Multi agent push and pull - group bAnAnA

Aleksandrs Levi
s162870

Arturs Gumenuks
s170466

Alina Gabriela Ciobanu
s155505

02285 - AI&MAS
May 30, 2018

Abstract

This report describes a solution to a multi-robot system for transportation tasks problem. It reviews theories and related work used over the course of the project and describes the overall algorithmic functionality of the developed client. The solution is based on using various map preprocessing methods, problem decomposition for separating overall level solution into granular tasks, and utilizing various search techniques for completing those tasks. The developed client is able to solve a wide range of single-agent and simple multi-agent levels; the paper concludes with highlights of experiments with the developed client and discussion of the potential improvements.

1 Introduction

Robot motion problems are an important topic of robotics and AI research, with many industrial applications like manufacturing, transportation systems for hospitals (Bolander et al.), harbors, airports and marshaling yards (Alami et al. 1998). Important resource savings could be made in those different applications by employing a system of multiple robots (further referred as agents) that can perform the necessary planning to satisfy a given goal (Ahmed 1997). This requires gathering knowledge about the world, constructing a model of it and using the model to create and execute a plan in a efficient fashion (Ahmed 1997). A common requirement is also having specialized robots to handle different types of transportation tasks i.e. in a hospital environment: beds, medicine, blood samples, mail (Bolander et al.). This report describes a solution for a simplified single and multi-agent transportation system, abstracted to a moving blocks problem. Focus is put on map analysis, task decomposition and goal prioritization, which makes the solution perform efficiently on a wide range of single agent problems.

2 Background and related work

The problem studied in this report belongs to the moving blocks class of planning problems with $Push$ and $Pull$ actions, previously described as *Pukoban* (Zubaran and Ritt 2011), (Pereira 2016) and shown to be PSPACE-complete. While an extensive literature is available on the moving blocks problem with $Push$ only actions known as *Sokoban*, previous work and results for Pukoban are limited (Pereira, Ritt, and Buriol 2016).

Compared to the Pukoban domain described in literature, further complexity is added by having agents being able to only move boxes that have matching colors. Several challenges were studied for implementing our solver, detailed in section 3:

- Serialisable subgoals

  A quick inspection of the example levels provided in the course material shows that most deal with serializable goals (Korf 1987), that is, subgoals must be satisfied in a specific order. This is not trivial in levels with many subgoals; in fact it was previously shown that finding a reliable subgoal ordering is as difficult as proving the existence of solutions (Yu et al. 2004).

- Effective path finding

  The problem is also a motion planning problem: finding a path the agent can take to move boxes from an initial position to a goal position without colliding with obstacles (walls, agents, or boxes it cannot move) (Ahmed 1997). According to Halldorsson, (Halldórsson 2015), this requires good path-finding techniques and map analysis to reduce computationally expensive searching. Map decomposition is a common approach in moving-blocks problems research (Botea, Müller, and Schaeffer 2003), (Pereira, Ritt, and Buriol 2015), including one of the best known Sokoban solver, the Rolling Stones(Junghanns and Schaeffer 1999), (Junghanns and Schaeffer 2001). (Halldórsson 2015) introduces two heuristics; the *dead-end heuristics*, intended at identifying, as the name says, dead ends (areas that cannot possibly be on an optimal path between two given points on the map) is relevant here for solving many of the levels given. Three map elements commonly discussed in previous work are *tunnels* or narrow areas that can be only crossed by one resource (man/agent) at a time, *rooms*, or larger free areas that can be shared by multiple resources and *articulation squares*, map cells connecting the first two elements (Junghanns and Schaeffer 2001). Identifying the presence of obstacles (walls) and certain configuration of the map, like tunnels or dead ends can be used in goal ordering and for guiding the search. Path clearing also needs to be guided so that it does not create more conflicts. Most common heuristic guided algorithm used in previous work is A* (Hart, Nilsson, and Raphael 1968).

- Planning as a search

  With a combinatorial explosion of the state space for planning at the atomic level (Gribomont), there is a need for a different planning strategy. (Gribomont) shows how hierarchical planning can decompose a complex Sokoban problem into simpler subproblems that can be solved with an effective search algorithm like A*: this is commonly seen as the strategy adopted by human players. A set of high level actions (HLA) was identified by studying the problem. The use of Hierarchical Task Network (HTN) planning method on to decompose HLAs is described in section 3.

