NotHard - Programming Project Report
Artificial Intelligence and Multi-Agent Systems, 02285, F18
May 30, 2018

Mads Filbert s152816 **Mathias Gammelmark** s134787 **Christian Hansen** s146498 **Martin Haubro** s133595

Abstract

This project is using multi-agent planning to solve Sokoban-like problems with 1 to 10 agents. A client which can solve these solutions is proposed. The client solves with a mixture of pre-analysis containing problem distribution, detecting static objects and estimating order of goals. The planning is done with online goal-prioritization, inspired from the blackboard model, along with conflict detection and offline search using weighted A*. The choice of using weighted A* is confirmed by experimenting with both BFS, regular A* and weighted A*. Furthermore as the project is part of a competition, the implementations performance is evaluated on all the levels hereof, where 28 out of 54 levels have been solved.

1 Introduction

Many companies today use artificial intelligence (AI) to run parts of their operation. Such as the Amazon warehouse in England run entirely by robots using AI (Vincent 2018). And with self-driving cars becoming a closer reality every day, everything points to the AI field to become larger and more important. For this project, we are trying to solve a Sokoban-like game, but with certain twists. This paper details a solution, a *decentralized multibody agent planner* using a blackboard system. the client does some pre-analysis by splitting the map into regions, computing all the shortest path from every location, and makes a scheduling order of how the goals should be completed. The client works for both single and multi-agent levels, and solves a total of 28 out of the 54 levels made for the *AIMAS Project Competition*, 11 multi-agent and 17 single agent levels. This paper will detail the background that laid ground for the client, how the client works and how we tested and measured the performance of our client. Lastly we will discuss the strengths and weaknesses of our client, as well as what needs to be worked on in the future.

2 Background

In this section we will go through some of the background that was done for this project. The research laid the background for how our client works. Other than the article and papers described below, we also consulted the class textbook (Russell and Norvig 2010).

2.1 Blackboard

The blackboard model, is a model for problem solving. As described in (Carver 1997) the blackboard model is based on the idea of multiple people contributing whatever they can forwards solving a larger problem. The idea of a group of experts sitting in a room watching as solutions are being made on a blackboard, and whenever one of the experts can contribute forwards solving the problem, the expert gets up and makes a contribution. The key idea is that problem solving is both *opportunistic* and *incremental*, such that a solution can be made up of multiple pieces. A blackboard system consists of three components, the blackboard, the knowledge sources and a control mechanism. The blackboard acts like a database that all the knowledge sources can access, and contains all the data and current hypotheses. The knowledge sources can modify, add or delete hypotheses on the blackboard whenever they can contribute. A key element is that the knowledge sources should act independently and should only interact with each other through the blackboard.

2.2 Planning

For planning we consulted (Bowling, Jensen, and Veloso 2005), the article describes multi-agent planning for problems with multiple goals. While the article uses a soccer-like domain to explain the reasoning, it was still applicable and useful for our project. The article describes 4 different possible plans for how a single agent reaches it goal, and the plans depend on how a single agent A can expect another agent B to perform. Do the agents act non-deterministic, like teammates, adversary or do they have overlapping goals? In our case we expect agents to act like teammates, and try to reach the same goal state, and should therefore work together.

2.3 Goal Scheduling

One major issue with many levels, is that goals often have to be solved in a specific order. Consider a single-width tunnel filled with goals, the agent might solve the closest goal within the tunnel, and therefore blocking the entire tunnel and making it harder (Or perhaps impossible in multi-agent problems). To solve this issue we found the article (Demaret, Lishout, and Gribomont 2008). The article is based on Sokoban, but the key considerations and aspects can still be applied to this type of project. If a Sokoban level has n goals, then there is a total of $n!$ permutations, of which one of these permutations represent a order of which solving goals will give a goalstate and avoid a deadlock. The article proposes a solution, it fills the goals one by one and then at every step checking if all unfilled goals are still reachable. The authors then explains that their solution in no way guarantees that the solution is effective, but rather consistent instead.

