TheRedDot
02285 AI & MAS

June 1, 2016

Elias K. Obeid s142952
Jens-Christian Finnerup s143120
Sune Bartels s155994
Mathias C. Lyngman s127719

Abstract

This paper presents a single-agent online centralised multi-body planner. We present and illustrate our goal prioritisation technique as our original contribution. Furthermore, the simplicity of our client has proven to yield sub-optimal solutions, but with a high performance w.r.t. time spent to find and solve the given levels, both for multi-agent and single-agent levels.

1 Introduction

This paper documents and presents our solution to the programming project. Our first attempt to solve the project was based on a partial-order planner for atomic plans. Our thoughts were that we could use the planner to re-plan if conflicts arose. However, we abandoned this solution as it would simply give us a backwards breadth-first search. As such we aimed for developing a better solution.

Our solution is a single-agent online centralised multi-body planner, where we attempt to prioritise the order in which goals are fulfilled. Paths were constructed in a hierarchical fashion using the weighted A* search algorithm in a relaxed domain. In this paper we showcase the resulting performance of our client and discuss its strengths, weaknesses, and potential ways for improvement. We chose to develop our client with the Python programming language.

1.1 Problem Statement

The aim of the project is to develop a client to solves levels by moving boxes to their respective goal positions. Challenges such as coordinating agent movement, resolving conflicting movement, communication between agents, moving obstacles from a path to reach a desired destination, etc. must be handled properly for a level to be solved. If the challenges are left unresolved the client may never be able to find a solution.

Similar systems with some form of artificial intelligence (AI) could be useful in many real life scenarios where repetitive work is needed. This could for instance be in a hospital where automated agents would be responsible for porting empty hospital beds to a destination where they are needed.

1.2 Background

This section briefly outlines the theories that act as the basis of our solution. It is our assumption that the reader is familiar with the course curriculum.

Throughout the course we were introduced to many techniques which we drew inspiration from to aid in developing our solution/client. The client is an online multibody planner, where only one agent moves at any given time. Plans are constructed iteratively in an hierarchical fashion using the best-first search algorithm A* with an admissible heuristic on a relaxed search space. (Russell and Norvig 2009; Geffner and Bonet 2013)

Furthermore, we will present an important technique used to increase the performance of our client, which we call the *goal prioritisation technique*. To the best of our knowledge, this technique is an original contribution. Without goal prioritisation an agent would attempt to move a box to its respective goal in an arbitrary order. This often leads to blocking future paths which were needed to achieve the other goals. To counter this we had to enable the client to order the different goals, according to a predetermined set of rules. To achieve the goal ordering ability we came up with our own technique of scoring/prioritising each goal according to its neighbouring cells. In section 2.1 we give a thorough description of our goal prioritisation technique.

1.3 Related Work

Our first plan was to make a partial-order planner for multiple agents on an atomic level. Knoblock examined how a partial-order planner would be able to construct parallel plans (Knoblock 1994).

Ephrati and Rosenschein had a similar approach to ours in 1994 where they suggested multi-agent planning with heuristics (Ephrati and Rosenschein 1994). They divided the goal in sub-goals and made sub-plans that were merged to a global plan. Their results showed that dividing into sub-goals and making the agents work in parallel can reduce the total elapsed time for the actual planning. Similarly, we used a high-level planner to divide goals into independent sub-goals and solve them independently.

Our goal prioritisation technique is an original contribution, however we have found papers that also use prioritised planning. For instance, (Van Den Berg and Overmars 2005; Bennewitz, Burgard, and Thrun 2001; Erdmann and Lozano-Perez 1987) present planning techniques where multiple objects must be moved, and the movement plans are constructed by prioritising either the paths or the objects to be moved. This differs from our goal priotization in that

they consider the object or the path, while we consider in which order to fulfill a goal. Furthermore, prioritised planning is not a new concept. Both in real life and in games plans must be prioritised, e.g. to stay alive in a game, the player or the game's AI agent must prioritise fleeing from danger or advancing on an enemy. (Orkin 2006)

2 Methods

This section describes our client's overall algorithmic functionality. The client is comprised of our goal prioritisation technique (section 2.1), the level representation (section 2.2), the path finding algorithm (section 2.3), and the overall planner (section 2.4). The client receives an incomplete plan from the planner, where the client makes sure to update the plan with the necessary `NoOp` operations for waiting agents.