- Multi agent coordination and cooperation

  Finally, a core problem is the choice of the approach for agent design. A multi-body design requires no complex communication or coordination, but is exponentially more computational expensive with the number of agents. The solution to this is using multi agent approach. The *Belief-Desire-Intention* (BDI) paradigm was previously proposed in Sokoban for designing agents that can do complex reasoning (Caillou et al. 2017).

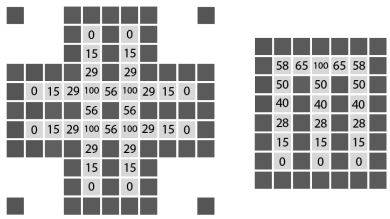
3 Methods

The client is built on the idea of using HTN in order to obtain atomic actions which could be guided by A* search. Moreover, a version of the BDI algorithm is used to facilitate replanning and reasoning of agents. The BDI control loop takes form of a state machine which also takes care of navigating through the refined actions. Preprocessing takes place before the start of the state machine, resulting in assigning each cell a weight based on how important it is. A more thorough description follows. (Note, that throughout the section we will be referring to a function for checking whether a path exists between two cells. The implementation of the function allows to treat or not to treat boxes, agents and walls as obstacles depending on the situation.)

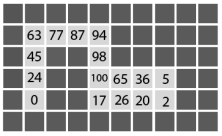
3.1 Pre-processing

Given the characteristics of the problem stated in Background and related work, pre-processing consisted of techniques for goal ordering, goal-box matching and map decomposition to aid the search.

Map decomposition. One problem considered was tunnel identification. First attempt was to check for all possible tunnel configurations, similar to the approach used by the Rolling Stones solver for dead-lock detection (Junghanns and Schaeffer 2001). 8 possible tunnel configurations were identified and checked against for each individual map cell. This resulted in false positives for some of the cells located in corners, next to level walls. Also, in order to be able to use this in the implementation, tunnels needed to be identified as a whole. The second attempt was to, starting from a specific cell, recursively search for all of the remaining tunnels and assign weights to the tunnel cell based on the tunnel type (i.e open ended, dead end), and to the cell position in the tunnel. This weighting covered only a subset of the map cells. Being able to determine the importance of all cells of a level could help further optimize the path finding, path clearing and goal ordering. To achieve this graph theory was used, building a map graph with cells defining vertices and all pairs of *SPACE* (non-wall) neighbour cells defining edges. The graph was used to calculate cell importance on the map graph based on the betweeness centrality (BC). BC is a measure of the ratio of shortest paths between any cell pair that passes through the specific cell. This way a good grading of importance of all cells can be achieved and crucial cells can be identified. For example in the levels shown in figure 1, (with normalized values of BC) most crucial cells have weight 100, while dead-ends have weight 0. Ordering goals only based on BC weighting can be tricky in symmetric levels like SAboXboXboX in (a), where the agent will try to satisfy all goals located on cells with weight 56 and afterwards goals on cells with weight 100, which results in a lot of wasted effort (solution is 762 moves). This can be optimized in combination with other techniques, like determining groups of goals (which was implemented in the same recursive manner as tunnels and is considered a future work to integrate it with the existing solution).



(a) SAboXboXboX and SAtowersOfHoChiMinh05 levels



(b) SAbotbot level

Figure 1: Betweeness centrality weighting on three different levels

Goal matching Current version of the client performs a very basic matching of boxes with goals. First, the relevant boxes are taken into consideration (i.e. boxes which have a letter corresponding to at least one goal in the level). Next, each goal is assigned a box which is closest to it (the distance is estimated using Manhattan distance) and for which a path exists between the box and the goal (to avoid trying to satisfy a goal with a box isolated from it by walls). This way, each goal is paired with a box with which an agent will try to satisfy it. This approach is certainly not flexible, and could be improved taking into consideration more factors - see 5.

