Distributed Artificial Intelligence and Intelligent Agents

Homework 3

by **Rémi Domingues** & **Ahmed Kachkach**

Task 1: N-queens problem

We want to solve the “N-queens” problem in a distributed fashion, where each queen would be an agent.

First, let’s introduce some background about the N-queens problem:

Introduction

The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other.

This problem can be generalized to the N-queens problem with N agents on a NxN chessboard. We choose in our case to always take a regular chessboard (8x8) with N varying between 3 and 8.

Distributed n-queens

We want to solve this problem in a distributed fashion. One way to do so is by representing each queen as an agent, and making these agents communicate in order to solve the problem.

We don't want to rely on a centralized design where each queen communicate with a central authority that manages all queens, so we designed our system in a way that each queen only needs to communicate with the previously positioned queens.

Our design radically changed when we wanted to explore all possible solutions instead of just observing one solution, so we'll describe both solutions

Solving n-queens: design

Each queen possesses the following attributes:

```
private int mNumQueens;
private int mQueenIndex;
private AID mPreviousQueen;
private List<Position> mPositionnedQueens;
```

`mNumQueens` is the the "n" parameter of our problem (6 when solving the 6-queens problem).
`mQueenIndex` is the index of the queen agent (0 for an agent representing the first queen, 3 for the 4th queen, etc.)
`mPreviousQueen` is the Jade AID of the previous queen's agent. It is used to forward results to that queen when backtracking or when a solution is found.
`mPositionnedQueens` contains the positions on the chessboard of all previous queens.

Algorithm

Each agent, when created, starts by picking a position where he can't be threatened by other previously positioned queens, knowing that the ith queen is necessarily on the ith row.

If this is not possible, the agent sends a "no-possible-solution" message to the previous agent and… suicides. (unless it's the very first queen, in which case it announces that there is no possible solution for this problem; which is the case for n=2 and n=3)

When a position satisfies all the problem's conditions, we create a new agent representing the next queen and give him an index equal to the queen's index + 1, and the positions of previously positionned queens (including the one that we just found for the current queen).

If a satisfying position is found and the agent is the last queen, we announce that a solution was found by sending a "found-solution" message to the previous queen and "suiciding", and each agent will do the same all the way to the first queen that will display the solution and then terminate the program.

This design has many advantages:

- The agents only communicate with the previous agent, which has many advantages:
    - No need to store a list of all the agents' IDs
    - No need to warn all agents each time a queen is moved
- The agents are created/destroyed dynamically, which makes it easier to deploy (no need to pre-allocate the agents and pass over a list of IDs).

Getting all possible solutions

If we want to get all possible solutions, we can't just kill the queens once they get one possible solutions: they must make sure they tried

all possible positions before returning their result.

One possible solutions using the previous design is to send a “found-solution” message each time we found a solution, and just move to the next possible position (like nothing happened).

The problem with this solution is that we can have a high number of messages transmitted (as each message must go all the way to the first agent) and the agents can’t be killed anymore after they forward the solution.

The alternate solution that we used is to pass over a list of solutions that is updated each time an agent dies. This way the last queen agents adds elements to this list (when a satisfying position is found), and it is brought back when backtracking.

Results

Here are the number of solutions for different values of N:

n	1	2	3	4	5	6	7	8	9
Solutions	1	0	0	2	10	4	40	92	352

Note that our design works even for n=1, n=2 and n=3 (which includes two cases where no solution is possible)

Bugs

We noticed a problem that happens for high values of N: message reception sometime blocks the thread and the other agents can’t find the solutions. We’re not really sure what causes this (because it’s nondeterministic), but we never had this problem for lower values.

Task 2: Intra-platform mobility - FIPA Dutch Auction

Introduction

The aim of this task is to showcase how does the agent mobility work with multiple containers in the same platform. We aim at working with the following agents:

- **Auctioneer** : *MainCurator* is a **Curator** agent which aim at selling an item to the highest bidder from every platform.
- **Buyers** :
  - *Heritage Malta*
  - *Galileo Museum*

  Both buyers are **Profiler** agents and want to be the highest bidder in every auction located in every container. We can here consider that the buyers are galleries interested in items sold in an auction, since they want to present that item in their next exhibition.

Each agent start in a different container. Hence we assume the following agent distribution:

In order to allow our agents to communicate, each of them will follow the scheme below:

1. Creating N clones (N is the number of containers - 1)
2. Moving each clone in a distinct container
3. Attend / Present the auction
4. Wait for each clone to return to the parent's container
5. Display the best result obtained from the parent and its clones

The clones will follow this scheme:

1. Attend / Present the auction
2. Return to the parent's container when the auction has ended
3. Send the results to the parent

Common behaviours

CloningBehaviour extends OneShotBehaviour

This behaviour allows an agent to clone itself and send its clones to every other container than the current one.

We first send a *QueryPlatformLocationsAction* request to the AMS agent, which replies us with the list of every containers in the platform. Then, for each of those container, we use the method *doClone()* of our agent in order to clone it and send it to the specified container.

HomingBehaviour extends OneShotBehaviour

The aim of this behaviour is for a clone to move back to its parent's container.
Therfore, we first identify the parent's container by sending a *WhereIsAgentAction* request to the AMS agent. We then use the method *doMove()* of our clone in order to move to that container. Eventually, we invoke its method *sendResult()* to notify the parent of the clone's completion and share the result of the auction attended / presented in the remote container. The clone agent is then deleted.

Specific changes

Directory facilitator

In order to provide a reliable and dynamic implementation of our agent mobility, we used the directory facilitator to discover the services available in our container.

- ***Curator***

Each curator (clones and parent) publishes its auctioning service as a *CuratorAuctioneer* service to the directory facilitator in the *setup()* method.

- ***Profiler***

At its initialization, each profiler agent (clones and parent) regularly search for every *CuratorAuctioneer* service registered in the directory facilitator, until it finds one in its container.
We used a *WhereIsAgentAction* request to the AMS agent in order to identify the current agent's container and the curators' container.

Extended behaviours

Both the Profiler and the Curator agents had a *CyclicBehaviour* which waited for auction messages then chose the suitable action to execute (these are respectively *DutchAuctionBuyerBehaviour* and *CuratorAuctioningBehaviour*).

These behaviours now accept the *AgentMessage* (object containing a *String* describing the message type and a serialized content *Object*) with the *<result>* type. Those messages contain an *AuctionResult* object as a content, which represent the result of an auction and is defined by an artifact ID (Integer), the highest bidder name (String) and the selling price.

Once every result has been received and the parent's auction is over, this behaviour displays the global result and delete the current agent.

Profiler

- ***MainCurator***

The MainCurator *setup()* method add a *CloningBehaviour* to the current agent in order to clone itself and send the clone in the other containers

- ***Clones***

When a message having the *<auction-end>* type is received, a *HomingBehaviour* is added to the clone, in order to join the parent's platform.