Before we begin describing the client's overall functionality, we will give an intuition of how the planner works. Consider the scenario presented in fig. 1. The red agent **0** must move the red box **A** to the goal. Multiple boxes are in the way of the desired plan. The state of the level in the figure is that the cyan agent **1** has moved its box out of the way, and so has the yellow agent **2**.

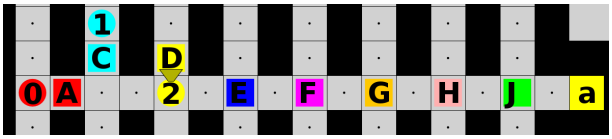

Figure 1: The red agent **0** must move box **A** to the goal **a**.

However, the yellow agent **2** is now itself in the way of the path. The planner thus informs that agent to move out of the desired path. The client will continue to detect obstacles in the desired plan and move them until no more obstacles are in the way. Then the red agent **0** will move the red box **A** to the goal.

2.1 Goal Prioritisation Technique

In order to explain how the functionality of our prioritisation works, we will break it down into three separate steps. For all the three steps we use an example set of goal cells, which are all only accessible form a single entrance.

Figure 2: Goal set with single entrance.

For the example in fig. 2 the optimal ordering would be to fulfil the goals in the following order: **a, c, b, e, f**, respectively. To accomplish this, our algorithm provides an ordering score to each goal.

Evaluating Each Goals Neighbouring Cells First, the algorithm evaluates each goal's eight neighbouring cells (including the diagonal neighbours) by giving them a score each. For every neighbouring goal-cell, the score is incremented by 1. For every free neighbour cell, the score is incremented by 2. The checking of neighbouring cells is illustrated with a grid in fig. 3. The middle cell is the one currently being checked, while the green represents a neighbour goal. A red cell represents an unreachable position (either a wall or out of bounds).

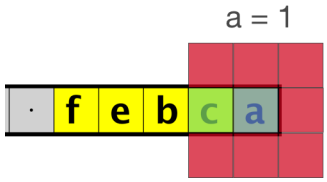

Figure 3: Neighbour checking of goal **a**.

The goal **a** is given a start score of 1. Goal **f** and **b** are given scores of respectively 3 and 2 as can be seen in fig. 4. A blue cell in the grid represents a free neighbour, while a green cell represents a neighbouring goal. The remaining goals not shown in the figures are all given a score of 2.

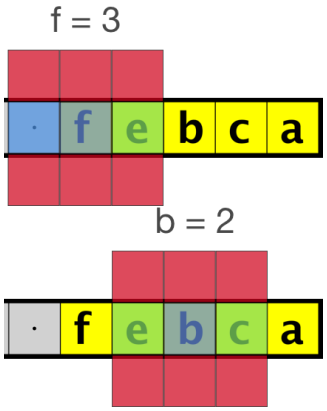

Figure 4: Neighbour checking of goals **f** and **b**.

Forcing an Order Secondly, after scoring the goals according to their neighbours, the algorithm creates an $n \times n$ size matrix, where n is the number of goals. Each column of the matrix will represent a goal cell and the first row holds the ordering scores that were computed in the first step.

Table 1: Example level - Ordering matrix
Incomplete (left) & Complete (right)

f	**e**	**b**	**c**	**a**
3	2	2	2	1
-	-	-	-	-
-	-	-	-	-
-	-	-	-	-
-	-	-	-	-

f	**e**	**b**	**c**	**a**
3	2	2	2	1
4	3	3	2	1
5	3	3	2	1
6	4	3	2	1
7	**4**	**3**	**2**	**1**

The initial state of the matrix is presented in table 1 (left). From this matrix we see the first ordering scores being filled

out while the rest remain to be computed. The remaining rows are inspected and for every element in each row, the corresponding goal's score is incremented if one of the following criteria are satisfied:

1. the score is equal the max value in the row
2. the score is less than the previous and next
3. the score is less than the previous and equal to then next
4. the score is equal to the previous and next

For each iteration for each row, the value of the goal being inspected is taken from the previous row. When we refer to "the previous score" it is the score of the goal to the left of the current goal being inspected (in the current row). Moreover, "the next score" refers to the score of the goal to the right of the goal being inspected (in the previous row). For instance, consider goal **e** at the first iteration (after the initial scoring) where the goal **f** already has been incremented because it was the max value in the previous row. The previous goal's score is the score of **f** which is 4 (the current row). The next goal's score is the score of **b** which is 2 (the previous row). The score of **e** is incremented because it satisfies the third criteria. The algorithm of course makes sure to only compare goals that are grouped together.