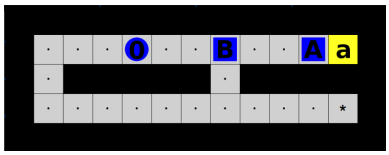
Goal ordering Initial attempt to prioritize goals (in SA track) was for every goal cell to check which combination of other satisfied goals prevents the agent from reaching that goal cell. This approach is in fact a partial order planning. A short description follows: Firstly, all unique ordering constraints (for example, goal *a* has to be satisfied before goal *b* is such a constraint if *b* is before *a* in a dead-end tunnel) were collected. Unique constraints were treated as logical sentences. Secondly, the algorithm found all logical interpretations which respected ordering transitivity property and made a developed logical formula evaluate to true. Any of these interpretations represented a legitimate partial order, so any sequence of goals respecting this order would work. One problem of this approach is that it does not consider changing agent position. Another problem is that it is heavy computationally as there will be 2^n interpretations where n is amount of unique constraints. Searching for a satisfiable interpretation leads to Boolean satisfiability problem (Larrosa, Lynce, and Marques-Silva) and was decided not to work with.

The second attempt has proven to be more successful. It prioritizes goals based on betweeenness-centrality calculated weights of goal cells, so the higher is the value of a goal cell (the more shortest paths go through this cell) the later this goal will be achieved. So, the list of *SolveAction* children (consisting of *AchieveGoal* actions, described in 3.2) is built based on this prioritization in a way similar to priority-queue building. The limitation of this approach is that it does not recognize goal groups, so a goal cell in the very end of one tunnel will be prioritized higher than, for example, an articulation square before another independent tunnel, while it would not matter which goal (of these two) is achieved first, if the tunnels are independent. Regardless, this approach significantly strengthened the client, as can be seen in 4.2.

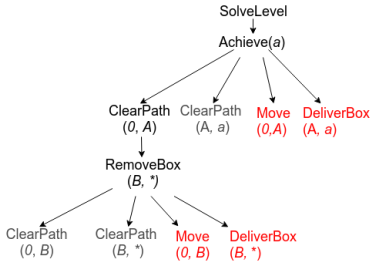
3.2 Hierarchical Task Network

The problem of solving a level of the game can clearly be quite complex and involve a multitude of actions undertaken by an agent. The approach taken in this project was aimed at having a data structure which would be feasible to traverse; it also needed to be able to cover situations which might be encountered in the game. This led to the task hierarchy which can be observed in Figure 2b. Looking at the figure, black colored nodes signify high-level actions which must be completed, gray - which have already been achieved and thus need no further action, and red - the atomic actions which are the actual actions carried out via search (A*).

In this example, the level consists of only one goal *a*. The top-level expandable action *SolveLevel* is thus decomposed into a single action *Achieve(a)*. In order to achieve the goal, the path must be cleared between the agent *0* and the box *A*, as well as between the box *A* and the goal *a*, resulting in the corresponding *ClearPath* actions. Every *ClearPath* action can be decomposed into a number of *RemoveBox* actions - equal to the amount of boxes on the path between the cells path between which must be cleared. Betweenness centrality algorithm is used to calculate an optimal cell where a box is moved (”parked”) when it must be removed from a path - in the example above, such cell is marked with *



(a) A simple level



(b) HTN task decomposition solving the level above

Figure 2: Using HTN task decomposition to solve a simple level

(which has weight 0). In this particular case, there is only box *B* standing on the way between the agent and the box *A*, and neither *ClearPath(0,B)* nor *ClearPath(B, *)* need to be processed as they are both already achieved. (In this case, the action is achieved when the path between the cells is already clear.) Thus, the agent proceeds to carry out the atomic actions *Move(0,B)* and *DeliverBox(B,*)*, resulting in removing *B* from the path between *0* and *A*. After completing the first two atomic actions, the *ClearPath* action between *A* and *a* is not further decomposed, as the path is already clear (i.e. action is already achieved). Thus, the agent proceeds with actions *Move(0, A)* and *DeliverBox(A, a)*. The level is now solved. Note, that the atomic actions *Move* and *DeliverBox* are carried out with the help of A* search (described more in 3.4). In order for A* to find a plan for either of the atomic actions, the path between the corresponding entities (cell-cell or box-cell) is assumed to be clear. This is the reason *ClearPath* actions come before any atomic actions take place.

3.3 BDI State machine

In order to establish certain way for the algorithm to traverse HTN tree and work with its nodes (expandable and atomic actions), a state machine was composed. The structure of the state machine resembles the BDI control loop. As described in 3.2, intentions (also referred to as actions) are hierarchically composed (i.e. one intention can be a child of another intention), so they are not on the same level. The state machine allows to build the HTN tree by repeatedly visiting the same state and traverse the tree to choose the intention to work with. A conceptual representation of the state machine can be observed in figure 3.

