An Actor-Based One Size Fits All AI
AIMuffins
02285 AI & MAS
02-06-2016

Jakob Larsen
s033092

Mike Berg Karstensen
s123237

Anders Rydbirk
s113725

Abstract

We present a single- and multi-agent client for a variety of the Sokoban problem. Our solution consist of a centralised planner using partially ordering of the goals and distribution of tasks to decentralised agents. For plan merging we have uses classic resource allocation and multibody conflict resolution. This approach has shown great performance in all disciplines and as our results shows, we have achieved our goal of creating an one size fits all client.

1 Introduction

Our aim in this project has been to develop an AI with overall great performance in both single-agent (SA) and multi-agent (MA) systems. We have designed our AI with the philosophy that the problem can be divided into an intelligent ordering of the goals and an excellent solver for a single goal. This relaxation allows us decrease the complexity while it gives us a great trade-off between the optimal solution length and the time used to find a solution. This philosophy does not limit us to SA problems but can easily be applied to MA problems using smart resource allocation and plan merging.

2 Background

We assume that the problem is partly decomposable, and we have assigned each goal field a specific box based on the initial state. We perform a static analysis of the level and add ordering constrains between goals to build a constraint graph (Rusell and Norvig 2010)[section 6.5]. We use single-agent multibody planning to decompose and distribute goals to the agents, while each agent uses decentralised hierarchical task networks to refine the goal into low level actions that can be solved using forward state-space search.

In the following a *plan* will refer to a solution devised by an agent for one of the goals in the set of partly ordered goals. We use a centralised plan merger where all plans are treated in a first come, first served manner. Our client uses a combination of online and offline planning to ensure no conflicts when the plans are executed. We simulate each plan and allocate resources to ensure no other agent uses the same square at the same time. In case of a conflict, we discard the plan. In specific cases, we switch to multibody planning as will be described in later sections.

The modeling of our client is heavily based on the actor model to facilitate the communication between the entities. This enforces that each actor in the actor system has clearly defined responsibilities and boundaries (Nobakht and de Boer 2014).

3 Related work

Others have applied static analysis to the level when solving the classic Sokoban problem, such as identifying deadlocks and other features of the level (Virkkala 2011). The problem at hand is a relaxed version of the classic Sokoban problem where the *Pull* action did not exist and therefore boxes could end up in positions which made the level unsolvable. We have applied much of the same techniques to identify *savespots* that is, fields to place a box so that it is not in the way of the agent's goal.

We have applied our own version of identifying rooms, tunnels and chambers (Virkkala 2011) combined with a goal decomposition similar to *goal scheduling* (Jean-Nol Demaret 2008) to add additional ordering constraints that we did not detect through our resource analysis of the level. Thus, we have applied existing techniques to the problem at hand to optimize our goal decomposition and limit the search space.

4 Methods

Our approach is heavily influenced by our modeling of the domain. We have approached the problem by adding centralised multibody Planners combined with decentralised agents very similar to the structure presented in the Martha Project (Alami et al. 1998). Our modeling of the domain contains three actors:

- **Planner:** The actor responsible for goal decomposition and prioritizing, distribution of goals to the agents, and keeping track of each agent's status.
- **Agent:** The actor responsible for devising a plan for each assigned goal. We operate with one agent actor for each agent present in the level.
- **Merger:** The actor responsible for merging plans and keeping track of the total accepted moves for each agent.

The rest of this section will be divided into the logically responsibilities of each of these actors.

4.1 Planner

Level Analysis and Goal Decomposition Our admissible heuristic is based on a static All-Pair-Shortest-Path (APSP) which ignores the boxes and agents in the level. However, to improve performance, we only calculate and store the shortest path from each square on request. In the worst case, the performance impact will be similar to APSP; $O(n^2)$ space and time where n is the number of squares in the level.

Our goal decomposition are comparable to Partial-Order Causual Link Planning as presented in (Geffner and Bonet 2013)[section 3.9]. However, we identify ordering constrains by analysing the map or by using other domain-specific information. Only after the ordering of the goals has been found, we apply STRIPS actions (Rusell and Norvig 2010)[section 10.1] as described further in the Agent actor.

