GOAL-DTU: Development of Distributed Intelligence for the Multi-Agent Programming Contest

Alexander Birch Jensen and Jørgen Villadsen(✉)

Algorithms, Logic and Graphs Section, Department of Applied Mathematics and Computer Science, Technical University of Denmark, Richard Petersens Plads, Building 324, 2800 Kongens Lyngby, Denmark
jovi@dtu.dk

Abstract. We provide a brief description of the GOAL-DTU system for the Multi-Agent Programming Contest, including the overall strategy and how the system is designed to apply this strategy. Our agents are implemented using the GOAL programming language. We evaluate the performance of our agents for the contest, and finally also discuss how to improve the system based on analysis of its strengths and weaknesses.

1 Introduction

In fall 2019 we participated as the GOAL-DTU team in the annual Multi-Agent Programming Contest (MAPC). We are using the GOAL agent programming language [1–3] and we are affiliated with the Technical University of Denmark (DTU). We participated in the contest in 2009 and 2010 as the Jason-DTU team [4,5], in 2011 and 2012 as the Python-DTU team [6,7], in 2013 and 2014 as the GOAL-DTU team [8], in 2015/2016 as the Python-DTU team [9] and in 2017 and 2018 as the Jason-DTU team [10].

In 2019 we had the new *Agents Assemble* scenario. The paper is organized as follows:

- Section 2 describes agent programming using the GOAL language.
- Section 3 covers the overall strategy of our agents.
- Section 4 describes the knowledge our agents acquire from the environment.
- Section 5 describes how our agents communicate.
- Section 6 describes the movement of our agents.
- Section 7 covers how our agents complete selected tasks.
- Section 8 evaluates the performance of our agents in the three matchups.
- Section 9 discusses improvements to the system.
- Section 10 is our conclusion.

© Springer Nature Switzerland AG 2020
T. Ahlbrecht et al. (Eds.): MAPC 2019, LNAI 12381, pp. 79–105, 2020.
https://doi.org/10.1007/978-3-030-59299-8_4

2 Agent Programming in GOAL

This section introduces the basic concepts of the GOAL agent programming language that are relevant to the implementation of our system.

Agents in GOAL are to be understood as self-controlled independent entities. Each agent interacts with the environment and communicates with other agents. Percepts and messages are treated as events that can be processed. This event processing then feeds into the knowledge, beliefs and goals (the three components that comprise an agent's cognitive state).

The programming philosophy behind GOAL is quite different when compared to other popular agent programming languages. Beyond the cognitive state, the core abilities of an agent are the just mentioned event processing capability, the ability to represent knowledge and reason about it, and finally, rule-based decision-making which allows an agent to select an action based on its current cognitive state.

2.1 The GOAL Execution Loop

GOAL features a simple execution loop of each agent. Beyond an initialization module that can process the initial state of the environment, and set up the initial cognitive state, GOAL follows the execution loop below:

1. **Check new events:** If there are no new events, the next step is skipped.
2. **Process events:** The event module processes new events. Recall that these events are either percepts from the environment or messages from other agents. It is the purpose of this module to update the cognitive state of the agent before selecting the next action.
3. **Action selection:** The main module defines the rules for decision-making. Based on the rules, the first valid action is selected. Note that several actions may be applicable based on the rules in the main module. GOAL allows for other strategies, but it is essential to our implementation that the first action is always the one that is selected.
4. **Perform action:** The selected action is sent to the environment (and communication actions are executed internally).

Technically speaking, there is also a final step that applies the post-conditions (effects) for the selected action from our action specification. However, it is not relevant for us since we rely solely on the event module to perceive the effects of actions.

2.2 Action Selection

GOAL advocates that agents are individual entities that reason about their environment. They react to changes in their environment rather than executing predetermined plans. For example, an agent may devise a plan for a goal to be achieved. At some intermediate step in the plan, the next step may no longer be

applicable due to (unforeseen) changes in the environment. GOAL tries to avoid the complexity of rebuilding "broken plans" by advocating a reactive model. We should consider how the agent can select appropriate actions based on the current state of affairs. However, note that it is still possible for programmers to represent plans via the cognitive state of agents using Prolog, but it is not facilitated explicitly by the language. The reactive approach is not flawless either: it can be difficult for programmers to come up with logical rules that produce the desired behaviour, but by overcoming this challenge, we often have a more flexible agent.

3 Strategy

In our current system, we have a universal agent type. By a universal agent type we mean that all agents share the same logic. While it is possible to have different kinds of agents in GOAL, i.e. via modules, it is not something we currently utilize. Some advantages of a universal agent type are that it is faster to implement, every agent is seamlessly capable of everything and we need not worry about when to switch the agent's type. The main disadvantage is that the code base becomes convoluted as development progresses due to growing array of logical rules for selecting the appropriate action.

During action selection, the agents apply heuristic measures to determine movement directions. We describe the different variants of heuristic functions in Sect. 6. It should also be noted that we currently do not perform any *clear* actions. We did not manage to implement use of the action in a meaningful way for the contest.

The following priority list describes the decision-making process of our agents (with some simplifications) where the first applicable rule determines the action to be selected in a given state:

- If the agent is assigned to a task:
  - Detach any attached blocks not needed for the task. The agent will only detach blocks if it considers it non-obstructive to future movement. If not, it will continue to move (using the detach movement heuristic) until it considers it safe to detach.
  - Rotate the block into the position dictated by the task plan. If rotation is blocked, move until rotation is possible (using the exploration movement heuristic).
  - If the agent observes part of the pattern to be handed in, or if the agent is assigned to submit the task (the submitting agent), and is on a goal, wait for other agents (by performing the *skip* action).
  - If the agent observes the entire pattern, connect with other blocks/agents as described by the task plan.
  - If all blocks in the pattern are connected, the assigned agent submits the task.
  - If the agent finds the submitting agent (waiting in a goal area), move until the attachment(s) form the (partial) pattern (using the task pattern movement heuristic).

  - If the agent is the submitting agent, move towards a goal area (using the go to movement heuristics).
  - Else, move towards the position of the submitting agent (using the go to movement heuristics).
  - If a goal area is known, move towards it (using the go to movement heuristics).
  - Move into the most promising direction (based on the exploration movement heuristic).
- If the agent is not assigned to a task:
  - If a block or dispenser is in vision, and the agent does not have four blocks:
    * Rotate such that a free attachment spot is facing the direction of the block/dispenser. If rotation is blocked, move (using the exploration movement heuristic).
    * If the agent is next to a block, attach it to the agent.
    * If the agent is next to a dispenser, request a block.
    * If not next to the block/dispenser, move towards it (using the go to movement heuristics).
  - Move around on the map (based on the safe exploration heuristic).
- Perform *skip* action.

4 Agent Knowledge

In this section, we cover the design of the knowledge stored in the agents' mental states. Generally speaking, the agents store and maintain knowledge about the map that is assumed be invariable (or alternatively: always perceivable). We consider invariable knowledge to be: the positions of goal cells, attached blocks, the agent's current position (relative to its starting position), positions visited by the agent, and the position of encountered agents from the team. Some of these involve communication between agents. The positions of blocks, dispensers and obstacles are only stored in the agent as long as they are within vision. The communication of our agents is described in Sect. 5.

The agent does not perceive a global view of the map via the environment, nor its own position on the map. Furthermore, random events and actions of other agents can change the structure of the map over time. Due to this complexity, we do not attempt to build up an internal representation of the map. Unfortunately, this comes at the cost of efficient and meaningful movement on the map.

4.1 The Current Position

By keeping track of performed *move* actions, and checking for a potential failed action, the agent maintains information about its own current position. With the starting position of the agent as the center of origin, we maintain two values that represent the agent's position in a two-dimensional space. Moving in a direction, either north, east, south or west; results in incrementing or decrementing one of these values.

