Applied Artificial Intelligence Project

Group 5

Alessandro Amici · Alvaro Rubio Gomez · Pietro Bolcato · Kai Jeggle

21.07.2019

Abstract The 2019 edition of the Multi-Agent Programming Contest raised a whole new set of challenges as the scenario changed completely compared to the one presented in the previous years. The complexity of the new simulation and the variety of tasks posed a problem suitable for a fully distributed multi-agent approach. This paper describes a possible approach to solve the problem as well as encountered issues and possible improvements for future works.

Keywords Multi-Agent Programming Contest; · Fully Distributed Multi-Agent Systems; · ROS

1 Introduction

Distributed systems perform computation by having multiple independent processes exchanging messages with each other. The resulting computation gives the illusion of a single, coherent system. It is a field of interest for many modern applications as they often rely on different interconnected components. The solution presented was developed in the context of a TU Berlin University course on Applied Artificial Intelligence. The team is an international group composed of four members with a computer science background.

1.1 MAPC 2019

The scenario of the *Multi-Agent Programming Contest 2019* [2] consists of two teams of agents moving on a grid map. The goal is to explore the simulation environment and acquire blocks to assemble them into complex patterns in order to complete tasks and earn points to win the simulation. The agents can perceive

A. Amici · A. Gomez · P. Bolcato · K. Jeggle
DAI Labor, Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin / Germany
Tel.: +49 30 314 74000
Fax: +49 30 314 74003

their environment with a vision range of 5 cells in a diamond shape. The complexity of the tasks requires the agents to collaborate and synchronize via a communication protocol.

1.2 RHBP Framework

A framework called *ROS Hybrid Behaviour Planner* [3] has been developed at the chair of *Agent Technology (AOT)* at TU Berlin in the last few years as base for solving the MAPC Contest. It provides a bridge interface between ROS and the MAPC simulation server as well as implementing a versatile task-level decision-making and planning mechanism. It combines adaptation capabilities with general decision-making and planning as well as coordination capabilities through the implementation of a hybrid behavioural network. Our strategy is built on top of the framework, as explained throughout the following sections.

2 System Analysis and Design

2.1 Distributed Strategy

Following completely the spirit of the MAPC Contest we realized a fully distributed multi-agent system. This means that each agent has its own perception of the map, the progress of the tasks, and the information received from other agents. There is no central logic, no central point of failure, no omniscient agent that dictates the perfect strategy. Unfortunately the time was not enough to think about an optimal distributed consensus algorithm, and that is why our strategy in computing different behaviours is clearly not the absolute optimal. However our agents can complete their tasks in finite time collaborating and avoiding obstacles without getting stuck.
Every agent has its own RHBP network that defines its possible actions with preconditions, effects and goals. In our final submission the goal is only one and permanent and drives the agents to make as many points as possible.
The explore behaviour is responsible to steer the agent to its local map's borders. It looks for the best value of unknown cells per step and it traces a path that will eventually discover the map completely. However, it may stop much before, because once the agent discovers the goal area and enough dispensers, it can start doing tasks in collaboration with other agents.
Once the goal area is discovered the agents will place bids for all available subtasks in the environment. The bids are based on the amount of steps needed to fulfill a specific subtask. The most efficient agents will then be selected by everyone and once all the subtasks of a task are assigned, the agents will start progressing to accomplish the task and earn the points.
Each agent will then go to the correct dispenser, dispense a block, attach to it, meet at a common meeting point with the other agents who are assigned to a subtask of the same task and connect all blocks. After that one agent will move to the goal area and submit the task, while the others will detach and continue exploring or start another task.

3 Software Architecture

The most up to date version of our code can be found in the master branch of [1]. In the following the structuring of the code base and the reasoning behind it is explained.

3.1 ROS Packages

The code base contains three ROS packages - *manual_player_package, mapc_rhbp_manual_player* and *strategy_1*. The first two are related to the manual player implementation which was used at the beginning of the project to get to know the simulation environment and the functionalities of the agents. The current agent strategy is implemented in the latter.