The reason for incrementing the max score, throughout all iterations, is to account for multiple groups of goals, for which we want to score the openings similarly. The idea is that we force an order on grouped goals so that we never have ambiguous priorities for neighbouring goals. Thus, the client should never be in doubt of which goal to choose. If two goals end up with the same score, they will either be a part of independently grouped goals, or equally accessible.

Sorting the Goals Lastly, after the matrix has been computed as presented in table 1 (right), the last row will contain the ordering of the goals. These can then be used to order the goals in ascending order (from lowest to highest). In our example case, the goals will be filled from **a** to **f**, respectively. The performance benefits and pitfalls of this prioritisation algorithm, will be discussed in detail in section 4.2.

2.2 Representing a Level

Cell of a Level We represent levels as grids of cells, where each cell can hold a wall, goal, agent, or a box. Otherwise it is a free cell. Each cell is uniquely referenced with coordinates, that define the x and y coordinates w.r.t. the top left corner. Thus, the first cell is referenced as $(0, 0)$ and is in the top left corner of the level.

The Grid A new grid object is instantiated with the level parsed from the server. The grid instantiation process takes care of assigning default colors to uncolored objects. The grid keeps track of the following; all object's positions (their cells) and colors, goals and their identifiers, the boxes and the agents. It knows which color boxes an agent can move. Furthermore, the grid representation must have the following functionality

- for a given cell, return its neighbouring cells
- for a given cell, is it possible to perform a swapping move
- for all goals, compute their priority (see section 2.1)
- for every server response (with every action we send), update the grid so it is up-to-date

The neighbours of a cell may be defined to filter results w.r.t. some criterion. Such a criterion could for instance be that we only want free cell (no agent or box at that cell). This method is used for finding possible movement from a given cell. Unlike in our goal prioritisation, where a cell has 8 neighbours, the diagonal cells are not considered neighbour cells here and thus a cell only has 4 neighbours.

We define a swapping move as swapping the position of an agent and a box it is moving. This could either be a pull-push combination or vice versa. The idea is to check if a given cell has enough neighbours for a swap to be performed, i.e. the cell must have at least three neighbours (because then we at least have a `T`-cell-combination, where a swap is possible).

The client performs online planning, i.e. an entire complete plan is not computed before performing actions. Because of this the client must keep track of all actions and if they succeed. The server's response must be used to keep the level representation up-to-date.

Movable Objects We define agents and boxes as movable objects. They have a position (a cell), some color, and an identifier that uniquely defines them in the level. Such an object can thus also be moved, which changes the position at which it is for that time step.

Updating the Representation For every action sent to the server, we make sure to update the client's perception of the current environment. Positions of agents and boxes are thus kept up-to-date while actions are performed.

2.3 Constructing Paths and Searching

Given a goal to achieve we split path finding into two parts; first find a box closest to the goal, second find the closest agent to move that box. These two paths are then combined to form a complete path for movement (we explain the path finding in more detail later in this section). The overall idea of our client is to iteratively remove obstacles from the combined path to reach the desired goal with the chosen box. We used the weighted A* search algorithm to find shortest paths (Patel 2016; Russell and Norvig 2009; Pohl 1969).

Distance to Goal We used the cross product as a heuristic to prioritise movement w.r.t. cells and their distance to the goal cell. This heuristic is admissible because the actual Manhattan distance will be longer than the Euclidean distance (Russell and Norvig 2009). We know that the A* algorithm considers both the heuristic value (distance to goal) and the distance already travelled (distance from start) as a measure of which nodes to expand in which order, i.e. as

$$f(x) = h(x) + g(x).$$

Many of the f values may thus be the same, but we would like a direct path to the goal. For this reason we add a tie-breaker to the heuristic value so the algorithm will prefer

nodes that get it closer to the goal, and thus gives a more direct path and less explored nodes. We used a tie-breaking value of 1.0001. (Patel 2016)

We search for paths in a relaxed domain, where boxes and agents are not considered as blocking objects, but the cells in which they are have a higher movement penalty. This is similar to the ignore preconditions heuristics, because we ignore that the cell must be free before movement is possible. The client then handles the fact that some other object must be moved before the initial plan can be carried out.