2.4 Conflict

For conflict handling we read (Pallottino, Scordio, and Bicchi 2004), the paper introduces many concepts of conflict handling in multi-agent systems. As we found out pretty quickly with multi-agent levels, is that conflict happens and are not easily solved. The paper proposes a scheme for conflict resolution in centralized planning. For centralized planning, the idea is to have a single decision maker, whom must have global knowledge of all the agents, positions and actions. The decision maker will then find admissible actions for each of the agents. For decentralized planning we read (Kai et al. 2010) which proposes different strategy based on negotiation and arbitrage, for arbitrage the strategies involves choosing one agent with a higher rank and letting that agent execute its action, while the other does nothing. Another strategy is seeing if one of the agents can execute another action that will make it closer to its goal.

3 Related work

There exists many multi-agent clients for Sokoban games, the only major difference is that in Sokoban agents can only push boxes (or stones) onto goals, and therefore the agents cannot undo wrong moves. Once a box is placed in a corner it will be stuck there. Googles DeepMind organization has a project called *pycolab* were users can create their own gridworld games (such as Sokoban), and can then be run using reinforcement learning agents communicating through a blackboard system (DeepMind 2018).

4 Methods

We will in this section describe how our client works. Some of the decisions were based on the research we did (described in section 2) and others were based on our testing (described in section 5).

4.1 Algorithm

Our client works as a blackboard-system with independent agents. Pre-analysis is done, in which the problem is potentially relaxed, simplified, and orders for goals to be solved are computed. Apart from this, the different goals to be solved are put on the blackboard, for the agents to grab. Whenever an agent grabs a task, it is removed from the blackboard, and then later, when solved, it is put back. Putting the task back, will allow another agent to pick it up, if e.g. the goal is invalidated by someone moving a box away from it.

Agents can also upload tasks where they request that space is freed by someone else, and take these down later. When an agent takes down such a task, everybody is noticed, to ensure that no one keeps on working on a task that is not active anymore.

The idea of this is to ensure that all goals will eventually be handled, and if a goal is invalidated, it will be fixed later, indirectly making the system self-healing. Apart from this, we would like to coordinate agents by having them request help from each other whenever necessary.

4.2 Tasks

We have defined a variety of tasks that the agent can solve.

- *Handle goal* - defines moving a box from one location to another. Most commonly used for moving a box to a corresponding goal. This type of task can have a number of predecessors (Explained in section 4.3) that should be completed before an agent begins this task.
- *Request free space* - defines requesting a location on the map to be free. Sometimes an agent or a box will block the path for another agent.
- *Get away from goal* - defines the act of a agent moving away from a goal. Sometimes an agent will end up on a goal, that another agent needs to be free in order to completes its task.

Handle Goal and Request Free Space are placed in a centralplanner (explained in section 4.4), and the agent can then request a task from the centralplanner, which will then find the best suitable task for the agent. If the centralplanner has no suitable task of any of these kinds, the agents will follow a generic Get Away From Goal-task.

4.3 Pre-Analysis

Before trying to solve any levels, the client will pre-analyze with the intention of simplifying the problem and predicting which order all the goals should be solved in.

All pairs shortest path To get a metric for the heuristics, all pairs shortest paths are computed to a large array. This is possible, as the levels doesn't go larger than 50x50. However, this would not scale well for larger levels, as doubling the dimensions leads to a 16 time larger level - size and speed is $O(w \cdot h)^2$.

Static Boxes For certain levels some boxes was impossible to move, and only served as a distraction, to drive up run-time and complexity. Because of this it makes sense to filter these out and treat them as walls. This can in simple cases easily be done by comparing the colors of the boxes to the colors of the present agents. By using BFS the client also analyzes if a box is in the same region [1] as a agent of same color, to avoid considering boxes which is out of reach for the agents.

