02285 AI and MAS Final report

Group Name: Starfish

Jules Belveze (s182291), Philip Butenko (s182824), Dimitri Khlebutin (s182450)

June 5, 2019

Abstract

This paper provides an overview of a artificially intelligent multi agent system developed for the purpose of solving fully observable simulated deterministic environments. Agents are tasked with moving boxes to designated locations in the process they must navigate around obstacles such as walls, boxes and other agents. The particular system features a custom navigation algorithm which also can determine if boxes or agents are placed in their optimal route.

1 Introduction

This report provides an overview of the AI and multi-agent system programming project developed with the purpose of solving logistical task designed for robots in a simulated environment. Hospitals tend to have a very high number of transportation tasks to be carried out: transportation of beds, medicine, blood samples, medical equipment, food, garbage, mail, etc. Letting robots carry out these transportation tasks can save significantly on hospital staff resources. To do so we have implemented a non-cooperative multi-agent system (Witteveen and Weerdt 2006), meaning that each agent is constructing its own plan to realize its goal. A joint plan is then created to reduce the number of conflicts between agents by revising the original plans. As opposed to what can be found in the state-of-the-art we do not rely on heuristics to determine the best path but use an algorithm that is derived from Q-learning.

2 Background

For route calculation a custom algorithm was developed and called Backwards Induction Gradient Mapping or BIG for short. As of the time of this report being written, the team is unaware of similar existing algorithm. But the idea is so fundamentally simple that it would be surprising if the MAS field did not already have a similar algorithm in its literature. However not knowing the proper name for it means that is very difficult to research. Christopher Watkins thesis: Learning from delayed rewards was the first publicized occurrence of the (Watkins and Dayan 1992) algorithm. His research and demonstrated results served as an underlying inspiration for the development of BIG, that is implemented in this project. While valuation matrix is directly taken from Q-learning the Valuation function has been heavily modified to only handle agent navigation in the level. Also BIG only calculates every cell once as oppose to Q-learning where many iterations can occur with the purpose of improving states selected by the agent.

3 Methods

3.1 Actions

For this project we have four different actions: $Pull$, $Push$, $NoOp$, $Move$. The action *NoOp* being always applicable, the actions schema for the three other actions is the following:

Action: $Move(agt, agtfrom, agtto)$
Preconditions: $AgentAt(agt, agtfrom) \land Free(agtto)$
Effects: $AgentAt(agt, agtto) \land Free(agtfrom) \land \neg AgentAt(agtfrom) \land \neg Free(agtto)$

Action: $Push(agt, agtfrom, box, boxfrom, boxto)$
Preconditions: $AgentAt(agt, agtfrom) \land Free(boxto) \land BoxAt(box, boxfrom) \land Neigh(agtfrom, boxfrom)$
Effects: $AgentAt(agt, boxfrom) \land Free(agtfrom) \land \neg AgentAt(agt, agtfrom) \land BoxAt(box, boxto) \land \neg BoxAt(box, boxfrom) \land \neg Free(boxto)$

Action: $Pull(agt, agtfrom, agtto, box, boxfrom)$
Preconditions: $AgentAt(agt, agtfrom) \land Free(agtto) \land BoxAt(box, boxfrom) \land Neigh(agtfrom, boxfrom)$
Effects: $AgentAt(agt, agtto) \land \neg Free(agentto) \land \neg AgentAt(agt, agtfrom) \land BoxAt(box, agtfrom) \land \neg BoxAt(box, boxfrom) \land Free(boxfrom)$

Each of the above actions can be executed using the four cardinal directions.

3.2 State

Since the walls are, by definition, immutable they are not used as attributes for each state but are part of the client. Each state contains information about the location of each box and agent on the map. That information is stored in two dictionaries (one for the boxes and one for the agents) where the key is the letter or number of the object and the value is tuple of the form: $(row, column, color)$. The state is updated after each joint action by the client, in order to keep track of the objects' position at each iteration.

3.3 Route Navigation

Instead of leaving agent movement to more conventional search graph exploration we decided to explore multiple different methods that could help the agents arrive at their targeted destinations at a low computational and time-step cost. The Hammington distance was only really useful for open space location mapping and doesn't fair well with larger obstacles in its way. This renders the Hammington distance almost useless in a majority of the maps where this system is to be used. Reinforcement learning was considered to assist in exploring the level however no one in the group had a good enough grasp of the technology in order to implement a solution in the given time frame. With a little more brainstorming BIG was developed.

