02285 Artificial Intelligence and Multi-Agent Systems
IIOO 05/06/2020

Thomas Heshe s164399 | **Christian Hjelmslund** s164412 | **Christoffer Krogh** s163959 | **Oliver Sande** s174032 | **Sebastian Specht** s164394

Abstract

Developing automated planners for the transportation industry leads to a more streamlined and cost-effective production. The IIOO client models a transportation environment through the Sokoban game and it manages to solve 44/60 levels in the Artificial Intelligence and Multi-Agent Systems 2020 competition (Thomas Bolander 2020). This result is achieved by an effective heuristic and a hybrid approach, where the client can switch between a centralized or decentralized instance based on the complexity of the given level.

1 Introduction

In this project, we have created an automated planner to solve levels in the Sokoban game. Sokoban can be used to model the domain of transportation within various industries, such as a hospital, where many transportation tasks are required and can be carried out by robots if the necessary automated planning has taken place. The focus of this paper is on the benefits and disadvantages of the centralized versus decentralized approach to tackle a planning problem. We have tried to use the best of both approaches by spending resources preprocessing the problem beforehand, and based on the characteristics of the problem, the client can be initiated as either a centralized or decentralized client. This hybrid approach makes our client more versatile and enables it to solve a broader range of problems.

2 Background

Modelling Sokoban as a search problem We create an automated planner by modelling the game as a search problem. Each state (node) contains the properties of each element of the game and their positions. The actions (as edges) that can be performed are moving, pulling and pushing described more detailed in (Thomas Bolander 2020). The initial state is defined with the initial positions of all elements, and the goal state as a state in which all goals are satisfied i.e each goal has a box or agent, with the same letter or number, placed on top of it.

Centralized planning We define a state graph consisting of states in the Sokoban domain, with directed edges representing joint actions of multiple agents. When using centralized planning, we perform best-first search on the state graph, starting at the initial state and ending in a goal state. The particular search is weighted A* with a domain-specific, non-admissible heuristic function. A* search along with admissibility is explained in detail in chapter 3 of (Russell and Norvig 2010).

Decentralized planning For decentralized planning we use concepts from hierarchical planning along with some of the same heuristic functions from centralized planning to solve sub-tasks. Hierarchical planning is explained in chapter 11.2 in (Russell and Norvig 2010). The concepts taken from hierarchical planning includes decomposition of goals into sub-goals and tasks, which we call *jobs*. We make the assumption of sub-goal independence. *Jobs* are calculated and distributed by a central planner. Each agent is then assigned different jobs by the central planner and has the responsibility of finding a plan that completes those jobs.

Offline and online planning Our centralized client uses offline planning. The search is done once and is only complete when the entire solution has been found. However, for our decentralized client, the search uses principles from online planning due to the need of replanning when encountering conflicts between the plans. Online planning in general is explained in 4.5 of (Russell and Norvig 2010).

Conflict management We discover conflicts through the use of execution monitoring. Specifically we use action monitoring and when conflicts are found we use a variation of *plan repair* to replan and thereby resolve the conflicts. This rather simple replanning technique along with the concepts of execution and action monitoring is explained in chapter 11.3 in (Russell and Norvig 2010).

Hungarian Algorithm Finally, as part of the heuristic function, we use the Hungarian algorithm to solve the assignment problem in polynomial time, by finding maximum weight-matchings in bipartite graphs. In other words, given two sets, where each element in one set has a cost connected to every element in the other set, we can find the pairwise matching between elements of the two sets which yields the lowest total cost. E.g. given two sets of boxes and goals respectively, where each box is connected to a goal with a cost (distance), we can match each goal with a box which yields the total minimum distance between boxes and goals. The algorithm runs in $O(mn^2)$, where m and n are the size of the two sets, and $m \geq n$(Cui et al. 2016)

3 Related work

In this section we will give an overview of some of the work that we have either drawn inspiration from or which describe alternative methods to solving the Sokoban problem.

The STRIPS automated planner introduced in (Fikes and Nilsson 1971) has made the foundation of the formal language STRIPS. In this project we have modelled the problem as a search problem, but an alternative could be to model it using the strict rules of PDDL. This could improve analysis of the program and could potentially also make the program's architecture more easily understandable.