Since BDI is a model, our state machine implementation does not repeat any particular BDI control loop version,

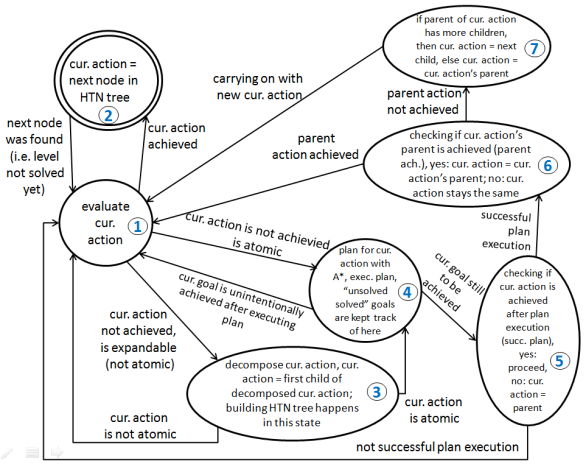


Figure 3: Conceptual state machine.

although being close to version 3 from the course slides (Bolander). In the Sokoban domain, there exist no contradicting desires. For example, a desire to achieve goal *a* in no way contradicts a desire to achieve goal *b*. The order might be problematic (which involves unsatisfying already satisfied goal), but goal prioritizer attempts to minimize the problem. Moreover, the way HTN is traversed will not allow to choose an intention if its preceding sibling is not achieved. Thus, all desires are considered intentions (also called actions).

The state machine will take a top level action and run decomposition until an atomic descendent is found. Once this is done, a plan for the action is computed using A* search algorithms in state 4. Atomic actions are designed in such a way that branching factor is low (compared to if A* was run on a higher level action). The plan for atomic actions is possible to find (except for one case, covered in discussion section), because paths are clear (thanks to HTN structure). The plan is executed immediately and stays sound at any step in SA track, because the environment is static (no other agents) and fully observable (current state of the world can always be accessed by an agent). This means that our system has offline planning. The domain is not fully deterministic because A* could move other boxes when planning for *DeliverBox* action, for example, even though the heuristic function includes punishments for diverging from the atomic action. In this way, it is possible to unintentionally undo some previously achieved action (see state 7 description further) or achieve, among other goals, the goal whose descendent the machine was working with, so there is no need to go further to states 5,6 and 7 and that is why link from state 4 to state 1 exists. If to stick to BDI terminology, these (5,6,7) are the states taking care of deliberation on intentions based on beliefs about achieved actions from the hierarchy.

Another point to mention is that in state 5 it is checked whether executing the plan for an atomic action led to achieving this action. State machine was developed with MA track in mind, where there can occur a situation that some other agent moves a box which current agent tried to move to and calculated the plan for. In SA track, plan execution will always lead to achieving that action which was planned for. State 6 is responsible for checking whether achieving an atomic action made the state machine achieve the parent action of the atomic action. As an example, a plan for *DeliverBox* is executed and some box is delivered. The question to ask here is whether the parent action, say, *RemoveBox*, is now also achieved, meaning that this box no longer stands on a path between some cells. If parent higher level action is not achieved, than the state machine goes to state 7, where two possibilities occur. The first one is when current atomic action is not the last child of its parent - there are more steps to do for a higher level action to be achieved. In this case, the state machine carries on with the next child in order to achieve higher level action. The second possibility is that that current atomic action has no more successor siblings. This means that all sub-actions of a higher level action were achieved one after another, but it did not lead to the higher level action being achieved itself. In MA track, it could mean that some agent undid the work of the current agent. In SA track it implies some changes in the environment (world) which the state machine did not take care of when planning. In either case, the state machine repeats the whole process for the higher level action (atomic action's parent) operating on the state of the world which has just been obtained during unsuccessful attempt to achieve this higher level action.

The execution of state machine terminates at state 2 if all goals are achieved in the current state of the world.

3.4 A* search algorithm

When an atomic action (either *Move* or *DeliverBox*) needs to be performed by an agent, the A* algorithm is used in order to find the path in the state-space which leads to fulfilling the action. The heuristic function *h* used for guiding the search differs depending on the action. Moreover, the client supports two different ways of calculating initial value of *h*. These are described below.

