Homework 1 **Kim Hammar**
ID2209 Due Date: 14 November 2016
Distributed Artifical Intelligence and Intelligent Agents

Problem Statement

The given task was to implement a very basic MultiAgentSystem (M.A.S) using the JAVA Agent Development Framework (JADE) [2], with the purpose of getting hands on experience with agent platforms and in particular the JADE agent platform. The assignment was also designed in a way to give experience programming agents in the context of a *practical scenario*.

Main problems and solutions

- *Connecting agents through a platform that can be used for interaction and service discovery*

  The JADE framework provides, among other things, a runtime environment where JADE agents can "live". The agents in the M.A.S for this assignment are distributed on different JADE containers that all are connected to a single platform, which enable the agents to utilize the JADE runtime environment to find and interact with each other.

- *Agent design (micro perspective): designing agents that can act autonomously in a given environment and make decisions.*

  Agents in this M.A.S perform tasks and perceive the environment to achieve their goals. Using JADE, the main mechanisms for designing and implementing agent tasks is through use of behaviours.

- *Society design (macro perspective): designing interactions for cooperation and coordination between agents in a M.A.S*

  The Foundation for Intelligent Physical Agents (FIPA) is used as the main Agent Communication Language (ACL) between agents in this M.A.S.

Connecting Agents

We refer to each running instance of JADE runtime environment as a Container, which can contain zero or more agents. The set of active containers is called a Platform and each platform have one container that is a special *main container* [1]. The most important aspect of the main container is to connect other containers together and to provide two essential services: **AMS and DF**, AMS (Agent Management System) is a naming service that ensures that each agent on the platform has a unique name. DF (Directory Facilitator) aka "The Yellow Pages" is a service where agents can *register* as providers of certain services and where agents can *search* for providers of specific services.

When running the M.A.S for this assignment the typical setup is to use 4 containers:

I **Main container**: required container for connecting the other containers. Does not host any agents directly in this setup.

II **Container 1**: Container where one ore more *curator* agents live.

III **Container 2**: Container where one ore more *tourguide* agents live.

IV **Container 3**: Container where one ore more *profiler* agents live.

Agent Design

There are three different types of agents in this system,

- **CuratorAgent**: Agent with the goal of monitoring an artgallery and respond to requests for art information from TourGuideAgents and ProfilerAgents. The CuratorAgent registers at the DF as a provider of the `artgallery-information` service, this agent perceives its environment mainly through the main container and its message-mailbox. The tasks/JADE behaviours of this agent are:
  - A `ParallelBehaviour` consisting of three `SubBehaviours`:
    * `GenreRequestServer` - A `CyclicBehaviour` that receives requests for a list of all genres of the monitored art gallery and responds to it.
    * `TourRequestServer` - A `CyclicBehaviour` that receives requests for a list of artifacts in the artgallery that matches a certain genre/interest and responds to it.
    * `ArtifactRequestServer` - A `CyclicBehaviour` that receives requests for details about a certain artifact in the artallery and responds to it.
- **TourGuideAgent**: Agent with the goal to build virtual tours upon requests from ProfilerAgents. The TourGuideAgent interacts with CuratorAgents to retrieve information for tours. The TourGuideAgent registers at the DF as providing the `virtualtour` service, and perceives its environment mainly through the main container and its message-mailbox. The tasks/JADE behaviours of this agent are:
  - A `ParallelBehaviour` consisting of three `SubBehaviours`:
    * `CuratorSubscriber` - A `SubscriptionInitiator` behaviour that subscribes to the DF service to receive notifications when new agents that provide the `artgallery-information` service (a service that is provided by CuratorAgents) registers.
    * `ProfilerMatcher` - An `AchieveREResponder` that receives requests from ProfilerAgents asking what kind of genres it can build virtual tours for. This behaviour is linked to the `FindSupportedInterest` behaviour (which is an `AchieveREInitiator`) that will be invoked when a request is received in order to: *(i)* ask discovered curator agents about their genres, *(ii)* build a list of all genres, *(iii)* respond to the requester with the list of genres.
    * `VirtualTourServer` - An `AchieveREResponder` that receives requests for virtual tours for specific interests/genres from profilers. When a request is received it will cause the `BuildVirtualTour` behaviour to be invoked which is an `AchieveREInitiator` that will send requests to all discovered curators and build a list of $\langle Artifact, Curator \rangle$ pairs that matches the given interest and finally it will respond with the built list to the requester.