3.2 Shared Functionalities

Most of the functionalities we use in our agent strategy (*strategy_1*) can be found in */commons*. The reason why this is outsourced from the agent strategy itself is that the functions and classes can be imported to different strategies, e.g. *mapc_rhbp_manual_player* or possible future strategies.
All functionalities that directly relate to the agents itself - mainly the behaviour model and the perception provider - can be found in */commons/agent_commons*.
possibly elaborate more
The implementation of the classes related to the auctioning system, the communication system, the task representation, mapping and path planning is located in the directories */commons/classes/auctioning, /commons/classes/communication, /commons/classes/mapping* and */commons/classes/tasks*. For the whole implementation we follow the paradigm of one file per class and separation of concerns. Variables, that need to be accessed from various classes are defined in */commons/global_variables.py*.
A collection of unit tests covering core functionalities of the mapping can be found in */commons/tests*. In total we produced 6035 lines of python code.

4 Implementation

In this section the core logic of the main components that are needed for the agents' strategy are presented. For further details see [1]

4.1 Communication

The communication mechanism covers a fundamental role for the solution of the problem as it enables the agents to share data and information between each other. The general idea is to share specific types of information through ROS topics between the agents, while allowing the flexibility of having custom callbacks. The implementation is realized in the class */commons/classes/communication*

/communications.py, which contains all the methods used to start the topics, send the messages and synchronization. The callbacks are logically separated from the Communication class itself and they're implemented in the classes */commons/classes/auctioning/auction.py* and */commons/classes/mapping /map_communication.py*.

The following communication messages are implemented:

- Map communication: allows the agents to share their map to others in order to merge them and have a global representation of the whole simulation environment as soon as possible. The communication is achieved through a custom ROS message composed of:
  - message_id (*string*): randomly generate unique id
  - agent_id (*string*): the agent that sent the map
  - map (*string*): the local map of the agent
  - rows, columns (*int32, int32*): the number of rows and columns in the map being shared
  - lm_x, lm_y (*int32, int32*): x and y position of the top left corner of the common landmark. This will be used as common origin point for the map merge

- Auction communication: allows the agents to share their bids for a subtask. The communication is achieved through a custom ROS message composed of:
  - message_id (*string*): randomly generate unique id
  - agent_id (*string*): the agent that sent the bid
  - task_id (*string*): the id of the subtask
  - bid_value (*int32*): the value of the bid
  - distance_to_dispenser (*int32*): distance to the closest dispenser of the type required by the subtask
  - closest_dispenser_position_x, closest_dispenser_position_y (*int32*, *int32*): x and y position of the closest dispenser of the type required by the subtask

- **Subtask update communication:** allows an agent to share an update for a subtask status. The communication is achieved through a custom ROS message composed of:
  - message_id (*string*): randomly generate unique id
  - agent_id (*string*): the agent that sent the status update
  - command (*string*): the command that triggers the update (eg: "done")
  - task_id (*string*): the id of the subtask to be modified

  Though not used in the current implementation, a personal communication mechanism is implemented. This enables two agents to share message between each other. The communication is achieved through a custom ROS message composed of:
  - message_id (*string*): randomly generate unique id
  - agent_id_from (*string*): the agent that sent the message
  - agent_id_to (*string*): the agent that receive the message
  - message_type (*string*): the type of message sent (eg: "connect", "information")

- params (*string*): the parameters to perform actions (eg: if the *message_type* is "connect", the param could be "north")

All the personal messages are published in a shared ROS topic between all the agents, but they ignore the messages unless they are the intended receiver of the message.

A basic synchronization mechanism is also implemented so that the sending of a personal message can be delayed until a message is arrived. In this way a naive question-answer system can take place.

4.2 Mapping (Kai)

The creation of a map representing the environment is the foundation of the agents ability to navigate through the simulation. The implementation of the mapping can be found in */commons/classes/mapping/grid_map.py*. The simulation environment is a two-dimensional grid and thus can be easily represented by a matrix of integers, where different predefined integers denote different properties in the map. The following properties are represented in the map: *obstacles*, *dispensers* of various types, *blocks* of various types, *entities* (other agents) and *goal cells*. Additionally we denote empty cells, unknown cells and the agents position itself. The full map specification can be found in */commons/global_variables.py*.
We differentiate between static and dynamic properties, where dynamic properties (*blocks and entities*) can change their position in the map over time and static properties (*obstacles, dispensers and goal cells*) have a fixed position. In the simulation it is possible for dynamic properties to move over static dispensers. Due to this circumstance we need two representations of the map, one with the static properties and one with the dynamic properties which is used for navigation, i.e. path planning (4.4)
In the first simulation step the map gets initialized with a quadratic matrix of length *(agent vision * 2) + 1*, where the center of the matrix is the position of the agent. In the map we have two points of reference: the origin of the matrix, which is always the top left cell of the matrix and the origin of the agent, which is the point where the agent was initialized. We can translate any given position between the two reference points with the methods *_from_relative_to_matrix()* and *_from_matrix_to_relative()*
Every simulation step the map gets updated based on the perception of the agent and its previous actions. If the agent moved in the last step, the position of the agent is updated in the matrix and if the agent's vision exceeds the current bounds of the map representation a row or column gets added to the matrix. By adding a row to the top of the matrix or a column to the left of the matrix, the matrix' origin gets shifted. This is also the reason why it is necessary to have a second fixed point for referencing the position of map properties - namely the origin of the agent.
For debugging purposes the map matrix of every agent is written to a text file and then periodically plotted. Thus we have a live representation of every agents map. The code for the live plotting is located in */commons/map_live_plotting.py*