We operate with three types of goal in our goal decomposition. The first and most simple is the actual goal squares of the level. We assign each goal a specific box and then investigate the resources between the box and the goal based on the shortest path calculation. By doing this, we identify if the resources of the goal contains another goal square. In that case, we add an ordering constraint between the goals.

We also apply a higher abstraction of goals. By analysing the level, we construct a graph of the level consisting of nodes and edges similar to rooms (nodes), tunnels (edges) and chambers (nodes) (Virkkala 2011). However, an edge is only an edge if it contains a goal square.

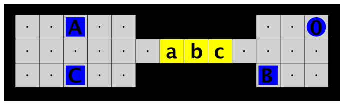

Figure 1: Limitations of resource analysis.

Consider the case in figure 1. With the resource analysis, we get the following ordering constrains: $c \prec b, c \prec a, b \prec a$, but placing C on c results in blocking of the access from B to b. Our graph analysis identifies that there exist an edge containing the set of squares a, b and c between the squares to the left and to the right, $node_{left}$ and $node_{right}$ respectively. A new goal is applied that requires all the assigned boxes to be in only one of the adjacent nodes. This goal is added as an ordering constraint to these three *goals* on the *edge*. When a plan for this goal is accepted, the resource analysis is updated to reflect the new box placements and the potential new ordering of the goals.

The last type of goals we operate with is in multiagent levels when one agent requests other agents to remove boxes that blocks its path. We consider this as new goals that are added as ordering constrains for the goal that required the boxes to be moved in the first place.

Notice that the agents cannot request other agents to move. This is only handled in the centralised multibody plan merger in case of resource conflicts.

We refer to the ordering constrains for a goal as the *preconditions of a goal* in the latter. When ordering the goals, we apply a topological sort to detect cyclic references in the ordering constrains. In case of a cycle, we simply relax the problem and delete one ordering constraint in the cycle. We argue that this pragmatic solution is applicable since we cannot solve a level with cyclic references with our current configuration since then at least two goals will never have all preconditions fulfilled. Thus, by relaxing the ordering constrains on cycles, we try to solve the level instead of flagging it as unsolvable.

Goal Prioritizing When an agent is assigned a goal to process, it is based on the agent's current *state*. This refers to, where the agent is placed and the configuration of the level based on the agent's current *time* that is, the number of moves the current agent has conducted so far. The filters used are:

- The goal can be reached by the agent,
- The goal is not completed,
- The preconditions for the goals are fulfilled,
- The goal is not being processed by another agent,
- The agent has not attempted to solve the goal beyond a threshold without success for the current *time*.

After all these filters are applied, we allocate the goal with the lowest estimate to the current agent. The goal is then locked until the agent reports a plan back to the Planner entity.

Agent Status When operating with MA, the Planner keeps track of each agent's status, which can be one of the following:

- Working
- Idle
- Replanning
- Completed

This is for handling limited resources among agents such as a narrow path, or for handling dependencies between agents of different colors. Notice that when operating in a SA level, this defaults to that the agent should always be set to *working* until completion.

In case an agent have completed all goals with fulfilled preconditions but not completed all goals for the agent, it will be awaiting the goals of other agents to be completed before it can continue. Thus, it will be flagged as *idle* in time t_{idle}. When other *working* agents reports a completed plan back that gets accepted in time t_{acc}, all *idle* agents will be set to *replanning*.

For the sake of argument, let all replanning agents have time t_{idle}. The Planner will now try to assign a goal to each replanning agent. If no goals are applicable, the Planner will request that the agent is pushed forward in time from t_{idle} by applying one or more persistence actions, which in this problem domain means that the agent applies *NoOp*s actions. The agent is pushed forward in time with a fixed time leap until it either reaches t_{acc} or acquires an applicable goal. In the latter case, it is set to *working*, or else it is set to *idle* and then awaits another agent to complete a goal in the future.

This works as a fallback mechanism and is also used to detect that no agent is working. In that case, the Planner resets its goal prioritizing and increases the number of allowed search depth and number of explored nodes in all *WA** searches. Then all agents are set to *replanning* to try to solve the level. This functionality detects if the client keeps restarting without any progress and terminates.