- **ProfilerAgent**: Agent that maintains the profile of a user and that has the goal of travelling around the network and collecting interesting (from the user's point of view) information about art. The agent perceives its environment mainly through the main container and input from the user. The tasks/JADE behaviours of this agent are:
  - A `FSMBehaviour` consisting of 7 states and 10 different state transitions. The states are:
    * `INITIALIZE_USER_PROFILE` - An `OneShotBehaviour` that interacts with the user for initializing the user profile (only invoked if command-line aguments wasn't supplied)
    * `SEARCH_TOURGUIDES_STATE` - An `OneShotBehaviour` that queries the DF for a list of all agents that provide the `virtualtour` service.
    * `FIND_MATCHING_TOUR_GUIDES_STATE` - An `AchieveREInitiator` that sends request-queries to all found TourGuideAgents asking what type of tours they provide.
    * `SELECT_TOURGUIDE_STATE` - An `OneShotBehaviour` for presenting the found tourguides and the types of tours they offer to the user and letting the user choose a tourguide.
    * `FIND_VIRTUAL_TOUR_STATE` - An `AchieveREInitiator` that sends a request to a chosen tourguide, requesting a virtual tour matching the interest of the user.
    * `SELECT_ARTIFACT_STATE` - An `OneShotBehaviour` for presenting the virtual tour to the user and letting the user pick artifacts to visit.
    * `RETRIEVE_ARTIFACT_STATE` - An `AchieveREInitiator` that sends a request for details about the artifact to the curator of the artifact in the virtual tour that was selected. When the details are retrieved they are presented to the user.