In the early stages of the project the main focus was to build a centralized client optimized with effective heuristics. In (Zubaran and Ritt 2011) some basic heuristic functions are introduced which we have also implemented variations of. This include simple heuristic functions using e.g. the *Manhattan distance* from box to closest goal.

One of the most famous Sokoban solvers is the *Rolling Stones* developed by Andreas Junghanns and Jonathan Schaeffer. In their second article about Rolling Stones (Junghanns and Schaeffer 2001), they present some very effective methods to improve their solver. This include the concept of tunnels which effectively can reduce the search space by seeing a tunnel only as one move. The concept of tunnels is also present in our client, but only to avoid agents moving undesirably into a tunnel with a box. Another interesting concept that improved the Rolling Stones solver was *overestimation* for the heuristic function. In the article it is argued that forcing the heuristic function to always be admissible can be very ineffective. We have adopted the overestimation approach, hence our heuristic is not admissible, which means that we are not guaranteed to find the optimal solution, but on the other hand we are able to solve more levels.

Finally, (Demaret, Lishout, and Gribomont) explore the concept of hierarchical planning in the context of the Sokoban problem. In our project some of the concepts of hierarchical planning have also been implemented. Specifically, we have used decomposition to improve both the centralized and decentralized versions of our implementation and a lot of these concepts are also touched upon in the before-mentioned article. It would definitely be interesting to implement more concepts described in (Demaret, Lishout, and Gribomont) for future iterations. This is discussed in greater detail in section 6

4 Methods

Our client makes use of domain-specific strategies to improve results in the Sokoban domain. We have implemented a hybrid-client, which chooses between two strategies based on the complexity of the current level. We use decentralized, online planning in case of relatively large maps with many agents, and otherwise centralized, offline planning. Both approaches are domain specific and therefore both rely on many mutual preprocessings of levels. The preprocessing will be presented first, followed by a more in-depth explanation of the two planning strategies.

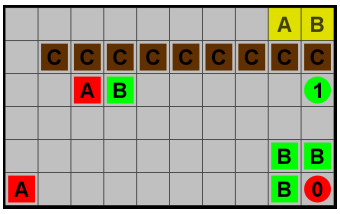

Figure 1: Example of multi-agent level

4.1 Preprocessing

Simplifying the level If a level contains boxes with no agents matching their color (like the C-boxes in Figure 1), we represent these boxes internally as if they were walls in the level. This simplification does not change the "true solvability" of the level, as the boxes are immovable if no agents match their color - thus these boxes has the same properties as walls. This improves the planning both in terms of time and memory consumption. Time, because when computing the shortest distance between two cells, the path through immovable boxes are no longer considered possible (more on this in the next section), which results in a faster and more accurate path planning. Moreover, it improves memory consumption, as only the movable boxes are represented in each expanded state - as opposed to walls being static for all states.

Precalculated distances Calculating the distance between two objects is an essential tool for our heuristic. A simple way to achieve this is by using the Manhattan distance but since this does not take walls into account it will sometimes yield misleading and too optimistic distances. This is exemplified in Figure 1 where the minimum actual distance from agent 1 to goal B is 20, but the Manhattan distance would be 2. To get a better indication of distance, we use a modified BFS that starts at the initial cell, and for each expansion of BFS it assigns the new points with the current depth of the search. The search only expands into free cells or the ones which contains movable objects i.e. the BFS will never consider cells with walls. This yields a matrix with an entry for each cell, where each entry contains the actual distance to the initial cell. Walls and other inaccessible cells get assigned a value equal to the products of the dimensions of the level. For each cell in the map, we run the BFS with the cell as initial cell, and add the distance matrix to a dictionary with the cell as key. This method of finding distances is still a relaxation since we do not take blocking boxes or agents into account but it is a good compromise between speed and precision. The downside is the heavy preprocessing required to generate the distances. Since computing each matrix has a time complexity of $O(nm)$ and since we in worst case compute one matrix for each cell in the map we have a time complexity equal to $O((nm)^2)$ for n being the vertical size of the level and m being the horizontal size.