4.2 Agent

Hierarchical Task Network Each agent uses a Hierarchical Task Network (HTN) to break down a goal into atomic actions. All HTNs in our solution are based on a High-Level Action (HLA) (Rusell and Norvig 2010)[section 11.2.1] which contains a specific goal, an agent, a state representing the level, and all the allocated resources. Thus, each HLA provides all the information that the agent need to devise a plan. The agent will add different actions to the HLA to analyse the required resources and to determine if any boxes is in the way or alike. However, all HLAs breaks down into one or several Low Level Actions which can be of the type *MoveBox* or *MoveAgent*. Both contains a set of one or more atomic actions, where the former moves a specific box while the latter only contains *Move*-actions for the agent.

Box Removal and Savespots A key issue in the domain is to detect if any boxes have to be moved to solve the given goal. Thus, we assume that we need to investigate and validate the path from the box to the agent and from the agent to the goal field. We identify the resources on the shortest path between these by using our APSP calculations and scan through the resources to identify if any boxes is in the way. Notice that we do not consider other agents at this point.

In case one or more boxes is detected on the path, we try to determine if they are blocking or not. An example of this is shown in figure 2. Three boxes are identified on the resources but in this case, we do not need to remove them. We try to search past the boxes from the field *F1* to *F2* using *WA**. We simply place a imaginary agent on *F1* and assign *F2* as the goal square of the agent, and the agent is not allowed to move any boxes when doing this. We apply an upper limit to the number of allowed moves in the search which is a function of the number of boxes in a row. Thus, in this case we can easily move by and decide not to move the boxes out of the way.

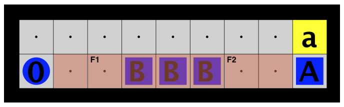

Figure 2: A case where boxes are placed on the resources from the agent to box *A*. *F1* and *F2* indicates the before and after fields for checking if the boxes has to be moved out of the way.

The interesting case here is that the optimal solution is to actually move the boxes out of the way. However, in a MA level where the *B* boxes might be of another color, removing them requires more moves. In that case we prefer that agent 0 can solve its goal without involving other agents. Thus, we prefer a slight relaxation of the optimal solution to decrease the number of boxes that has to be removed.

When the agent has detected all boxes that has to be removed, it scans through them decide if it can remove them itself or if other agents have to move them. In case it has to be removed, the agent will return an incomplete plan which contains all the boxes that has to be removed. These will then be added as ordering constrains for the goal that the agent tried to solve.

In case it can remove all of the boxes by itself, it will continue and analyse the map to find a *savespot* for each box using a BFS. By using the static analysis of the map and different characteristics of the map such as detecting how many savespots are available, the agent decides how aggressively it should allocate savespots. The more aggressive, or *greedy*, the less the agent cares about keeping corridors open and the accessibility of boxes irrelevant to its current goal. Thus, the more limited resources, the more greedy the agent becomes.

Search We uses a *WA** forward state-search using our APSP as heuristic (Rusell and Norvig 2010)[10.2.1]. The search is always limited to only moving the given agent and the assigned box that the agent tries to move to a specific square. This is due to the assumption that the agent do not need to move other boxes as these have been identified and moved during the validation of the path. Thus, by paying the cost of the analysis, we limit the search space considerable by focusing the search.

When assigning a goal to an agent, we make a simple estimate of how the level will look after the completion of the goal. Given this educated guess, the agent is assigned an expected next goal in the HLA so that it can plan for which direction it should be facing after completion of the current goal. This decision is also based on the APSP heuristic.

During the search the agent cannot use resources already allocated by other agents. However, to enable the agent to await another agent that passes by, the agent is allowed to add up to 5 *NoOp*-moves during the search. This is configurable but we found that this factor was a great trade-off between the increased size of the search space and the ability of the agent to find a valid plan.

4.3 Merger

Merging The Merger treats plans in a first-come, first-served manner and simulates each move before accepting a plan. This ensures that there will not be any conflicts when sending commands to the server. It works under the assumption that any accepted plan can never be invalidated by a plan accepted afterwards. For this assumption to hold, we need to detect all possible conflicts in simulation. By simulating it, eventual resource conflicts are detected and mitigated before the plan gets accepted. We enforce the domain restrictions and detect eventual plans that are out of sync with the accepted plans. This could be that the agent tries to move a box that has already been moved or that the agent has been moved by the Merger in a multibody conflict resolution.

