Mandatory Assignment 3 - Group null
02285 AI & MAS

Daniel Schougaard
s103446

Kasper Reindahl Rasmussen
s103476

Martin V. Ottesen
s060186

Abstract

In this paper we present a planning system for solving Pukoban problems in a coordination effort of up to 10 agents. This involves route planning, conflict resolution, multiple heuristics, domain relaxation, distance metrics, partial-order planning and clustering. It describes how we have used these techniques in practice and attempts to justify the choice of distance calculation methodology, search algorithm and level of goal independence based on selected test cases concluding that in spite of high start cost a good distance map is central to the success of our solution. We discuss the strengths and weaknesses of our solution based on the completion level results. Finally we describe what would be the logical continuation of the project if we were to continue its development either by improving on existing functionality like algorithm selection or adding entirely new functionality like agent coordination, backtracking and improving the goal decomposition through e.g. goal ordering.

1 Introduction

Inspired by the developments of robotics used for transporting assignments to lower the need for humans to carry out these trivial tasks, the challenge is to have multiple robots capable of distributing these tasks among themselves, resulting in one combined multi-agent system capable of efficient and reliable execution of such tasks.
This paper describes a solution to this planning problem on an abstract level, where the hardware of the physical robots have been disregarded and the environments have been relaxed in such a way that they make up grid like patterns. The moving entities (agents) are given the ability to push and pull objects (boxes) around in the grid map. The objective is to move these boxes to a desired destinations (goals). The agents have to navigate around the obstacles in their way walls, other agents and boxes they are not allowed to move (restrictions are defined by agent/box colour), by enforcing these rules, restrictions and relaxations the resulting problem domain is of a single-/multi-agent Pukoban nature.

To solve the problem the level is first analysed mapping all agents, boxes, goals and walls, calculating distances from any cells to all other cells in the space, clustering the goals for the agents and then prioritizing them so that blocking goals can be avoided.

2 Background

The Pukoban problem is closely related to the classical automated planning problem called Sokoban. The difference is that you are also allowed to pull the boxes in the Pukoban case. As an automated planning problem we need to calculate action/command sequences to solve the problem/levels these sequences of commands have to be sent to to a provided server application.

The levels are solved using progression planning, implemented using a best-first heuristic search to explore the state space as the roles in the assignment implies completeness in the levels. The search algorithms we have found interesting to look at are the AStar and Greedy, as they are well documented and suit our purpose well.

2.1 Multi-Agent System

As part of the solution, we have to take the requirement for multiple agents into consideration. This means that some of the basic assumptions in automatic planning have to be re-evaluated as these are normally single-agent by default. Multi-agent systems are generally neither static nor fully observable this makes offline planning problematic as centralised planning is very computationally heavy. An alternative to this is to make the agents work autonomously. Full autonomy however opens up the possibility of agents obstructing each other and the introduction of colour restrictions on which agents are able to move which boxes can make some goals initially unattainable, this opens up the requirement for some sort of conflict resolution in the form of online replanning.

2.2 Relaxation

In order to make sure that an agent has a plan to carry out, even though none of its goals are actually attainable a relaxation of the problem may be necessary to relax the search space in some way. The relaxation has to be done in such a way that the plan is still feasible, so that the time spent on calculation a relaxed solution is not a waste of time and processing power only non-permanent obstacles should be disregarded and agent and box parameters are final. By following these rules we can end up with an emulated single-agent system. A relaxed solution is used to find a solution in optimal conditions. The length this solution is then used

to set a cutoff point to stop the non-relaxed search, before it expands the state-space too far.

2.3 Partial-order Planning

Another useful relaxation is only to plan on a subset of the goals at a time instead of trying to solve them all in one go, at the same time if you only look at one goal at a time the agents might constantly undo attained goals in order to fulfil new ones. In order to make this function optimally ordering of goals is also necessary so that the goals come in an optimal order (Han Yu and Siegel 2004).

2.4 Clustering

In order to optimise the levels with multiple agents with the same problem space the goals are decided into clusters. This is done using a center-based K-means algorithm (Likas, Vlassis, and Verbeek 2003).

3 Related Work

Sokoban is, as stated earlier, a classical automated planning problem. It was created in 1981 by Hiroyuki Imabayashi, and published in 1982. During the 33 years since its conception a myriad of solutions have been created to solve it. The Pukoban and Multi-Agent variations however are not as well documented.