Splitting Regions This analysis of finding active regions, gave rise to the idea of splitting each of such regions into its own sub-level, which could be solved independently from the others. This region analyses is done as an extension of

[1]Region is seen as a part of the level, completely surrounded by static objects.

the detection of static objects using BFS. As an active regions needs at least one agent inside, the agents will be used as a base of the BFS search. To keep track of which agents can go where, each location in the level will be represented as a bit-field, where each bit will be corresponding to whether or not the agents can go to this location, or if the location is a box or a wall. This approach will also detect static objects, by checking if the correct agents can go to the location of the boxes. However this is a simplified approach and will not take into account if agents can block each others paths or be blocked by dynamic boxes. Therefore we'll only detect a subset of static objects, but for the sake of this assignment and the given levels, is more than enough. each object will the get assigned which region they belong to, and there will be a separate planner for each region, ensuring complete decoupling among the regions. A region with a single agent, will act a single-agent level, while multiple agents will act as multi-agent.

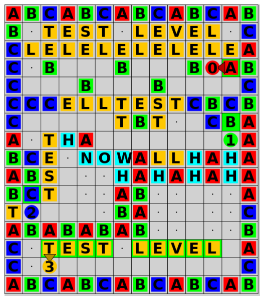

Figure 1: Extreme Level without walls and multiple regions, split by static boxes

Predecessors Based on the background done on scheduling (section 2.3), an algorithm was created to set the predecessors. The algorithm is relatively simple, in that it checks for all pairs of goals, whether goal a is a predecessor of goal b, by relaxing the problem. In practice a wall is but on goal a's location, and it is checked whether goal b can reach a fitting box despite this. If it cannot, then b is a predecessor of a, as this must be solved first. In terms of runtime and space it is $O(g \cdot w \cdot h)$, making it negligible, as it is much smaller than the all pairs shortest path. Having a list of predecessors is equivalent to Partial Order Planning, knowing what to do first.

4.4 Planning

The planning is done individually in the agents, considering them separate entities, who plan separately.

Agent The agents' thought process can be described as follows:

```
for each agent:
  if (I have a plan):
    if (I can follow the plan)
        Follow the plan
    else
        noPlan()
  if (I don't have a plan):
    noPlan()

noPlan:
    if (I don't have a task)
        get a task from the planner
    Find a plan for the task
    if (Plan found)
        Return, and get ready for executing
    else
        Ask if I can help someone
        if (I can help)
            help
        else
            Replan()

Replan:
    if (our task is a requestFreeSpaceTask)
        Do nothing, grab a new task later
    if (our task is a HandleGoalTask)
        Relax the problem* and solve
        if (not solved)
            Relax the problem and solve**
            if (not solved)
                get a different assignment
            else
                Request Free Space
        else
            Request Free Space

*First Relaxation: Remove all agents.
**Second Relaxation: Remove agents and boxes
```

The planning by the agent is done offline, considering the current world as static, not considering the actions that other agents are about to do. The least aggressive (first) relaxation is done first, since it often can be assumed that an agent will move at some point, while it is much harder to move boxes of different colors, while the second relaxation also limits the state space a lot, as only the box to be moved is kept, removing the option for the agent to move other boxes. If it's the case that there is something located on the goal destined for a task, the problem is relaxed by removing the item, as a goalstate is not achievable if another agent is placed on top of the goal, making a search meaningless. Whenever an agent has completed a goaltask, it is, as previously mentioned, pushed back to the central planner. The way agents follow plans is merely by doing action monitoring. There is no plan monitoring or goal monitoring.

Conflicts Whenever the action monitoring sees a conflict, it must be resolved. This is done by replanning, finding a new way to the goal, by the replanning-algorithm described in section 4.4. Afterwards, at a 30 % probability, agents will sleep two joint operations. This is done to ensure that the system does not deadlock, with two agents replanning, colliding, replanning, colliding indefinitely. This also leads to the system being non-deterministic, and employing a human-like behaviour.