Movement Cost While searching for a path to a desired end position four scenarios may occur; (1) the object can move freely in the given direction, (2) the object must move an obstacle to move in the given direction, (3) the object must receive help to move an obstacle, (4) another agent must be moved to move in the given direction.

These scenarios are associated with different costs, and they are as follows; cost of receiving help is 20, cost of moving an obstacle (without help) is 4, cost of asking an agent to move is 3, otherwise object can be moved freely and the cost is 1. We wanted the movement of one agent to be as independent as possible from other agents. For that reason we associated higher costs for receiving help from other agents, and if an agent can move around an obstacle it will prefer that over moving the object itself.

Ignoring Heuristics In some scenarios an object must be moved from a desired path, because it is blocking it. For this we ignore the distance to the goal because we do not have a desired end position, we simply need to find the first free cell where the object can be moved to, which gets it out of the desired path.

Structuring Paths and Converting to Actions Path movement is initially a list of coordinates dictating the movement of an object. Thus, consecutive elements of the list comprise a single step in one direction. The first element in the list gives the position of the object to be moved. The grid can inform what object is at that position. The path of cells is converted to a list of actions, which the server can interpret.

The list of coordinates also defines when an agent is to move a box. This is defined as a list in the outer list, i.e. in this scenario an element in the outer list is not a cell but a nested list of cells, e.g. `[(0,0),[(0,1),(1,1)],(0,1)]`. The example will be interpreted to `[Push(E,S)]`, where the nested list defines the initial position of the box and the following movement. The element after the inner list defines the end position of the agent that was involved in the moving action.

2.4 Planning on the fly

Our client is an online planner, i.e. it creates sub-plans and continuously executes them in the aim of solving all plans. It only moves one agent at a time. Given a constructed plan (as described in section 2.3) the client is in charge of handling the agents and sending them instructions. A given plan may contain positions on which obstacles are in the way. If an obstacles is in the way, the client will create a new sub-plan, which must be achieved before the original plan can continue. This process may continue for many steps before all obstacles are moved for the original objects to be moved in the desired path. The client will iteratively create and solve the sub-plans for every blocking obstacle (one at a time). Note, that sub-plans of a sub-plan may also occur.

As mentioned in the previous section, each movement has an associated cost, and the agents are to be as independent as possible. However, independence is sometimes not possible and thus the agents must aid one another. The planner thus tries to create a sub-plan, where that plan's goal is to move the obstacle. This is done in an iterative manner, where the planner checks if the current plan/sub-plan is conflict/obstacle free and thus returns a list of desired actions to the client, which then sends actions to the server. The client monitors the execution of the agents in a similar fashion to action monitoring (Russell and Norvig 2009), but where we use the server's response to validate that our action was successful and from that update the representation of the level's state. The planner is continuously called with new goals to be achieved until all goals are occupied by a corresponding box.

3 Results

This sections presents some results and experiments for our client. First we present the results of running the competition levels, and then we present an example to illustrate why our goal prioritisation was useful. We performed our tests on a laptop with an Intel Core i5 4210U CPU with 2.7 GHz, 4GB DDR3L 1600 MHz SDRAM, with a Solid State Disk.

Table 2 presents the results of our client on the different levels developed by different groups. The time is defined in milliseconds, and actions are number of moves performed to solve the level. The table is split into two columns; one for single-agent levels, and one for multi-agent levels. There were four levels that were unsolvable, and they are thus ignored. Our client entered an endless loop for three of the levels (`SAAIMuffins, SASolo, SABoXBoXBoX`).

It is clear from the results presented in the table, that the client needs improvement because the client crashed for 11 levels (two of which were multi-agent levels). Our solution did not handle a specific scenario well. We were not able to locate where the specific problem was. Our client was the only client in the competition to solve the two levels `MAteamhal` and `MAAIMuffins`.

