Artificial Intelligence, Multi Agent Systems

Albani Dario and Daniele Nardi

Overview

This project addresses multi agent cooperation, in particular, task assignment. The scenario is the management and control of wildfires, where a set of UAVs is deployed to automatically extinguish fires in a forest. In order to complete the project, you have to implement in JAVA your own protocol for multi-agent coordination of UAV fire fighting agents, that, given a grid world representation, must cooperate so that all the fires are extinguished.

To complete the assignment you have to:

i) Design a task allocation protocol for fire depletion under communication constraints, particularly you can choose among:
   a. Contract Net Protocol (CNP)
   b. Collective Decision Making (CDM)

ii) Design an inside-task exploration strategy based on Random Walks (RW)

Optional: extend the point ii) to manage an alternative exploration strategy based either on CNP or CDM.

If you work in a group, each member of the group has to implement a different method (i.e. CNP and CDM); comparisons among the two approaches are expected.

Expected outcome: working code and a brief report (up to 4 pages), containing an abstract description of the implemented approach and its experimental evaluation.

Problem Set Up

The setup for the project assumes that the area to patrol can be reduced to a squared 2D grid made of cells. There is a scout fixed-wing drone flying at high altitude that is able to see all the fires in the given area and to estimate the extensions of the former (only a rough estimation). This information is then transmitted in real time to all the other UAVs in the region, allowing them to share a coordinated knowledge of all the active tasks. On the other hand, due to payload constraint, the UAVs are not equipped with efficient communication hardware and have a limited communication range.

We associate the concept of task with the concept of fire; a group of neighbor cell undergoing a fire represents a task. A task has a centroid (i.e. the origin of the fire) and a radius (i.e. its expansion radius starting from the centroid), cells within the area of the task are cells eligible for cell selection. The UAVs will select a task among those communicated by the scout drone, and then they must handle the assignment of the task. When they get assigned a task they must navigate towards the task location. Since the scout communicates only the centroid and the extension of the fire, UAVs do not have any knowledge about the precise status of each single cell around the centroid and therefore they must have a strategy to act when they reach a cell on fire.

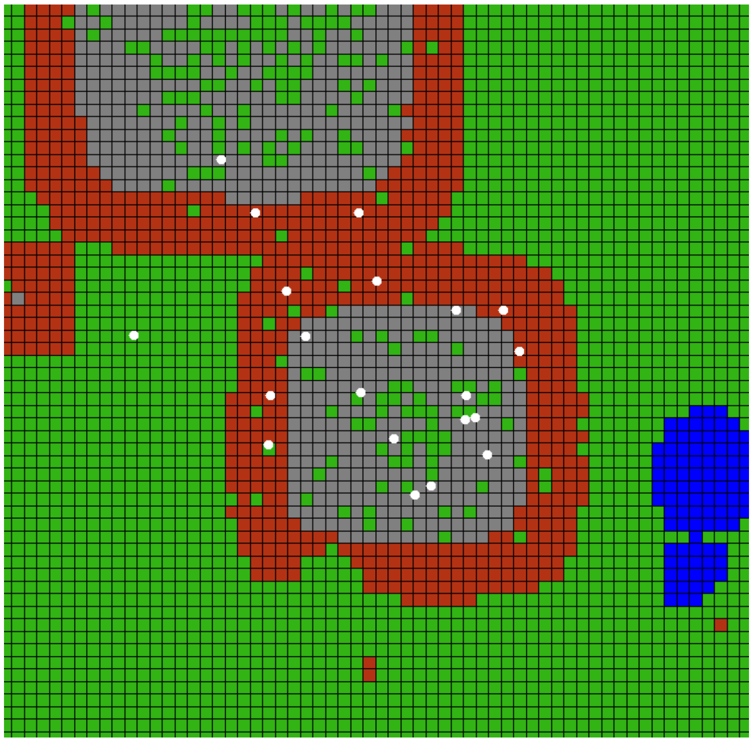

Figure 1, overview on the GUI of the simulation. Green square are either normal trees or foamed trees; blue squares represent water; red squares fire; grey cells are cells burned out with no possibility to be recovered.