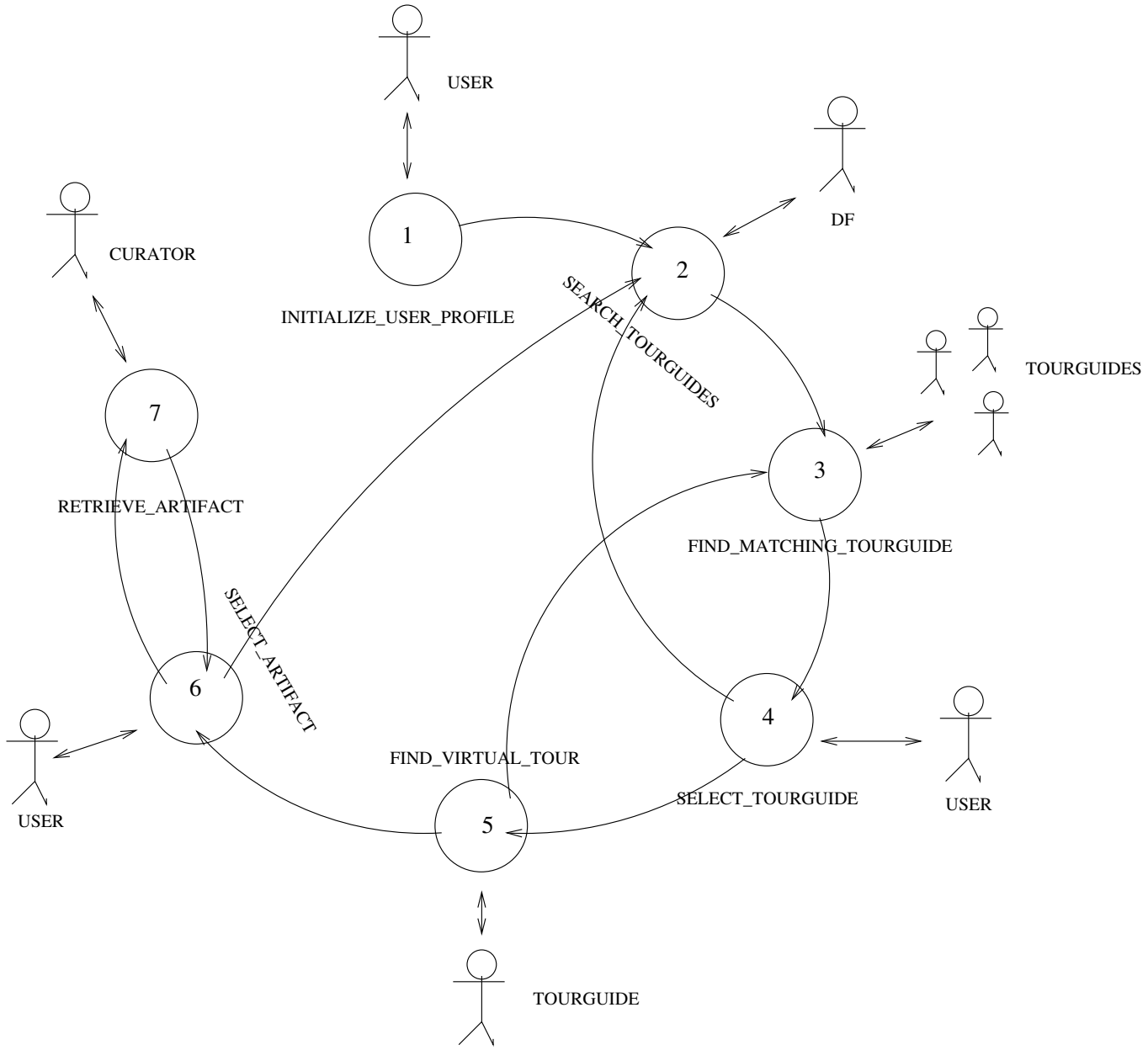

Figure 1: FSMBehaviour of ProfilerAgent

Society Design

From the presentation of the agent designs it should be clear that they need to interact in order to complete their individual goals. As mentioned, FIPA ACL is used for communication to gurarantee a consistent syntax of messages. The central concept for communication is that agents explicitly state a *performative-verb* together with the content of each message. The performative verb decides how the agent at the receiving side will interpret the message. FIPA defines a large set of performative verbs, the main ones used in this M.A.S are:

query-ref Used by one agent to determine the specific value for an expression [3], for example when a profiler agent queries tour guides for the types of virtual tours they support.

request Allows an agent to request another agent to perform some action [3], for example when a profiler agent requests a tourguide agent to build a virtual tour for a specific interest.

agree Used to indicate that the agent has agreed a request made by another agent [3], for example when a tourguide receivs a request to build a virtual tour it responds with a message with this performative before actually building the tour.

inform Basic performative for communicating information [3], used for example by an tourguide agent to inform a profiler after having successfully built a virtual tour upon request.

failure Indicates that an attempt to perform some action failed [3], used by tourguide agents to respond to profilers if they fail to build a virtual tour (for example if curators did not respond).

Conclusions

Building a M.A.S consisting of autonomous agents that interact with each other is different to building "regular" distributed systems. In this assignment the agents were *benevolent* towards each other which eased the development quite abit, since there were no need to be concerned about negotiations to solve conflicts. The main hurdle to overcome when developing this M.A.S as opposed to regular distributed systems was to understand how the agents interact through performative verbs, JADE libraries and predefined behaviours were useful in this aspect.

Attachments

Documented source code can be found in the attached zipfile. See README.MD in the root directory for instruction on how to execute and build the program.

References

[1] Giovanni Caire. Jade turorial. `http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf`, 2009. [Online; accessed 12-Nov-2016].

[2] Telecom Italia. Java agent development framework. `http://jade.tilab.com/`, 2016. [Online; accessed 11-Nov-2016].