3.1 Different Goal Prioritisations

To showcase how efficient our goal prioritisation technique was and how we have improved upon it throughout the project, we present some test runs using the `SAOptimal` level. We have chosen this level, as it requires a good goal prioritisation in order to solve efficiently. In the following we will refer to our main goal prioritisation technique (section 2.1) as the "complex" prioritisation, and use the term "simple" to mean a prioritisation that does not require a strict ordering of grouped goals, as presented in section 2.1 (the second step). The intuition is that goals placed in dead-ends or corners must be achieved before any goals that would otherwise block the path to the inner most goals.

level creator	single-agent		multi-agent	
	time	actions	time	actions
Lazarus	91	243	62	291
Sojourner	144	417	116	255
Optimal	209	730	161	552
DangerBot	402	741	170	716
TheAgency	crash		55	39
botbot	crash		69	87
ButterBot	crash		72	175
TheRedDot	crash		123	489
extra	crash		402	1409
teamhal (!)	crash		334	1186
AIMuffins (!)	live-lock		3695	1417
WallE	unsolvable		157	506
FortyTwo	crash		unsolvable	
NoOp	crash		unsolvable	
TAIM	crash		unsolvable	
Solo	live-lock		crash	
BoXBoXBoX	live-lock		crash	

Table 2: The stats of our client's solutions for the different levels. The exclamation mark (!) defines levels that only our client solved. Time is in ms, and actions is the number of actions performed.

We illustrate three runs on the same level; one with no goal prioritisation whatsoever (fig. 5), one with simple goal prioritisation (fig. 6), and one with complex goal prioritisation (fig. 7).

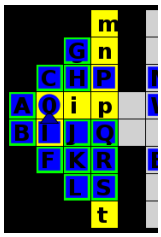

Figure 5: Arbitrary prioritisation.

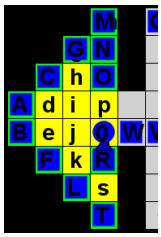

Figure 6: Simple prioritisation.

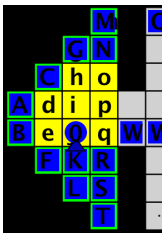

Figure 7: Complex prioritisation.

The main difference between the complex and simple prioritisation is that the complex forces a specific order on goals which are grouped, as mentioned in section 2.1. As fig. 7 illustrates the complex prioritisation results in the goals next to the level's bounds to be fulfilled first. The lack of strict order for the simple prioritisation (as illustrated in fig. 6) results in the agent choosing to place **R** before **S** and thus blocking the path of a goal (the priority of these two boxes is the same in the example, because they have the same number of walls, goals, and free neighbour cells). With arbitrary prioritisation (as presented in fig. 5) even more goals are blocked by the fulfilment of others.

Our final client utilized the complex prioritisation method, and with this we managed to solve this level (SAOptimal) with 31 fewer actions than the simple prioritisation as presented in table 3.

Prioritisation technique	Actions
None	> 1000
Simple	761
Complex	730

Table 3: No. actions performed on SAOptimal level with different prioritisation techniques.

We found that a client with no goal prioritisation requires almost 50 % more actions on this level than with some kind of prioritisation technique. This is due to the fact that many actions are wasted because the inner most goals are left open, and thus as the client wants to achieve them, all the previous boxes must be moved.

3.2 Performance vs. Optimality

We have developed a client with simplicity in mind. The only technique we used to yield more intuitive solutions was to develop the goal prioritisation technique, so the client did not fill in goals arbitrarily. From the course's competition presentation we found that, as an example, our client was beaten in SAOptimal with about 7 % less actions performed. However, our client solved the level approximately 700 times faster than the opponent.[1] We presented our client's statistics in table 2, and the level mentioned here was created by Optimal for single-agent movement.

3.3 Multi-agent and Single-agent Levels

Though the level SAOptimal showcases the importance of some goal prioritisation, it is important to note that out of the 16 levels we managed to solve, most of them were multi-agent levels. For the multi-agent levels, we only move one agent at a time and therefore lose a lot of performance w.r.t. the number of actions performed. On the time performance however, again here we perform very well as with the single-agent levels. As presented in table 2 our client was able to solve most of the multi-agent levels.

4 Discussion

This section presents some of the pros and cons of our client. For instance we will discuss the con of only moving one agent at a time and the difference between online and offline planning.

4.1 On Multi-agent Movement

Our solution only sends commands to one agent at a time. This increases the length of results in multi-agent levels but it also give us better performance. By performance we mean the time spent to find a solution (from starting the client to the server responds with success). By only moving one agent at a time we do not have to look for conflicts between the agents' paths but simply move obstacles. The simplicity of our client was advantageous while competing against

[1] **opponent**: 143,297 ms with 680 actions & **our**: 209 ms with 730 actions. (source: competition results)

the other group's more complicated solutions. It was clear that we could out-perform them in time spent on finding a solution.

