AI and MAS (02285)
Programming Project Report (05/06/2020)
AIStars

Magnus Peter Eilersen s144210
Carlos Parra Marcelo s192770
Jacob Chen Jensen s145419
Miklós Kristóf Jásdi s192748

Abstract

In this report we propose a planning approach that aims to provide optimized solutions for the multiagent Sokoban problem. We implemented a centralized offline planning solver, that utilizes a Best-First Search strategy (with forwarding search) using domain-specific heuristics to find a solution with a lower number of steps. It uses a graph to represent the levels and computes the distances between all positions efficiently. Then it generates the search tree with permutations of possible actions for each of the agents, computes a heuristic value for each of the states and orders them into the frontier in ascending order. It performs well for small and normal agent-complexity levels but encounters many difficulties with a high number of agents and possible actions. This is due to not diverting time towards creating a decentralized system, however we have a good base structure for developing towards it, if more time was available.

Introduction

Automated Planning is a widely used method for planning problems. It allows for solving complex problems by specifying states and actions of a particular problem, and finally searching through the states to find a suitable solution. There are many variations and implementations of automated planning that can be chosen depending on the specific requirements for the solution.

In this paper, we are applying automated planning to solve a multi-agent Sokoban problem. We have explored the possibilities and challenges of designing and creating efficient solutions for it, especially by managing as many agents as possible at the same time. In conclusion, we discovered that, with our methods, such a solution is very complex and scales poorly with more than 3-4 agents. The client performs well in non-complex levels in terms of cooperation and communication between the agents, but it shows some problems when facing chains of goals/agents, removing several boxes to clean paths to goals, or trying to get close to the goal state only with the help of the designed heuristics. In addition to that, we have explored the possible improvements that could be added to our solution to make it more scalable, efficient, and successful with many different types of levels.

Background

After taking into account the available options for our solution, we decided to develop a centralized domain-specific planner that utilizes a Best-First Search (informed-search) strategy using domain-specific heuristics, performing forward search, and planning the solution. Furthermore, we have also used problem relaxation to simplify the problems and reduce state complexity when designing our heuristic functions.

Our solution aims to always keep the agents busy, trying to minimize the number of moves required to complete their tasks. This means we chose to optimize the final solution provided by our client, resulting in a lower number of steps for the provided solution but also experiencing more difficulties reaching the final solution.

The distances between the different positions within the level are pre-computed using a graph that is built at the beginning of the whole process with the help of the NetworkX Python library.

The planner employs a classic allocation mechanism based on a minimum-cost matching algorithm known as the Hungarian algorithm. Our implementation assigns first boxes to goals and then agents to boxes to find the optimal combination for these assignments in a way that the total action distance (the required actions to push the boxes into their goals) is minimized. With this implementation, each of the agents can focus on one box and its corresponding goal at a time, significantly reducing the breadth of the state space search tree.

	Box #1	**Box #2**	**Box #3**
Goal #1	10	3	5
Goal #2	12	4	6
Goal #3	1	7	3

Table 1: An example of cost matrix based on goal-box distances for a certain level

The Hungarian algorithm (Kuhn 2015) takes a so-called cost matrix as an input, as shown in Table 1. Each value of the cost matrix represents an assignment cost of its corresponding row and column (the distance between the pairs box-goal or agent-box). The algorithm matches rows and columns (using the graph) in a way that the total assignment cost is minimized.

Related Work

In our Sokoban implementation we use a graph representation of the level. This is a very common technique to represent maps (Patel 2020) because it can make a huge difference in the performance and path quality. Instead of exploring the corners of an obstacle (walls, agents and boxes in our case), a graph allows us to explore paths without taking care of these obstacles. In our implementation, when the graph is built, nodes represent available positions in the grid, and edges represent possible moves. It is only necessary to identify each node with its position on the original grid. There are a lot of types of representations using graphs for maps and it is very easy to find classic games using this famous technique, such as Super Mario (where some actions can be jumps or similar with different behaviors) or Pac-Man. This approach allows us to pre-compute the path lengths between nodes (only taking into account static obstacles like walls, but not agents and boxes), whose computational cost is high, letting us avoid doing it for every state.