World Description

The world is a two dimensional grid of size *w* and *h*. Each cell in the resulting matrix has the following important properties:

- **Position**, a pair *(x,y)* identifying the position of the cell in global coordinates
- **Type**, each cell assumes one of the following types:
    - NORMAL, cell is in a good shape and trees are safe
    - WATER, the cell contains water and cannot be ignited
    - FOAMED, a fire has been previously extinguished and trees are safe
    - FIRE, fire is devastating the cell
    - BURNED, the cell is completely lost

Cells normally start with status NORMAL. After a while, a fire starts to propagate, cells interested move from NORMAL to FIRE. Cells reached by UAVs in time, are treated with a special foam that can extinguish the fire. After the treatment, a cell cannot be ignited again; cells treated with the special foam move from FIRE to FOAMED (see action extinguish). After a certain time, a burning cell moves from FIRE to BURNED and cannot go back to the NORMAL status again.

Fires propagate from cell to cell in a non-constant manner. If cell at position *(x,y)* is on fire it will decrease the NORMAL status of all its neighbors until neighbors cells move from NORMAL to FIRE. Cells that are on fire are not affected by other fires (i.e. the transition from FIRE to BURNED only takes count of the cell itself).

Agents

An agent is a small unmanned aerial vehicle (UAV) (no need to model the fixed wing). Agents present an internal system for collision avoidance (i.e. you do not have to worry about collisions). Agents have limited perception range: the camera mounted on each UAV has the footprint that coincide with the size of a cell; i.e. agents perceive only the status of the cell above which they are located.

Agents store their own local knowledge as a collection of known cells. When a cell is inspected from a close range (i.e. at the level of a single cell), the agents automatically update their knowledge with the information about current location.

As previously specified, agents have **limited communication range**. You must consider this whenever you implement an action that requires to communicate with other agents (see the method *isInCommunicationRange* in class *UAV*). In the code, at the bottom of the class UAV, some signatures require you to implement your own communication protocol. Functions **sendData(DataPacket)** and **receiveData(DataPacket)** need to be implemented in order to allow the agents to share information about their behaviors (e.g. start an auction or signaling that a certain agent is joining a certain task).
Moreover, the function **retrieveAgents()** allows to query about the status of other agents in the field.

HINT (optional): To improve your implementation you can write an efficient communication protocol to send information to other agents, yet taking into account the communication constraints.

The primitive set of actions to use for your implementation are the following:

- SELECT TASK
- SELECT CELL
- MOVE
- EXTINGUISH

The selection of the next action to execute is already implemented in the code, see the method **nextAction()** in the class UAV.

Action selectTask() - Cost 1 time step**.** To be implemented!

Implements either CNP or CDM.
It allows the agents to allocate a task that requires attention (see the class *Task* and list *tasks* in the class *Ignite* for a better understanding of this concept).
Agents have to select a task within the set of available tasks coming from those issued from the fixed-wing drone (i.e. the list tasks in the main class Ignite).
Note that, is really important that you focus on the question: "how many agents should go for a task?". Thus, you have to define a heuristic that allows your team to deploy the right number of agents to every task (e.g. proportional w.r.t. the utility).

Action selectCell() - Cost 1 time step**.** To be implemented!
Implements Random Walks and, in the case you choose the optional strategy, CNP or CDM. It allows the agents to select a cell that requires inspection and might have FIRE status (see the class *CellType and WorldCell* for a better understanding of this concept). An agent has to select a cell within the set of available cells coming from those that lie in the area of agent's current task. If you decide to implement the optional strategy in ii), you can assume that a Task, as defined in the simulator, is decomposable and that a Cell, a WorldCell, is a simple task (i.e. non decomposable).

Action extinguish() - **Cost 10 time steps.** It starts to spray the fireproof foam at current location. After a fixed amount of time (10 time steps) the cell is covered with foam and the simulator changes the status of the cell from FIRE to FOAMED. Note that this action requires time and should not be performed when unnecessary.

Action **move(), Cost 5 time steps/cell.** It allows the agent to move toward its next target as set by the action select. This is a point to point navigation and do not allow intermediate waypoints.

Contract Net Protocol

1) Deliver a working implementation of CNP. If CNP is your choice, write your own custom methods that allow your system to bid on auction (refer to the slides for more details). The whole procedure is related to the action SELECT_TASK of the agents. A working implementation of CNP requires to implement the four main stages (remember that they take place all in one step of the simulation):
    a. **Announcement**: Manager sends a task description to all possible suppliers
    b. **Bidding**: Suppliers evaluate the offer and send a proposal to the Manager
    c. **Awarding**: Manager allocates the contract to the best supplier
    d. **Expediting**: The chosen supplier replies positively or negatively to the Manager

   HINT: try to avoid to make auctions for tasks that already have enough agents involved

2) Deliver your custom exploration strategy. The whole procedure is related to the action SELECT_CELL of the agents.

