Implementation of a Single and Multi Agent system for transportation tasks
02285 AI and MAS
June 5th, 2020
NeverAI

Alexandros Dorodoulis s182148
Håkon Westh-Hansen s154313
Amalia Matei s153986

Abstract

The goal of this paper is to explain the implementation of a simplified simulation of how a multi-robot system might work in the hospital domain. The single agent (SA) solution is optimized to be as robust as possible. while for the multi-agent solution, the conflict management has been implemented inspired by the Conflict based search (CBS) algorithm. The main target has been improving competition results.

1 Introduction

The main problem that is being solved in the MA system is returning a set of actions that will get the agents closer to achieving the goals without conflicting with each other. This is interesting in the context of the hospital domain since they have a high number of transportation tasks that need to be carried out. We implemented conflict management which was influenced by the solution described in the (0inSharon et al. 2015) paper.

2 Background

2.1 Informed search: A*

Achieving the solution to the SA problem can be seen as the sum to sub-pathfinding problems. The shortest path between each box and goal has to be found, this can be solved by the A* algorithm. Whether or not this method is optimal heavily depends on the chosen heuristic: if it's admissible and consistent then the algorithm is both complete and optimal (0inRussell and Norvig 2016).
The chosen heuristic for the algorithm is the sum of Manhattan distances from the agent to the box and the box to the goal, where the boxes are pre-assigned goals in the processing phase of the algorithm (0inVirkkala 2011).

2.2 Conflict based search

The multi-agent pathfinding (MAPF) problem consists of planning paths for multiple agents, where the agents can follow their individual plans without colliding with each other. In the (0inSharon et al. 2015) paper, a new search algorithm called CBS is presented as an alternative to the current way of solving the previously mentioned task. To start with, the algorithm makes a plan for each agent without considering the other agents. This is followed by a check for conflicts, where agents are associated with constraints. The constraint is defined as a tuple where the agent is prohibited from being at a certain point in time and space. A new plan is then recalculated accordingly. This step is repeated until a solution where there are no conflicts, is found.
All in all, CBS searches a constraint tree where each node contains: a set of constraints, a solution, and the total cost. Each level of the tree depicting an algorithm iteration. CBS does return optimal solution but it comes at the cost of a high computational expense (0inSharon et al. 2015).

3 Related work

3.1 Sokoban solver

Sokoban is a single agent game where the goal is to push the stones into the any of the goal squares. This is done by pushing the stone one square at the time, along the four cardinal directions. Despite having quite simple rules, the problem is not, as it's NP-hard and PSPACE-complete (0inDor and Zwick 1999).
There have been many studies on algorithms that solve Sokoban in the scientific literature. Some of the popular ones are as follows: deadlock tables (calculating the positions of the deadlocks in the game), PI-corral pruning (reduces the number of positions expanding), hashing (avoids searching the same position multiple times), BFS and DFS, bidirectional searching, etc (0inVirkkala 2011).
Multiple versions of the Sokoban problem exist, some of which are closer to the problem described in this paper more than the original version.

- Sokomind plus : Boxes and goals are numbered and for the goal to be achieved the numbers have to match. This is similar to the SA problem solved in this paper, since the box and goal have to match by letter.
- Hexoban : The board is composed of hexagons, hence the agent can move in six directions.

- Multiban : More than one agent is included in this version, this is very similar to the MA agent task in this paper.
- Modern Sokoban : The blocks and floors have different abilities.

3.2 Multi-agent pathfinding

This MAPF problem has applications in automated warehouses, autonomous vehicles, and robotics and has thus received a lot of attention from the academic community (0inStern et al. 2019).
There are two main techniques when solving the MAPF problem: the decoupled path approach where the plans for the agent are made separately and a coupled approach where the problem is formulated as a SA search problem (0inSharon et al. 2015).
A decoupled implementation was described in the (0inDresner and Stone 2008) which was created in order to guide cars in an intersection. The approach is based on a reservation table mechanism, where the agent search has to avoid time and space points that have been reserved for previous agents.
The big issue with MAPF implementations is the fact that there is no universally dominant algorithm, as they are very susceptible to the characteristics of the problem at hand (0inSharon et al. 2013).