We have used the Python library NetworkX (NetworkX 2020) instead of other alternatives (like iGraph) because when building graphs for approximately 50x50 grids (though they could be bigger outside of the competition levels) NetworkX is enough to have a good performance (NetworkX performs very good for graphs with thousands of nodes). It is interesting to mention that we decided to use this library after having faced a lot of problems for big levels with our own non-optimized graph implementation.

The other implementation that we have included in our work is the Hungarian algorithm. This algorithm is a costly $O(n^3)$ operation (where n represents the number of boxes) (Kuhn 2015) and, as a result, some papers on Sokoban solvers disregard it in favor of faster, but less precise approximation algorithms (Froleyks 2016). The papers that apply this minimum-cost matching algorithm either do not go into the implementation details (Junghanns 2009), or only apply it to the single-agent Sokoban levels (Virkkala 2011).

In contrast to these solvers, we have come up with an approach that generalizes the use of Hungarian algorithm, so it can work in both single and multi-agent levels.

Methods

Before we start exploring the states, our client runs some pre-processing to minimize some of the computational cost for the rest of the process. Our client directs the agents by using two mechanisms - the heuristic function and collision avoidance during state generation.

Level Pre-processing

While reading the level information from the server and generating the start and goal states to solve the problem, our client performs some sorted tasks once to load in memory some useful information for the incoming search:

1. **Generate the level graph using the original grid map** - We generate the corresponding graph (previously mentioned in the *Related Work* section) in order to compute the shortest path lengths for the distances that we will use in next processes.
2. **Convert unmovable boxes to goals** - This algorithm checks if there are some boxes of a particular color without at least one agent of the same color. If this is true, it converts these boxes to walls and removes their nodes and edges from the graph.
3. **Compute the distances between nodes** - Finally, using the Python library NetworkX we compute the distances between nodes and we save all of them in memory. This is computationally expensive for big maps, but the implementation is optimized compared to the first approach we implemented by ourselves. After this we save a $n*(n-1)$ sized data structure (concretely a dictionary), with n being the number of nodes in the graph. At this point, we relax the problem by assuming that there would be no obstacles in the way, i.e. other agents or boxes (we only take into account the walls and unmovable boxes when generating the graph).

Hungarian Algorithm

The heuristic calculations are based on a number of tasks (or agent-box-goal allocations) that are generated by the Hungarian algorithm. The algorithm runs using two iterations: the first one to assign boxes to goals, the second one assigns agents that will move those previously allocated boxes.

Allocations take place at the program's initialization and in each state (whenever a box is pushed into a goal we repeat this process). An allocation step consists of the following procedures:

1. Let G_U denote the set of non-fulfilled goals.
2. Let B_U denote the set of boxes that have not been pushed into their goals.
3. Let A denote the set of all agents on the map.
4. For each goal/box letter L:
   - 4.1. Calculate distances between all possible goal-box pairs of G_{UL} and B_{UL} - resulting in a distance (cost) matrix.
   - 4.2. Run the Hungarian algorithm with the above cost matrix as an input and match a box to each of the unfulfilled goals. Disregard any extra boxes.
5. Let B_A denote the set of boxes that have been assigned to a goal in the previous step.
6. For each agent/box color C:
   - 6.1. Calculate distances between all possible agent-box pairs of A_{UC} and B_{AC}, resulting in a distance (cost) matrix.
   - 6.2. Run Hungarian algorithm with the above cost matrix as an input and match a box to each of the agents. Disregard any extra boxes or agents.

After this process agents are always occupied with tasks, and they receive a new task (a particular box to be pushed into a particular goal, or moving into their own agent goal if no boxes are left of their color) every time they complete their current task or run out of goals to fulfill.

Heuristics

The heuristic function is composed of five terms that are weighted differently.