Manhattan distance based heuristics For *Move*, the Manhattan distance (explained in (Virkkala 2011)) between the agent and the cell towards which he is moving is calculated; moreover, to discourage the agent from moving boxes while the *Move* action is performed, any *Push* or *Pull* commands result in the increase of *h* value. Thus, the heuristic is not admissible; in particular, it will prefer a path in which the agent does not need to push/pull any boxes to a potential shortest path.

For *DeliverBox*, the Manhattan distance between the box and its destination is calculated; in addition, agent is heavily discouraged from moving other boxes (as opposed to the one he is supposed to be moving in the action), and slightly discouraged from pulling boxes instead of pushing them, both of which result in increase of the *h* value. Hence, this heuristic is not admissible either: it encourages pushing instead of pulling even when pulling would result in a shorter path; and, just like with *Move* action, it will prefer a path in which no other boxes need to be moved to a potential shortest path.

Shortest path found by BFS based heuristics In this case, BFS is used to calculate the length of the shortest path

between either the agent and the cell (*Move*) or the box and the cell (*DeliverBox*). (The shortest path in this case is calculated taking walls into consideration, making the heuristic more precise than using Manhattan). This value is the initial value of *h*, which gets further increased in same way as for the other distance based heuristics (discouraging agent from moving boxes when executing *Move*, and from moving other boxes except the one he is supposed to be moving when executing *DeliverBox*). The heuristic is not admissible for the same reasons as the Manhattan-based heuristic, but proved to be better at guiding A* to the goal state, as seen in 4.2.

3.5 MA environment

Current version of the client has a basic MA solver. In essence, a multiagent approach is used, where each agent attempts to make his own plan, and then proceeds to execute it. If a simple conflict occurs, such as a collision with another agent, the first agent does a NoOp and replans with the help of the state machine for the current state of the world. This approach works for levels which only involve this kind of conflicts (such as MAKJFWAOL, MAAiAiCap from 2018 competition levels); but not for levels requiring agents to cooperate in some way. More on potential for implementing advanced MA solver within our client's current structure is described in section 5.2.

4 Experiments/results

Over the course of development the client has undergone several improvements and modifications. By the time of competition, it was able to solve 11/26 SA levels (not counting the team's own level). However, further advancements were made after the competition was over, leading to being able to solve 18/26 levels; moreover, for most levels the solution consisted of fewer number of actions than before. The most notable changes include addition of goal prioritization(desc. in 3.1) and another type of heuristics (based on BFS shortest distance, desc. in 3.4). This section aims to summarize and compare various versions of the client. To draw comparisons, competition levels from 2018 programming project are used. Our team's levels submitted for competition are not particularly challenging (do not involve any unblocking, agent cooperation, etc.), and were simply what the client was able to solve at that time. This is why their discussion is omitted here.

Note: All experiments were performed on a computer with a quad core Intel i7-3630QM CPU and 6GB of RAM.

4.1 Client without goal prioritization

Parking cell selection One of the interesting points of comparison is how a "parking cell" is selected when an agent needs to dispose of a box (for instance, to clear a path). This is mainly based on BC weighting. However, in many cases the algorithm alone is not sufficient as parked box might need to be moved somewhere else, e.g. if it was placed in a wrong location beforehand - a common situation when there is no goal prioritization. To avoid a situation where an agent places a box in the same location over and over again because an algorithm thinks it is the optimal parking cell, we introduced a "punishment", i.e. a value "p" by which the price of a cell is incremented after it has been selected for parking a box. Tweaking the value changes the length of the solution for many levels, as can be observed in table in figure 4. Note, that the testing was done on the version of the client without any inbuilt goal prioritization (the one used for competition). It can be observed that for some levels the

Level Name	SL(p=1.5)	SL(p=50)
SAEasyPeasy	2355	1225
SANotHard	448	394
SAGreenDots	487	483
SABeTrayEd	**9410**	**1684**
SALobot	641	183
SAAlphaOne	127	125

Figure 4: Different solution lengths (in actions) depending on the p value used. SL=Solution Length. Most notable change is highlighted.

difference between solution lengths is rather large (e.g. SABeTrayEd is solved with about 5 times fewer actions with p=50 as opposed to p=1.5), while for others it is quite small. It can be explained by the fact that levels differ in size and probability that an agent makes a poor parking choice. Overall it seems that it could be beneficial in many cases to discourage agents from parking a box in the same cell twice.