[3] Michael Woolridge and Michael J. Wooldridge. *Introduction to Multiagent Systems, 2nd Edition.* John Wiley & Sons, Inc., New York, NY, USA, 2009.

Homework 2 **Kim Hammar**
ID2209 Due Date: 22 November 2016
Distributed Artifical Intelligence and Intelligent Agents

Problem Statement

Implementing the FIPA Dutch Auction Interaction Protocol [1] using the JAVA Agent Development Framework (JADE) [2] and laying out a theoretical model of the the game mechanisms involved in dutch auctions between autonomous self-interested agents. The purpose of the homework was to get further practice in developing MultiAgentSystems (M.A.S) and in particular M.A.S that involves *negotiatons* between agents.

Main problems and solutions

- *Implementing the FIPA dutch action protocol in JADE*
  - Agent design - Designing and implementing auctioneer and bidder agents.
  - Society design - Implementing the interactions of the protocol as defined by the specification [1].
- *Establish a theoretical model for a specific scenario and apply concepts from game theory*

  All possible strategies for the agents were considered to find which strategies are in Nash equilibrium.

Task 1: Implementation

Agent design

Two type of agents are used in the dutch auction scenario for this assignment:

- `ArtistManagerAgent`

  The `ArtistManagerAgent` is the *auctioneer* in this scenario; it auctions out artifacts from artists through dutch auctions. When carrying out a dutch auction the auctioneer tries to find the market price for a given good, i.e. the auctioneer starts out offering the good at some artificially high price and then continously lowers the price for each round until some bidder accepts the price or the price reaches the reserved price. The reserved price is the lowest price that the auctioneer is willing to sell the good for, if no bidder accepts that price then the auction is cancelled and the good is not sold.

  The `ArtistManagerAgent` is self-interested and wants to sell goods for as high price as possible to increase its personal revenue. For each auction i the `ArtistManagerAgent` will select a strategy s_i consisting of:

  I Initial price

II Rate of reduction (i.e. how much to lower price from one round to the next one if no buyer was found)

III Reserve price

Frankly, a self-interested auctioneer with infinite time would put the initial price arbitrary high and the rate of reduction arbitrary low since that would guarantee that the auctioneer would receive the highest price possible for a certain good. However if we assume that the auctioneer to some degree values his time, then he would start the auction at an initial price that is higher than what he expect that good to be sold for, but still within a realistic price-range. Further more, the auctioneer would choose a rate of reduction high enough such that participans in the auction that chose not to bid in a previous round might consider to bid in the next round.

The best strategy for the auctioneer is the strategy that optimizes the expected revenue. The expected revenue depends on the type of auction as well as the strategies of the bidders. Dutch auction gives best expected revenue if the agents use *risk-averse* strategies.

The `ArtistManagerAgent` is implemented with a `FSMBehaviour` with all the different states of a dutch auction, i.e.: `FIND_BIDDERS_STATE`, `OPEN_AUCTION_STATE`, `SEND_CFP_STATE`, `COLLECT_BIDS_STATE`, `MODIFY_PRICE_STATE`, `SELECT_WINNER_STATE`, `CLOSE_AUCTION_STATE`.

- `CuratorAgent`

  The `CuratorAgent` acts as a *bidder* in this scenario, it receives the current price of each round in the dutch auction and can choose to either bid and accept the price or to ignore the price (i.e. not bid). The agent is self-interested and wants to obtain artifacts that it is interested in for as low price as possible.

  Typically for each dutch auction i a participating bidder would have a personal valuation of the good that is being auctioned and then follow a strategy s_i that decides when to bid and when not to bid. The agent could be *risk-neutral* and bid for the auction at the first round that has an announced price less than or equal to its private valuation. The agent could also be *risk-averse*, if the true value of some good is unknown an agent might be willing to bid higher than its private valuation. Or, the agent could be *risk-inclined* and try to obtain the good for a price lower than its private valuation but with a increased risk of loosing the good to another bidder.

