief sets implemented reflect all the variable elements of the environment:

- **Environment properties**: After connection of the agent, 3 important variables are set: parcels and agents viewing distances and the parcels decaying interval (if present). These values are used to implement visibility and decay of parcels and visibility of enemy agents.
- **The Agent Itself**: Each time the listener `onYou` is called, score and coordinates of the agent are updated.
- **Game Map**: After connection to the server, through `onMap`, the agent receives information about the cells that constitute the game map. Each cell is initialized and then stored into a matrix to use later during planning. The attributes of the cells describe their position in the map and whether they are deliverable cells, spawning cells, normale walkable cells or walls. Furthermore, each cell is initialized with an attribute `lastSeen` that counts how much time has passed since the agent has last seen the cell; this value is necessary to implement and plan the idle searching movement.
- **Parcels**: Each parcel is identified by an id and described by its position in the map, reward and whether it is currently being carried. Along with these values, we added a visibility attribute. This lets us keep track of parcels that are not currently within the viewing area of the agent to improve planning capabilities. Also, the decaying interval is used to update parcels that are not visible by the agent. Knowing the agent's viewing distance is useful also to correct inconsistencies in the parcels stored; for example, when an enemy agent picks up a parcel that is not currently visible, the agent will not know that the parcel is not there anymore until the old coordinates are seen by the agent. The revision of this belief set happens whenever `onParcelsSensing` is called.
- **Agents**: Similarly to parcels, each agent is identified by an id, and is described by its position, score and visibility. These values get updated whenever `onAgentsSensing` is called. The visibility attribute is computed by knowing how far away our agent can see enemies. An agent that is not visible for more than three seconds is deleted by the belief set, this allows us to remember the position of enemies in the recent past without really impacting future decisions since it's very likely that after that amount of time the last known position is no longer accurate.

As explained, the beliefs revision process happens asynchronously with respect to the computation and execution of the plan. This allows the agent to always have at hand the most recent and consistent information about itself and about the various elements of the environment.

2.2 Intentions and Intentions Revision

2.2.1 Utility Functions

2.3 Planning

This task is implemented into the related and It takes as input all the elements of the beliefset (map, agents and parcels list and the current agent) and, after the intentions are computed it creates the

plan that the agent is going to use. More precisely, the task that the solver have to solve consists into a BFS (Breadth-First Search) that:

- Is performed on the game map, considered as an undirected graph, where the cells are the nodes and the proximity to other cells describe the edges
- Uses the agent position as root
- Ends when the shortest path to the target cell is found

Tho compute the plan the planner relies on an online PDDL1.2 (Planning Domain Definition Language) online solver. As the standard declare, the model of the planning problem is composed by two major parts:

1. Domain description: define the predicates that characterize the environment and the action that can be performed
2. Problem description: describe objects in the environment, the initial state and goal descriptions that will characterize the current plan

2.3.1 Domain description

Since the tasks consists of a BFS, the domain file model the environment, composed by cells and the agent, into an undirected graph. More precisely, the domain is composed by:

- Predicates:
  - *cell ?x_y*: it indicates that the object *x_y* is a cell and a node of our graph
  - *near ?from_x_y ?to_x_y*: It models that the cell *from_x_y* is near the cell *to_x_y*, i.e., there is a unidirectional edge from the first to the second (in the Problem description section is explained how the graph become undirected)
  - *in ?x_y*: it keeps track that the agent is in the cell *x_y*. In the graph parallelism it refers to the node that the planner is visiting
- Actions:
  - *move*: it represents the agent movement through the map (i.e,, the graph). It takes as parameters two objects *from_x_y* and *to_x_y*, which respectively represent the starting and the target cell of the movement. Then it checks if the agent is in *from_x_y* (i.e., *from_x_y*), if the object *to_x_y* is a cell (i.e., *cell to_x_y*) and if the two cells *from_x_y* and *to_x_y* are close (i.e., *near from_x_y to_x_y*). After that, the effect of the action consists in moving the agent to the target cell (i.e., *in to_x_y*) and making sure that it is different from the starting cell (i.e., *not (in from_x_y)*

In the move action we avoid to put the precondition *cell from_x_y* because we assume that the starting point of the plan taken from the intentions is a cell. After that, thanks to the combination of the precondition *cell to_x_y* and the effect *in to_x_y* we know that all the subsequently objects are cells.