3.1 Pukoban

In AGENT MOTION PLANNING WITH PULL AND PUSH MOVES (Zubaran and Ritt 2011) is a solution to a single-agent to the Pukoban problem. To the single-agent implementation this solution has a very similar approach it however uses the Kuhn-Munkres(Hungarian)algorithm, instead of Floyd-Warshall this gives them some overhead as they have to run it multiple times. (Zubaran and Ritt 2011) pairs up the boxes to goals they also analyse for bottlenecks called choke points and clean them for what they call "Clogs", they have no partial order planning so they only have a success rate of 30 out of 90 on fairly simple levels.

3.2 Multi-Agent

A MULTI-AGENT PLANNING APPROACH INTEGRATED WITH LEARNING MECHANISM (Tao Zhang and Ueno 2009) describes a solution to Sokoban problems by means of centralized planning. It integrates machine learning in order to build up a knowledge base capable of solving complex Sokoban problems. Since the problems the self-learning system is solving is Sokoban the state-space does not explode in the same way as a Pukoban would, this makes it possible to go throw possible combinations at a higher speed. By rewarded in accordance to how many goals were attained before the game deadlocks, the system learn what no to do in order to continuously improve on its solution.
It is hard to compare this approach to the one we are using for the Pokuban problem as but for a Sokuban type of problem a centralized should theoretically be better suited as cooperation is what gives the multi-agent approach an edge it could with enough time solve a NP-hard problem, and the central control system is a better choice for Sokuban as backtracking is not a possibility.

4 Methods

The general structure used is a multibody approach with preprocessing. The complexity is lowered by decomposing the problem by assuming that goals of the same colour, located close to each other are related and therefore can be grouped together (using k-clustering). It is also assumed that solving each grouping is independent of the others. One agent is assigned to solve each grouping. Each agent in this system is responsible for solving its own goals independently of the other agents, as far as possible. To lower the complexity further for each grouping the goals are solved cumulative. The order of solving the goals is based on a heuristic. It cumulative solves the goals, to avoid destroying already achieved goals, since they are assumed to be dependent on each other. To find a solution for a set of goals a search is performed. After each search the found plan is executed until it is completed or it fails. If it fails replanning is commenced.

4.1 The Heuristic

A series of heuristics is used, each one to solve a different problem. All the heuristics have that in common, that to enhance the heuristics the distances used, is the shortest path, when the domain is relaxed by removing all movable objects, these values can be precomputed (the all-pairs shortest paths problem).

Of all the heuristic used only two are noteworthy Goal Heuristic and Goal Selection Heuristic, the rest is basically just the distance to or from a point.

Goal Heuristic The main heuristic used, is the one for placing boxes on goals. It consist of two parts, one to ensure progress and one to ensure as little interference as possible with the other agents.

The first part calculates the shortest path from each of the agents' assigned goals to the agent, going through a box that matches the goal (boxes already on goals are not used). All these parts are summed together.

The second part is basically a goal count. It works as punishment for moving boxes away from goal fields, even though the goals are not part of the agents' goal condition.

This results in an non-admissible heuristic.

Goal Selection Heuristic This heuristic is used to choose the order in which the goals is solved. Goals are selected from a agents grouping mainly based on the shortest distance from a goal to the nearest similar box (not on a goal) and to the agent. The location of a given goal is also taken into consideration so goals likely to be blocked (e.g. dead ends and corners) is prioritised higher by a factor.

4.2 Distances in the Map

To increase the performance of our heuristics, we have used something we've called a "distance map". A distance map is a certain way to calculate the distance from one location to another.

One important fact to realise, is that the levels we work on here can be represented as graphs: Each cell is a vertice, V, and each connection to another cell is an edge, E. With this knowledge in hand, we quickly realised that we could employ various algorithms, originally designed to work on graphs. At the time of writing, we've implemented, tested, and worked with a distance map generated using the Floyd-Warshall algorithm.

4.3 Search

To find a plan to solve a given set of goals, a search is performed. The search is either a Greedy or A* search. A* is used in the single-agent world and when the problem is considered small enough, such as moving out of a given path. The search works under the assumption, that nothing changes in the world, except what the agent performing the search do. This will sometimes fail on the multi-agent levels, and we rely on simply letting the server *tell* the client, that there is a conflict and then replanning is commenced. This is by far the simplest possible way to detect a conflict, but it does inevitably result in a greater overhead.