3) Vary the parameters of your simulation, to have statistical meaning, you should start at least 50 runs simulations for each parameters and with different seeds. You have to change:
    a. The number of agents involved in the simulation
    b. The communication range
    c. The number of initial fires
    d. And, if implemented, the single cell selection strategy

4) Estimate the performances of your approach and discuss possible solutions and/or improvements. Some comparison parameters might be the number of extinguished fires, the number of burned cells and more. It might be useful if you plot this information with a specific data visualization library (e.g. pyplot).

Collective Decision Making

1) Deliver a working implementation of a Collective Decision Making. If CDM is your choice, you have to write your own custom methods that allow your systems to perform decentralized decision based on the local knowledge of the agents. The whole procedure is implemented by the action SELECT_TASK of the agents.
   To deliver a working implementation of CDM you have to implement at least three utility-based concurrent processes: i) spontaneous commitment ii) spontaneous abandon and iii) at least one induced transition (e.g. recruitment, self or cross inhibition).

   HINT: For the spontaneous commitment, you can use a step function whose value is 1 if there is room for other agents in the task and 0 otherwise. A similar (but not same) procedure might be implemented also for other processes and might help you to easily manage the number of agents for each task.

2) Deliver your custom exploration strategy. The whole procedure is related to the action SELECT_CELL of the agents.

3) Vary the parameters of your simulation, to have statistical meaning, you should start at least 50 runs simulations for each parameters and with different seeds.
    a. You have to change:
    b. The number of agents involved in the simulation
    c. The communication range
    d. The number of initial fires
    e. And, if implemented, the single cell selection strategy

4) Estimate the performances of your approach and discuss possible solutions and/or improvements. Some comparison parameters might be the number of extinguished fires, the number of burned cells and more. It might be useful if you plot this information with a specific data visualization library (e.g. pyplot).

MASON Multi Agent Toolkit

In order to accomplish the project, you have to use MASON from GMU: http://cs.gmu.edu/~eclab/projects/mason/

You can find a detailed description of the simulator here (more than 300 pages): http://cs.gmu.edu/~eclab/projects/mason/manual.pdf

The code for the project, presents a custom class, outside the MASON toolkit, named FireController. The latter, continuously controls termination conditions of the simulation and kills the job, terminating the current simulation run, when such conditions are satisfied. While doing this, it generates a text file in MASON root directory where you have to store all the information and values that you deem of any importance for building your metrics.

To facilitate your job in evaluating the performances of your implementation, you can call MASON with the *–repeat x* flag, where x is the number of runs (see MASON specification for more details). Please note that this do not work with the GUI.

MASON requires JAVA and is working on all main operative systems and linux distributions. Below we provide details on how to install it on Ubuntu.

1) Be sure to have JAVA installed on your machine, both the oracleJDK and the openJDK will do the work

2) Clone the main git repo from https://github.com/eclab/mason.git by using the command **git clone**

3) Copy the project folder containing all the *.java files (as provided) to **mason/sim/app/**

4) Download all the required additional libraries from the MASON web site and store the jar somewhere in you pc (e.g. mason/lib)

5) **cd** to mason root folder. You will find a file named **Makefile**, edit the file with your preferred text editor and:

- Modify the string: **FLAGS = -target 1.5 -source 1.5 -g -nowarn** to **FLAGS = -target 1.8 -source 1.8 -g -nowarn** (i.e. change to java 1.8)

- Tell the compiler where libraries are, add to the above string the following parameter **-cp YourPathToLibs/*** (e.g. if you extracted the jar files into the MASON root and put them into a folder name lib: -cp lib/*)

- Add the complete path of your project below **DIRS = ** (line 16). Lines from 16 to 19 should look like

```
DIRS=\
sim/app/firecontrol/*.java\
sim/app/hexabugs/*.java\
```

6) go to MASON root folder and type **make** to compile the project

7) run the project from the MASON root folder by typing:

java sim.app.firecontrol.IgniteWithUI (WITH GUI)
java sim.app.firecontrol.Ignite (NO GUI)

7b) use **java sim.app.firecontrol.Ignite -repeat 50** to run 50 simulation (with different seeds)