  The essential thing here is that the best strategy for a particular `CuratorAgent` depends on what strategies the auctioneer and other bidders choose. For instance if a curator agent knows that the auctioneer has a reserved price far lower than its own valuation of the good and that the other bidders will not bid unless the price reaches some even lower value, then the best strategy for the agent is to be *risk-inclined*. Unless the winner of the auction has complete information about the auctioneer's and the other bidder's strategies then it will always be susceptible to the winner's curse, i.e. the agent does'nt know if it should be happy with winning the auction or worried because he might have overvalued the good.

  The `CuratorAgent` is implemented with a `ParallelBehaviour` for receiving different messages of the FIPA Dutch Auction Interaction Protocol and taking the appropriate action.

Society design

The implementation follows the specification of the FIPA Dutch Auction Interaction Protocol [1] to the full, it uses exactly the same messages and interactions as the specification suggest. In case of multiple, competing and simultaneous bids the first-come-first-served principle is applied.

Task 2: Game mechanisms

To find a *pure-strategy* nash equilibrium we consider each possible combination of strategies, and for each combination we check whether this combination forms a best response for every agent.

Agents

$Agents = \{$ProfilerAgent (PA), ArtistManagerAgent (AMA), CuratorAgent (CA)$\}$

Actions

$Ac_{AMA} = \{$sell-high-quality (HQ), sell-low-quality (LQ)$\}$
$Ac_{CA} = \{$quote-based-on-demand (QD), quote-based-on-interest (QI)$\}$
$Ac_{PA} = \{$buy (B), not-buy (NB)$\}$

Outcomes

3 agents having 2 distinct actions each means $2^3 = 8$ different combinations of actions and outcomes.

$\Omega = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6, \omega_7, \omega_8\}$

Environment function

The outcomes depend on the *combination* of actions performed by the different agents, which can be modelled with an *environment function* τ.

$\tau(HQ, QD, B) = \omega_1 \quad \tau(HQ, QD, NB) = \omega_2 \quad \tau(HQ, QI, N) = \omega_3$
$\tau(HQ, QI, NB) = \omega_4 \quad \tau(LQ, QD, B) = \omega_5 \quad \tau(LQ, QD, NB) = \omega_6$
$\tau(LQ, QI, B) = \omega_7 \quad \tau(LQ, QI, NB) = \omega_8$

Utility functions

The utility function u_i maps outcomes to utility for Agent i. The utillities are based on the assumptions listed in the assignment.

ArtistManagerAgent

$u_{AMA}(\omega_1) = 1 \quad u_{AMA}(\omega_2) = -2 \quad u_{AMA}(\omega_3) = 1 \quad u_{AMA}(\omega_4) = -2$
$u_{AMA}(\omega_5) = 2 \quad u_{AMA}(\omega_6) = -1 \quad u_{AMA}(\omega_7) = 2 \quad u_{AMA}(\omega_8) = -1$

Preference ordering of ArtistManagerAgent for the different outcomes:
$\{\omega_5, \omega_7\} \succ_{AMA} \{\omega_1, \omega_3\} \succ_{AMA} \{\omega_6, \omega_8\} \succ_{AMA} \{\omega_2, \omega_4\}$

CuratorAgent

$u_{CA}(\omega_1) = 1 \quad u_{CA}(\omega_2) = 1 \quad u_{CA}(\omega_3) = 1 \quad u_{CA}(\omega_4) = 1$
$u_{CA}(\omega_5) = 1 \quad u_{CA}(\omega_6) = 1 \quad u_{CA}(\omega_7) = 1 \quad u_{CA}(\omega_8) = 1$

ProfilerAgent

$u_{PA}(\omega_1) = 2 \quad u_{PA}(\omega_2) = 0 \quad u_{PA}(\omega_3) = 2 \quad u_{PA}(\omega_4) = 0$
$u_{PA}(\omega_5) = 1 \quad u_{PA}(\omega_6) = 0 \quad u_{PA}(\omega_7) = 1 \quad u_{PA}(\omega_8) = 0$

Preference ordering of ProfilerAgent for the different outcomes:
$\{\omega_1, \omega_3\} \succ_{PA} \{\omega_5, \omega_7\} \succ_{PA} \{\omega_2, \omega_4, \omega_6, \omega_8\}$

Since the CuratorAgent cannot affect the outcome and has a role of a simple intermediate point in the buying and selling between ArtistManagerAgent and ProfilerAgent, we can neglect it in the analysis of strategies and nash equilibriums.