However, other bookkeeping techniques are required to ensure that the accepted plan is valid. An agent or a box is not allowed to end its moves on a field already allocated in the future. By book keeping a timestamp for each field for the latest time that the field was allocated, we can ensure that the assumptions hold.

In case of an unresolvable resource conflict, the entire low level action is rolled back and all the move actions are discarded. However, in case a plan consists of multiple low level actions, only the low level action resulting in the unresolvable resource conflict is rolled back and the rest is discarded. Thus, plans can be partly accepted.

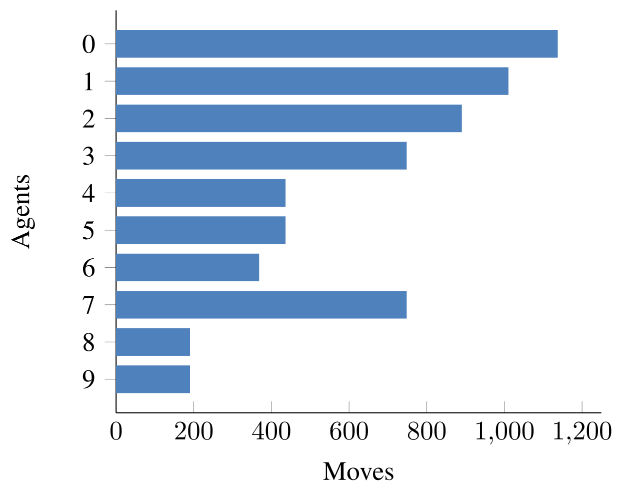

Figure 3: The merged plans for the level *MaWallE* (Levels 2016).

An example of a completed solution can be seen in figure 4.3. Here it is shown how the number of moves varies for each agent. However, a few agents have exactly the same number of moves, such as agent 4 and 5, and 8 and 9. In the given level, they are of the same color, respectively. This is an example of how *replanning* works in cooperation between the Planner and the Merger. Agent 4 and 5 have attempted to solve the same goals but due to eventual preconditions and that the goals are locked when being processed, one of them has been set to *idle* and then set to *replanning*. As the agent has been pushed forward in time, at some point the agent reaches a point where all the goals for the given agent color are completed. Thus, at this point, the agent is then set to *completed*. The Merger is responsible for moving the agent forward in time and applies the *NoOp* actions for the agent upond request from the Planner.

Multibody Planning We only switch to multibody planning when a resource conflict occurs with a non-working agent. This is a strict requirement, or else we would violate the assumptions for the Merger since moving a working agent would invalidate an already accepted plan for that agent.

When one or more non-working agents are detected on the required resources for a plan being simulated, the rest of the plan is applied. Then, the Merger will apply a recursive function on the first non-working agent. The Merger attempts to move the non-working agent by moving it to a field not allocated by the plan. The non-working agent cannot move across the requesting agent.

1. The non-working agent could move out of the way of requesting agent and the two plans can be merged and both are accepted.
2. The non-working agent could move out of the way of the requesting agent but the two plans could not be merged. The plan to move the non-working agent is accepted.
3. The non-working agent requires a third agent to be moved. The merger will now try to move the third agent out of the way and then recursively apply the plans for the requesting agents. The third agent cannot move across the requesting agents.

The goal of this is to move the non-working agent that is the furthest away from the requesting agent first. Whether this conflicts with the plan of the requesting agent is of second interest. In case there is a conflict, the non-working agent has been moved and the requesting agent will replan with respect to these allocated resources. However, there is one clear drawback to this approach. If the requesting agent blocks the path for the non-working agents, they cannot request the first to move, resulting in a deadlock.

5 Test Results

Test Setup The tests are performed on a machine with the following specs:

- Processor: Intel Core i7-6500U 2.5GHz
- Memory: 16GB DDR3 ram

Test Levels In the benchmarks, we use a selection of levels from (Levels 2016) to test out the capabilities and properties of our client. The levels are selected to illustrate how the client performs in levels of different sizes and complexities.