Of course, if the client could move multiple agents simultaneously, this would have been a great improvement on number of actions performed. For example consider fig. 1 again, if all agents moved at the same time then all the blocking boxes would be moved out of the way simultaneously. But by moving all agents simultaneously we would risk getting conflicts if two or more agents' paths crossed. To solve these conflicts we could either do offline planning and make a complete plan before execution. This would hurt our solving times on multi-agent levels since we would have to cross check all paths. We could also try to solve the conflicts as they come in our online planning. This way we would solve conflicts as we go, but we risk live-locks if two agents get stuck in each others paths. We would be forced to handle these challenges, which would most likely decrease our client's performance.

A third option is to inspect if multiple paths cross, and simultaneously execute them if they are disjoint. Otherwise, the path without dependencies should move first, and the others that cross the original path must wait. This way it would most likely only hurt our performance a little, but the number of actions would not be improved a lot and only on some levels.

4.2 Goal Ordering

As shown in section 3, our goal prioritisation algorithm enables us to solve 16 out of the 30 given solvable levels in an intuitive manner. It is however important in this discussion segment to stress some of the flaws that come with our algorithm. In this section we will outline some of the strengths and weaknesses found in our goal prioritisation.

Strengths From the competition we learned that our client was successfully able to prioritise goals in levels such as `MAteamhal`. This level includes one opening which branches into two different one-way paths, as can be seen in the included fig. 8. This can be attributed to the fact that our ordering algorithm increments the opening of the goals (in this case the middle), for each iteration of the scoring matrix (see section 2.1). As such the opening in the middle will get the highest score, and therefore everything following will decrease. With this technique, we are able to solve both problems like `MAteamhal`, as well as levels with different groups of goals.

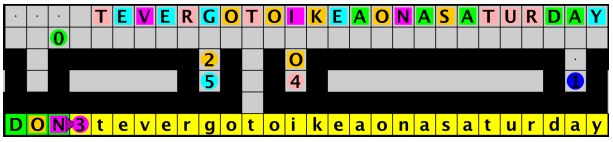

Figure 8: Our client solving `MAteamhal`.

Weaknesses Defining the opening (or entrance) to a group of goals as the goal with the most neighbouring free cells, does have it issues however. While working great in cases such as the one seen in fig. 8, problems arise if the entrance to a group of goals is not actually the goal with the heighest number of neighbouring free cells. An example of this can be found in fig. 9. Here we see that goal **c** has the highest number of neighboring free cells, and as such, our ordering algorithm will find it to be blocking the other goals, even though it is not. In the class competition, we did not encounter a problem like this, however the problem is indeed a known weakness of our prioritisation algorithm.

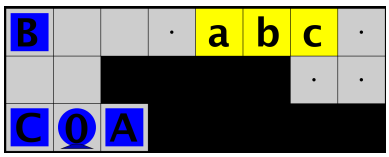

Figure 9: An example level for illustrating a weakness.

4.3 Online and Offline Planning

In this section the aim is to discuss the differences between online and offline planning, where we worked with online planning.

Both online and offline planning are applicable in the domain of the project. The domain is deterministic, static, and fully observable and we know that there are no unforeseen side-effects such as some event that can change the state of the level. The biggest challenge of offline planning is the combinatorial explosion/branching factor for planning for all possible contingencies for an entire plan. All possible conflicts must be found and resolved before any action is performed.

We can consider online planning as performing small steps in aim of solving a grand goal. We believe this is a more natural approach to solving the programming project. Unable to look into the future, the online planning approach will most likely not find an optimal plan. On the other hand the online approach gives us the advantage of performing actions early on.

Furthermore, if the domain for instance was not fully observable then with online planning the client can plan for unseen future effects of past actions. By monitoring the execution of actions the client can keep an updated perception of the world in which the agents are performing actions. By dividing the grand goal into smaller sub-goals and solving obstacles as they appear for the sub-goals, we do not experience the dramatic branching factor, compared to an offline approach.

We found that online planning was more intuitive and simpler to develop. We believe that our solution is simple because of the simple nature of online planning. We monitor the server's response and from that we construct and achieve the next goal in the line.