4 Methods

4.1 Preprocessing

In order to increase performance and cost estimation a number of different preprocessing methods were used. These are essentially the methods that are carried out once after a level has been parsed. A lot of the following actions are done in parallel in order to minimize the time that the program spends on preprocessing. This section covers the methods pertaining to this part of the client.

Immovable boxes translation

As the problem allows for boxes that don't have an agent to move them, also known as immovable boxes, a translation of these boxes into walls is carried out initially in the preprocessing phase. This is done because it is computationally cheaper to treat them as walls when the agents are searching for possible solutions. This is due to the method used to check if a move is legal, where boxes require an additional check to determine if the particular agent can indeed move that box. Moreover, boxes are used in the cost calculation part as described in section 4.2. Thus translating the boxes into walls is beneficial in two aspects.

Algorithmically the translation method is done by iterating over the boxes and checking if there is a corresponding agent of the same color as the box. If not then the box is translated into a wall.

Box to goal assignment

In order to minimize the total cost, in a situation where there are more boxes than goals, a way to assign boxes to goals has been implemented. The cost calculation as explained in section 4.2 has been used in order to designate the box with the lowest cost to the respective goal. It is important to note that the agent to box cost is not included in the assignment process, thus there are situations where a sub-optimal solution is achieved.

Agent to box assignment (MA)

For the multi-agent levels, to avoid multiple agents targeting the same box, each box is assigned to one agent. This has the added benefit of limiting, a bit, the amount of generated states, it effectively distributes work amongst the same colored agents and finally, it also improves the performance and the accuracy of the cost calculation as described in section 4.2. This is achieved by implementing the following code:

Algorithm 4.1: Agent to box assignment algorithm.

```
for box in boxes
  minCost = MAX_INTEGER
  agentWithLowestCost = ∅
  for agent in agents
    if ¬isBoxMoveableByAgent
      skipIteration
    end if
    coordsToConsider = agent.coordinates
    if agentAlreadyHasBoxAssigned
      coordsToConsider =
          ↪ lastGoalOfBoxAssignedToAgent.
          ↪ coordinates
    end if
    cost = costCalculation(coordsToConsider,
        ↪ boxCoordinates)
    if cost < minCost
      minCost = cost
      agentWithLowestCost = agent
    end if
  end for
  assignBoxToAgent(box, agentWithLowestCost)
end for
```

Where the `costCalculation` refers to the function described in section 4.2.

Wall preprocessing

To improve the performance and the accuracy of the cost calculation, the inner walls are extracted, meaning all the walls except the ones that surround the level. From there, we are extracting walls that are occupying continuous cells, either horizontally or vertically and we use them to improve both the performance and the accuracy of our cost calculation, as explained in section 4.2. If the previous process does not find any walls in the level it stores that information. That information is used by the cost calculation to completely skip trying to calculate the cost of walls. Moreover, it's also used during the multi-agent search to stop searching for agent

moves once the boxes that are assigned to that agent are in a goal state. This is done because of the assumption that once the agent has completed its goal and there are no obstacles in the level, it is no longer needed.

Final information extracted

In order to improve the memory usage of the program, during this phase, the program calculates the exact number of bytes required to store a unique identifier of a state. This allows for both less memory consumption as well as fewer memory allocations. Finally, the amount of goals in the game is calculated and a mapping is created between the index of the box and the index of the agent that is assigned to that box.

4.2 Cost calculation

There was a lot of time devoted in order to improve the accuracy of the cost calculation, without sacrificing performance. This part only mentions the methods that provided an improvement over the ”previous best” method and not all the methods that were tried.