If it is possible for the agent, from the conflict, to still find a solution to its problem, it just follows it. If not, but there is a solution to a relaxed problem, the agent will ask for the space it needs to fulfill its desire to be freed, and other agent may then aid the agent.

Search All searches are weighted A*-searches. We have used a min-heap for our frontier, therefore we want as small a heuristic as possible. The heuristic function consists of four components α, β, δ, ϵ such that for an agent a the heuristics function is:

$$h(n, a, g) = \sum_{b \in\ boxes} \alpha(b, a) + \beta(b, g) + \delta(b, a) + \epsilon(b) \tag{1}$$

where n is the current node, a is the agent and g is the goal that agent a tries to solve. The functions are defined as follows:

$$\alpha(b, a) = \begin{cases} d(b, a) & \text{agent a should move box b} \\ 0 & \text{else} \end{cases} \tag{2}$$

Where $d(x_1, x_2)$ is the distance in the map between x_1 and x_2.

Function 2 is the distance from an agent a to the box b it is assigned.

$$\beta(b, g) = \begin{cases} d(b, g) & \text{box b is the box that should reach goal g} \\ 0 & \text{else} \end{cases} \tag{3}$$

Function 3 is the distance from box b to the goal g that it should reach.

$$\delta(b, a) = \begin{cases} -1 & \text{agent a should move box b and } d(b, a) < 2 \\ 0 & \text{else} \end{cases} \tag{4}$$

function 4 is a reward if the agent is next to the box it should move.

$$\epsilon(b) = \begin{cases} -20 & \text{box b is put on a goal that does} \\ & \text{not have predecessors and that is not in risk of blocking} \\ -10 & \text{box b is put on a goal that} \\ & \text{does not have predecessors but might block another goal} \end{cases} \tag{5}$$

function 5 rewards the heuristic if a box is moved to a goal. There are two separate rewards. The first case is given if a box is moved to a goal that does not have predecessors and that is not in risk of creating conflict later on i.e. a goal that is not in a tunnel. The second case is given if the box is put on a goal that does not have predecessors but might cause conflict later on. The rewards adds the benefit that if an agent sees a box on the way to its assigned goal it is rewarded for solving it. The values are set in a way such that the agent still has a benefit of moving a bit out of its trajectory in order to solve a goal. Furthermore the rewards ensure that boxes are not moved away from a goal again. For a better implementation these values should be set dynamically such that they are dependent on the map size. This would probably improve performance as the rewards are fairly large for small maps. If the heuristic only included our distance measures $d(x_1, x_2)$ it would be admissible. However a side affect of our reward is that our heuristic becomes inadmissible, for solving the current goal of the agent. We can thereby not guarantee optimality of A*.

Centralplanner The centralplanner can be seen as a cloud that keeps track of which tasks is not solved yet, which tasks should be solved before others and which task an agent should be assigned if a task is requested. Thereby it is both a blackboard by containing tasks and a control mechanism. The centralplanner assigns the task to the agent that has the best heuristics i.e. the task it can solve the fastest. If an agent has pushed back a task that it could not solve, a penalty is added to the heuristics such that the task is down prioritised so it will not be solved until later on.

The way that agents are responsible for getting tasks by their own will, and giving goaltasks back, the planner can also be viewed as a tuplespace with assignments.

5 Experiments & results

For the purpose of testing, and comparing a benchmarking tool was built that uses the competition server, to run all levels in a directory. Then the output is parsed and that checks all the logs to see what levels was solved, in what time and in how many actions. A directory with levels the client without a doubt could solve was made, and as time passed more levels were added to this directory. This made it easy to compare benchmarks when changes were being made to client. This also served as a great way to quickly check whether ones changes did not ruin our solutions for certain levels. All the competition levels were run multiple times, to see if more levels could be solved.

All experiments and results in this section were performed on a laptop with a Intel i7-4800MQ CPU and 16GB of RAM.