4.2 Client with goal prioritization

Comparison with older client With the addition of goal prioritization, the client became capable of solving more levels and producing more efficient solutions. For the sake of comparison, the table in figure 5 can be observed. It is clear

Level Name	SL, No GP	SL, GP
SAEasyPeasy	1225	1155
SANotHard	394	162
SAGreenDots	483	328
SABeTrayEd	**1684**	**238**
SALobot	183	123
SAAlphaOne	125	95
SACybot	N/A	216
SAPushPush	N/A	834

Figure 5: Solution lengths (in actions) for versions of the client with and without goal prioritization enabled. SL=Solution Length, GP=Goal Prioritization.

based on the values presented that goal prioritization aids greatly in finding efficient solutions to levels (e.g. SABeTrayEd is now solved with almost 7 times fewer actions); moreover, solutions to some levels which were not solved before (SACybot, SAPushPush) have now been obtained.

Number of nodes generated by A* when using different heuristics Even with addition of goal prioritization, amount of nodes generated by A* had in some cases been too large, resulting either in longer computation time or in not being able to solve a level. With introduction of BFS heuristics described in 3.4, this problem has been mitigated, which is illustrated in the table in figure 6. Changing to a BFS based heuristic improves memory consump-

tion of the client, leading to being able to solve some levels which were not solved before (such as SAZEROagent, SAAntsStar, SAKaldi). This is achieved thanks to A* generating fewer nodes due to a more precise heuristic.

Level Name	NG, Manh.	NG, BFS sh. path
SAEasyPeasy	22106	7355
SANotHard	1649	1616
SAGreenDots	4868	2384
SABeTrayEd	6722	2452
SALobot	1364	1145
SAAlphaOne	661	647
SACybot	4934	4075
SAPushPush	27079	18395
SAZEROagent	N/A	11116
SAAntsStar	N/A	957
SAKaldi	N/A	36009

Figure 6: Number of nodes generated by A* with Manhattan vs BFS based heuristics. NG=Nodes Generated. N/A means that the level was not solved.

5 Discussion and future work

5.1 Strengths

The developed client is capable of solving a wide variety of SA levels. Among its advantages are the following:

- BDI-inspired state machine facilitates reasoning and replanning, and thanks to HTN structure the search carried out for atomic actions does not result in the explosion of state space.
- Cell weights based goal prioritization allows efficient (in most of the cases) solving of levels where goal satisfaction order is crucial and less repetitions of the same goal achieving actions occur.
- Dynamically changing decision making. The weight of a cell is increased whenever a box is parked there, so depending on the value, after some repetitions the box will not be parked in the same cell. This means that there can be two different parking decisions in two identical states of the world occurring at different points of state machine execution.
- Presence of modifiable variables (punishing prices, multipliers, etc.) allowing for adjusting the system to a particular level without modifying the algorithm and the structure. As an improvement, the values of these variables can be calculated automatically based on a level's traits and pattern (size, highest and lowest cell weights, similarity to Towers of Saigon level, for instance).
- Precise BFS-based heuristics resulting in significant reduction of generated state space nodes with insignificant loss in time (compared to Manhattan distance), such that levels involving high resource usage are solved as well.

5.2 Weaknesses

There also exist certain limitations of the developed client.