To avoid using to much time on searches, which are impossible or just to time consuming, we employ a sort of double search. By first relaxing the domain, into a subdomain, only containing the agent itself and boxes, it can move. We then search this reduced domain. Using the length of the found route, we get a threshold for determining whether or not the normal search is worth progressing with or it should be cut short. This approach rely on the assumption that relaxed search is significantly faster than the normal search. This assumption is however not the case for the single-agent level, and it is therefore not performed in that case.

Another area worth remarking on here is that a randomized approach is used in expanding the search space, which means that a search is often not likely to result in the same plan each time. This is even more so the case, when combining it with a greedy approach and a non admissible heuristic.

4.4 Conflict Resolution

Since we're using multiple agents, each planning their own route, conflicts are bound to happen. We have chosen a method to solve these conflicts, which somewhat resembles online planning. But first and foremost we need to describe exactly how we *detect* these conflicts.

Detecting the Conflicts Currently we rely on two different methods for detecting if an agent is stuck. As was mentioned in section 4.3, we employ a double search. From the solution of the search in the relaxed subdomain, we derive a threshold. Should this threshold be exceeded, during the search on the original domain, we can safely assume that a conflict must be present.

The second approach is using something we call a `History`. Due to the way we implemented searching and re-planning, occasionally two agents could find themselves in a situation, where both of them would "cycle" between two locations. The `History` is a memory of a certain length, containing information about where an agent has been for the past x moves. Should more than a certain fraction of this history be of the same location, a cycle is detected.

We handle conflicts based on a priority system: At first, the agents' numbers determines their priority. Agent 0 has priority over agent 1, and so forth. Should new conflicts arise, while handling the conflicts, we always prioritise the newly found ones the highest. This is done to avoid unfortunate conflict-of-conflict situations.

Types of Conflicts Having detected a conflict, we partition the cause(s) into one of two groups: When it is an agent obstructing the route, and when it is a box. As a rule, all boxes found in the route are handled first. This is done to avoid cases, where an agent otherwise asked to move an obstructing box, is told to get out of the way first. An example of this can be seen below. 0 is a `BLUE` agent, and 1 is a `RED` agent. The box B is also `RED`.

```
...+++++++++...
      0 B1
...+++++++++...
```

In this example 0 would like to move right. Obviously, this is impossible, due to the fact that both agent 1 and box B is obstructing the route. Should we choose to handle agents first, agent 1 would move out of the way, leaving no one to move the box B.

Resolving a Box Conflict To move a box out of the way, we first need to identify which agent should "help". This is done by first trying to find the nearest agent, which currently does not have a plan. If no such agent is found, the nearest one is simply selected – disregarding its current plan. Once the selected agent has reached the box, we employ a very naive search: We breadth-first search, until both agent and box are out of the route. Once both are out of the route, a number of `NoOp`s are injected into the plan, to ensure that the agent receiving help has left the problematic area.

Resolving an Agent Conflict If an agent is obstructing the route, that agent is simply told to leave, much like described in section 4.4, only without a box. The exception to this, however, is if the length of the agent's plan, is below a certain threshold. Should the plan be below this threshold, the obstructing agent is allowed to finish its current plan, before getting out of the way *(if it is required at that point)*.

Resolving a Cycle A cycle is resolved by simply injecting `NoOp`s in one of the agents, enabling the other to re-plan itself out of the problem area. A cycle conflict falls under the category of an Agent Conflict, but is resolved and detected separately.

5 Experiments and Results

Throughout the development and implementation of the client for this project, we have tested out various ideas and designs. This section contains a mix of experiments performed during all stages of the development.

Tables 1 and 2 show benchmarks using a Greedy search and a A* search, with distances pre-computed using the

Level	FloydWarshall		Manhattan	
	Steps	**Time** *(ms)*	**Steps**	**Time** *(ms)*
SACrunch	104.0	167.0	105.4	574.8
SAFirefly	100.4	160.6	62.0	202.2
SABispebjerg	1520.8	2023.2	1568.0	9439.8
SAGroup42F13	375.0	31846.2	481.4	756.4

Table 1: Benchmarks using A* search.

Level	FloydWarshall		Manhattan	
	Steps	**Time** *(ms)*	**Steps**	**Time** *(ms)*
SACrunch	106.4	162.2	296.6	1849.6
SAFirefly	100.4	164.0	98.4	155.8
SABispebjerg	1530.0	1913.6	-	-
SAGroup42F13	375.0	31110.4	-	-

Table 2: Benchmarks using greedy search.