4.2 Visited Positions

The bookkeeping of visited positions is essential to avoiding that the agent repeatedly gets stuck, or does not make progress. When an agent performs a *move* action, in the next step the following information about the visited position is stored: the relative position of the agent, the current step in the simulation, and a flag for if the position is a goal cell. The position is stored relative to the initial position of each agent. That is, each agent is initially at $(x, y) = (0, 0)$. The relative position of each agent is updated based on successful *move* actions.

As the simulation progresses, the database of visited positions gains additional entries. Due to the way we utilize this, we are only interested in visited positions with respect to a specific subtasks. For instance, if the agent is trying to find a goal cell, it is not relevant which positions the agent visited in an attempt to find blocks. Therefore, we define a number of events that trigger a clearing of the agent's knowledge about visited positions:

- The agent has completed a subtask: Attached, detached or requested a block.
- We submitted a task. In our system, once a task is submitted, most agents will work on different matters.

The visited nodes are useful for steering the agent away from repeating the same movement patterns when they do not make progress. The idea is that visited positions are only relevant locally – at later points in time it may be relevant to visit those positions again. Essentially, this means that the visited positions are only remembered for the duration of smaller subtasks. Intuitively, it seems non-optimal to remove knowledge that could help steer the agent away from dead ends that it has found earlier. However, the ever-changing structure of the map quickly invalidates this knowledge anyway.

4.3 Positions of Goal Cells

The positions of goal cells are assumed to be invariable throughout the simulation. Due to this assumption, once the agents store knowledge about the positions of goal cells, they are never removed.

The position of goal cells are stored relative to the position of the agents and are thus updated following successful *move* actions.

The agents learn about positions of goal cells either through perceiving them within their own vision or via communication with other agents.

4.4 Blocks, Dispensers and Obstacles

The information about blocks, dispensers and obstacles are only perceived when the agent is within vision. A *clear* event may remove blocks from the map, or they may be moved by other agents. Due to this, the positions of blocks is not maintained when outside of the agent's vision.

Obstacle positions could potentially be maintained by perceiving *clear* events and remove information about affected obstacles. The position of obstacles currently plays no part in any sort of route finding algorithm and we do not maintain this knowledge when outside of the agent's vision.

Dispensers are different from blocks and obstacles as their positions do not change during the simulation. In the current implementation, agents always go towards an available dispenser, if they do not have a block on each side, and if they are not trying to solve a task. Our agents will always prefer to go to the nearest known position of a block or dispenser. Thus to avoid the agent always going back to the same dispenser, we currently do not keep information of dispenser positions outside of the agent's vision.

Neither the position of blocks, dispenser or obstacles are shared between agents via communication. Since we do not keep and maintain their positions, it does not make sense to share the information between agents – it should only be part of an agent's mental state when within vision.

4.5 Attached Blocks

Each agent keeps track of its own attached blocks with coordinates relative to its own position. The environment makes available any attached blocks in vision, but it is not immediately visible which agents the blocks are attached to. To make sure that the agent only keeps stored knowledge about the blocks attached to itself, we check for successful *attach* actions to insert the knowledge of a block being attached. Successful rotations update the stored coordinates accordingly. We always make sure that any knowledge about attached blocks is also perceivable in the environment – if not, the knowledge is removed. This is due to the fact that submitting tasks and *clear* events may invalidate the knowledge.

We will also briefly mention that current attached blocks are communicated between agents. This is used for devising plans to submit tasks. The details are covered in Sects. 5.3 and 7.2.

5 Agent Communication and Shared Knowledge

Sharing knowledge between agents by means of communication is essential for efficiently exploiting the multiple agents available. The environment presents a number of challenges in enabling effective agent communication. Also, the volatility of the scenario map does not suggest an easy way of building up a shared representation. Our current implementation could utilize more shared knowledge and communication, and it is something we hope to improve in the future.

Specifically for agent programming using GOAL, communication between agents are part of the core loop. One important aspect is that any messages sent in one step will only be available for processing by the receiving agent in the following step. This requires some deliberate implementation to make sure that

the information received is up to date – in our implementation this is extremely relevant as we often share information that is relative to the current position of agents.

5.1 Encountering Other Agents

The environment only gives information to agents about the position of other agents when within their vision. The agent is able to perceive which team an encountered agent belongs to, but no further identification is provided (i.e. the name of the encountered agent). To be able to identify which pair of agents that have encountered each other we apply the following: when two of our agents meet, they exchange information about what other objects they are able to identify within their vision. Only if they agree on everything in their shared vision, they acknowledge that they did in fact encounter each other. A check is performed to prevent two agents from mistakenly concluding that they encountered each other. We do this by checking that the given pair of agents agree on objects in their shared vision. Our initial implementation was solely based on the agents' relative position to each other, with no additional conditions, which yielded occasional false positives.

5.2 Goal Cells and Agent Positions

When two agents agree that they encountered each other, they exchange information about the positions of goal cells and other agents from the team. Each agent adds the shared information to its belief base relative to its belief about its own current position. Currently, the agents do not continue to share new information after encountering other agents.

When an agent successfully moves in a given direction, it informs other agents about which directed it moved in. This information is used by each agent to maintain the knowledge about positions of other agents.

5.3 Attached Blocks

We assign one of our agents as the *planning agent*. At each step of the simulation, each agent, that is not currently assigned to solve a task, sends a message to the planning agent containing a list of its currently attached blocks. The planning agent uses the received messages to (possibly) assign a task to a subset of the agents that sent messages. The details of the task planning assignment are covered in Sect. 7.2.

6 Agent Movement

The *Agents Assemble* scenario provides only partial vision of the map to agents, limited to a small area around each agent. Combining the knowledge of agents over time will provide more and more knowledge of the map. However, random

clear events happen over time around the map that remove and randomly add new obstacles on part of the map. Other agents also have the ability to remove obstacles. As such, a usual route finding algorithm requires substantial alterations to be usable for the scenario. Due to the volatility of the map, such a route finding algorithm will necessarily have to support re-planning when the planned route is invalidated.

The above mentioned challenges for a route finding algorithm means that we have opted for a more naive implementation of agent movement. The overall strategy is to evaluate each of the (up to four) possible directions: north, east, south and west; and then select the direction which has the optimal heuristic value. When multiple directions share the same optimal value, a direction is selected pseudo-randomly (the current simulation step is used as seed). Our agent movement algorithm has five different variations:

- **Exploration** favors directions towards positions the agent has not visited recently.
- **Safe exploration** is similar to the above, but further favors directions that increase the distance to goal areas and other agents.
- **Go to** favors directions towards a given relative position and penalizes movement to recently visited positions on the map.
- **Task pattern** favors directions that realize a given task pattern and penalizes movement to recently visited positions on the map.
- **Detach** favors directions away from obstacles.

The choice of movement algorithm depends on the current strategy of the agent.

6.1 Evaluation Functions

As described above, each variation of the movement algorithm is distinguished by its heuristic function h. We will use h^+ to denote that a higher value is better and h^- when lower is better. Neither of the mathematical formulations are perfect in any sense, but give some approximation of the optimal choice.

Exploration

$$h^+_{exp}(d) = \sum_{visited} \begin{cases} \frac{|\Delta x(d)|+|\Delta y(d)|}{\Delta S^2} & \text{if } |\Delta x(d)| + |\Delta y(d)| \leq 30 \text{ and } \Delta S > 0 \\ 0 & \text{else} \end{cases}$$

where $\Delta x(d)$ and $\Delta y(d)$ are the differences in x and y coordinates between the visited position and the agent's position after performing *move* in direction d. ΔS is the number of steps since the position was visited.

Safe Exploration

$$h^+_{s-exp}(d) = h^+_{exp}(d) + \sum_{(x_t, y_t) \in P} c(x_t, y_t) * (|x_t| + |y_t|)$$

where P is a set of coordinates of goal cells and nearby agents in the team. x_t and y_t are coordinates relative to the agent's current position. $c(x_t, y_t)$ is a constant factor used for heavily favoring moving away from nearby agents.