Here is the domain description just described:

```
;; domain file: domain.pddl
(define (domain default)
    (:requirements :strips)
    (:predicates
        (cell ?x_y)
        (near ?x_y ?x_y)
        (in ?x_y)
    )

    (:action move
        :parameters (?from_x_y ?to_x_y)
        :precondition (and (in ?from_x_y) (cell ?to_x_y) (near ?from_x_y ?to_x_y))
        :effect (and (in ?to_x_y) (not (in ?from_x_y)))
    )
)
```

2.3.2 Problem description

The problem description is created in two different phases. First of all, when the map is initialized, we visit the matrix and for each cell that isn't a wall we identify it as *c_x_y*, where x and y are its coordinates, and declare (i.e., add the proper objects and put the proper predicate in the *init* section) that it is a cell (i.e., *cell c_x_y*) and, for every neighboring cell (identified with the same notation) that is not a wall, we sign that they are close (i.e., *near c_x_y c_x_1_y_1*). Then, when we have to compute a plan, we look at what cell *c_x_y* the other officers are in and we treat each of them as walls (i.e., *(not (cell c_x_y))*). After that, we declare in which cell *c_x_y* the agent is (i.e., *in c_x_y*) and we set the target cell *c_x_1_y_1*, taken from the intentions as the goal of the plan (i.e., *in c_x_1_y_1*).
These two actions are performed always taking the problem description of the map described before and updating it at the current situation as just described.

2.3.3 PDDL example

Consider a situation like this:

The problem description is created is as follows:

```
(define (problem BFS)
    (:domain deliveroo)
    (:objects c_0_0 c_0_1 c_0_2 c_0_3 c_1_1 c_1_2 c_2_0 c_2_1 c_2_2 c_2_3 c_3_0)
    (:init
        (cell c_0_0) (cell c_0_1) (cell c_0_2) (cell c_0_3)
        (cell c_1_1) (cell c_1_2)
        (cell c_2_0) (cell c_2_1) (cell c_2_2) (cell c_2_3)
        (cell c_3_0)
        (near c_0_0 c_0_1)
        (near c_0_1 c_0_0) (near c_0_1 c_1_1) (near c_0_1 c_0_2)
        (near c_0_2 c_0_1) (near c_0_2 c_1_2) (near c_0_2 c_0_3)
        (near c_0_3 c_0_2)
        (near c_1_1 c_0_1) (near c_1_1 c_2_1) (near c_1_1 c_1_2)
        (near c_1_2 c_0_2) (near c_1_2 c_1_1) (near c_1_2 c_2_2)
        (near c_2_0 c_3_0) (near c_2_0 c_2_1)
        (near c_2_1 c_1_1) (near c_2_1 c_2_0) (near c_2_1 c_2_2)
        (near c_2_2 c_1_2) (near c_2_2 c_2_1) (near c_2_2 c_2_3)
```

```
        (near c_2_3 c_2_2)
        (near c_3_0 c_2_0)
        (in c_0_3))
    (:goal (in c_3_0))
)
```

And the plan that is created is:

```
(move c_0_3 c_0_2) -> down
(move c_0_2 c_1_2) -> right
(move c_1_2 c_2_2) -> right
(move c_2_2 c_2_1) -> down
(move c_2_1 c_2_0) -> down
(move c_2_0 c_3_0) -> right
```

2.3.4 Considerations

After the planner has computed the plan we translate it in the language that the agent can understand. In fact, we parse the coordinates contained into the objects *from_x_y* and *to_x_y* and compute if the movement corresponds to *up*, *down*, *left* or *right*. After that we append to the plan the intention *pickup* or *putdown* computed before.

Anyway, the choice to assign to the planner "only" the BFS task relies on two factors.