Floyd-Warshall algorithm, and computed on the fly as a Manhattan distance, respectively.

5.1 Testing Methodology

All testing is performed over the course of five iterations, after which the average is calculated. Results shown as "-" denote that a test failed all five attempts. The limits have been set to approximately two gigabyte RAM and five minutes. The benchmarks in Table 1 and Table 2 have been run on a system powered by an Intel i5-3570k (Ivy Bridge) processor, equipped with 8 gigabyte 1866MHz DDR3 RAM, running Ubuntu 12.04. This was also the system that was used for the competition.

Table 3 and Table 4 have been run on a system powered by an Intel i7-3680QM (Ivy Bridge) processor, equipped with 8 gigabyte 1600MHz DDR3 SDRAM, running Windows 8.1.

5.2 Comparison of Search Strategies

First off, it could seem like our benchmarks are flawed: Where a greedy search fails to find a solution using the Manhattan distance, A* actually manages to solve it, cf. table 2. This might seem slightly counter-intuitive but the reason is actually fairly simple: We apply goal decomposition, as mentioned earlier. Each search is treated as its own. Hence the difference here, stems from how Greedy and A* fares in search across the level. In larger levels, with a rather high branching factor, the Greedy approach would simply end up reaching the memory limit, before even getting a box *close* to the goal.

An interesting phenomenon, is that using the Manhattan distance results in a better solution, as seen on table 1. This is caused by a very odd set of events. We select the goal order based on the distance, cf. 4.1. When using the Manhattan distance, the inherent flaw in the calculated distance actually makes the goal heuristic select a *better* goal order.

Having said that, we can conclude using a Manhattan distance with a greedy search, is a horrible choice. Not only does it result in sub-par solutions, but it simply fails to solve both `SABispebjerg` and `SAGroup42F13`. This is due to the branching factor on the two levels, combined with the inherent source of error, from calculating distances using the Manhattan distance.

Finally, we see that by applying A* we are able to solve levels, which a greedy search failed on. Unfortunately the time for it to solve `SABispebjerg` is far from good. Impressively, we managed to solve `SAGroup42F13` using only 756.4 ms in average.

Having said all this, while limiting ourselves to only using the Manhattan distance, A* is the superior option.

5.3 Distance Maps

Having compared A* and greedy search with Manhattan distances, we can now compare this to their respective performances, using a pre-computed distance map. As we've mentioned before, we are applying the Floyd-Warshall algorithm to generate said distance map. This comes with a trade-off: Time. In terms of running time, the algorithm is bound by $O(|V|^3)$ in pre-computation time, where V is the number of cells. On some levels this trade-off is non-noticeable, simply due to the increased effectiveness. This is very much the case on `SABispebjerg` and `SACrunch`, cf. tables 1 and 2. However, when a level is of a certain size, the time explodes. This is clearly seen on `SAGroupF42F13`.

One *very* interesting effect of the Floyd-Warshall distance map, is that it makes the greedy search return an optimal result, cf. table 2. This is caused by the fact that for each location on the level, we know *exactly* how far there is to any given destination.

In the current test set, the only case where we see an actual *increase* in time is on `SAGroup42F13`. A theory as to why the pre-computation time is much more significant on this level, compared to e.g. `SABispebjerg`, is because `SAGroup42F13` can be considered a sparse graph. Because of this, the pre-computation time exceeds the benefit of the reduced search time. While the time penalty *is* a concern in this case, the fact that we increase the quality of the solution by roughly 29%, we feel that the Floyd-Warshall distance map is a good choice, in this domain.

5.4 Goal Dependency

Here we will be comparing how our solution performs in regards to total subgoal independence contra cumulative subgoal dependence. The results of this experiment are found on Tables 3 and 4, respectively.

The results show a clear difference between the single-agent and the multi-agent levels. The single-agent levels show that the performance is more or less equal, when comparing the number of steps and the amount of time. The only major difference is the success rate on `SAnull`, where a livelock occurs 4/5 of the time, due to the proximity of the goals. This tendency can however not be seen on the `SAFirefly`, since the agent has a fairly limited set of paths. This means that the agent can not easily move around previously achieved goals, and ends up undoing them, instead of progressing. This in return, results in a longer solution.