- $h_{goal-box}$ - Sum of distances between allocated goals and boxes. It motivates agents to move boxes towards their corresponding goals.
- $h_{agent-box}$ - Sum of distances between allocated agents and boxes. It incentivizes agents to move towards their corresponding boxes.
- $h_{fulfilled-goal-box}$ - Sum of distances between fulfilled goals and their corresponding boxes. It punishes agents for moving boxes that are already in their goals.
- $h_{unassigned-agent}$ - Distance traveled by unassigned agents compared to the previous state. Punishes the unnecessary movement of jobless agents.
- $h_{unassigned-box}$ - A punishment added for each unassigned box that does not fulfill a goal. Its role is to make the heuristic function consistent.

The final heuristic function takes the form of

$$h(s) = w_1 * h_{goal-box}(s) + w_2 * h_{agent-box}(s) + w_3 * h_{fulfilled-goal-box}(s) + w_4 * h_{unassigned-agent}(s) + w_5 * h_{unassigned-box}(s)$$

in which $w_1 ... w_5$ are (positive integer) weights denoting the "importance" of each heuristic term. The weights significantly affect the agents' behaviour - for instance, we must assign a very large weight to $h_{unassigned-box}$ to keep the heuristic function consistent. The actual values used by our planner are the following:
$w_1 = 2, w_2 = 1, w_3 = 5, w_4 = 2, w_5 = 1000$
We set the weights w_1, w_2 in a way that the planner values the movement of boxes towards their goals ($h_{goal-box}$) more than the movement of agents towards their boxes ($h_{agent-box}$). The large w_3 weight incentivizes agents to search for alternative solutions before pushing a box that is already in its goal. Finally, the weight w_5 is large enough to deter unassigned agents from unnecessary movements (and state generation), while it is low enough to let agents move out of the way of other agents, if needed.

Due to these larger-than-one weights and the presence of push/move distance-unaware punishment terms, the heuristic usually overestimates the actual goal distance, making the function non-admissible.

Centralized State Generation

One important part of our code relapses in the state generation and its management. We use a Best-First Search strategy combined with a limited frontier structure containing a maximum of 100 promising states (the ones with the lowest heuristic values) in order to find the most optimal solutions.

Our state generation consists of the following steps.

1. We take the first state from our priority queue, this is the state with the lowest heuristic value and possibly the lowest distance to the goal state.
2. After this we go through all the agents in the state while generating and saving all the valid actions for each of them. A valid action is one matching the preconditions (without taking into account the actions of the rest of the agents yet).
3. Then we make all possible permutations of these agent actions, generating every possible child state with its corresponding joint-action. The computational cost of this step is remarkably high.
4. We apply these joint-actions to the new states and we generate the heuristics for each of them. Here is where we check the possible collisions between agents, discarding all the states matching one of these situations.
5. Afterwards we add all the new states to the frontier. Remember that we are using a priority queue as a data structure for our frontier so, the states will be sorted by their heuristic value in ascending order.
6. Finally if the frontier is containing more than 100 states we remove the states with the highest heuristic values in order to maintain a reasonable length for the frontier. Without this condition the process becomes very heavy in terms of computational cost and our client could not progress properly.

This process is repeated for each of the states that our automated planning implementation takes from the frontier until we find a state matching the goal state conditions.

Experiments/Results

Before going with the rest of the section, it's important to notice that all experiments mentioned in this report were performed on a computer with a single Intel i7-8750H CPU 2.20GHz core and 8GB of RAM.

Our Levels

For the competition we submitted our own two levels, where we focused on different challenges that we thought would be difficult for clients to solve.

For our single agent level (check Figure 1), we focused on one simple challenge: moving a box away from a goal in order to get on the other side of it. Our idea behind this was to confuse agents moving the box away from the goal, since we assumed that many agents would use some kind of heuristic that would punish moving boxes away from their goals. The solution to this challenge is to have some kind of path checking feature that figures out when there is something in the middle of the way to the goal and, finally, manages to move the agent behind the box.

Figure 1: SAAIStars

Now if we take a look to our multiagent level, Figure 2 we wanted to add a more advanced version of the Blocks-World

stacking problem, where you have a stack of blocks and you have to order them in a certain way. The optimal solution consist of making the agents work in parallel, but also making room for the rest of the agents when the boxes are being put on their goals. An easy solution would be to just have one agent doing all the work, but with a lot of open space it is possible to have the agents working in parallel (after moving them out of their respective start points). Having all the different agents working together to push the correct boxes inside the dead end in the correct order seemed to be an interesting challenge. However, we were not able to solve the level ourselves in the end because of challenges when handling many agents, so the level was not considered in the competition.