Big - How it works

BIG draws a null matrix the same $M * N$ size as that of the level. This matrix is called the value grid. From here BIG starts at the destination (in terms of rows and columns), where the agent is to travel to and assigns a value of 1000 to that location in the value grid. This is done by comparing the newly created value grid to the level grid. The assigned value is arbitrary and could be both positive or negative, but the important thing is that the rest of client follows the same syntax. From here the 4 surrounding cells (north, east, south, west) are calculated based on the average of the surrounding 4 cells value in the value grid and divided by a number slightly $>$100% of the newly calculated average. If one or more of the nearest cells are 0, they are not taken into account. In a worst case scenario 3 out of 4 cells are 0. Wall cells are not calculated and therefore continue to have a value of 0.

The Frontier

The newly calculated cells are added to a list dubbed the frontier. This list maintains a list of indices of cells which have received a value. Once BIG selects a cell for calculation from the frontier it is then dropped from the frontier. The frontier takes values from the front of the list and adds new values to the back of the list. This results in an even exploration in all directions away from the destination location and also delivering more accurate directions for determining the quickest route.

Navigation

In principle an agent continually choosing the largest value(from its corresponding value grid) in any of its movable directions will eventually arrive at its targeted destination given that there are not obstacles in its path.

Navigation Accuracy

Previous versions of BIG resembled more Q-learning in the way the grid was calculated. This meant that values were calculated based on an imaginary agent moving through the level in a similar way as if one was to take a paint brush and paint within the walls of a given level. While averaging the surround cells with the one to be currently calculated helped alleviate the problem of early versions of BIG creating unevenly descending gradients, there were still levels that caused problems and sent the agent on slight detours on the way to their designated location, and even down dead-end corridors in worst case scenarios. This was greatly fixed by implementing the frontier which came much later in development and assured an even descent of the valuation numbers in all directions away from the destination. It almost goes without saying the inspiration for the idea came from (Russell 1994)

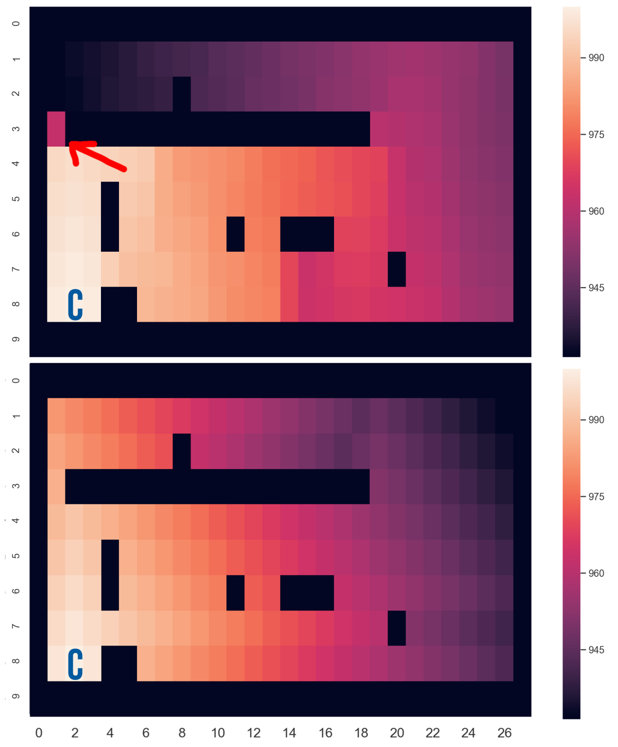

Figure 1: BIG v4 on the top and BIG v6 on the bottom: The red arrow points to the corridor that reveals the weakness of BIG v4)

Figure 1 displays the value grids in terms of heat maps as they are generated by BIG v4 and BIG v6 respectively. The higher the value the closer an object or agent is to the target location. In both scenarios the BIG algorithm is tasked with mapping the quickest way to arrive at the location C which is index (2,8) that has a value of 1000.

In version 4 of BIG the valuation function first explores and calculates the cells upwards until it arrives at the corridor, then turns right and travels along the central horizontal wall and all the way to the end of it. At this point no faults have occurred. The valuation function then continues around the wall and fills in all the cells in the upper 3 rows. If an agent is located in the upper left area of the map, it would travel the long way round to get to the destination instead of passing through the corridor. This is avoided in BIG v6 by maintaining a frontier that explores all directions away from the destination at the same time. Thus maintaining the same even amount of descent. Implementing the frontier has the added benefit of only calculating the space which is physically accessible in the same space as the goal location. This might have to change if teleportation is ever invented.