5.1 Overall results

The competition levels were used metrics as for how good the client was doing. The final results for the weighted A* client in summary is:
It solved a total of 28 out of 54 levels, of which 11 is multi-agent levels and 17 is single agent levels. Appendix A.1 shows the complete results of the levels solved.

5.2 Comparing search strategies

For comparing different search strategies, every competition level was runned with three different client, one running weighted A*, A* and BFS (results can be found in appendix A). A quick summary of the three search strategies:

A* Did the second best out of the three clients, solved 26 of the competition levels. With an average time of 4 seconds and 164 actions.

Weighted A* Did the best out of the three clients, solved 28 of the competition levels. The client solved the same 26 levels as A*, but with an additional 2 levels. Running the same 26 levels as A* completed, we get an average time of 6 seconds (on average 3 seconds slower than A*) and an average of 167 actions (on average 3 actions more than A*). The reason Weighted A* on average was slower, was due to a single competition level *SAbAnAnA* (figure 4) where A* was approximately 10 times faster. The reason for this is explained in section 5.5. Removing *SAbAnAnA* from the benchmarking levels, weighted A* is on average 0.5 seconds faster (ca 12 percent), but uses 1 more action than A*.

BFS Did the worst out of the three clients, unsurprisingly, and solved only 13 of the competition levels and did not solve any levels that A* or weighted A* could not solve. For the same levels weighted A* was on average 33.4 seconds faster and used 11 less actions. We could not solve our own competition levels with BFS.

5.3 Comparing heuristics functions

A comparison between two different heuristics functions has been done, both running the weighted A* client. The first heuristics function is the one used for the overall results, and as described in section 4.4, the heuristic is rewarded for certain actions. The second heuristics function is the most *simple* one, summing all distances from goals to matching boxes, and then to matching agents. Interestingly, running the *simple* heuristics is much quicker, but takes more actions (results in appendix A.4). The heuristics function was only able to solve a subset of the same levels as the one currently used by the client, solving a total of 25 levels. Comparing the averages on the 25 levels:
Using the sum of distances heuristics functions, the 25 levels were on average 5 seconds faster, but used 25 more actions.

Interestingly once again the competition level *SAbAnAnA* comes up again, the *simple* heuristics solves it in only 1.3 seconds (compared to almost 1.5 minute), but uses 122 actions more (602 in total).

5.4 Levels

Two levels were built, not just for the competition, but also for internal testing of methods and functionality. For the single agent, it made sense to test scheduling and the setting of predecessors for tasks. And for multi-agent, it made sense to test conflict solving and searching in a larger map.

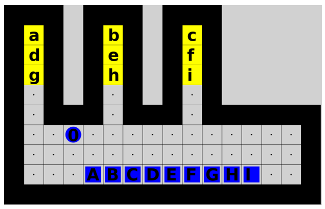

Figure 2: SANotHard - submitted competition level

Figure 2 shows our single agent level, the intent of this level is to test goal prioritization and analysis of predecessors, to make sure that the goals are achieved in correct order. The pre-analysing of the level will set the predecessors for every goal accordingly. I.e. goal *A* and goal *D* should be solved before the agent tries to solve goal *G*. So the level served the purpose testing that the predecessors for tasks were set correctly, and that the agent then solved the tasks in a correct manner.

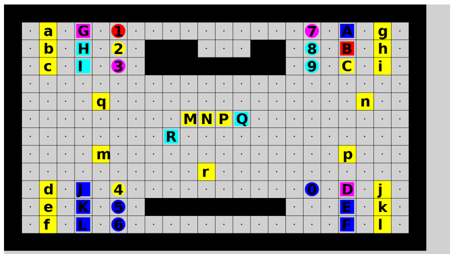

Figure 3: MANotHard - submitted competition level

Figure 3 shows our multi-agent level, the intent for this level is to test the solving of conflicts, by having a relatively chaotic level with a lot of agents. Furthermore it's a test of branching factor by having a large space, which will penalize wide searches.