Pairing boxes and goals We use the Hungarian algorithm to assign each goal to a box, where the assignments yields the minimal total distance between boxes and goals. This is done using the implementation in (almikael 2018), where we construct a $n_b \times n_b$ matrix m, where n_b is the number

of boxes. Rows and columns represents each box and goal respectively, and an entry $m_{i,j}$ is the distance between the box represented at row i and goal represented at column j. Using the level in Figure 1 as an example, we would create a 6×6 matrix. The distance between boxes and goals of different letters are set to ∞, so that any other pairing is more preferable. Moreover, as we only have two goals, the last four columns in the matrix represents "fictive" goals, where the distance from these to any box is set to 0. By that, the total minimum distance will remain unchanged regardless of which box is assigned to these. Therefore the algorithm will match the two real goals with the two boxes that yield the minimum total distance and randomly "assign" the rest of the boxes to the fictive goals. In our example, the algorithm would pair the uppermost A and B box with the two goals, and the rest to fictive goals. We then assign the two boxes to their goals and ignore the rest.

Blocking detection Having boxes that are inaccessible to their assigned agents can be a major problem e.g. in Figure 1 agent 0 is blocked by B-boxes. We solve this issue by initially performing a modified BFS that finds the paths between agents and their boxes. The search tries to avoid finding paths that go through boxes, but allows it if it is inevitable. We then prioritize moving boxes away from these paths, resulting in agents clearing paths for each other.

Goal Dependencies Satisfying one goal sometimes lead to blocking another. To counter this we find the order of which goals must be satisfied. We do this by once again performing a modified BFS that finds the paths between boxes and goals. We prioritize finding paths that do not contain goals, but if this is inevitable, we allow it. By doing so, we can compute which goals must be crossed to access another goal. We use the notion of dependency for when a goal is depending on other goals to be satisfied before itself. In the case of Figure 1, A depends on B.

Placing Boxes in Corridors If an agent has to place a box at the very end of a corridor, it might pull the box through the corridor which can lead to the agent blocking the goal. To avoid this, we mark all cells that have less than 3 adjacent walls. If an agent and a box steps onto a marked cell, we encourage the distance from box to goal to be less than the distance from agent to goal. This ensures that the agent will always push the box into the corridor, avoiding trapping itself.

Avoiding cluttering boxes We generally want to avoid clutter and having space around boxes helps the agents navigate the map and ultimately solves the level faster. We therefore try to avoid having boxes adjacent to other boxes that are ready to be moved onto their goal.

4.2 Centralized Planning

Our first strategy is planning based on an offline centralized graph search. Here the primary focus has been to construct a good heuristic for a weighted A* search within the Sokoban domain.

When expanding a state s in centralized planning for n agents, n sets are initially computed, where each set $actions_i$ contains applicable actions of agent i in the given state s. A Cartesian product between the n sets are then computed resulting in N lists, where each list contains a unique combination of n actions (one for each agent) - i.e. we now have a set of N unique joint actions. Hence, s is expanded into N new states to consider. Note that N is exponential in the number of agents: $N = \prod_{i=1}^{n} |actions_i|$, where $|actions_i|$ is the size of the set $actions_i$, i.e. the number of applicable actions for agent i. Thus, for a state s' with $n = 10$, where for every i, $|actions_i| = 5$, expanding s' yields $|actions_i|^n = 9,765,625$ states, i.e. the state space grows exponentially with the number of agents. Due to the large state space, our focus have been on designing a heuristic that finds solutions before running out of memory. This, combined with using weighted A*, motivates searching in depth rather than breadth. This is well-suited for the Sokoban domain in many cases, as we assume that there exists many solutions to levels slightly deeper in the search graph. Consider for example how many ways the level in Figure 1 could be solved. Therefore, our heuristic is not designed towards being admissible and consistent, as this guarantees optimal solutions (ensure by somewhat searching in breadth) rather than just focusing on finding a solutions (by exploring in depth).

Our heuristic is inspired by hierarchical decomposition in HTNs. Solving a level is reduced to placing proper boxes and agents on every goal. We have further decomposed this into minimizing distances from agent to the proper boxes, minimizing distances from boxes to appropriate goals and then finally minimize distances from agents to their goals. This is incorporated into the following heuristic function:

$$h = h_1 \cdot w_1 + h_2 \cdot w_2 + h_3 + h_4 + h_5 + h_6 + h_7$$

$$d_{1,i} = \min(\{D(a_i, box) \mid c_1\} \cup \{\infty\}), \quad d_{2,i} = \min(\{D(a_i, box) \mid c_2\} \cup \{\infty\})$$