Go To

$$h^{-}_{go-to}(d) = |\Delta x(d)| + |\Delta y(d)| + \text{size}(V_{p_d})$$

where $\Delta x(d)$ and $\Delta y(d)$ are the differences in x and y coordinates between the goal position and p_d is the agent's position after performing *move* in direction d. size(V_{p_d}) is the number of times position p_d has been visited recently.

Task Pattern

$$h^{-}_{pat}(d) = \text{size}(V_{p_d}) + \sum_{(x,y,t)\in(pat/att)} min\left\{|\Delta x| + |\Delta y|, |\Psi(d, x, y, t, \Delta x, \Delta y)|\right\}$$

where size(V_{p_d}) is the number of times p_d has been visited recently (p_d is the agent's position after performing *move* in direction d). The set pat/att contains the relative position and type of every block in the pattern excluding the blocks the agent itself is providing (has attached). The predicate $\Psi(d, x, y, t, \Delta x, \Delta y)$ gives the difference in x and y coordinates to every observed non-attached block of type t assuming a move in direction d.

Detach

$$h^{+}_{det}(d) = h^{+}_{exp}(d) + \sum_{(x,y)\in\text{obstacles}} |\Delta x(d)| + |\Delta y(d)|$$

where obstacles is the set of the positions of observable obstacles. $|\Delta x(d)|$ and $|\Delta y(d)|$ are the relative differences in x and y coordinates between the agent and the obstacle following a move in direction d.

7 Solving Tasks

This section describes how the agents solve tasks. We consider solving a task to consist of four parts: Collecting blocks, planning tasks to complete based on the collected blocks, executing task plans (assembling the pattern) and finally submitting the pattern.

7.1 Collecting Blocks

One core aspect of our strategy is to collect blocks before committing to any of the available tasks, and to only commit to tasks for which we already have the blocks to solve.

If an agent, that is not assigned to a task, and does not hold a block on each side, encounters a block or dispenser, it will generally try to go towards it. It will

only ignore the possibility to collect the block(s) in case another from the same team is adjacent to it. This is to avoid race conditions for the same resource and improve efficiency. Dealing with this issue via communication would likely be a better approach, however.

In case the agent sees a dispenser or block, that is not occupied by another agent from the same team, the agent will rotate if necessary to ensure that a free attachment spot is available in the direction of the block or dispenser. Attaching blocks takes priority over requesting blocks from dispensers. The agent will repeatedly attach blocks on each of the four spots until they are all used.

In some cases the position of the block or dispenser may not allow the agent to attach from that angle due to its current attachments. This is not currently checked and avoided. In such a case, the agent is likely to enter a state of not making progress – unless it is assigned to a task, or if it moves outside of vision of the block due to being penalized for going to similar positions repeatedly.

7.2 Task Planning

If the combined attachments of all agents are sufficient to solve a task the planning agent will compute a task plan for it. The task with the lowest reward (and thus, likely the easiest to complete) is selected. We select the easiest task based on the observed performance of the agents, and we only ever try to complete one task at a time. The logic for task planning supports computing multiple non-overlapping task plans, but we observed that the agents would be likely to obstruct each other. We only ever commit to a task that has some amount of steps available to complete the task before the deadline. For the contest, we set this to a minimum of 50 steps to complete the task. Later experimentation has yielded better results with a higher number. Due to the abundance of available tasks, the improved results seem logical. So far we have not conducted tests of the distribution of task completion times for our agents.

The task plan specifies how each agent provides part of the pattern including how it should be rotated. The expected completed pattern is computed for each agent relative to its own position in the aligned pattern. This is used to ensure that the agents align their attachments correctly. The task plan also specifies how agents should connect to each other once the pattern alignment step is complete. For each task, one of the selected agents is assigned to submit the task.

The task plans are rigid in the sense that two agents that provide the same block type cannot swap their respective sub-patterns. While there potentially could be some benefit in supporting this behaviour, we do not consider it worth the effort to implement when considering other potential improvements to the agents.

7.3 Executing Task Plans

When a task plan is sent to the agents, the agents not involved will keep on moving around the map, but their heuristics for movement will penalize moving

close to other agents. This is done in an attempt to avoid obstructing agents that are working on completing a task. Each of the involved agents will immediately detach any of the blocks that are no longer needed and rotate the remaining attachments to align with their part of the pattern for the task. The agent responsible for submitting the task will move towards the nearest goal cell if its position is known (or scout for a goal cell if not). Once positioned at a goal cell, the agent will wait for the other agents to show up. If the other agents assigned to the task know the position of the agent responsible to submitting the task, they will move towards that agent. If not, they will methodically visit each of the known goal areas. If this fails, they will scout around the map. In most practical cases, the agents eventually learn of the position of the agent responsible for submitting the task.

We are not currently able to resolve situations where assigned agents are stuck. However, the task plan is discarded if an assigned agent is disabled due to a *clear* event. The idea is that if an agent is stuck the task plan is to be discarded, but our implementation does not work properly. As previously mentioned, we also do not utilize the *clear* action in any way currently, which could solve potential pathing problems. At any time, if the task deadline is exceeded, the task plan is deleted.

In case an agent assigned to the task finds its way to the agent responsible for submitting the task, waiting at a goal area, the agent will start to align itself to complete the pattern. This is achieved by the task pattern heuristic that favors directions that minimize the expected deviance from the pattern to submit. It should be noted that the agent responsible for submitting the task will try to position itself such that the other agents have room to align themselves to complete the pattern.

Once the pattern is complete, the agents connect their attachments and the task is submitted.

8 Evaluation of Matches

With a total of four participants, we played three matches against each of the three opponents. In the following, we evaluate the performance of our agents in each matchup. See Figs. 1–45 for key statistics over the 500 steps of each match.

GOAL-DTU vs. TRG

In two of the simulations, we manage to complete a single task early on (Figs. 10, 11, 12). Team TRG completes a single task in the first simulation, but are unable to do so in the other simulations. TRG has a strategy where some of their agents defend goal areas by attempting to perform *clear* actions on our agents trying to complete tasks. Since part of our task execution plan is to assemble the pattern in the goal area, the strategy of team TRG denies a fair number of submits from our agents.

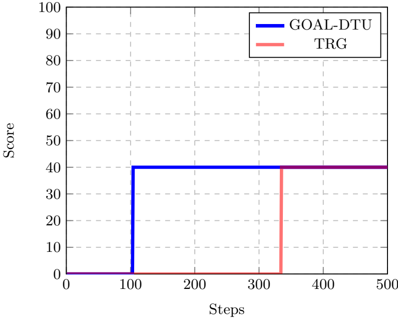

Fig. 1. Score: GOAL-DTU vs. TRG (1)

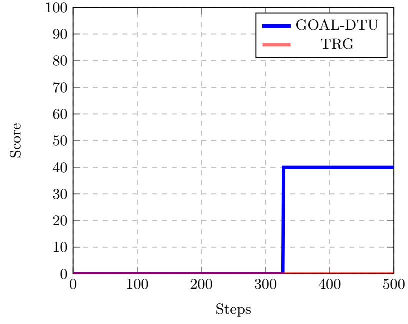

Fig. 2. Score: GOAL-DTU vs. TRG (2)

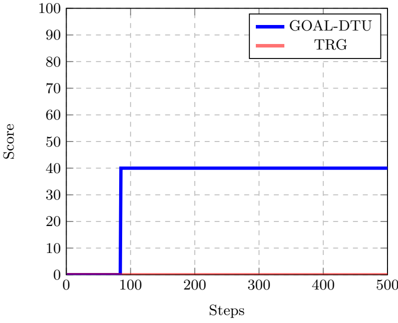

Fig. 3. Score: GOAL-DTU vs. TRG (3)