Simple implementation

Initially, a simple Manhattan distance, as described at (0inVirkkala 2011) between the box and the closest goal was used. This solution had some serious drawbacks, for example, levels, where more than one box could be assigned to a given goal, were challenging for this initial implementation.

Manhattan Plus Plus

The next version was achieved by assigning boxes to goals and using the Manhattan distance between the predefined box to goal. On top of that distance, the Manhattan distance of the agent to its closer box was also added. This method improved the previous implementation but it was hard for the agents to find paths when there were walls between them and the box or between the box and the goal.

Manhattan Plus Plus with walls

This implementation expanded on the previous cost calculation method by incorporating the cost of the walls. Walls cost was calculated both between the agent and the box as well as between the box and the goal. The first implementation counted the walls inside of a grid that was defined between the target and its desired position. This count is normalized by multiplying it with the percentage of the rectangle area that the walls are occupying. The final step was done in order to try to minimize the overestimating of the cost of reaching the target.

Final implementation

Even with the ”normalization” that was done in the previous step, there were quite a few levels where the cost was overestimated. In order to address that, the final cost calculation used a different approach. The wall cost is bigger than zero only if the wall spans the full width, or more, of the rectangle mentioned in the previous method, this process is really quick because the information required for this is already calculated during the initial preprocessing stage (section 4.1). If the previous statement is met, then we calculate the minimum horizontal distance from our target to either end of the wall and that minimum distance is used as the wall cost. Figure 1 illustrates the aforementioned way of calculating the cost of walls.

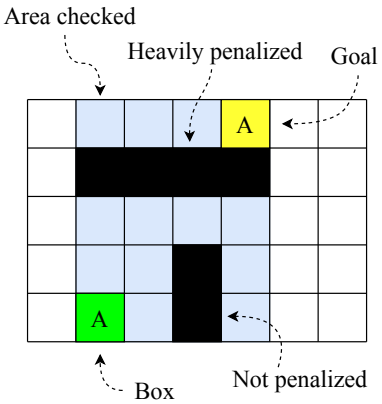

Figure 1: Visualized example of the walls part of the final cost calculation implementation.

4.3 Conflict resolution

In order to make the MA system work, conflict resolution is needed. Two different versions of it have been implemented and tested. We started with a very simple, naive implementation. The next possible step for each agent is planned separately and then all of the steps are merged together, during this merging conflicts are detected by checking all the possible combinations, in pairs of two. When a conflict is found, two new possible branches are created. One where for example agent 0 continues with the plan while agent 1 does a *no-op*, and another one where the opposite is done.

The second implementation was a bit more complex, inspired by the previously discussed CBS algorithm. It starts in the same way as the naive implementation, but now instead of automatically assigning *no-op* to one of the agents in conflict, the algorithm is asking both agents to generate a new plan, taking into consideration that the other agent does the move that caused the conflicts. Once all of the possible plans are generated the algorithm selects the plan with the lowest cost and uses that plan to resolve the conflict.

5 Results

During the development of the client a continuous test approach was used. Here is a selection of iterations that were carried out on the client:

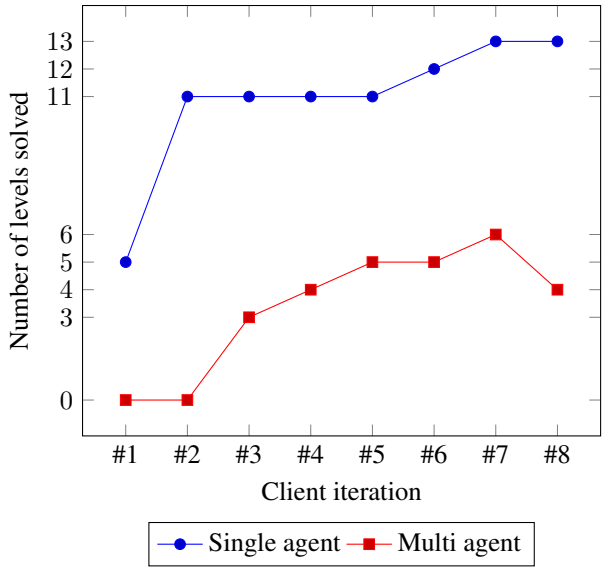

Figure 2

The client iterations are in chronological order and corresponds to the following:

1. *Competition client*: This was the client used in the competition. It performed quite poorly both in the single- and multi-agent setting, as can be seen in the above figure. It only managed to solve five single-agent levels and no multi-agent levels. The client had a cost calculation that used both Manhattan distance as well as the walls consideration described in section 4.2. Although it did not solve any multi-agent levels it did have a simplistic conflict resolution algorithm.
2. *Fix of competition client*: After the competition, a few bugs were discovered. The most crippling of which was one found in the function determining if the goal state had been reached. After fixing this issue the number of single-agent levels solved rose to 11. Essentially these solutions were also found with the competition client, but as the client was erroneous in determining that the goal state had been reached it never terminated.
3. *Fix of multi-agent sorting issue and addition of immovable boxes translation*: In the previous client a sorting error was discovered in the multi-agent part, which was causing severe problems. This issue was mended in this iteration and the translation of immovable boxes into walls was introduced, resulting in three multi-agent levels being solved.
4. *Fix of multi-agent intent issue*: In some edge cases there was an issue with some mixed pointers in the previous client, this client fixed those issues so it managed to solve one additional multi-agent level, compare to the previous client
5. *Fix for multi-agent empty agent plan*: This client fixed the handling of empty action plans returned by an agent, which is translated into a no-op action for that agent. This resulted in one more multi-agent level being solved.
6. *Improve wall cost approximation and assign boxes to agents*: In this iteration of the client, the wall cost approximation was changed to account for columns and rows as described in 4.2. Furthermore, the assignment of agents to boxes was added as described in 4.1. A test was also carried out on each of the changes mentioned here separately. This showed an interesting result wherein for the improvement of the wall cost approximation 13 single-agent levels were solved, but only four multi-agent levels were solved. However when this change was combined with assignment of agents to boxes change then 12 single-agent levels were solved, but the amount of multi-agent levels solved jumped back to five.
7. Leverage agent to box assignment for cost calculation: This iteration uses preassigned boxes for each agent to improve both the accuracy and the speed of the cost calculation for each state. Some other notable changes for this iteration, are improvements for conflict detection, better cost estimation for the cost of the agent to reach the box, and a drastic reduction in memory usage by optimizing the generation of unique IDs for each state.
8. *Use CBS variation for Multi agent*: This iteration switched our naive conflict resolution with a more complex one, which resulted in losing 2 multi-agent levels. This is discussed in more detail in part 6.

Level	*Actions*	*Time* [s]	*States Expanded*
`SAAIstars.lvl`	38	0.012	137
`SAChuligans.lvl`	274	60.677	323181
`SACoronAI.lvl`	136	0.033	237
`SAGLaDOS.lvl`	239	0.041	515
`SAKaren.lvl`	489	107.926	1205469
`SAMAAIStro.lvl`	11	0.008	12
`SAMulle.lvl`	134	0.065	2930
`SANicolAI.lvl`	204	0.188	4571
`SAThree.lvl`	21	0.007	37
`SATrueGlue.lvl`	114	0.050	969
`SAaiaioh.lvl`	15	0.012	39
`SAaicecubes.lvl`	27	0.012	60
`SAfootsteps.lvl`	270	61.267	80894
`MADeepMinds.lvl`	53	0.094	1635
`MAGLaDOS.lvl`	245	68.202	46040
`MAMAAIStro.lvl`	45	0.110	2915
`MAMulle.lvl`	157	20.843	72067
`MAaiaioh.lvl`	14	0.017	213
`MAaicecubes.lvl`	84	13.203	76284

Table 1: Results from the solved levels of client #7.