$$h_1 = \sum_{i=1}^{n} \begin{cases} d_{1,i} & d_{1,i} < \infty \\ d_{2,i} & d_{2,i} < \infty \wedge d_{1,i} \not< \infty \\ 0 & \text{otherwise} \end{cases}$$

$$h_2 = \sum_{j=1}^{m} \begin{cases} D(b_j, AssignedGoal(b_j)) & c_3 \\ -1 \cdot reward & c_4 \\ 0 & \text{otherwise} \end{cases}$$

$$h_3 = \sum_{i=1}^{n} \begin{cases} k_1 & AgentStandsStill(a_i) \\ 0 & \text{otherwise} \end{cases}$$

$$h_4 = \sum_{i=1}^{m} \begin{cases} D(a_i, Goal(agent_a)) & c_5 \\ 0 & \text{otherwise} \end{cases}$$

$$h_5 = \sum_{i=1}^{n} \sum_{j=1}^{m} \begin{cases} D(a_i, b_j) \cdot w_3 - D(b_j, BlockedAgent(b_j)) & c_6 \\ 0 & \text{otherwise} \end{cases}$$

$$h_6 = \sum_{i=1}^{n} \begin{cases} punishment & c_7 \\ 0 & \text{otherwise} \end{cases}, \qquad h_7 = \sum_{b_1=1}^{m} \sum_{b_2=1}^{m} \begin{cases} k_4 & c_8 \\ 0 & \text{otherwise} \end{cases}$$

In which w_i are weights, k_i are constants, n = number of agents, m = number of boxes, a_i is the i'th agent and b_j is the j'th box. $D(elem_1, elem_2)$ is the true distance between two elements, as explained in the earlier chapter. We use the notion of a "ready box" to indicate boxes that have an unsatisfied assigned goal whose dependencies are satisfied i.e. a box that is ready to be moved onto its goal. We also use the notion of "an agent's boxes" meaning the boxes that have the same color as the agent.

h_1 encourages minimizing the distance from each agent to its nearest box, by adding the value to the heuristic value. The agents prioritize minimizing the distance to ready boxes but if none are ready they minimize the distance to any box

(a) The initial state. (b) `deadlock-conflict`. (c) `freeCell-conflict`. (d) `blocker-conflict`.

Figure 2: The different conflict types. The agent goal 0 has been assigned to agent 0, and agent goal 1 is assigned to agent 1. The arrows indicate the plan of each agent in Figure 2a, and in the remaining figures the arrows indicate the chosen direction of the move action of each agent. Above each level there is a timeline that shows how far each agent is in their respective plan. Each plan consists of five actions. The unfilled circle indicates that the agent has not performed any actions of their plan, and the filled circle indicates that the agent has executed the entire plan.

of their color. This encourages the agents to focus moving boxes that are ready to be placed on their goal. Even if no boxes are ready it is usually still an advantage to have agents close to their boxes so they can move them out of the way of other agents. $d_{1,i}$ is the minimum distance from agent a_i to a ready box of matching color and $d_{2,i}$ is the minimum distance agent a_i to a box of matching color that is not located on its goal. c_1 is the following conditions: the box must be ready and the same color as the agent. c_2 is the condition that the box must be the same color as the agent.

h_2 encourages minimizing the distance from ready boxes to their assigned goal. If the box is already located on its assigned goal, we reward the agent with the constant $reward$. This encourages the agents to move ready boxes onto their goals. c_3 is that b_j is ready. c_4 is that b_j has an assigned goal, that the dependencies of the assigned goal is satisfied and the goal is satisfied. We have chosen to add weights to the first two heuristics since these are the most essential.

h_3 punishes agents for standing still. The method $AgentStandsStill(a_i)$ is true if the agent is standing on the same spot as in the parent state. This makes agent move around boxes in situations where agents otherwise would wait for an adjacent box to be moved, as going through the occupied cell would be the shortest path to a goal or box.

h_4 encourages minimizing the distance between each agent and its goal but only if all boxes of the same color as the agent are satisfied. This encourages the agents to go to their respective goals when there is no more tasks to do. c_5 is the conditions: all boxes of the same color as the agent, that has an assigned goal, are satisfied, and there exists an unsatisfied agent goal with the same number as the agent.