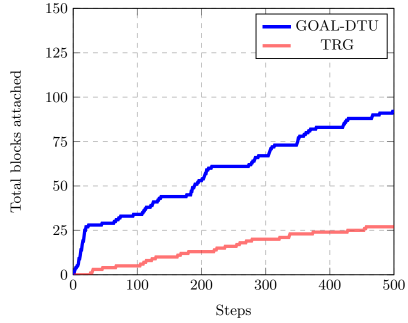

Fig. 4. Blocks: GOAL-DTU vs. TRG (1)

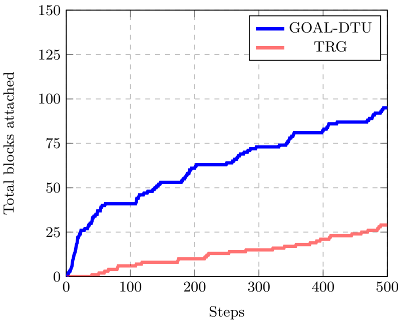

Fig. 5. Blocks: GOAL-DTU vs. TRG (2)

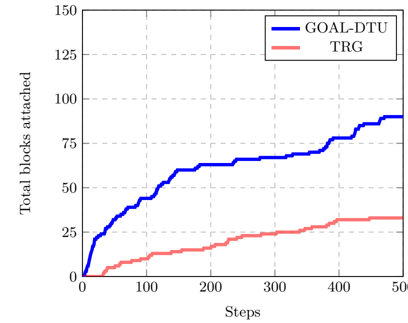

Fig. 6. Blocks: GOAL-DTU vs. TRG (3)

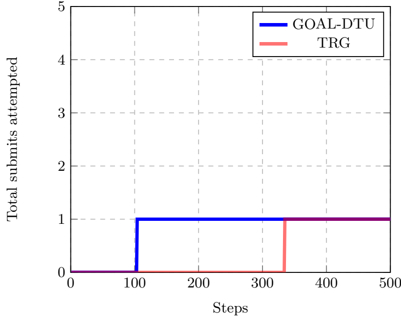

Fig. 7. Submit: GOAL-DTU vs. TRG (1)

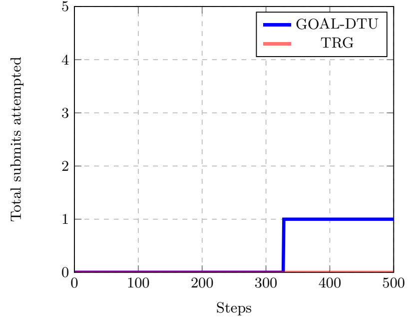

Fig. 8. Submit: GOAL-DTU vs. TRG (2)

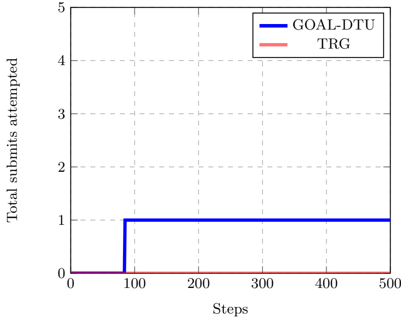

Fig. 9. Submit: GOAL-DTU vs. TRG (3)

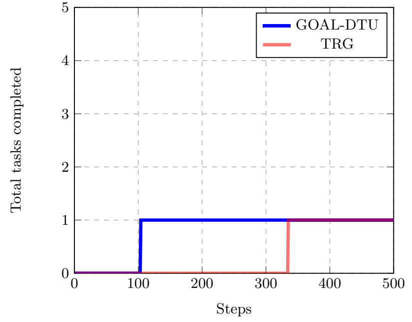

Fig. 10. Tasks: GOAL-DTU vs. TRG (1)

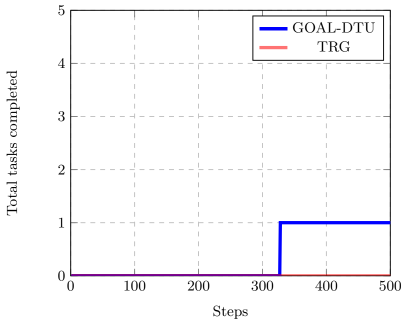

Fig. 11. Tasks: GOAL-DTU vs. TRG (2)

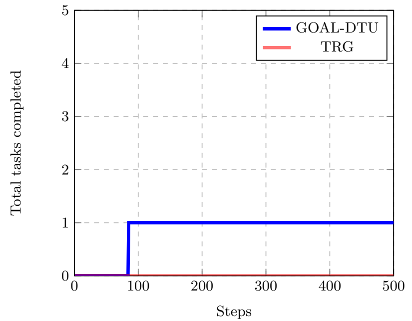

Fig. 12. Tasks: GOAL-DTU vs. TRG (3)

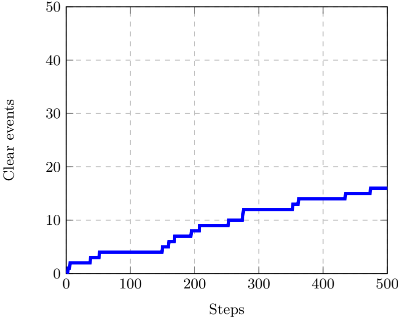

Fig. 13. Clear: GOAL-DTU vs. TRG (1)

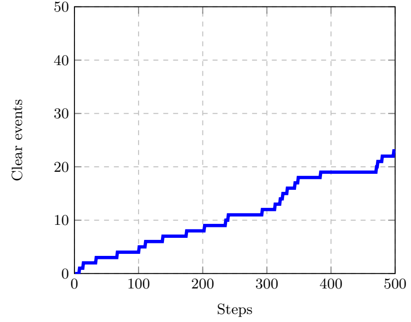

Fig. 14. Clear: GOAL-DTU vs. TRG (2)

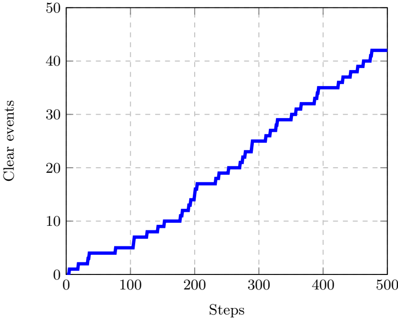

Fig. 15. Clear: GOAL-DTU vs. TRG (3)

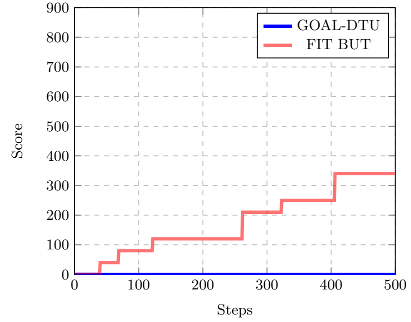

Fig. 16. Score: GOAL-DTU vs. FIT BUT (1)

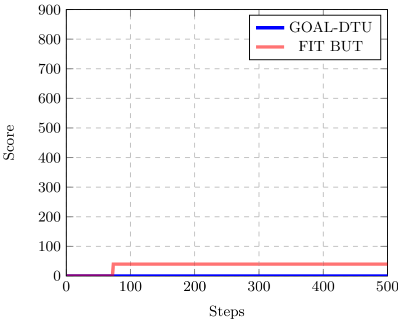

Fig. 17. Score: GOAL-DTU vs. FIT BUT (2)

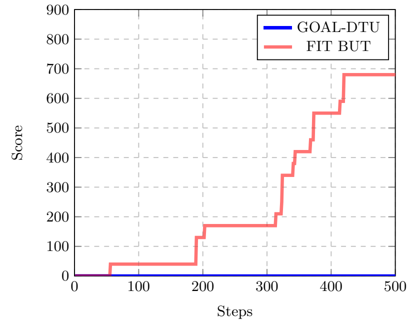

Fig. 18. Score: GOAL-DTU vs. FIT BUT (3)