Performance

While the main motivation for improving the BIG algorithm was to achieve higher navigation accuracy, implementing the frontier introduced a performance benefit. The previous iteration of BIG relied on recursion to navigate and map a given level, which could become slightly problematic on the biggest of levels as the allocated heap memory would run out very quickly. The new version of BIG switched to using a loop, which changed the limiting factor from the heap to how much RAM was allocated to the client.

Assessing Accessibility
Before sending an agent to a given destination it is important to know if the agents path is obstructed. This function addresses this issue by exploring the space in similar fashion to BIG. However instead of maintaining a value grid, the grid is just used to keep track of empty cells visited. The level is explored at the same rate from both the destination location and the starting point. Unlike BIG, two frontiers are maintained. If one frontier list reaches null before colliding with the other frontier it means that the agent does not have access to the item. From here the BIG algorithm is used to map out the level and a path is drawn from the agent to its target location. Each cell in the path is added to a list and all obstacles from that list are returned.

3.4 Client

The client can be seen as our *main* script in the project. It is parsing the wanted level and creating agent instances in order to independently assign them their respective goals. Each agent then tries to find a path to its desired box and from this box to the given goal cell. Agents with multiple goals will treat them consecutively.

Since agents are computing their goal independently they are not aware of the other agent moves and thus incompatible joint actions can arise. This is why we are checking the compatibility of the decisions before sending the joint action to the server. We are able to do it by updating the current state after each joint action is executed. In case of incompatibility the agents that are producing conflicts will have their action switched to *NoOp* and recompute a new path to their goal with the current state as initial state.

In order to send consistent instructions to the server, the agents that have already fulfilled their goals wait for the other agents to accomplish theirs. Once the set of actions for each agent is empty, we check if the state is the goal. If not, we reassign the remaining objectives to the respective agents.

Binding actions to states
As previously mentioned, a state object contains the locations of all agents and boxes at a given point in time. In addition to this, it is able to store the action that was performed to reach the state. During the creation of a child state, an action is computed based on the provided directional values by the agent and the box. The set of given directions is matched with a hash table containing all possible actions, grouped by cardinal direction, in order to retrieve the positional modifiers associated with moving in a direction. These modifiers are used to compute new locations for the agent and the box in the child state, as well as verify whether the action is applicable in the current state.

Agent navigation
Upon being given a goal, an agent is able to attempt to satisfy it by completing two subgoals in sequence, which are to reach the box and then move said box.

Once a valid path is discovered(provided by BIG), the agent is able to move itself by simply following the values in each cell in ascending order until reaching the destination. This destination is chosen to be current location of the box that needs to be moved, and the subgoal is complete once the agent is one cell away from it(i.e. next to the box). In order to satisfy the second subgoal, a path is calculated from the location of the box to the goal. The agent does not compute a new path for itself, and instead follows along the same path as the box. Since both entities can individually pick a direction they want to go based on a cell with the biggest value, a situation could occur where the two entities wish to head in completely different directions. This is problematic since their directional desires need to be converted into valid actions the server is able to accept. For this reason, additional rules and checks exist to make sure that the agent is not able to split from the box.

Some general heuristics are followed here, such as the fact that when an agent pushes the box, the new agent position is always the old box position. This also applies to the agent pulling the box, where the new box position is always the old agent position. Another heuristic of note, is the observation that pushing is more beneficial than pulling in the majority of scenarios. For this reason, should the algorithm detect the agent is pulling the box, it will attempt to perform a switch operation which orientates the agent to be able to push. An example of this process is illustrated in figure 2 below.

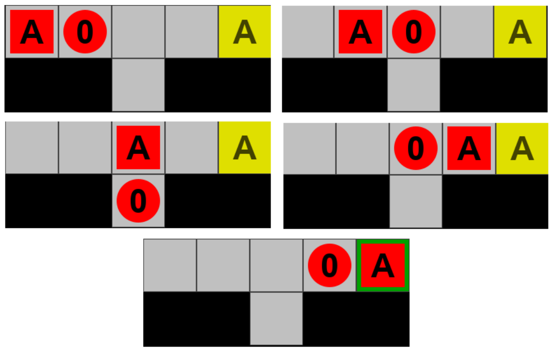

Figure 2: An example of the switch process, where agent goes from pull to push. Should be read left to right, top to bottom.