h_5 encourages minimizing the distance from agents to their boxes that are blocking other agents. It also maximises the distance from the blocking box to the blocked agent. By adding weights to that distance it becomes highly prioritized. This maximizes the distance from the blocked agents to the blocking boxes, which essentially unblock agents. $BlockedAgent(b_j)$ refers to the agent that is blocked by box b_j. c_6 is the condition that box b_j is the same color as agent a_i and that it is blocking another agent.

h_6 punishes if an agents is pulling a box through a corridor. This will encourage the agents to push instead, eliminating the possibility of trapping itself. c_7 is the condition that an agent a_i is adjacent to a box of the same color, one of them is located in a corridor and that the distance from agent to the box's assigned goal is less than the distance from the box to its assigned goal.

h_7 punishes if there are boxes adjacent to a ready box. This rewards having space around ready boxes and helps unblock the path to its goal. c_8 is the condition that box b_{b_1} is adjacent to box b_{b_2} which is a ready box.

4.3 Decentralized Planning

In this section, 'agent' is used interchangeably to describe the agent objects in the levels and the intelligent planning agents.

The decentralized client consists of multiple agents. At the highest level we have a centralized planning agent, the *distributing agent*. This agent identifies all goals in the level i.e. required positions for boxes and agents. Afterwards, the *distributing agent* distributes all goals in the form of jobs to the decentralized planning agents, which we call the *searching agents*. There is exactly one *searching agent* for each agent object in the level. The *searching agents* are responsible for computing plans that achieve their individually assigned jobs using only the actions available to their corresponding agent. These plans are computed using weighted A* search with a heuristic function similar to the heuristic function introduced earlier. However, using completely offline planning is not sufficient in this dynamic setting, as the plans of multiple *searching agents* might be conflicting. We use execution monitoring from online planning to figure out when replanning is needed. More specifically, we ask all *searching agents* to submit one action from their plan, and then we use action monitoring. If an agent cannot find a plan, they simply submit a `NoOp` action and wait to be able to find a plan in the future. If all preconditions of the submitted actions are still met we execute all actions at once. Otherwise, we must have encountered a conflict. We consider three types of conflicts: `deadlock-`, `freeCell-` and `blocker-conflicts`. All these conflicts are illustrated in Figure 2. The `deadlock-conflict` occurs when two agents try to move through each other - with or without accompanying boxes. An example hereof can be seen in Figure 2b. The `freeCell-conflict` occurs when multiple agents submit actions that reserve the same cell of the level which is initially free. An example can be seen in Figure 2c. Lastly, the `blocker-conflict` occurs when the precondition of one action is violated, but the conflict is not categorized as a `deadlock-` or `freeCell-conflict`. An example can be seen in Figure 2d.

When we encounter conflicts, we need to replan to resolve the conflicts. However, computing plans can be computationally expensive, so we use a variation of *plan repair* in hopes of reusing as much of the original plans as possible. The exact approach to resolving conflicts depend on

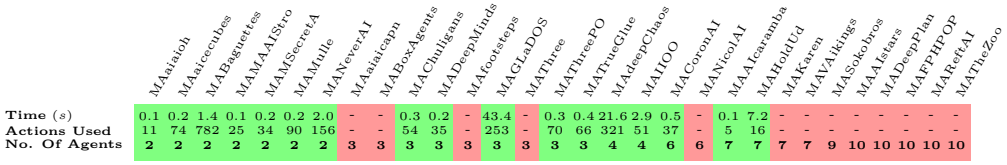

Figure 3: Results for MA levels - Performed on an Intel I7-8705G CPU @ 3.10GHz and 8GB RAM

which type of conflicts need resolving. When we encounter a `deadlock-conflict`, we tell one of the conflicting agents that they need to go back one step in their plan. Whereas, the other conflicting agent is told to stall the execution of their plan by one action. We decide which agent needs to go back in their plan according to the social law: *agents with the lowest unique identifying number always have the highest priority*. An example of this approach to conflict resolution can be seen in the transition from Figure 2b to Figure 2c. Looking at the timelines at top of the figures, we see that agent 1 has gone back one step in its plan, while agent 0 has postponed the execution of its plan by one action. This leads us to the `freeCell-conflict`. When multiple agents try to move into the same free cell, we only allow one agent to execute its action. This agent is chosen according to the same social law as before. All other agents postpone the execution of their plan by one action. Finally, we have the `blocker-conflict`, where something occupies the cell, x, which must be free for some agent to execute its action. There is no guarantee that whatever occupies x can be removed by making another agent go back one step in its plan. Instead, we create a *freeCell*-job. The *distributing agent* assigns this job to the *searching agent* that is closest to x and capable of moving the object that occupies x. The *freeCell*-job is completed when x is free. The conflicting agent waits until x is free. Agents will often solve their newly assigned *freeCell*-jobs by simply continuing with their original plan. Therefore, agents will always continue with their current plan until completion. Only then will they recompute the plan such that it takes new *freeCell*-jobs into account.