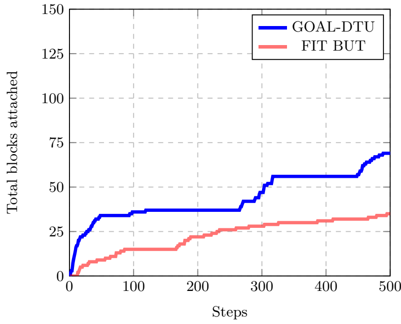

Fig. 19. Blocks: GOAL-DTU vs. FIT BUT (1)

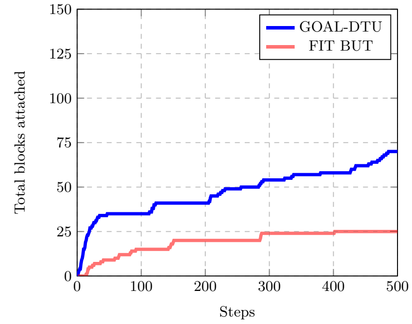

Fig. 20. Blocks: GOAL-DTU vs. FIT BUT (2)

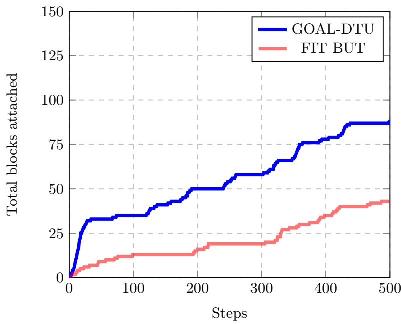

Fig. 21. Blocks: GOAL-DTU vs. FIT BUT (3)

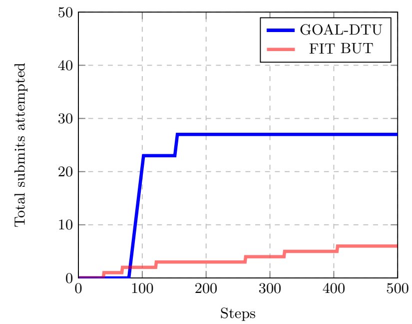

Fig. 22. Submit: GOAL-DTU vs. FIT BUT (1)

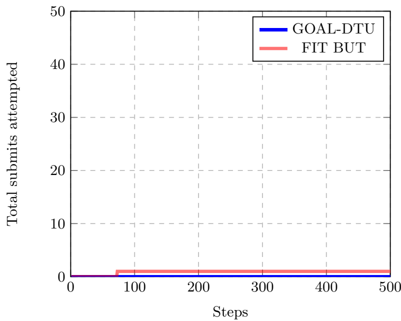

Fig. 23. Submit: GOAL-DTU vs. FIT BUT (2)

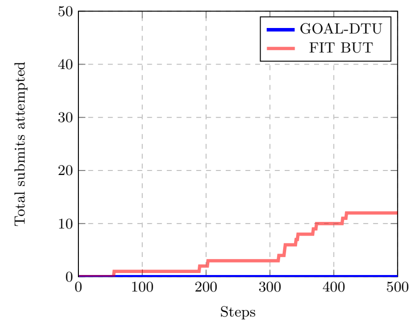

Fig. 24. Submit: GOAL-DTU vs. FIT BUT (3)

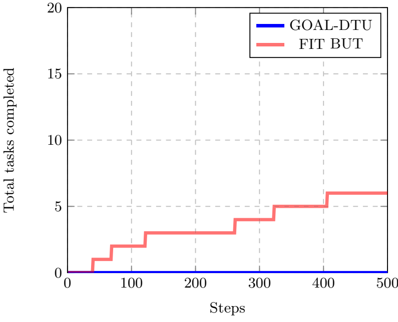

Fig. 25. Tasks: GOAL-DTU vs. FIT BUT (1)

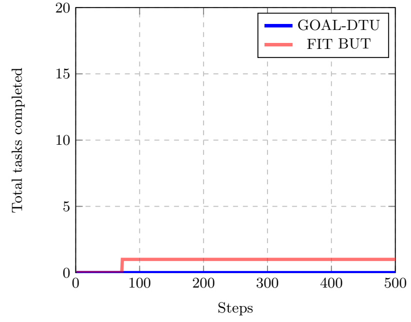

Fig. 26. Tasks: GOAL-DTU vs. FIT BUT (2)

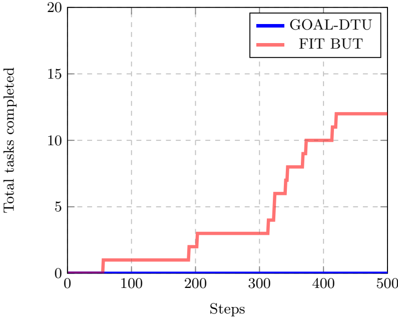

Fig. 27. Tasks: GOAL-DTU vs. FIT BUT (3)

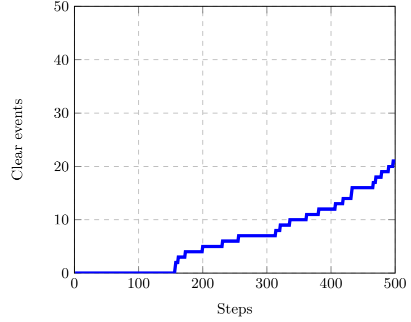

Fig. 28. Clear: GOAL-DTU vs. FIT BUT (1)

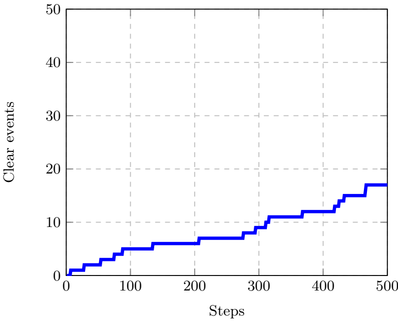

Fig. 29. Clear: GOAL-DTU vs. FIT BUT (2)

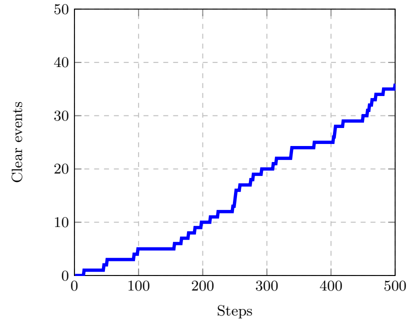

Fig. 30. Clear: GOAL-DTU vs. FIT BUT (3)

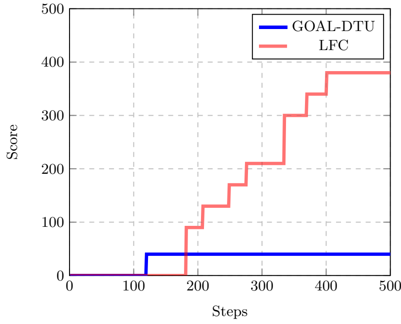

Fig. 31. Score: GOAL-DTU vs. LFC (1)

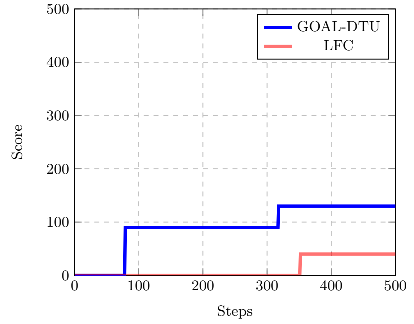

Fig. 32. Score: GOAL-DTU vs. LFC (2)