4.3 Map Merging

A map merging algorithm is used to come up with a common global map for all agents as the result of merging all local maps. The implementation of the map merging is located in */commons/classes/mapping/map_merge.py* To merge the local maps of different agents a common landmark is needed. For simplicity, the landmark chosen is the top left corner of the goal area. This decision was made based on the assumption that the goal area is unique and due to the necessity of discovering the goal area as a common requirement for all agents to be able to complete tasks.

The map merging algorithm consists of the following three steps:

1. Finding the goal area: Agents explore the map by following the exploration behavior (4.7.2)
2. Share local maps: Agents that have discovered the goal area share their local maps through a ROS topic, as well as the common landmark in their own relative coordinates.
3. Merging maps: Agents receive the local maps from other agents every time an agent shares it through the ROS topic. The local maps get loaded into a buffer to avoid synchronization issues. The maps are then merged based on the position of the common landmark and the agents add all new information to their local map.

4.4 Path Planning

The path planning is based on a well-known path planning algorithm called *A Star* [4].
The main idea of this algorithm is to find the greediest way get from one point to another. To do so, the algorithm starts generating nodes from a starting position and calculates their cost. Nodes are possible consecutive steps and their cost is calculated as follows:

$$\textit{Cost of node} = \textit{Distance from starting point} + \textit{Distance to end point} \quad (1)$$

We based our path planning on an open source code [5] and modified it for our particular purpose in */commons/classes/mapping/grid_path_planner.py*

The modifications made were:

- Movements are constrained to the four cardinal points (north, west, south, east), i.e no diagonal movement.
- Added clockwise and counterclockwise rotations to the possible movements.
- Added validator to check for invalid end points.
- Added validator to understand if a node is blocked due to the existence of an obstacle (walls, entities, blocks).
- Added a translator between consecutive path positions and its analog in cardinal points or rotation moves (counterclockwise, clockwise).

4.5 Task Representation

The goal of an agent team is to complete tasks provided by the simulation. A task is the arrangement of two or more blocks - of a certain type - to a predefined shape. Each block of a task defines a subtask, which needs to be completed by an agent (4.7.3).
Every simulation step the agent perceives all available tasks and updates its list of current tasks with the newly available tasks. Each task gets decomposed in its subtasks and assigned to an agent through the auctioning process (4.6. The implementation of the task -and subtask representation and the updating process is located in *task.py, subtask.py* and *update_tasks.py* in */commons/classes/tasks*.

4.6 Auctioning

The auction system is used to assign the different subtasks to the agents. The algorithm is designed such that the approach is fully distributed. In general, it is divided in four steps:

1. Send the bids: Every agent calculates the bid value for the currently analyzed subtask. The bid value for a subtask is calculated based on (2).

$$\textit{bid value} = \textit{distance to dispenser} + \textit{distance from dispenser to meeting point} \tag{2}$$

   The cost calculation is iterated and summed over every subtask already assigned to the agents in order to calculate the final bid value. It is important to point out that if the agent has not discovered the goal area, it places an invalid bid of -1. The computed final value is then sent to the shared ROS topic (4.1).

2. Wait for the bids: After the bid value is calculated and sent, the agents wait for the bids of all the other agents or for a safety timeout. The timeout is to avoid that the agents get stuck waiting indefinitely for the other agents' bids in the case of a crash of one of the agents. The bids are received in the ROS message callback and saved in a dictionary of bids.