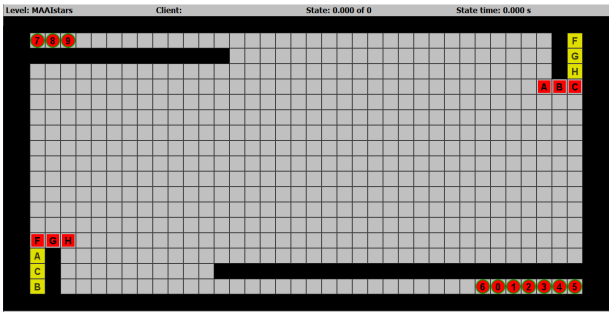

Figure 2: MAAIStars

Benchmarks

The final client ended up being able to solve 16 competition levels compared to the 9 levels it was able to solve for the competition date itself. Our choice for both performances was the Weighted A* strategy, with a weight value of 5. The rest of the strategies we were using during the development of our solution, Greedy and A*, were performing slightly worse (they are able to solve 1-2 levels less).

This improvement during the last two weeks is due to a number of changes and some debugging that we have been applying shortly after the competition.

When analyzing the client, we found out a huge amount of time was spent processing the maps and that is why we focused on improving that part towards the end, improving the pre-processing time while computing the distances between the graph nodes for large maps (more than 20x20 grids approx.). The previous iteration was not using an optimized library for converting levels into graphs.

In addition to that, checking, sorting and adding new states to the frontier also became an issue, causing us to shorten down the maximum size of the frontier to 100. Since the frontier set is sorted by the heuristic value removing states that were not in the top 100 did not cause the client to lose the quickest solution path. In terms of runtime, the mentioned improvement caused the client to go from a planning time of 180.2 seconds to only 17.4 seconds, and handling the frontier from 177.2 seconds to 14.7 seconds if we take a look to the *MAaicecubes* level for example.

```
========TOTAL TIMES========:
Heuristic calculations: 22.31895136833191
get children: 2.6988625526428223
run through children: 177.20679664611816
Reset Assignments: 0.227494478225708
Planning: 180.2187032699585
```

Figure 3: Old Client Runtime

```
========TOTAL TIMES========:
Heuristic calculations: 12.634941816329956
get children: 2.5393741130828857
run through children: 14.703521966934204
Reset Assignments: 0.24449944496154785
Planning: 17.36110258102417
```

Figure 4: New Client Runtime

Other problems that we saw were related to the joint-action. Our agent was failing to keep track of agent numbers in the solution path, so it was sorting the actions (into the joint-action) in wrong positions. This caused our client to send the actions to the server with an incorrect agent order for some of them, resulting in an unsuccessful level completion even though the level was correctly solved.

We were also not aware of agent goals being part of the goal state, resulting in good solutions not being approved by the server because of the final position of the agents. This was quickly implemented.

One of our last attempts to improve the performance of our solution was to include a fifth term in the heuristic's formula. This new term was trying to measure the impact of some obstacles (agents and boxes) present in the path between the current agent-box and box-goal assignments for a concrete state. If we define it mathematically we should add this new term to the heuristic formula previously mentioned in the *Methods* section: $w_6 * h_{path-obstacles}(s)$, where $w_6 = 5$ and $h_{path-obstacles}(s)$ is the sum of agents and boxes already mentioned. This path variant of the client was able to solve 14 levels (2 levels less than without the new term). Although a lot of the levels solved were the same ones, it was able to solve *MAHoldUd* for example (the final version of our client could not). This shows that it is better when dealing with small levels containing many agents compared to our final client, but it falls short on larger maps, such as *MAChuligans* or *SATheZoo* which the final client could solve, due to the path computation time.