There exists a *dominant* strategy for the ProfilerAgent, which is to always buy, and furthermore there exist a dominant strategy for the ArtistManagerAgent aswell which is to always produce a low quality product (it follows from the assumption in the assignment that the ProfilerAgent does not know the quality of the product until after buying it). The payoff matrix between the agents looks like the following.

		ArtistManagerAgent	
		HQ	LQ
ProfilerAgent	B	$(2, 1)$	$(1, 2)$
	NB	$(0, -2)$	$(0, -1)$

As the matrix shows, there is a single nash equilibrium, namely the actions: (B, LQ). Assuming that ProfilerAgent chooses to buy, the best strategy for ArtistManagerAgent is to produce/sell a low quality product and likewise assuming that the ArtistManagerAgent chooses to produce/sell a low-quality product the best strategy for the ProfilerAgent is to buy, i.e. neither agent has any incentive to deviate from the equilibrium. Note that the nash equlibrium in this case is also Pareto efficient.

The strategy for finding the nash equilibrium is to eliminate dominated outcomes, since there exists a dominant strategy for both of the agents we can exclude all outcomes that are dominated, because in either of the dominated outcomes there is atleast one agent will be better of choosing another strategy, no matter what the other agent chooses.

		ArtistManagerAgent	
		HQ	LQ
ProfilerAgent	B	~~$(2, 1)$~~	$(1, 2)$
	NB	~~$(0, -2)$~~	~~$(0, -1)$~~

Conclusions

Auctions are mechanisms to reach agreement on the issue of allocating resources between entities and can be used just as well for agents as for humans. Dutch auctions are a special type of auction that is *open-cry descending*, which requires minimal communication between bidder and auctioneer agents to reach agreement on allocating resources.

In a M.A.S with self-interested agents that are operating in the same environment, finding the *best decision* for each agent in many cases resembles studies from game theory. Concepts like dominant strategies, Pareto efficiency and Nash eqilibrium can often be applied when analyzing interactions between self-interested agents in a M.A.S.

Attachments

Documented source code can be found in the attached zipfile. See README.MD in the root directory for instruction on how to execute and build the program.

References

[1] Foundation for Intelligent Physical Agents. Fipa dutch auction interaction protocol specification. `http://www.fipa.org/specs/fipa00032/XC00032F.pdf`, 2001. [Online; accessed 18-Nov-2016].

[2] Telecom Italia. Java agent development framework. `http://jade.tilab.com/`, 2016. [Online; accessed 11-Nov-2016].

Homework 3 **Kim Hammar**
ID2209 Due Date: 29 November 2016
Distributed Artifical Intelligence and Intelligent Agents

Problem Statement

Implement the following using the JAVA Agent Development Framework (JADE) [1]

Task #1 M.A.S for solving the N-Queens problem where agents coordinate with each other to choose the right positions.

Task #2 M.A.S with mobile agents that can perform dutch auction in different places. Mobility means in this context that agents have the possibility to move between containers.