We continue requesting actions, finding and resolving conflicts until we arrive at a state where all goals in the level are achieved.

5 Results and Discussion

In this section our client's results in the 2020 competition from the DTU course 02285 is presented, analyzed and discussed. Implementation and all levels can be found in (IIOO 2020)

5.1 Competition Results

Our client solved **44/60 levels**, where 27/30 and 17/30 solved levels where single agent (SA) and multi agent (MA) levels respectively. Our client finished at second place out of 26 clients in total.

We believe the primary reason for our client's good performance is due to our initial focus on improving the heuristic for a centralized system, which improved our results on both SA and MA levels. We managed to construct a heuristic effective at guiding a search to a goal state through concepts which applies to both SA and MA levels - e.g. placing boxes on goals in correct order, minimizing distances between agent and appropriate boxes, and between boxes and goals. Moreover our client improved on MA levels from our hybrid approach, by switching to decentralized planning for levels of high complexity.

Our initial focus on developing a centralized client is clearly reflected in the results on MA levels shown in Figure 3. We solve the majority of levels with few agents, including somewhat complicated ones like our own MAIIOO (level can be seen in appendix, Figure 5), which requires handling several boxes and placing them in a correct order. However, for levels containing more than 7 agents in larger maps, our client struggles finding solutions due to exceeding maximum memory, because of the exponential growth with the number of agents. Moreover, our client fails to solve levels which require more complicated unblocking mechanisms, such as in *MAaiaicapn*, where agents have to remove boxes from a goal in a corridor, before placing them again in a specific order.

Observing these two challenges - sophisticated unblocking and rapidly expanding state spaces - motivated our hybrid client approach, as addressing these problems seemed more feasible with online decentralized planning. Solving the rapid expanding state space with centralized planning would be to prune the search graph in a smart way, e.g. only consider a single action for an agent in levels where it cannot move any boxes and does not block anything. However, this would typically only fix edge cases and not the general problem of having 10 agents all required to search for a plan. Moreover for the anti-blocking problem - when attempting to implement an advanced anti-blocking mechanism into the heuristic of the centralized client, we experienced that this was very time-consuming in terms of repeatedly searching the map for potential blocks of agents and/or boxes - therefore, we stayed with the anti-blocking mechanism described in section 4.1, where we try to predict potential blocks from the configuration of the initial state. With a online decentralized planner, both of these problems seemed better addressed. Here, the space complexity of planning is only linear with the number of agents. Moreover, the decentralized structure encourages decomposing the problems into even more simple tasks - e.g. when an agent is executing a plan and suddenly observes that another box is blocking its planned path, it may communicate a new job for another agent to free that cell. However, the decentralized client struggles when agents are unable to compute plans that completes their jobs. A natural extension to the decentralized client would be enabling agents of determining what is preventing them from computing plans. They could then use this information to create *freeCell*-jobs with the aim of unblocking them.