3. Compute the assignment: The dictionary with the bids is iterated and every subtask gets analyzed in the following way: If an agent is not already assigned to a subtask of the same parent task and there's at the least one valid bid, the agent with the lowest bid is virtually assigned to the task. If all subtasks of a parent task are successfully virtually allocated, a dictionary with the allocation informations is returned.

4. Execute the assignment: The last step modifies the subtask object and effectively assigns the agents based on the information of step 3.

4.7 Behaviour Model

The behaviours are responsible for the actions the agent chooses to execute in a specific point in time in the simulation. Each behaviour is conditioned by one or more sensor values and has an effect on one or more sensor values when the behaviour is completed. The goal of the agent is to earn as many points in the simulation as possible. Points are earned by completing tasks, which consist of several subtasks (4.6) .
All behaviours get initialized in */strategy_1/src/rhbp_agent.py* and every behaviour is defined in its own module in the package */commons/agent_commons /behaviour_classes*.

4.7.1 Sensor Manager

The sensors for all behaviours are organized in the module */commons/agent_commons/sensor_manager.py*. The sensor values get updated every simulation step based on the simulation environment and the actions executed in the previous simulation step by the agent.

4.7.2 Exploration Behaviour

The exploration behaviour is activated at the beginning of the simulation when there is no subtask assigned to the agent yet. To get a subtask assigned the agent must have discovered the goal area through exploration. The goal here is to explore the whole simulation map. The point with the highest exploration score in the map is chosen as the next point to explore and the agent navigates to that point.

$$exploration\ score = \frac{unknown\ cells\ in\ vision\ range\ of\ the\ point}{distance\ to\ the\ point} \tag{3}$$

4.7.3 Task Completion Behaviours

Once the agent has discovered the goal area and dispensers, it will place valid bids in the subtask auctioning (4.6). As soon as the agnet is assigned to a subtask, it executes the behaviours necessary to complete the subtask. The conditions are designed in a way that only one behaviour is activated at a time and the behaviours are activated in the sequential order given below:

1. Move to dispenser
2. Dispense block
3. Attach to block
4. Reach meeting point
5. Connect blocks
6. Detach from block
7. Go to goal area
8. Submit task

Behaviours *1.* - *3.* are straight forward - the agent localizes the closest dispenser of the same type as the current subtask, navigates to it, dispenses a block from

it and then attaches to that block. Then the meeting point with the other agents needs to be calculated (4.8) and the agents navigate to the meeting point. Once the agents arrived at the meeting point (*4.*), they connect the blocks (*5.*) and all but one agent detach from their blocks (*6.*). For agents executing the detach behaviour the subtask is complete and they start executing the next subtask in the queue. The agent, that is still attached to its block is now in fact attached to the whole shape of blocks and moves to the goal area (*7.*) where the task can be submitted and the reward for the task earned. The submit behaviour is always done by the agent that is assigned to the subtask at the origin of the task (the task next to the red dot in the GUI). On the task submission (*8.*) the last subtask is complete and since all subtasks were completed, the whole task is complete.

4.8 Meeting Point

The meeting point algorithm seeks for a common meeting point where the agents that complete the subtask of a common task will place themselves and their attached blocks to create the shape needed for completing a particular task.
The logic of the meeting point algorithm is implemented in the method *meeting_position()* that is part of the *GridMap* class in the module */commons/classes /mapping/grid_map.py*. It returns the meeting position for a single agent and its blocks attached around a common meeting point between all agents.

The method relies on three other class methods of *GridMap*:

- *create_figure()*: This method creates a list of arrays representing the final shape of the blocks and the agents assigned for each block of this shape.

- *get_common_meeting_point()*: This method computes a common meeting point where the agents can group to compose the shape for a particular task. This meeting point is uniquely computed by each agent depending on the distance to the closest dispenser (for completing their subtask) and the distance between this dispenser and the other agents' closest dispensers.

- *agent_position_in_figure()*: This method finds a possible shape composition (list of pairs of agents and its blocks attached) around the common meeting point for a specific task. It establishes the common meeting point to be the position of the submitting agent and it fixes the position of the blocks in the shape of blocks with respect to the submitting agent. Then, it tries all possible compositions by changing the position of the other agents around their blocks attached, until it finds a shape composition that is not blocked in the map.
  For now the computations in *get_common_meeting_point()* and *agent_position_in_figure()* are done by every agent individually and the results are not shared between the agents. This requires that all agents have a perfect representation of the map. This approach is quite error prone and will be enhanced with inter agent communication in the future.