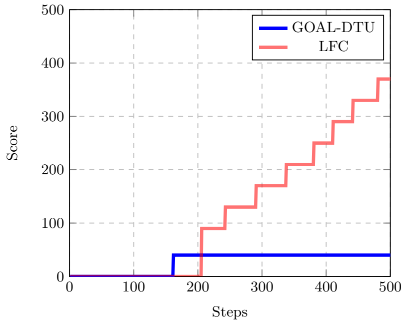

Fig. 33. Score: GOAL-DTU vs. LFC (3)

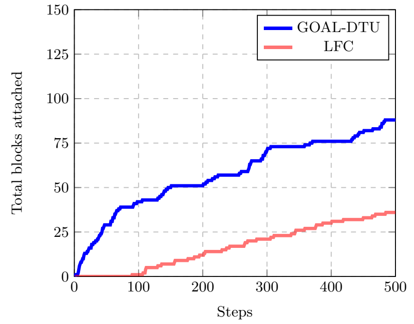

Fig. 34. Blocks: GOAL-DTU vs. LFC (1)

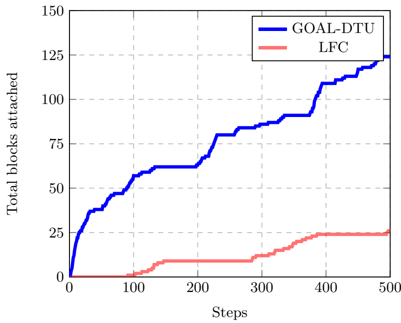

Fig. 35. Blocks: GOAL-DTU vs. LFC (2)

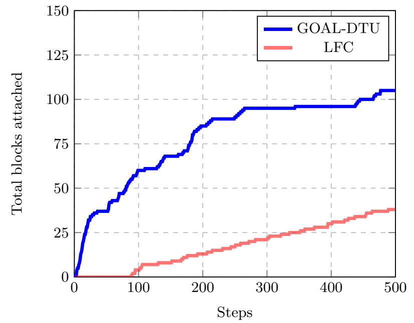

Fig. 36. Blocks: GOAL-DTU vs. LFC (3)

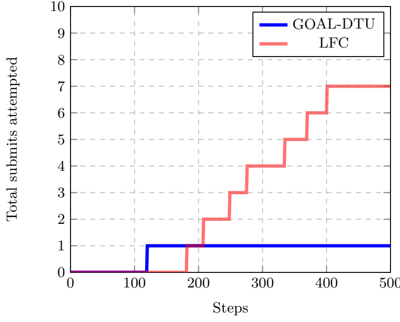

Fig. 37. Submit: GOAL-DTU vs. LFC (1)

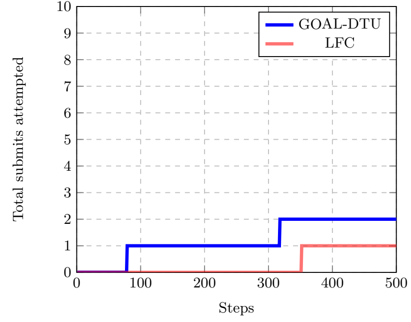

Fig. 38. Submit: GOAL-DTU vs. LFC (2)

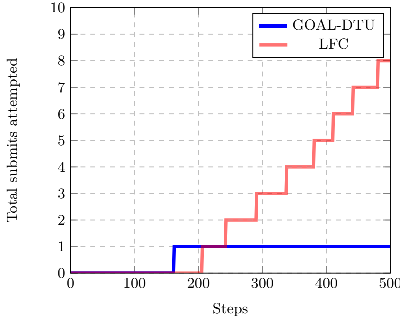

Fig. 39. Submit: GOAL-DTU vs. LFC (3)

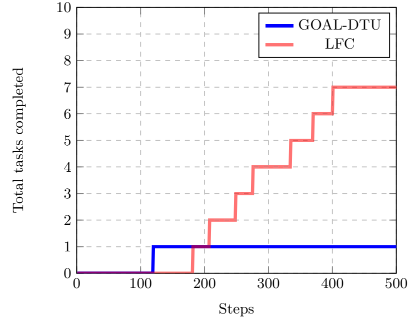

Fig. 40. Tasks: GOAL-DTU vs. LFC (1)

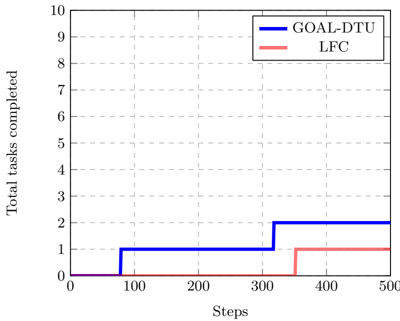

Fig. 41. Tasks: GOAL-DTU vs. LFC (2)

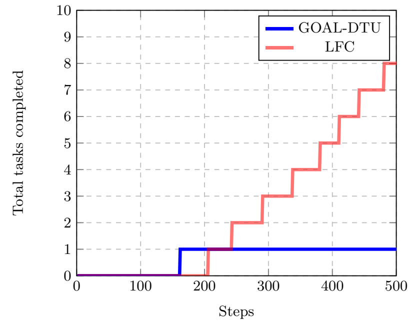

Fig. 42. Tasks: GOAL-DTU vs. LFC (3)

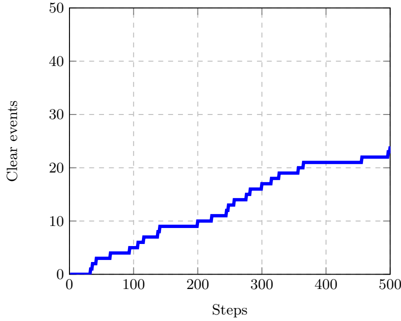

Fig. 43. Clear: GOAL-DTU vs. LFC (1)

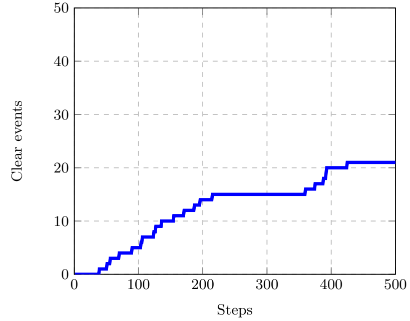

Fig. 44. Clear: GOAL-DTU vs. LFC (2)

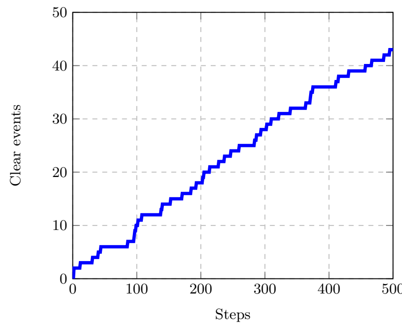

Fig. 45. Clear: GOAL-DTU vs. LFC (3)

We experience problems with many agent getting stuck in every simulation. Around halfway through, we can usually observe that half of our agents are now stuck. This is especially detrimental if one of those agents is assigned to complete a task.

It seems that our greedy approach for collecting blocks, which causes the map to become even more convoluted, also causes serious issues for the agents of team TRG.

In the third simulation, the number of blocks we manage to collect stagnates (Fig. 6). This could be correlated with the number of *clear* events that is significantly higher (Fig. 15). While this presumably has no direct impact our score, it could be an indication towards our agents' ability to move around on the map.

See Figs. 1–15 for all of the collected statistics in matches vs. TRG.

GOAL-DTU vs. FIT BUT

For this matchup we experienced issues with the agents. In the first simulation, our agents manage to assemble two patterns for tasks, but in one instance they seem to have a wrong pattern, and in the other instance they try to submit

outside of a goal area (Fig. 22). At this point, we try to restart our agents, but they do not mange to make meaningful progress since our implementation is not robust in case of crashes (Fig. 19).

Also in the second simulation we have to attempt a restart, but to no avail. It seems our agents obstruct the map so severely that team FIT BUT has issues (Fig. 26). Before our agents crash, they are relatively close to assembling a pattern.