4.4 Centralised and Decentralised Planning

Our client incorporates a single-agent multibody planner, which plans to achieve goals in a prioritised order. Compared to a multi-agent planner, our client will never encounter conflicts with other agent's plans that need to be

resolved. On the other hand, if multiple agents could have performed actions simultaneously now they must wait for the other to finish, and thus the number of actions performed will be higher.

If we consider that the planner could handle multiple agents simultaneously, then the client must be able to handle conflicts with other agents. This would for instance be in the form of social laws, where agents consider each others identifier to decide which agent must make room for the other. Agent's could also communicate their intentions to each other, and thereby coordinate conflict free paths. Moreover, it could also be possible to perform some kind of agent prioritisation based on the agent's identifier or the path which the respective agent is currently moving along. As we presented in section 1.3, some of these ideas have been proposed in related work.

Furthermore, if multiple agents move simultaneously then it could happen that their movement would be caught in a live lock, i.e. they move in an endless loop trying to move past each other.

5 Conclusion

From our results, we can see that we have succesfully developed a single-agent online multibody planner. In terms of success rate, it was able to solve 16 of 30 solvable levels, two of which our client was the only one to solve. According to the course's competition results, our client was superior w.r.t. performance (the competition's time category) compared to the other group's clients.

To the best of our knowledge, our goal prioritisation technique is an original contribution to the project solution. It proved very effective at prioritising the goals in the correct order, making the client able to avoid blocking of goal cells and reduce the amount of actions taken.

The centralised planning we implemented made us able to reduce the complexity of the multi-agent levels, by removing the risk of conflicts caused by agents bumping into one another. This along with the online planning approach drastically reduced the time needed for our client to compute a solution, but increased the number of actions performed because only one agent moves at a time.

5.1 Future Work

It would be worth the effort to attempt to change the client to handle moving multiple agents simultaneously. This would lower the number of actions performed. If we consider having 10 agents working together instead of a single agent, then we could potentially have an order of magnitude less number of actions performed depending on the level. The client could be modified to not only achieve goals in a specified order, but also check if any goals can be achieved in the same order, and if multiple agents could move simultanouesly to achieve these goals. If it is not possible for more than one goal to be achieved simultaneously, then it might be possible for the waiting agents to move their respective boxes closer to the wanted goals. It would of course only be moved in such a manner that it would not block a path to the other goals. Moreover, moving multiple agents simultaneuosly would also require us to handle live-lock situations.

As a matter of fact we currently experience live lock situations where the agents try to solve goals in a specific order, but because boxes are in a dead-end they end up performing the same set of actions over and over without progress. This could be handled by finding all relevant boxes that are in a dead-end and moving them out to some safe zone from which they could be transported freely.

A way to establish *safe zones* for the different agents to place boxes, could be to use our goal prioritisation algorithm on free cells instead of goals. This could be achieved by mapping all the cells that are not in direct paths towards the goals, and score them according to the number of neighbouring free cells. With some adjustment to the scoring algorithm, this could prove to be a simple way of detecting safe zones (or buffer areas) for agents and boxes to move freely, without blocking the way of others.

References

Bennewitz, M.; Burgard, W.; and Thrun, S. 2001. Optimizing schedules for prioritized path planning of multi-robot systems. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 1, 271–276. IEEE.

Ephrati, E., and Rosenschein, J. S. 1994. Divide and conquer in multi-agent planning. 1(375):80.

Erdmann, M., and Lozano-Perez, T. 1987. On multiple moving objects. *Algorithmica* 2(1-4):477–521.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis lectures on artificial intelligence and machine learning. Morgan & Claypool.

Knoblock, C. A. 1994. *Generating parallel execution plans with a partial order planner*. University of Southern California, Information Sciences Institute.

Orkin, J. 2006. Three states and a plan: the ai of fear. In *Game Developers Conference*, volume 2006, 4.

Patel, A. 2016. Amits A* Pages. `http://theory.stanford.edu/~amitp/GameProgramming/`. Viewed in April, 2016.

Pohl, I. 1969. *First results on the effect of error in heuristic search*. Edinburgh University, Department of Machine Intelligence and Perception.

Russell, S., and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd edition.

Van Den Berg, J. P., and Overmars, M. H. 2005. Prioritized motion planning for multiple robots. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, 430–435. IEEE.