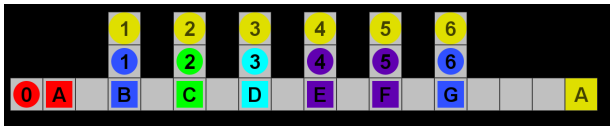

Figure 5: MAHoldUd

Finally we want to highlight some types of stages that our planner completes particularly well, these stages can be seen in Figure 6 and Figure 7.

This type of level where there are some spread out tasks that can be allocated to different agents work really well in our planner. We are able to pretty quickly find a close to optimal plan, that utilizes as many agents as possible. In both the highlighted levels we are able to assign the agents to each their own goal, and utilize all the agent that we have. The way that our planner works also means that in levels such as Figure 7 where there are more goals than agents, we are able to assign boxes to each goal, and when an agent is done with it's first goal we can reassign it to a new task immediately.

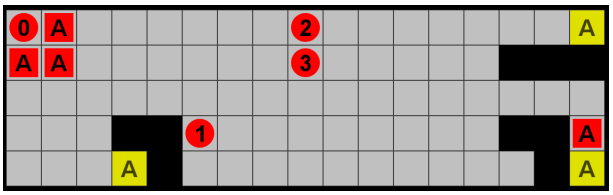

Figure 6: MAD3, we can solve it in 5 seconds at 19 moves

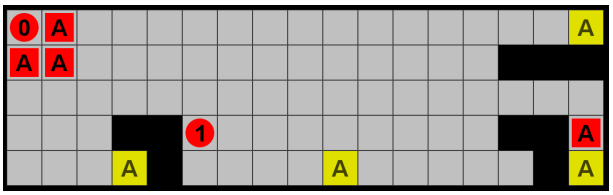

Figure 7: MAD4, we can solve it in 30 seconds at 26 moves

Discussion

As seen from our results, our planner cannot solve big complex levels, especially those containing a high number of agents (like our own multiagent level *MAAIStars*, for example). We can however solve smaller levels with a low amount of moves, especially levels where tasks can be delegated out to 2-3 agents we can solve efficiently.

The reason for the poor performance in larger levels is a combination of several issues, but they all stem from the same basic problem of scalability and combinatorial explosion.

The two main aspects of our planner that create this problem are:

- Multibody (centralized) planning
- Offline planning

A big weakness in our planner is the use of multibody planning, this means the practice of having a centralized planner that controls all the agents and plans them according to each other (in contrast to decentralized planning, where every agent acts on its own and communicates with the rest to coordinate and solve possible conflicts).

As seen from our results centralized planning scales very poorly with the number of agents. In fact, it scales exponentially: the number of possible actions in each state is x^n, where n is the number of agents and x is the possible moves for each agent. This is not a sustainable solution for more complex levels and, in hindsight, an approach using decentralized planning would have been much more suitable for this problem. We were able to counteract the combinatorial explosion somewhat by applying heuristics and limiting the amount of exploration that each agent did, but we could not, through heuristics, limit the number of actions that each agent can perform for each state.

Another big problem in our solution was the use of offline planning in contrast to online planning. We assign tasks to each of our agents but these tasks are only used to calculate the heuristics for the entire state. With online planning in combination with decentralized planning, it would be possible to create individual plans for each agent and start applying them straight away. This could sometimes cause conflicts between tasks or take us far from an optimal solution (increasing the total number of steps used) depending on how much each agent knows about the rest of the solution, but we would avoid the combinatorial explosion and we could have more flexibility to design a proper solution for it.

We did try to counteract our problems by simplifying the levels, things such as converting unmovable boxes to walls and pre-computing a lot of information about the level (such as distances and paths). In the end all of these countermeasures were only a minor improvement that couldn't make up for the loss of performance in agent-complex levels and we did not manage to have enough time to change our initial approach by a new one.

We can find one interesting example with the competition level *MAAIcaramba* (see Figure 8). Here our solution cannot generate the necessary permutations of all the possible actions due to the number of agents and the number of available actions per agent (there are no obstacles or possible conflicts in the start state), at least none of our computers can deal with this kind of complexity in less than 3 minutes. It is clear that we cannot use this approach for this kind of level.