The story repeats itself in the third simulation, although team FIT BUT successfully complete multiple tasks (Fig. 27).

While we do not expect us to have been able to beat the agents of team FIT BUT, we would likely have been able to complete a few tasks if the agents did not crash.

See Figs. 16–30 for all of the collected statistics in matches vs. FIT BUT.

LFC vs. GOAL-DTU

We managed to complete tasks in each of the three simulations (Figs. 40, 41, 42). Yet, as for other simulations, as the simulation progresses, the agents' ability to move around the map degrades (Figs. 34, 35, 36). In comparison, it seems that the agents of team LFC make steady progress throughout the simulation (Figs. 40, 41, 42). Another note about the agents of team LFC is there seems to be a correlation between the tasks they submit and the number of blocks they collect which suggest a different strategy (Figs. 34, 35, 36).

We manage to complete more tasks in the second simulation (Fig. 41). By inspection of the map layout, there are two goal areas in the middle of map, not close to any obstacles. This is a lucky coincidence for our agents, as they often experience more problems when close to obstacles (maneuverability in confined spaces is more likely to degrade over time).

See Figs. 31–45 for all of the collected statistics in matches vs. LFC.

9 Discussion

While our implementation achieved satisfying results, it can still be improved on several fronts. During the contest, we learned that the performance of the agents could be improved by tinkering with parameters, while other issues were due to technical difficulties.

9.1 Changes Since the Contest

After analysis of the replays of our matches in the contest, we realized two possible improvements to the implementation. The first improvement is concerned with the assumed time our agents need to complete a task. Through experimentation, we have learned that increasing the minimum amount of steps needed to complete a task improved the performance of the agents significantly. The value used for the matches in the contest often lead to agents missing task deadlines

resulting in a lower score, when considering that a task with a later deadline could have been chosen instead.

Another issue occurs when agents detach blocks (for getting rid of blocks not needed for completing the assigned task). The agents will try to detach the unneeded blocks away from goal areas and obstacles. This is done to avoid that the agent potentially obstructs itself and other agents. By increasing the minimum distance there should be to goal cells and obstacles when detaching a block, we observed improved performance.

9.2 Technical Issues During the Contest

During the contest, we experienced a number of issues during the simulations that we had not encountered before. Unfortunately, for some of the matches this made our agents break down completely, practically leaving us with no way to continue. Since we did not experience this before, our agents are not very robust in the sense that they are not well-suited for restarts during the simulation in case of crashes.

One of the issues we experienced occurs when the steps are performed very rapidly, at which point it seems as if the GOAL execution and the server get out of sync. We did not experience this problem earlier since in our testing, there were always agents not performing actions, thus using the full server timeout for each step. Experimentation following the contest shows that the problem is not related to connection issues and will have to be investigated further. We find that two seconds for each step prevents the issue.

9.3 Known Problems and Bugs

In our implementation, we have discovered a number of problems over time, and there are still some unresolved bugs to fix.

One problem is related to agents getting stuck. Often in simulations, we will experience that one or more of our agents end up getting stuck. This could be due to a number of random *clear* events, or potentially the map has disconnected parts. One of the major issues with this is that we have not been able to implement a way for agents to deduce that they are in fact stuck, or that some agents are unable to reach each other. Another problem is that we do not utilize the *clear* action to help the agents become unstuck.

Lastly, we have experienced problems when agents assigned to a task visit goal areas in search for the agent responsible for submitting the task. Due to an unresolved bug, the agents will not properly scout all the goal areas, but tend to always stay in the same area. This obviously means that we will never be able to submit unless the problematic agents learn about the position of the agent responsible for submitting the task.

9.4 Improvements

There are a number of directions to take in terms of improving the implementation. We will consider some of our high-level ideas to improve the performance of the system by targeting some of our weaknesses.

Our agents are universal in the sense that every agent is based on exactly the same logic rules. One way to improve the performance, could be to assign different roles to agents, or to assign agents into smaller teams that move together. Examples of roles could be agents that explore the map, some that request and collect blocks and some that complete tasks using the collected blocks.

Another weakness of our agents is poor movement. Since we do not build an internal representation of the map during the simulation, our agents often move blindly. We expect substantially improved performance if we are able to make the agents better at moving around the map, for example by building up such an internal map representation (which could then be shared among agents). The initial reason not to attempt building up a map representation, is the complexity of the map being dynamic.

Another problem arises from our greedy approach to collecting blocks. Since each agent always tries to collect one block on each side, movement around the map becomes much harder afterwards. Furthermore, we do not consider the possibility that an agent may be able to move past narrow corridors by rotating its attachments, or potentially even moving the blocks past corridors a few at a time.

The last obvious improvement is to implement some logic to perform *clear* actions. Multiple *clear* events are likely to make it difficult to move around the map. Getting rid of obstructions is exactly one of the purposes of the *clear* action. It can be used to reconnect parts of the map that has been disconnected completely, and to save valuable time by creating shortcuts through obstacles.

Lastly, there are number of technical improvements that we would like to implement. Since it was impossible to monitor the agents live during the simulations of the contest, it would have been helpful to have better output (in the console) about the behaviour and progress of the agents. Furthermore, we would like to increase the robustness of the agents in case of crashes such that they can be restarted and still make progress.

10 Conclusion

We have provided an overview of the multi-agent system that the GOAL-DTU team developed for the Multi-Agent Programming Contest 2019. We have explained our choice of the GOAL programming language; we have also described the main strategy of our agents and how they execute that strategy. This year was the first iteration using the *Agents Assemble* scenario, and we have developed our implementation from scratch using GOAL. Our implementation features a universal agent type in which each agent is based on the same set of logical rules.

The strengths of our system are the flexible nature of our agents. Our agents always react to the current state of affairs and do not rely heavily on predefined

plans to reach their goal of completing tasks. The weaknesses are primarily the agents' poor movement around the map and rigidity in the way tasks are assigned and submitted where stuck agents have a severe negative impact on the performance of the system.

Finally, we have described how to improve the system by coming up with ideas that target its weaknesses. Some of these potential improvements are related to minor issues and bug fixes while other potential improvements require designing and refactoring large parts of the system.

In conclusion, we are satisfied with the performance of our system, ending at a 3rd place in the final rankings, when considering that we have built the system from scratch. We consider our current implementation a good platform to built on for future iterations of the *Agents Assemble* scenario.

Further details about the previous DTU teams are available here:
https://people.compute.dtu.dk/jovi/MAS/

Acknowledgement. We thank Tobias Ahlbrecht, Asta Halkjær From, Benjamin Simon Stenbjerg Jepsen, John Bruntse Larsen and Simon Rumle Tarnow for discussions.

A Team Overview: Short Answers

A.1 Participants and Their Background

- **What was your motivation to participate in the contest?** To work on implementing a multi-agent system capable of competing in a realistic, albeit simulated, scenario.
- **What is the history of your group? (course project, thesis, ...)**
  The name of our team is GOAL-DTU. We participated in the contest in 2009 and 2010 as the Jason-DTU team [4,5], in 2011 and 2012 as the Python-DTU team [6,7], in 2013 and 2014 as the GOAL-DTU team [8], in 2015/2016 as the Python-DTU team [9] and in 2017 and 2018 as the Jason-DTU team [10]. The members of the team are as follows:
  – Jørgen Villadsen, PhD
  – Alexander Birch Jensen, PhD student
  Asta Halkjær From, MSc student and now PhD student, was a consultant until the tournament started.
  We are affiliated with the Algorithms, Logic and Graphs section at DTU Compute, Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU). DTU Compute is located in the greater Copenhagen area. The main contact is associate professor Jørgen Villadsen, email: `jovi@dtu.dk`
- **What is your field of research? Which work therein is related?** We are responsible for the Artificial Intelligence and Algorithms study line of the MSc in Computer Science and Engineering programme.