6 Discussion

The main target of this project was to create a single-agent (SA) and a multi-agent (MA) system that can solve transportation tasks. The SA implementation was based on A*

with added enhancement, mostly through preprocessing like box to goal assignment, optimized cost calculation that takes walls into consideration, and memory usage improvements. As can be seen in figure 2, the solver has been greatly improved compared to the one used in the competition. Our best SA implementation managed to solve 13 of the competition levels, compared to initially only being able to solve 5.

The MA solver had two different versions, a very naive conflict resolution using *no-op* and a more complex version that was based on the CBS algorithm. Surprisingly, the simple version manages to solve 6 competition levels, while the more complex one falls short with only being able to solve 4. While trying to figure out what the issue was with this implementation, we could see that the algorithm is choosing the expand the wrong move according to our cost calculation. Also, merging plans for the multi-agent system has proven to be trickier than expected with a lot of opportunities for small bugs to creep in and create major issues in the agent.

In order to keep track of the results for each of the implementations a script was created. That script parses the output of the server and creates a pretty table reporting if the agent solved the level or not, how many actions the agent used, how much time did the agent needed to solve the table, and how many states the agent searched through to find the solution. This script can be run as by itself or as part of a GitHub action. If it's run as a GitHub action it comments the results on the pull request, which provides an easy way of keeping track of progress. An example of how that looks can be found here [1].

All in all, going from theory to implementation has been a challenging experience. We did manage to parallelize the vast majority of the code, the exception being the planned merging. This has led to a quick state exploration and with a few more improvements we also achieved a low memory consumption,especially considering that A* was used and that previous states where stored in order to not be re-explored.

There are many improvements that could have been done such as adding a goal prioritization, using bidirectional search instead of A*, full implementation of CBS, etc. These topics are further expanded upon in the future work section below.

7 Future work

The first important improvement would be adding goal prioritization. During the different test runs for our AI solution, it was discovered that in a lot of the cases sub-optimal solutions were found due to the agents not ordering the completion of goals in an optimal way. One way to fix this issue would be to implement an algorithm that checks whether or not the completion of one goal obstructs the completion of another. In that situation, the goals should switch places in the priority queue. All the goals are gone through in this way until we have a final priority queue.

Secondly, while A* works well and it's a relatively easy to implement, the bi-directional search would be the next step to take in our list of improvement since it has the potential to yield enormous savings from the perspective of the number of generated nodes (0inVirkkala 2011). Bi-direction search is essentially composed of two searches starter simultaneously, one at the initial state and one at the goal state. At each step of the way, the algorithm checks whether or not the two searches are intersecting. The moment they do, the searches stop as the path has been found.

Another important issue is that the possible states grow exponentially, relative to the number of agents present in the level. To improve that a more informed state expansion could be done. For example, when there is no wall between the box and the goal (or the agent and its target box) and the goal position doesn't block another agent from achieving its goal, the state expansion could be trimmed down to expand only towards directions that bring the box closer to the goal, expanding 2 directions instead of 4.

Last but certainly not least, full implementation of the CBS as described in the (0inSharon et al. 2015) is to be desired. Furthermore, a more thorough investigation of why the more complex version of the conflict resolution is not performing as expected should be carried.

References

[Dor and Zwick 1999] Dor, D., and Zwick, U. 1999. Sokoban and other motion planning problems. *Computational Geometry* 13(4):215–228.

[Dresner and Stone 2008] Dresner, K., and Stone, P. 2008. A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research* 31:591–656.

[Russell and Norvig 2016] Russell, S. J., and Norvig, P. 2016. *Artificial intelligence: a modern approach*. Pearson, third edition.

[Sharon et al. 2013] Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence* 195:470–495.

[Sharon et al. 2015] Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219:40–66.

[Stern et al. 2019] Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks.

[Virkkala 2011] Virkkala, T. 2011. *Solving sokoban*. Ph.D. Dissertation, Master's thesis, University of Helsinki.

[1] `https://github.com/alexdor/dtu-ai-mas-final-assignment/pull/10?#issuecomment-639386727`