- MAYSoSirius
  A small and relatively simple MA level, that contains few goals. It is interesting because it does not use a lot of memory and time to process, giving a good image of the base resource usage of the client.
- SASkynet
  A SA level of medium size and complexity. Interesting because it contains different kinds of goal ordering problems combined with the medium size.
- MAWallE
  A very large, very complex MA level. It is interesting because it holds the maximum number of agents and it contains a bottleneck that all agents need to pass through several times which causes many resource conflicts, pushing the client to its limits.

5.1 All Pair Shortest Path Performance

It is interesting to consider the performance of our APSP heuristic, since it is essential in our estimation, search calculations and resource analysis. The interesting parameters here is the memory and time used to find a solution for a level given the asymptotic analysis of APSP.

On Request vs. Initial Computation Our initial approach was calculating APSP before solving the actual level. This meant potentially calculating APSP for fields never touched by any agent, which on larger levels such as *MAWallE*, lead to a noticeable delay upon start of the client. To increase performance, this approach was changed to computing the shortest path only as requested.

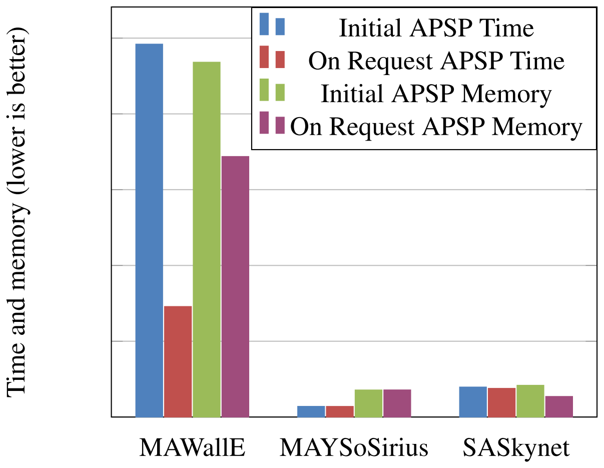

Figure 4: APSP benchmark: On Request vs. Initial Calculation.

As seen in Figure 5.1 the impact on smaller levels is practically non-existant. This is reasonable given that on smaller levels, the resource utilization is higher, meaning that the total work of the on request-approach ends up being close to calculating APSP initially. Both memory and time usage are almost the same in both approaches.

Conversely, the impact on large open levels such as *MAWallE* is clear. We take advantage of the fact that a lot of the fields are never reached by any agent, from which the client gains a huge performance increase. Since we save not only memory by calculating less shortest paths, we also gain a solution speedup resulting in a lower overall time to find a solution.

Ultimately the APSP performance is bound by the size of the level. Thus, the on request APSP will in the worst case be calculated for the same number of fields as calculating them all initially.

5.2 Search Algorithm

Since moving boxes is an essential action in the problem domain, it is interesting to examine the performance of one of the core components in the execution of such actions. Essential to the movement performed on the level is the search algorithm, as mentioned in the description of the Agent.

To quantify the effectiveness of our current solution of searching using the *WA** algorithm, we will substitute it with a *Greedy best first* algorithm and compare the performance. The main interest will be the solution length and the number of moves used to complete a level using each search algorithm.

	Memory(mb)	Time(ms)	moves	Level
WA*	40.56	2653	705	SASkynet
Greedy	42.88	2736	717	SASkynet
WA*	471.26	18270	1100	MAWallE
Greedy	461.42	16942	1387	MAWallE

Figure 5: Search benchmark: Weighted $A*$ vs. Greedy.

As seen in figure 5, the impact on memory and calculation time are relatively small. Running the test multiple times show that they fluctuate in the same amounts of system resources used. The penalty on levels such as *SASkynet* is not significant as the greedy solution is only 1.5% longer. The low impact on solution length is in part because of our goal decomposition, and the limitations we apply on our search algorithm, limiting the search space and thereby the difference in the efficiency of the algorithms.

For levels of considerable size and complexity such as *MAWallE*, there is a significant difference with an around 26% longer solution from the greedy algorithm. In more complex levels our goal decomposition and search limitation still solves the problem, but the greater complexity adds more possibilities for the greedy algorithm to find sub-optimal solutions.

The efficiency of the search algorithms in our solution greatly depends on the complexity of the level. Because of the goal decomposition and search limitations the overall memory and time impact are not all that noticeable considering the different algorithms. More important is to find an optimal solution for each goal since that naturally will give an overall more optimal solution.