5.5 SAbAnAnA

This section will discuss the performance of the client based on the heuristics chosen. This level is selected as it showed some interesting behaviour from the client. Figure 4 shows a state in which the weighted A* gets stuck. The weighted A* will try the most direct way to its goal. At this point all the best solutions go through the middle of the map. But as seen in the figure there is a lot of boxes in the middle which will expand the state space a lot creating many branches to explore. Therefore search time will be much longer than for A*. In this special case the weighted A* start will therefore be a lot slower at finding a solution than A*. This is a contradiction to the relationship that we would normally expect between weighted A* and A*. Usually there will be a tradeoff between optimal solution and search time where a more greedy function will be faster while pure A* will obtain a better solution.

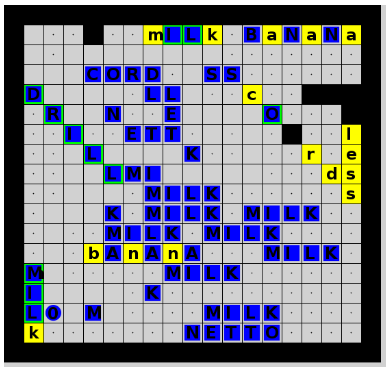

Figure 4: SAbAnAnA state where the weighted A* gets stuck for a while

Furthermore it was seen that using the *simple* heuristic was even faster but with a much worse solution. The reason for it using more actions is that it does not solve nearby goals, that it can solve on the way to solving its own goal and that boxes that are placed on goals might be removed from the goal again, when the agent moves towards a new goal. However a lot fewer nodes are explored as there are fewer nodes with low heuristic scores and it therefore very fast at solving the level. The results for the *SAbAnAnA* level with the different clients are shown in table 1.

Table 1: Benchmark results for competition level *SAbAnAnA*.

Client	**Milliseconds**	**Actions**
Weighted A*	89503	480
Weighted A* w/ *simple* h	1295	602
A*	10358	438

5.6 Scheduling

To test the strength of the goal scheduling, results of benchmarks was done with the goal scheduling removed from the client. The benchmark was run on the same 28 levels as the weighted A* client could solve. The results were that now the client only solved 18 levels out of the 28 levels. The client was no longer able to solve our own single agent client, which of course was designed to test the scheduling feature. Unsurprisingly, most of the levels the client was no longer able to solve, had specific orders in which the goals should be solved. Full results shown in appendix A.5.

6 Discussion

We experimented with our client using different search strategies, and different heuristics functions. In the end we used the solution that solved the highest amount of levels, our weighted A* search strategy with a heuristics function with different rewards, solved the most levels. Every other combination solved only a subset of the same levels, and never solved any levels that our solution could not solve. Whether or not one solution is better than the other is hard to say, some of the solutions solved less levels, but solved them faster and by using more actions. The solution we ended up choosing, solved the most levels, and had the lowest average of actions used.
A weakness of our solution is that the planning is offline, and there is only action monitoring, but no goal monitoring or plan monitoring. This leads to agents following potential impossible plans, and on several occasions choosing suboptimal solutions, where replanning on the go would lead to quicker solutions. However, the system, as it is, finds solutions eventually. Another weakness is possible bugs, ensuring that agents don't always come helping.
We manage to successfully analyse the level, and make some simplifications, finding a usable goal ordering, albeit not perfect. We have perfectly split into regions, decomposing the problem, which also is handy when solving e.g. *MABahaMAS*, which consists of 10 different regions. By removing the goal ordering functionality from our client, we solved 9 fewer levels, proving the usability of this. Our client solves 27 (17 single agent levels and 10 multi-agent) out of 54 levels, which isn't perfect, but shows a certain level of usable functionality.

7 Future work

In this section we will describe the future works related to this project, which implies what we would have liked to see in this one.

7.1 Scheduling

The scheduling is not entirely complete, as it may still find wrong predecessors. On top of this, it doesn't handle the case, where two goals may be predecessors for each other, which might possibly be handled by improving the goals boxes first.