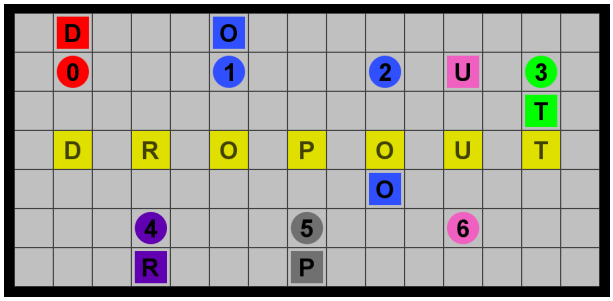

Figure 8: MAAIcaramba

Although we did end up fixing the problem in the end, throughout the project the graph representation that we had created ourselves ended up timing out on all large maps, resulting in us not completing many competition levels in the pre-processing phase alone. This caused a lot of issues and with better debugging / analysis tools at our disposal earlier

we would have noticed this and had more time on to use on other elements.

In conclusion, we could have made many smarter decisions than what we ended up with. We tried to make a very robust planner that would explore all the possibilities and by using smart heuristics, thinking that it would be able to counteract combinatorial complexity and edge cases. This was however a naive idea, and we have realized that our approach is simply not viable for big and complex levels. However, the way we assign boxes to goals could be used to develop decentralized planning with agents bidding for certain box-goal jobs.

Future Work

Taking into account that a major weakness of our planner is the extensive use of centralized planning, for future work a transition from a purely centralized solution to a more decentralized planning approach would be needed in order to solve complex levels.(Borrajo and Fernández 2019) This would also require us to implement more conflict handling for each of the agents, which was an advantage of the centralized planner. This individual conflict handling will probably require smarter domain-specific solutions, and it should allow us to scale more linearly with the number of agents (instead of exponentially).

A solution like this would also bring the possibility of applying online planning in order to make the decentralized planning easier. Our solution includes some kind of replanning if we think about the reassignment of agents-boxes and boxes-goals, but it's included in the offline planning because we make it before we start sending steps to the server.

A big goal of our planner was to have efficient minimal move solutions. With a decentralized planner, this goal is harder to achieve but, with smart conflict resolution and heuristics, it is definitely possible and computationally more efficient.

Going further into this decentralized approach, the most prioritized extension would be to create a queue of subgoals, similar to the POP-algorithm. We already save goals and assign boxes to them, but implementing a queue system (where maybe the agents can publish tasks using some of the multiagent protocols studied in this course) and/or converting the goal-box assignments to subgoals would greatly help the AI planning. This can be seen in the Blocks-World stacking scenarios where we do not have a priority system for which box and goal should be handled first, like in the multiagent level we proposed for the competition.

Another extension that could be used in combination with the previous subgoal extension is handling obstacles, more precisely agents and boxes, in order to avoid them in the way to its assigned goal or move them out of the path. This obstacle algorithm would calculate if it was worth it to move the box away or go around it. If it is worth moving away then it could be created as a subgoal that will be prioritized (making it kind of a prerequisite to complete the original goal) to the original goal and could potentially be handled by another agent for maximum efficiency.

Some other suggestions for the future could be related to deal with some interesting corner cases that are usually a handicap for the current implementation: more boxes than goals to push them, choose the best strategy for a level (depending on the number of agents, the size of the level, etc.), modify the weights of the heuristic function depending on the level (in the pre-processing), etc.

References

Borrajo, D., and Fernández, S. 2019. Efficient approaches for multi-agent planning. *Knowledge and Information Systems* 58(1):425—-479.

Froleyks, N. 2016. *Using an Algorithm Portfolio to Solve Sokoban*. Karlsruhe Institute of Technology.

Junghanns, A. 2009. *Pushing the Limits: New Developments in Single-Agent Search*. University of Alberta.

Kuhn, H. W. 2015. The hungarian method for the assignment problem. *Naval Research Logistics* 52(1):7–21.

NetworkX. 2020. Networkx. `https://networkx.github.io/`. Accessed: 2020-06-04.

Patel, A. 2020. Map representations. `http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html`. Accessed: 2020-06-04.

Virkkala, T. 2011. *Solving Sokoban*. University of Helsinki.