5.3 Competition Results

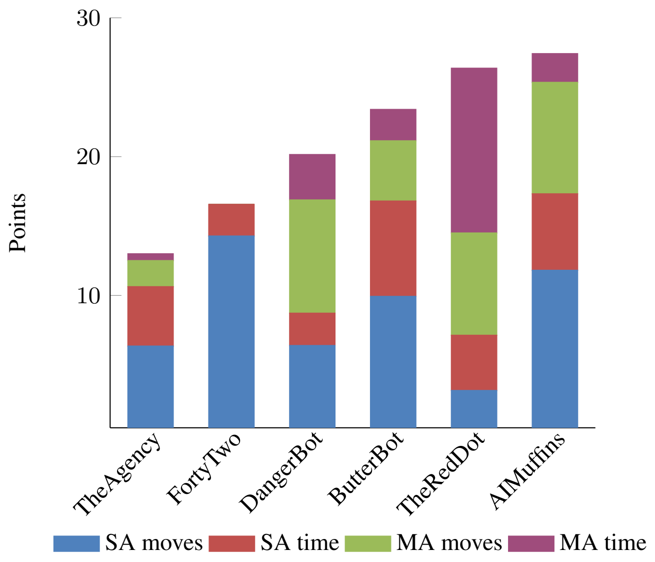

Figure 6: Aggregated scores from Competition Day.

Overall Score Seen in figure 6 is an aggregated score of the top 3 best teams in each category. It is worth noting that while we have the highest overall score, we are not the best in any of the four categories.

Another interesting aspect is that we are the second-best in all categories except for the time it takes us to calculate the solution for MA levels in which we are placed fourth. It could be a indicator of that the way we merge plans and discard unaccepted plans could be improved. As the scoring in the competition is relative to the best solution in each level for the given category, it is clear that the best performing AI in the MA time track outperforms all competitors. When comparing their solution with ours, we have approached plan merging and conflict resolution in completely different ways. TheRedDot uses a MA strategy where only one agent moves at a time, reducing the complexity of conflict resolution and therefore minimise time used to find a solution. This simple approach allowed TheRedDot to solve three more levels in the MA moves track compared to us, but we still scored higher due to our ability to produce more optimal solutions.

Problem Areas Among the levels that were in the competition, there were a few levels we were not able to complete at all. The levels we could not solve involved a type of problem our client could not handle.

In the SA levels, the two levels we could not solve were both mass-box, limited-resources levels. These levels contained too many boxes blocking the path to solve a goal for us to be able to use our savespot algorithm. Other teams seems to have beaten the level trough simply searching their way using a *WA** search, thus solving levels we could not.

Our general problem in the MA levels, turned out to be the levels where there are a lot of agents in limited space with high inter-dependency. The problems caused deadlocks in our multibody conflict resolution handling and *replanning* fallback functionality.

The results of the competition provides a nice indication that we have achieved our initial goal of creating a client that is not necessarily the best at anything, but more of a *one size fits all* solution.

6 Discussion

We have used APSP as our primary heuristic and all resource analysis are based on this. This algorithm uses $O(n^2)$ time and space where n is the number of fields, which in larger levels could lead to poor performance and high memory usage. Due to the upper limit on the level size, the scalability of APSP has not been an issue. In much bigger representations of the problem, the use of this heuristic should be compared to other heuristics that are less time and memory consuming. In our case APSP has been an excellent heuristic providing exact information in constant time after preprocessing.

The choice of multibody conflict resolution was primarily based on the assumption that a non-working agent blocks another agents path is relatively rare. We based this assumption on the problem domain that contains at max 10 agents while the number of boxes and the size of the levels can be considerably larger. Our client falls short in many cases where levels have limited free resources and MA conflicts. If the ratio between the level size and number of agents changed, one could argue that we should switch from centralised multibody conflict resolution to direct communication between agents. Many cases exist that our multibody planner cannot solve. However, there exists possible optimisations and in the given problem domain we could solve most if not all of these limited resource problems with further use of centralised multibody conflict resolution.