- The first one is regards the complexity of PDDL solver. In fact, since it is exponential in the number of actions and polynomial in the number of states, reducing the number of actions to only one (the move action) we have reduced the overall complexity. In this way the latency is considerably reduced, since the major complexity of the solver is simplified, and the remaining latency are the one given by the polyomial part of the solver algorithm and the one of the network
- The second one take into consideration how the PDDL planner works. Indeed, considering that starting from the starting cell (initial state) and moving through the nearby cells (action move) until it reaches the target cell (goal), the first plan found (i.e, the shortest) coincides with the shortest path bewteen the starting and the target cell

2.3.5 Offline version

Since the online solver adds a latency due to the network latency we created an offline version of the BFS method that works completely synchronously with the code and reduce considerably the time taken by the planner to compute the plan.

3 Multi-Agent Architecture

3.1 Communication

3.1.1 Initialization

In order to allow the two agents to exchange information about intents and beliefs, a communication needs to be established in the first few moments of the execution of the programs. A simple protocol, composed of 3 types of messages, allows the agents to recognize each other and discover the agent ids that will be used after the initialization is completed successfully.

The agents are assigned two different roles: sender and receiver. The sender forwards the first INIT message while the other agent waits to receive the request of initialization. After the request is received, a SECRET message is sent back in order to confirm the identity to the first agent which will respond with an OKAY message that will end successfully this first phase of communication. After these 3 messages are exchanged, both agents know the id of each other; this id will be used in all future exchanges of information.

3.1.2 Handling of messages

Once the communication is initialized, each agent will start sharing the information about its beliefs sets and intentions. A buffer is used to store temporarily the messages waiting to be processed by a function that modifies variables or information stored according to the type of the message received. The implementation of the handler affects the following information kept by the agents:

- **Friend Agent**: Each agent sends constantly its updated position, score, total reward of parcels carried and current target of the plan in execution. This allows better decisions while choosing for targets and avoids the generation of conflicting plans.
- **Enemy Agents**: Since each agent has its own list of enemies that it sees, when this list is updated a message is sent so that the other agent can utilize the information during planning.
- **Parcels**: When the belief set regarding parcels is updated, a message is sent in order to let the other agent know the most recent confirmed state of visible parcels.
- **Map**: The handling of this type of messages has the sole purpose of provide better information for the idle searching movement implemented in the planner. Each time an agent moves it sends information about the cells that are visible so that both agent's maps are consistent with each other.
- **Exchange Intention**: Whenever the planner decides to perform the exchange of parcels there is the need to communicate the intention to the other agent and to also update internal planning targets whenever they are not computed locally. Moreover, there is the need to handle the exit from the intention exchange whenever an error occurs or the process ends successfully.
- **Block Strategy Intention**: Similarly to the exchange intention, there is also the need to handle messages that allow the agents to coordinate when a planner decides to perform the blocking strategy.

3.2 Beliefs and Beliefs Revision

3.3 Intentions and Intentions Revision

3.4 Planning

4 Robe da fare

Roba da mettere:

Un agente

- Beliefs implementate
- Come abbiamo fatto belief revision
- Le intention che abbiamo considerato (pickup, delivery, idle(search))
- Implementazione delle intention e intention revision
- formulazione del plan con il pddl e cenno alla versione BFS

Due agenti

- come abbiamo gestito la comunicazione
- belief aggiunte rispetto alla versione con un agente solo
- come facciamo belief revision con la comunicazione per sincronizzare i due agenti
- Come abbiamo cambiato le intention singole rispetto ad un agente solo (non sceglie il target se è dell'altro agente)
- le intention e i multi-agent plan che abbiamo aggiunto
- come abbiamo gestito l'implementazione delle nuove intention
- anche qua dire sul pddl cosa abbiamo fatto

C'è da chiarire la questione dei desire e spiegare che magari noi abbiamo astratto solo le intention. Sarebbe carino fare degli pseudo codici per spiegare il while principale del planner per far vedere come abbiamo effettivamente gestito le cose. Anche potrebbe essere interessante spiegare le funzioni di utility che abbiamo creato, specialmente per la prima parte del progetto.