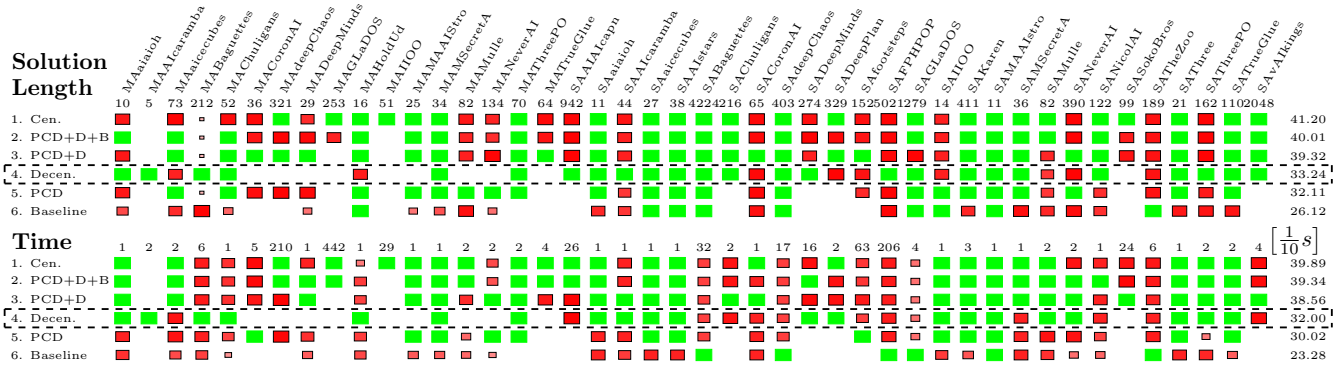

Figure 4: The rows represent five centralized and one decentralised (Decen.) clients' performance on the solved levels from the competition. Active features are specified on the left, with abbreviations PCD = Pre-calculated distances, D = dependencies (orderings) and B = unblocking. Cen. is our final version of the centralized client which also rewards agents for not standing still, and pushing rather than pulling boxes into corridors. No boxes means levels are not solved, green boxes are best found solutions and size of red boxes is relative performance to the best (green) found solutions. The total score is shown to the right.

5.2 Client versions and features

An overview of how our client performs with major features toggled on and off is presented in Figure 4. All versions include minor unspecified optimizations such as changing immovable boxes to walls, avoiding boxes cluttering, etc.

We see that the effect of activating each of the four features in the centralized client solves additional levels, but with reduced effect every time for both time and solution lengths - the total scores (on the right) is only minorly improved for the two top-most rows. This suggests that the optimisation from adding additional complexity to the heuristic somewhat converge - i.e. it might become over-complicated by adding too many features, which in some cases are conflicting. E.g. rewarding a state when an agent moves a box towards its goal, but at the same time punish the state because that box blocks another agent.

The two most effective features in terms of solving additional levels and improving total score are DE (placing boxes in correct order) and PCD (using the precalculated "true" distances between all cells). Only using the Manhattan distance (the bottom row "Baseline") finds equally good solutions - and faster - in a few cases compared to using PCD (bottom row has a large or green box and the row above has a smaller or red box). This is because the Manhattan distance requires no preprocessing, and is therefore more efficient than using "true" distances in some cases, e.g. with levels like *SAGLaDOS* which has no static obstacles.

This compromise between adding features for solving more levels but occasionally finding solutions a little slower happens in general, as features are sometimes unnecessary - e.g. for the "B" feature, when searching for corridors even though there are no corridors present in the level as with *MANeverAI*. In other cases like in the level *MABaguettes*, the baseline finds shorter solutions due to randomness when choosing between equally rated states by the heuristic. For example when there seems to be several shortest paths from an agent to a box, but one path is temporarily blocked by another movable box placed on its goal.

Finally it is worth noticing that the decentralized client (surrounded with dotted lines in the Figure) is the only client who solves MAAIcaramba, which truly reveals the strength of the decentralized approach compared to centralized. The level is very simple but contains many agents, and therefore the centralized client runs out of memory before finding the relatively simple solution requiring only five actions.

6 Conclusion and Future Work

In this project, we have made an automated planner to solve levels of the Sokoban game. We have achieved good results with a hybrid approach switching between a centralised and a decentralised client depending on the complexity of the level. For the competition results we solve 44 of 60 levels which resulted in a second place in the overall competition. For future work our first priority would be to develop the decentralised client further. This could include more ideas, concepts and theories from hierarchical planning such as more refined decomposition, goal scheduling and more intelligent replanning strategies. Some of these techniques has shown great success in similar projects like (Demaret, Lishout, and Gribomont). Specifically we would like to implement a clear way of prioritizing goals, which could be used to make the social law for conflict resolution more advanced. We expect that the decentralized client would be substantially improved if agents that are unable to compute a plan that achieves their goals could identify what is preventing their search from succeeding. The agents could then create goals to remove the obstacle(s) - thus solving the issue of advanced unblocking. This leads us to another enhancement to the decentralized client: instead of one agent being responsible for the distribution of goals, the distribution of goals could be done according to the principle of bidding. Closely related, it would be interesting to see how the decentralized client performs if agents only considers one job at a time. Another feature that would improve our client in a multi-agent setting is extending our client with a version of the BDI-model (belief-desire-intention). This would provide an abstraction level to our client which would make further development easier and the client itself faster.