This could possibly be handled by different algorithms, however, it is a limitation in the current implementation. While it is desirable to find i an efficient plan, it is most important to actually solve the goals

7.2 Path Finding

For the future, it could make sense to try out alternative pathfinding algorithms, e.g. a greedy best-first search, and see how it compares against A* in terms of time spent as well as actions. This also involves experimenting with heuristics.

Distance Oracles The all pairs shortest path distance that's used in the heuristics is feasible for levels of the current size, however, if larger, would be unusable. This has to be resorted, which can be done through e.g. distance oracles, having a much smaller space usage. This would, however, lead to more inaccurate heuristics, as it is only approximation of distances (Thorup and Zwick 2001).

7.3 Finessing Multi agent coordination

The current implementation leaves something to desired with respect to multi-agent coordination, especially actually fulfilling the requestfreespacetasks. This is much more implementation based than conceptual. Currently agents working on a FreeSpace task can't upload other freespacetasks, for example, leading to potential deadlocks, and that not all are completed.

7.4 Ranking assignments

Related to pre-analysis, it would be desired to be able to rank assignments, e.g. for the more stressed agents to have priority, while the agents with fewer or easier assignments should be more active in helping these.

7.5 Local Multibody Planning

Doing multibody planning on a small scale as conflict resolution would be an interesting concept. Having the agents decoupled, as the design is, but, at a conflict, with an agent colliding with another, constructing a good, short multibody plan in which the agents get clear of each other, while getting closer to completing their subgoals.

This would especially let agents making space for each other better, and most likely lead to less actions when solving levels.

An implementation was briefly attempted, but the complexity of the problem made us focus elsewhere.

7.6 Subgoals

To further reduce search space for the agents you could divide the goals into two. The first part of the goal would be to move the agent to the box, while the second goal would be to move the box to the goal. The search space is reduced when doing goal decomposition like this, as all branches are discarded after solving a subgoal. Different heuristics for the different subgoals would also be desirable. The first would simply want the agent to get closer to the box and solve ”easy” boxes along the way, while the other would want the box to get closer to the goal.

References

Bowling, M.; Jensen, R.; and Veloso, M. 2005. Multiagent planning in the presence of multiple goals. http://www.cs.cmu.edu/ mmv/papers/05planning.pdf. Accessed: 2018-05-23.

Carver, N. 1997. A revisionist view of blackboard systems. http://www.cs.siu.edu/ carver/ps-files/maics97.ps.gz. Accessed: 2018-05-29.

DeepMind. 2018. pycolab repository. https://github.com/deepmind/pycolab. Accessed: 2018-05-30.

Demaret, J.; Lishout, F.; and Gribomont, P. 2008. Hierarchical planning and learning for automatic solving of sokoban problems. https://orbi.uliege.be/bitstream/2268/5895/1/bnaic2008.pdf. Accessed: 2018-05-28.

Kai, H.; Zhonghua, Z.; Zheng, Q.; and Bing, L. 2010. Conflict resolution in multi-agent systems based on negotiation and arbitrage. https://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=5478068tag=1. Accessed: 2018-05-29.

Pallottino, L.; Scordio, V.; and Bicchi, A. 2004. Decentralized cooperative conflict resolution among multiple autonomous mobile agents. http://www.centropiaggio.unipi.it/sites/default/files/vehicles-multi-cdc04.pdf. Accessed: 2018-05-23.

Russell, S., and Norvig, P. 2010. *Artificial Intelligence: A Modern Approach*.

Thorup, M., and Zwick, U. 2001. Approximate distance oracles. http://www.cs.jhu.edu/ baruch/teaching/600.427/Papers/oracle-STOC-try.pdf. Accessed: 2018-05-30.

Vincent, J. 2018. Welcome to the automated warehouse of the future. https://www.theverge.com/2018/5/8/17331250/automated-warehouses-jobs-ocado-andover-amazon. Accessed: 2018-05-30.