4 Results

Only levels that were solved are included in table 1 below. The results reveal that the multi-agent system is quick to find a solution compared to more conventional methods explored during the course. It is important to note that since the path is calculated during an initial state, it is considered to be the most optimal at the time of calculation. It could be the case that a more optimal path may appear at a later point. However, the path will not be recalculated should that happen unless a collision has occurred. This means that many of the easier levels can not be solved in fewer actions, however there is still room for improvement in levels that require more logistical complexity and management.

While the solution is able to quickly and effectively solve moderately complex levels, it will struggle when agents have conflicting interests in narrow spaces, such as a scenario where two agents wish to go in opposite directions in a corridor. Furthermore, scenarios where significant foresight is required will also be difficult to solve, as the client lacks a proper implementation of a planning solution.

Level	Actions	Time
SAsimple1x8	5	0,009
SAsimple1x16	14	0,011
SAsimplev	7	0,010
SAsimple2x8	17	0,013
SAushape	11	0,010
SAgalaxy	147	0,042
SAcolourSpam		
SAgrid64x64	120	0,106
SApull	9	0,009
SApullingSolvesNothing	38	0,018
SAsoko10x10	107	0,085
SAMess	80	0,046
MAexample		
MA3simple	8	0,013
MA2conflict	13	0,018
MA3conflict	22	0,033
MApriority	10	0,015
MA2boxnogoal	27	0,019
MA2serverconflict	11	0,014
MA3multiboxes	55	0,042
MAagentmultibox	21	0,016
MAagentnogoal	21	0,016
MAagentnogoaltricky	53	0,057
MAintheway		
MAMess	63	0,048
MAEasyPeasy	47	0,569
MAAiAiCap	114	0,355
MAbAnAnA	125	0,114
MAKJFWAOL	16	0,037
MABahaMAS	187	0,275
SAAiAiCap	168	0,604
SAAntsStar	238	0,119
SABahaMAS	92	0,043
SAbAnAnA	743	1,052
SAKJFWAOL	69	0,037
SAora	213	0,063
SAZEROagent	482	0,238
SANotHard	158	0,081
MAMKM	143	0,106
MAStarfish	58	0,118
SAdeepurple	168	0,441
SAMKM	63	0,024
SAStarfish	80	0,046
SAGronhoff	4	0,022

Table 1: Starfish performance results of completed levels

5 Discussion

5.1 BIG algorithm

The Big map takes advantage of full level visibility in order to determine agent navigation. In a scenario where mapping the level was a part of agent movement in the level, would mean that there was always a constant cost to the amount of steps to acquire the information. If the domain contains many agents this might still be advantages since once the level has been mapped efficient navigation can be achieved. However if there is only one agent and a single goal this might prove to a waste of resources. use gradient decent instead of gradient ascent for improved scalability.

5.2 Child-state and action search

During earlier versions of the client, the way in which actions were bound to states followed a different approach compared latest version. Upon discovering a path(provided by BIG), the algorithm would generate child states with all the possible actions from a given state. These would be sorted based on a heuristic value and placed into a stack. The heuristic used the value in the BIG path corresponding to the current position of the agent or box that wanted to advance. This approach made sure that the algorithm prioritized the child state which moved the target closer to the goal. However this approach was superseded by the algorithm discussed earlier in the report, due to the fact that it displayed occasional unpredictability when prioritizing actions. Furthermore, since it relied on a single heuristic it was not as flexible should additional logic be needed when prioritizing states. An added benefit of moving away from this approach was a boost in the overall performance, especially on more involved levels, as generating all possible child states from every state gets expensive very quickly in the scope of this problem.

5.3 Multi-agent planning

The chosen planning approach was Partial-Order planning(POP), due to the fact that it offers the flexibility of maintaining a partial order between elements, until some constraints need to be introduced or the final plan needs to be topologically sorted. This approach also allows for exploration of nodes in parallel. The action scheme has been kept high-level, in a sense that the algorithm would attempt to create a sequence of planned actions and each action could have a series of small operations it would perform. For example, if an action requires the agent to move from one location to another, there would be no actions that make sure the destination is free. Instead, this would be assumed to take place inside the action and should one of the checks fail, the action would not be satisfiable.