A.2 Statistics

How much time did you invest in the contest (for programming, organizing your group, other)? Approximately 200 man hours

How many lines of code did you produce for your final agent team? Approximately 1000 lines

How many people were involved? 3 (1 programming)

When did you start working on your agents? August 2019

A.3 Agent system details

How does the team work together? (i.e. coordination, information sharing, ...) How decentralised is your approach? A task is delegated to a set of agents that are attached to the needed blocks. One agent is assigned as the so-called submit agent and the other agents follow/search for this submit agent before aligning the pattern in a goal area. Beyond this, each agent keeps track of the position of other agents. This information is exchanged when two agents are within vision range. The agents confirm their identify by agreeing on the part of the environment they both are able to perceive based on vision.

Do your agents make use of the following features: Planning, Learning, Organisations, Norms? If so, please elaborate briefly. Planning is used when delegating tasks. Agents have set positions in the final pattern for submission.

Can your agents change their behavior during runtime? If so, what triggers the changes? The agent's behavior changes if they are delegated a submission task. Furthermore, other agents will try to avoid blocking agents with a task.

Did you have to make changes to the team (e.g. fix critical bugs) during the contest? We encountered timeout problems when the simulations ran too fast. We did not manage to resolve this beyond putting artificial limit. Furthermore, we did not manage to handle the automatic transition between simulations in each matchup.

How did you go about debugging your system? Partly using the debugger and partly using console output.

During the contest you were not allowed to watch the matches. How did you understand what your team of agents was doing? Did this understanding help you to improve your team's performance? We tracked them using console output although this feature could be vastly improved. It did not help towards performance beyond discovering timeout problems in fast simulations.

Did you invest time in making your agents more robust? How? Some robustness comes almost for free using GOAL as we never deeply commit to a plan. We also considered tracking if an agent ending being stuck, but ultimately the feature was not completed.

A.4 Scenario and Strategy

What is the main strategy of your agent team?

- If the agent is selected to hand in blocks for a task (part of a task plan):
  - Detach any attached blocks not needed for the task. The agent will only detach blocks if it considers it non-obstructive to future movement. If not, it will move until it reaches a position where it considers it safe to detach.
  - Rotate the block into the position dictated by the task plan. If rotation is blocked, move until rotation is possible.
  - If the agent observes part of the pattern to be handed in, or if the agent is the one to submit the task and is on a goal, wait for other agents (skip action).
  - If the agent observes the entire pattern, connect with other agents as described by the task plan and then submit (the *submit* action is performed by the submit agent).
  - If the agent finds the submit agent (waiting in a goal area), move to place the attachment(s) as described by the task plan to form the final pattern.
  - If the agent is the submit agent, move towards a goal area.
  - If not the submit agent and believe that submit agent is in a goal area, move towards the position of the submit agent.
  - If a goal area is known, move towards it (to see if we can find the submit agent there).
  - Move into the most promising direction based on the exploration heuristics.
- If the agent is not selected to hand in any task (not part of the current task plan)
  - If a block or dispenser is in vision:
    - Rotate such that a free attachment spot is facing the direction of the block/dispenser. If rotation is blocked, move.
    - If it is a block, attach it to the agent.
    - If it is a dispenser, request a block.
    - If not next to the block, move towards it.
  - Move into the most promising direction based on the safe exploration heuristics.
- Perform *skip* action.

Your agents only got local perceptions of the whole scenario. Did your agents try to build a global view of the scenario for a specific purpose? If so, describe it briefly. No global view is attempted beyond the position of other agents in the team.

How do your agents decide which tasks to complete? Based on the currently collected blocks.

Do your agents form ad-hoc teams to complete a task? Yes, see above.

Which aspect(s) of the scenario did you find particularly challenging? The random map change events and deciding which blocks to clear (ultimately, we avoided trying to clear blocked paths).

If another developer needs to integrate your techniques into their code (i.e., same programming language tools), how easy is it to make that integration work? That entirely depends on the programming language. Prolog is deeply integrated into much of the code.

A.5 And the moral of it is ...

What did you learn from participating in the contest? We learned about using GOAL and general training in solving complex problems with no obvious solution.

What are the strong and weak points of your team? Our agents are rather flexible and rarely idle. Weak points are that we are possibly too greedy collecting blocks which makes it harder to navigate the map as the simulation progresses.

Where did you benefit from your chosen programming language, methodology, tools, and algorithms? GOAL helps our agents become flexible. We are forced to think in moment-to-moment reasoning and not just plans.

Which problems did you encounter because of your chosen technologies? The freedom can make it harder to keep things simple as the complexity grows. Furthermore, GOAL had some integration issues with the provided EIS interface. We have to attempt changes to the source code to run.

Did you encounter new problems during the contest? We were unaware of the feature that allows for multiple simulations without restarting. Furthermore, we had not tested GOAL with very fast simulations (the fact that we did not sent idle actions in our testing created an artificial slowdown).

Did playing against other agent teams bring about new insights on your own agents? We learned that with another team playing the map became even harder to navigate based on our approach. However, we probably also won some matches by creating the same problem for the opponent.

What would you improve (wrt. your agents) if you wanted to participate in the same contest a week from now (or next year)? Less rigid task submission plans and a less greedy approach to mindlessly collecting all blocks possible.

Which aspect of your team cost you the most time? Navigating the map and trying to make the agents find each other for submission.

What can be improved regarding the contest/scenario for next year? Set up test matches early using the contest setup to discover technical difficulties.

Why did your team perform as it did? Why did the other teams perform better/worse than you did? We did not use roles for agents to help with different tasks. We saw other teams using interesting strategies to solve the tasks. Ultimately, we also had some false assumptions about the scenario which created artificial problems that could have been avoided. In the end, some parts of the design should be completely redone.

References

1. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J.C.: Agent programming with declarative goals. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS (LNAI), vol. 1986, pp. 228–243. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44631-1_16
2. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming, pp. 119–157. Springer, Boston (2009). https://doi.org/10.1007/978-0-387-89299-3_4
3. Hindriks, K.V., Dix, J.: GOAL: a multi-agent programming language applied to an exploration game. In: Shehory, O., Sturm, A. (eds.) Agent-Oriented Software Engineering, pp. 235–258. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54432-3_12
4. Boss, N.S., Jensen, A.S., Villadsen, J.: Building multi-agent systems using Jason. Ann. Math. Artif. Intell. **59**, 373–388 (2010)
5. Vester, S., Boss, N.S., Jensen, A.S., Villadsen, J.: Improving multi-agent systems using Jason. Ann. Math. Artif. Intell. **61**, 297–307 (2011)
6. Ettienne, M.B., Vester, S., Villadsen, J.: Implementing a multi-agent system in python with an auction-based agreement approach. In: Dennis, L., Boissier, O., Bordini, R.H. (eds.) ProMAS 2011. LNCS (LNAI), vol. 7217, pp. 185–196. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31915-0_11
7. Villadsen, J., Jensen, A.S., Ettienne, M.B., Vester, S., Andersen, K.B., Frøsig, A.: Reimplementing a multi-agent system in Python. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) ProMAS 2012. LNCS (LNAI), vol. 7837, pp. 205–216. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38700-5_13
8. Villadsen, J., et al.: Engineering a multi-agent system in GOAL. In: Cossentino, M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 329–338. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45343-4_18
9. Villadsen, J., From, A.H., Jacobi, S., Larsen, N.N.: Multi-agent programming contest 2016 - the Python-DTU team. Int. J. Agent-Orient. Softw. Eng. **6**(1), 86–100 (2018)
10. Villadsen, J., Fleckenstein, O., Hatteland, H., Larsen, J.B.: Engineering a multi-agent system in Jason and CArtAgO. Ann. Mathe. Artif. Intell. **84**, 57–74 (2018)