Appendix

A Overall results

A.1 Weighted A*

Table 2: Performance of client with weighted A*

Level	Milliseconds	Actions
MAAiAiCap	318	118
MABahaMAS	503	65
MABeTrayEd	323	200
MADaVinci	284	108
MAEasyPeasy	313	56
MAGreenDots	50299	200
MAKJFWAOL	286	19
MAMagicians	820	72
MANikrima	278	38
MANotHard	379	83
MAbAnAnA	290	90
SAAIFather	321	102
SAAiAiCap	1596	73
SAAlphaOne	304	108
SAAntsStar	329	239
SABahaMAS	295	95
SABeTrayEd	296	259
SAGreenDots	300	233
SAJMAI	276	130
SAKJFWAOL	275	56
SALobot	613	80
SAMagicians	328	450
SANikrima	305	253
SANotHard	297	163
SAZEROagent	332	497
SAbAnAnA	89503	480
SAbongu	278	89
SAora	297	220

A.2 A*

Table 3: Performance of client with A*

Level	Milliseconds	Actions
MAAiAiCap	285	115
MABahaMAS	513	65
MABeTrayEd	356	189
MADaVinci	271	108
MAEasyPeasy	478	54
MAGreenDots	61960	240
MAKJFWAOL	298	19
MAMagicians	299	70
MANikrima	294	38
MANotHard	587	80
MAbAnAnA	289	86
SAAIFather	652	96
SAAlphaOne	304	92
SAAntsStar	347	239
SABahaMAS	285	95
SABeTrayEd	315	253
SAGreenDots	313	227
SAJMAI	281	130
SAKJFWAOL	317	56
SALobot	897	74
SAMagicians	363	452
SANikrima	300	242
SANotHard	317	161
SAZEROagent	354	497
SAbAnAnA	10358	438
SAbongu	324	93

A.3 BFS

Table 4: Performance of client with BFS

Level	Milliseconds	Actions
MAAiAiCap	534	115
MABahaMAS	6699	65
MABeTrayEd	39950	197
MAEasyPeasy	7931	50
MAGreenDots	175712	225
MAKJFWAOL	296	19
MAMagicians	6129	86
MANikrima	278	38
MAbAnAnA	44709	91
SAAIFather	7190	203
SAAlphaOne	6405	97
SABeTrayEd	136374	259
SAZEROagent	77124	524

A.4 Weighted A* *w/simple heuristic*

Table 5: Performance of client with weighted A* using a simple heuristics function

Level	Milliseconds	Actions
MAAiAiCap	290	118
MABahaMAS	519	65
MABeTrayEd	309	200
MADaVinci	285	108
MAEasyPeasy	313	56
MAKJFWAOL	272	19
MAMagicians	1350	66
MANikrima	284	38
MANotHard	456	91
MAbAnAnA	287	99
SAAIFather	602	108
SAAiAiCap	568	73
SAAlphaOne	340	90
SAAntsStar	324	239
SABahaMAS	284	129
SAGreenDots	293	253
SAJMAI	281	117
SAKJFWAOL	284	70
SALobot	360	78
SAMagicians	406	856
SANikrima	300	265
SANotHard	283	163
SAZEROagent	301	497
SAbAnAnA	1295	602
SAbongu	280	89

A.5 Weighted A* without scheduling

Table 6: Performance of client with weighted A* without scheduling of goals

Level	Milliseconds	Actions
MAAiAiCap	302	118
MABahaMAS	486	65
MAEasyPeasy	308	56
MAKJFWAOL	276	19
MAMagicians	301	100
MANikrima	297	38
MANotHard	381	83
MAbAnAnA	287	90
SAAIFather	308	231
SAAiAiCap	1548	73
SAAlphaOne	721	122
SAAntsStar	335	239
SABahaMAS	287	95
SAKJFWAOL	298	56
SALobot	613	80
SAMagicians	329	450
SANikrima	294	253
SAbAnAnA	88654	480
SAora	292	220