The process would begin by analysing the level once and creating a priority of which goal cells needed to be completed first. This is done to make sure a completed goal does not block other goals that have yet to be completed. Once an agent is assigned a task, the planning begins by defining the possible actions and states of the problem. These are used as blueprints, since upon the initial creation they do not bind to any objects and only suggest which objects are required, by naming the variables. An example of the action schemes used is shown below. Not listed are the start and end actions, as these are dynamically created based on the initial level. These two actions are also unique in the fact that the start only has effects, and the end only has preconditions.

Action: $DetachFrom(agent, box)$
Preconditions: $AttachedTo(agent, box)$
Effects: $Detached(agent) \wedge \neg AttachedTo(agent, box)$

Action: $AttachTo(agent, box)$
Preconditions: $Detached(agent) \wedge NextTo(agent, box)$
Effects: $AttachedTo(agent, box) \wedge \neg Detached(agent)$

Action: $MoveAgentNextTo(agent, box)$
Preconditions: $\neg NextTo(agent, box) \wedge Detached(agent)$
Effects: $NextTo(agent, box)$

Action: $MoveAgentTo(agent, row, col)$
Preconditions: $\neg AgentAtLocation(agent, row, col) \wedge Detached(agent)$
Effects: $AgentAtLocation(agent, row, col)$

Action: $MoveBoxTo(agent, box, row, col)$
Preconditions: $AttachedTo(agent, box) \wedge \neg BoxAtLocation(box, row, col)$
Effects: $BoxAtLocation(box, row, col)$

The plan search starts by adding all open preconditions to a queue. Then, one by one, a precondition is picked and the algorithm searches for an action that would satisfy it. Once an action is found, its preconditions are added to the queue and an ordering constraint is created. This is important for threat resolution, as well as performing the topological sort at the end.

6 Future work

6.1 Fully working partial planner

The current version of the client does not utilize the POP scheme described above, since it was not brought to a reliably working state. In order to bring the algorithm to a fully working state, elements such as threat resolution need to be fully supported. This would be done by keeping track of any conflicting effects, and fixing a conflict by shifting the ordering constraint. The current implementation is already able to keep track of the relative order between actions, so this would be extended.

Another improvement would be to implement a more sophisticated plan search algorithm, as currently the search favours breadth. A viable alternative could be the AND-OR algorithm, where AND decisions would be the exploration of all preconditions of an action and OR decisions, where the algorithm would have to pick which plan to follow and which one is more likely to lead to a quicker solution. This is also where heuristics could be introduced to aid these decisions, such as ranking plans based on the amount of open preconditions they have (XuanLong Nguyen 2009).

6.2 Multi-agent communication

Currently there is no flexible way for the agents to be able to communicate, and in case of failure to complete a task they are to simply wait or try again. However, the potential implementation of this has been given some thought. The client would run in the following cyclical scheme. All agents would receive their goals and attempt to complete them. Should an issue occur, an agent is able to leave a message stating what the issue was. This message is collected and placed into a central location. Once the agents have performed their round of goal completion, they check for any messages that might be relevant to them. Should this be the case, a task described by the message is assigned to the appropriate agent who will attempt to complete the task when it is free in the next goal solving cycle.

This messaging approach can be considered centralized, since the messages are not sent directly to an agent. While that could be a viable alternative, it would require that each agent is running in its own processing thread, detached from the central client. This tends to be more error prone, which is why the centralized approach was favoured.

7 Conclusion

The solution is competent enough to solve moderately complicated levels. Agents are intelligent enough to give way to other agents in various scenarios, and given enough time a variety of collisions can be resolved by the client using incremental steps. Compute times are fast compared to conventional search graph methods. This is by large due to the implementation of the BIG algorithm, which is able to discover a path in linear time, depending on the $row \times column$ size of the open-space in a given level. Furthermore, the conversion of a path into valid actions is done efficiently by picking the desired directions and performing a look up in a hash table, which is also done in constant time.

Overall, the current version of the implemented *Starfish* client is no where near deployment ready for hospitals or similar environments. However, that is not to say there is potential. Several of the components critical to the client, from the algorithm responsible for path discovery to the underlying agent-to-agent behaviour and structure of the program have come a long way in its development. Given enough time to work out the remaining components, the client can become a viable solution for solving the hospital navigation problem.

References

Russell, N. 1994. *Artificial Intelligence – A Modern Approach. 3rd edition.*

Watkins, C. J., and Dayan, P. 1992. Q-learning.

Witteveen, C., and Weerdt, M. D. 2006. Multi-agent planning for non-cooperative agents.

XuanLong Nguyen, S. K. 2009. Reviving partial order planning.