5 Evaluation

Due to lack of time and valid competitors, it was not possible to test our solution against other teams. It is however clear to us that we are not ready for an actual competition, because at the actual state, we managed to make our team of agents accomplish tasks, but with strong limitations such as:

- max number of agents: 2
- max number of blocks per task: 2
- dimension of map: tested until 20x20

Our system is designed to pass all this limits, however, due lack of time we managed to implement and confirm that our solution is stable only under this constraints. Unfortunately, we are now facing a bug we were not able to solve yet. The bug causes that at a certain point the internal representation of the map of an agent gets incorrect, making the agent crash.
Before this problem occurs, however, our implementation is very efficient, able to take a decision always in less than a second (tested on a Dell XPS 15-2018). Moreover, the agents are very efficient in finding the best way to reach any point and avoiding obstacles thanks to our A* implementation.
Given the short time given, we consider our results to be promising. We know we are very close to make it work with any number of agents, any number of blocks per task and any map dimension, but we need time to solve the bug that ruins the agent's local map representation. Furthermore, we are happy with our system design and implementation, we managed to create a fully distributed system in accordance to the MAPC Contest spirit that can become very robust after just some minor refactoring operations.

6 Future Work

While working on this project we came up with many potential optimizations to our code which unfortunately, due to the lack of time, we were not able to implement. However, we believe they would improve the performance of our implementation considerably.
We have divided these optimizations following the same structure as the original implementation presented in this document 4.

Communication:

- Improve synchronization mechanism. E.g., use the time-stamp of the server which is uniquely shared between all agents

Map Merging:

- Add a validator to ensure the merging was right. Due to timeouts or unexpected bugs, maps can end up being shifted and because most of our functions relie on the map representation to be correct, failures on the map representation can lead to the crash of an agent.
- Allow dynamic landmarks, as for example, other agents. When two agents run into each other, their positions can be used as unique landmarks between that pair of agents. This will allow for a faster discovery of the whole map.

Path Planner:

- Adapt the algorithm to work with a class position.
- Make it more robust when receiving wrong parameters.

Auctioning:

- Make it dynamic and not fixed, based on what is better to be done at every time step.
- If it remains fixed, set a limit amount of task that can be allocated at a time in order to avoid assigning tasks which will be executed very far in the future.

Meeting Point:

- Make it more robust by adding communication between agents about their computed meeting position. In this way every agent would be able to understand if a bug occured and thus reducing the dependence on a perfect global map representation.
- The common meeting point is computed every time step but it is based on distances that do not change over time. The closest dispenser is assigned during task allocation and the distance between dispensers is fixed. If we base the computation of the common meeting point in the actual position of the agent and we share this information with all other agents, it would be able to converge in the most optimal meeting point between any number of agents.

7 Conclusion

As a conclusion, we want to summarize our experience during the whole project, the problems faced and the lessons learned.

The main issues we have encountered are:

- Debugging asynchronous and distributed systems is harder than expected. Asynchronous systems make it really difficult to replicate bugs.
- Integration between different code parts is not easy without coming up with a common understanding of the structure needed for each code part beforehand.
- Type checking is not mandatory in python and it has to be enforced by using classes. This came up to be a good practice due to some variables were accessed multiple times by different functions and it was easy to pass or return wrong types which pass unnoticed until the code crashes.

The lessons we learned are:

- Spending more time modelling before implementation saves time and effort in the long term.
- Pair programming is useful.
- Extensively test every function with the test framework to ensure an easier integration.
- Simplifying the environment helps to develop faster.
- We improved our Python skills. We are now definitively more comfortable with OOP in Python, the ROS library, the IDE Pycharm, the testing framework pytest, the PEP8 specification.
- How to handle big software projects and working in distributed team, subdividing different tasks, respect internal deadlines.

References

1. Group 5 implementation - mapc 2019 (2019). URL https://gitlab.tubit.tu-berlin.de/aaip-ss19/group5
2. Multi-agent programming contest (2019). URL https://multiagentcontest.org/2019/
3. Christopher-Eyk Hrabia, S.W., Albayrak, S.: Towards goal-driven behaviour control of multi-robot systems. In: 2017 3nd International Conference on Control, Automation and Robotics (ICCAR) (2017)
4. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968). DOI 10.1109/TSSC.1968.300136
5. Swift, N.: Easy a* (star) pathfinding