We have applied different fallback techniques to relax a problem we cannot solve. Examples of this could be the different levels of aggressiveness of the *savespot* functionality, relax cyclic references in our ordering constrains or increase the search space if all agents end up *idle* and no solution is found. There still exist several cases where fallback techniques could be applied when our first, preferred solution does not apply. We have assumed that all types of goals matches the pattern that all blocking boxes can be moved out of the way before moving the goal box to the goal field. In some cases, this assumption falls short and our client stalls. An example could be mass-boxes, limited-space problems as shown in figure 7. Many boxes are blocking the way and the entire path cannot be cleared all at once. These cases can often be solved using simply a *WA** search, but our constrains are too strict.

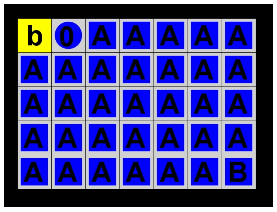

Figure 7: The mass-boxes, limited-space problem.

We prefer solutions that can be swapped out and applied when the primary method fails. These different kind of relaxations of the problem could be used as last resort fallbacks that loosen up a deadlock.

Applying the actor model to our solution has helped decreasing the complexity in the project and defining clear responsibilities of the different entities in our client. This is recommendable for similar problem domains.

Our goal was to create a one size fits all client. We have put great focus on ordering constrains among goals and excellent single goal solving. When it comes to prioritizing among applicable goals, we apply a simple estimate based on the distance from the agent to the box and from the box to the goal. Different estimation techniques could be applied as well as bidding from different agents of the same color. However, we settled with this suboptimal solution for estimates given that we aimed for solving many problems great instead of solving a few close to perfect. Considering the results from the competition, our client did just that. Placed second in three of four categories and a combined overall highest ranking client, we believe we achieved our goal.

7 Future Work

The HTN used by the Agent actor are very specialised towards the current problem domain, but many of the actions have been possible to reuse in different use cases when the Agent are creating a plan for different kind of goals. Thus, in future works if adding a new Top Level Actions to the HTN, several low level actions could be reused.

The partial ordering of the goals are highly generalised and decoupled from the problem domain. To highlight this: We ourselves was surprised how easily we could add new types of goals very late in the development process without any delay.

Two clear improvements could easily be applied to highly increase the change of success.

- An improved multibody conflict resolution handling in the Merger would increase our chance of success in MA problems with limited resources considerably.
- A relaxation of our constrains in our search that allows the agent to move all boxes to accomplish a single goal. This could be implemented with a simple *WA** and a new Top Level Action easily in our existing framework.

Our Merger cannot handle simple MA problems as shown in figure 8. Our current solution does not detect that agent 0 in this case is the blocking agent since agent 1 never will request agent 0 to move. Thus, we need to detect dependencies between agents in conflict.

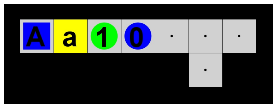

Figure 8: Illustrating MA conflict resolution problem.

Another less obvious improvement could be trying to minimize the time the agent is moving "empty-handed". We have seen this in several cases when the agent is moving long distances, where it could minimize the solution by picking up nearby boxes and bringing them closer to goal even though the agent is working on another goal. We are considering ways where we could use our existing goal ordering to implement second priority goals for cases like these. This could also be used to improve MA performance in cases where agents are *idle* due to unfulfilled preconditions. Often the agents can still perform work to decrease the cost of the goal until the preconditions are fulfilled. The benefit of this future improvement is hard to predict.

References

Alami, R.; Fleury, S.; Herrb, M.; Ingrand, F.; and Robert, F. 1998. Multi robot cooperation in the martha project. *IEEE Robotics and Automation Magzine*.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool, 1 edition.

Jean-Nol Demaret, Francois Van Lishout, P. G. 2008. Hierarchical planning and learning for atuomatic solving of sokoban problems. *University of Lige, Lige*.

Levels. 2016. *Sample levels handout*. 02285 AI & MAS.

Nobakht, B., and de Boer, F. S. 2014. Programming with actors in java 8. *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications* 8803(1):37–53.

Rusell, S., and Norvig, P. 2010. *Artificial Intelligence, A Modern Approach*. Pearson, 3 edition.

Virkkala, T. 2011. Solving sokoban. *Master's Thesis, University of Helsinki, Helsinki*.