N-Queens

Agent Design

The M.A.S for task #1 consists of only one agent-type, the `QueenAgent`. The `QueenAgent` is designed to be reactive and receiving messages from other agents, upon receipt of a message from another queen, the agent will compute a "safe" position on the board according to the following algorithm (randomness is used to be able to find different solutions to the same puzzle):

Algorithm 1 QueenAgent algorithm for selecting a slot on the board

Require:
B ▷ Board
id ▷ Id of the agent
T ▷ Safe positions on the current board that have already been tried
Ensure:
p is the selected safe position which have not previously been tried. If no safe position is found, $p = -1$.
procedure SELECT_SAFES_SLOT(B,id,T)

```
R ← Shuffle(B[id])          ▷ random shuffle the row of possible positions
p ← −1
for r ∈ R do
    i ←indexOf(r)
    if safeDiagonally(i) ∧ safeVertically(i) ∧¬i ∈ T then
        p ← i
        break
    end if
end for
end procedure
```

Society Design

The `QueenAgent`'s will take turn selecting slot on the board, the first queen will initialize the puzzle by selecting a slot on the board and then notifying the *next* queen to indicate that it is its turn to select a slot. If a queen finds a safe slot and there is no queen left that have'nt selected a slot, the puzzle is solved. Further more if a `QueenAgent` fails to find a safe slot on the board it will notify the *preceeding* agent about this and ask it to change its slot on the board. If an agent fails to find a safe slot on the board that have'nt already been tried and there is no preceeding queen to notify, the puzzle is considered unsolvable. In order for the queens to find each other they register at the `DirectoryFacilitator` (DF).

Mobile Agents

Agent Design

The `CuratorAgent` and `ArtistManagerAgent` in this scenario uses the same behaviours as for Homework2 with a few modifications:

- Both agents runs in parallel to their other behaviours, a cyclic behaviour `ReceiveCommands` that receives commands from a controller agent to do one of the following: (*i*) move to a container (*ii*) clone itself (*iii*) kill itself
- Both agents use GUI's for interacting with the user and start/stopping auctions rather than the command-line as used in Homework2.
- The `ArtistManagerAgent` will run two cyclic behaviours `ClonesServer` and `AuctionResultServer` in parallel with other behaviours for receiving auction-results from clones and for receiving the final winner-bid from parents, respectively. Consequently, after finnishing an auction, if the `ArtistManagerAgent` is a clone and has a "parent"-agent, it will send the result to that agent and if a `ArtistManagerAgent` is a parent, it will collect results from clones, choose a winner, and notify the clones about the winner.
- Auctions are ran locally on containers, not globally on the platform.

In addition to the `CuratorAgent` and `ArtistManagerAgent`, a third agent called `ControllerAgent` is used. The `ControllerAgent` and associated `ControllerGUI` is inspired from the example code at the tutorial [2] that was recommended as a guide for this assignment. The `ControllerAgent` creates containers as well as agents and issues commands to existing agents to move, clone or die. The `ControllerAgent` communicates with the user through the `ControllerGUI`.

Society Design

The dutch auction interactions from Homework2 have been extended for intra-platform mobility where auctions are performed locally on containers instead of globally on platforms. Further more the results of auctions are forwarded by clones to parent-agents which will synthesize the results and present the best price from its point of view, the result is then forwarded to the clones who will notify the bidders about the result. The interactions between bidders and auctioneers in the actual auction is identical to the interactions presented in Homework2.

ControllerAgent communicates with the `AgentManagementSystem` (AMS) to retrieve a list of all containers on the platform. The ControllerAgent also communicates with agents that it have

created by sending simple one-to-one commands to agents, where no reply is expected. The commands that might be sent from the ControllerAgent to created agents are: clone, move, kill.

Conclusions

Extending simple agents to be intra-platform mobile allows for designing agents with further capabilities for acting autonomously. The agents for this homework were limited to mobility within a single platform but the same design principles could be applied for inter-platform mobility if the underlying agent-architecture allows for it.

Attachments

Documented source code can be found in the attached zipfile. See README.MD in the root directory for instruction on how to execute and build the program.

References

[1] Telecom Italia. Java agent development framework. `http://jade.tilab.com/`, 2016. [Online; accessed 11-Nov-2016].

[2] Jean Vaucher and Ambroise Ncho. Jade tutorial and primer. `http://www.iro.umontreal.ca/~vaucher/Agents/Jade/JadePrimer.html`, 2003. [Online; accessed 26-Nov-2016].