- Agent blocking a path is the most serious problem in the present approach. If an agent is located on a path between a box and some cell, *ClearPath* action will be achieved (as there are no boxes on the path), while a plan cannot be found in case there is no space for the agent to turn around (if the path is a tunnel made by walls and/or boxes). This results in A* algorithm being stuck, which is the main reason why some levels were not solved. In order for an agent to know how to turn around an additional action would have to be introduced to HTN which would imply a significant change to the system.
- Current actions are being achieved in a pre-defined order (clearing path from agent to box, clearing path from box to destination, moving to box, delivering box) and are treated separately. In reality, it might be more efficient to move freely from one action to another (for instance, clearing a part of a path, delivering a box to some intermediate cell, continuing clearing the path and so on).
- The box assigning is done in pre-processing, so the agent will aim for achieving a specific goal with a specific box even if at some state of the world it is more beneficial to achieve the goal with another box of the same letter.
- Immediate unsolved goal achieving. If the agent is achieving some goal g_1 and after execution of a plan of some atomic descendent it unsolves some already achieved goal g_2, the latter will be achieved right after g_1 is achieved. In fact, it might be beneficial to postpone unsolved goal g_2 achieving until the state of the world is more favorable for doing so.
- The path between two cells is either the shortest path with **no** boxes on it, or the shortest path with boxes on it, **all** of which will have to be removed. In some situations it might be better to choose a longer path with fewer boxes on it to be removed. This would require to develop a technique for finding an optimal relation between path length and boxes to be removed.
- Lack of goal grouping. All goals are perceived as belonging to one group. So, achieving a goal with lower weight is unconditionally done before achieving the goal with higher weight even though these goals might not be related and any ordering is applicable. For another problematic scenario please see map decomposition part in 3.1.
- Multi agent. As mentioned in 3.5, the MA development of our client is still in its infancy and this would be the major focus of further work. Firstly, a better conflict resolution was discussed, such that when a conflict arises between two agents and they both re-plan, their re-planned plans are not in conflict. Furthermore, agent communication e.g. using a agent communication language like FIPA is considered in order to be able to solve unblocking other agents. A sketch for MA is based on the current design (BDI state machine + HTN tree) and requires few modifications such as making a generic higher level action *RemoveObject* (object being a box or another agent) and asking for help in case the object is another agent or a box of color different from the color of the help calling agent. In terms of state machine execution, every agent would run its own instance of state machine planning for his subset of goals. Thus, this would be a multi-agent approach.

References

[Ahmed 1997] Ahmed, N. 1997. Robot motion planning.

[Alami et al. 1998] Alami, R.; Fleury, S.; Herrb, M.; Ingrand, F.; and Robert, F. 1998. Multi-robot cooperation in the martha project. *IEEE Robotics Automation Magazine* 5(1):36–47.

[Bolander et al.] Bolander, T.; Garnæs, A.; Jensen, M. H.; and Andersen, M. B. Programming Project. 1–12.

[Bolander] Bolander, T. *Slides 05s, Multiagent planning and agent architectures*. Provided as the material for the course.

[Botea, Müller, and Schaeffer 2003] Botea, A.; Müller, M.; and Schaeffer, J. 2003. Using abstraction for planning in sokoban. In Schaeffer, J.; Müller, M.; and Björnsson, Y., eds., *Computers and Games*, 360–375. Berlin, Heidelberg: Springer Berlin Heidelberg.

[Caillou et al. 2017] Caillou, P.; Gaudou, B.; Grignard, A.; Truong, C. Q.; and Taillandier, P. 2017. Advances in Social Simulation 2015. 528:15–29.

[Gribomont] Gribomont, L. P. Hierarchical planning and learning for automatic solving of sokoban problems.

[Halldórsson 2015] Halldórsson, K. 2015. Using Map Decomposition to Improve Pathfinding. (December).

[Hart, Nilsson, and Raphael 1968] Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

[Junghanns and Schaeffer 1999] Junghanns, A., and Schaeffer, J. 1999. Pushing the limits: new developments in single-agent search. (November 1999):195.

[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.

[Korf 1987] Korf, R. 1987. Planning as search: a quantitative approach. artificial intelligence 33, pp. 65-88.

[Larrosa, Lynce, and Marques-Silva] Larrosa, J.; Lynce, I.; and Marques-Silva, J. Satisfiability: Algorithms, Applications and Extensions. `http://sat.inesc-id.pt/~ines/sac10.pdf`. Accessed: 2018-05-05.

[Pereira, Ritt, and Buriol 2015] Pereira, A. G.; Ritt, M.; and Buriol, L. S. 2015. Optimal Sokoban solving using pattern databases with specific domain knowledge. *Artificial Intelligence* 227:52–70.

[Pereira, Ritt, and Buriol 2016] Pereira, A. G.; Ritt, M.; and Buriol, L. S. 2016. Pull and PushPull are PSPACE-complete. *Theoretical Computer Science* 628:50–61.

[Pereira 2016] Pereira, A. G. 2016. Solving moving-blocks problems. (October).

[Virkkala 2011] Virkkala, T. 2011. Solving Sokoban. 80.

[Yu et al. 2004] Yu, H.; Marinescu, D. C.; Wu, A. S.; and Siegel, H. J. 2004. Planning with Recursive Subgoals. 17–27.

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