The idea of using a hybrid approach can be generalized to other domains, but our preprocessing and heuristic function are domain specific, so we would have to adapt those to be able to generalize to other domains.

References

[almikael 2018] almikael. 2018. HungarianAlgorithm. https://github.com/aalmi/HungarianAlgorithm.

[Cui et al. 2016] Cui, H.; Zhang, J.; Cui, C.; and Chen, Q. 2016. Solving large-scale assignment problems by kuhn-munkres algorithm.

[Demaret, Lishout, and Gribomont] Demaret, J.-N.; Lishout, F. V.; and Gribomont, P. Hierarchical Planning and Learning for Automatic Solving of Sokoban Problems. Technical report.

[Fikes and Nilsson 1971] Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3):189 – 208.

[IIOO 2020] IIOO. 2020. our repository. https://github.com/christianhjelmslund/ai_programming_project.

[Junghanns and Schaeffer 2001] Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence* 129(1-2):219–251.

[Russell and Norvig 2010] Russell, S., and Norvig, P. 2010. *Artificial Intelligence A Modern Approach Third Edition*.

[Thomas Bolander 2020] Thomas Bolander, Andreas Garnæs, M. H. J. M. B. A. 2020. Programming Project. https://cn.inside.dtu.dk/cnnet/filesharing/download/b77640e3-8421-40e4-8685-eac2f589b745.

[Zubaran and Ritt 2011] Zubaran, T., and Ritt, M. 2011. Agent motion planning with pull and push moves. *Eniac*.

7 Appendix

Figure 5 and Figure 6 show the levels that our group submitted to the competition.

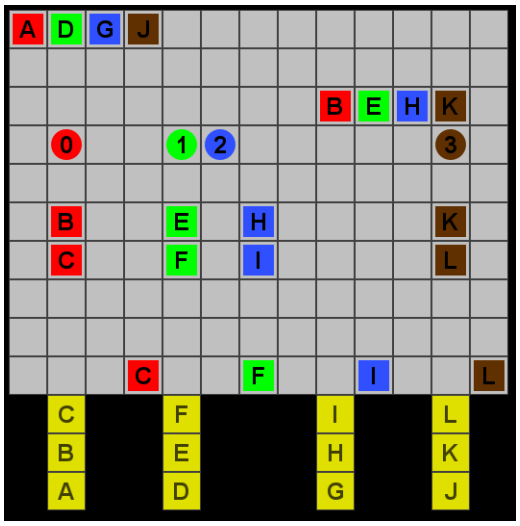

Figure 5: This MA level tests that a client can coordinate several agents at the same time. Moreover it tests that agents are smart enough to place boxes in a correct order, even though they might be tempted to first place the nearby boxes on their goals. Furthermore agents has to navigate around each other when collecting the boxes in the top left corner, possibly resulting in different types of conflicts. Finally, the level tests that agents pick the boxes which are the most near their goals (e.g. the C-box in the bottom-row rather than the one in the middle row).

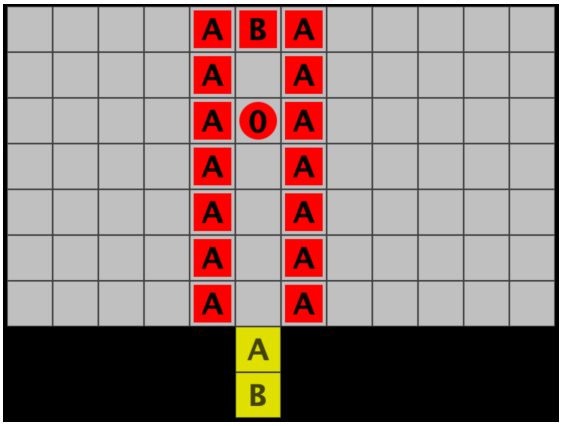

Figure 6: The purpose of the SA level is to challenge the client's skill in placing boxes in a correct order. Box B has to be placed before A, but the agent is surrounded by so many A boxes that it would be tempted to pick one of those first and thereby block the B goal.