Level	Cumulative Steps	Time (s)	Rate
SACrunch	104	0.13	100 %
SAnull	97.8	0.15	100 %
SAFirefly	97.2	0.13	100 %
SABispebjerg	1526	1.8	100 %
MAAteam	338.6	13.9	100 %
MABullFightF11	335.0	14.7	100 %
MAdeepblue	252.4	8.2	100 %
MAnull	106.8	8.7	100 %

Table 3: Benchmarks using cumulative subgoals.

Level	Independent Steps	Time (s)	Rate
SACrunch	102.8	0.12	100 %
SAnull	95	0.17	20 %
SAFirefly	123.2	0.13	100 %
SABispebjerg	1523.6	1.7	100 %
MAAteam	228.8	6.1	80 %
MABullFightF11	303.8	8.2	100 %
MAdeepblue	229.4	8.1	100 %
MAnull	112.4	11.3	100 %

Table 4: Benchmarks using independent subgoals.

The interesting part of the data is that the independent goal strategy finds faster and shorter solutions for the multi-agent levels. This can be explained by the fact that if an agent interferes with another agent's goal, it is not necessarily best to fix the goal immediately, but instead wait for a more opportune time. The only apparent disadvantage is the same as for the single-agent: It can end up in livelocks.

Part of the explanation is also that the heuristic helps to avoid destroying already achieved goals, and thereby negating most of the disadvantages that the independent goal approach has.

For the multi-agent levels, `MAnull` breaks the pattern. This is due to the fact that the most time consuming part of the level, is in reality a minor version of the single-agent level `SAboxesofHanoi`. This means that a performance similar to that of a single-agent level, is to be expected.

6 Discussion

With any choice there is bound to be pros and cons: In this particular case, it is primarily the eternal question of the quality of the solution, compared to the time required to generate it. An example of this trade-off is that we feared our double-search would impact our performance in a negative way. What we were happy to discover was that it actually increased it.

This is due to the fact that we effectively terminate any search, which would otherwise take too long to complete. The benefit of this approach was clearly shown in the results of the competition.

One unfortunate thing that turned up during the competition, is that we're only able to successfully solve `SAHoldKaeft` at around 25% success rate. Our investigations show that this has to do with the way we prioritise goals. The problem is that we treat the prioritisation as a weight, instead of an actual sequence. The way we've chosen to do it is *fast*, but obviously this has certain shortcomings. In this case it unfortunately results in a rather low success rate.

Levels such as `SAWatsOn` and `MAmultiagentSort` unfortunately unearths another weakness in our solution: Multiple goals and boxes with the same letter in close proximity *can* result in the agent moving a box between two goals, while also moving other random boxes, as to create an entirely new search space. When caught in this behaviour, it will result in a practically infinite search space. The inherent cause of this issue, is that this new "goal state" is reachable within a single move. Thus when expanding the search space, we will always reach this problematic goal state, no matter what the heuristic returns.

However, this is just our apparent weaknesses. Our solution also showed to excel at other types of problems. When the level is effectively partitioned into several independent sub-problems, we see how our approach of decomposing the overall problem excels. In this case – both single- and multi-agent – we see how we solve the problem quickly, while also obtaining a solution of high quality. Additionally, because multi-agent levels often are highly independent levels, we managed to perform goal decomposition to a much greater extent, than we did with the single-agent levels. We reckon that this is the primary reason, as to why we perform better at multi-agent levels. One of the methods we employ to create this decomposition on the multi-agent levels, is the process of clustering the goals. This is a far more logical process to perform on these levels, than on single-agent levels.

While we admittedly are in a fully observable world, where a perfect master plan could be devised, we've chosen to rely on relaxation and online planning. This obviously comes with some drawbacks, but also has its advantages. When performing a search to plan a solution, we don't take the other agents into account. This greatly reduces the complexity of the problem, while also simplifying the algorithm.

Additionally it also introduces an obvious opportunity to employ parallel programming, to further increase performance. However, taking the restraints of this project into consideration, resource-wise, we chose not go down this path. Another reason not to implement this, is that we fear non-obvious drawbacks, complications, and bugs would arise.

7 Future Work

As with any project, there is a deadline. As the deadline approaches, one often finds that there are certain aspects that could still be improved. In this section, we will go through the most significant of these aspects, which could be implemented beyond the deadline of this project. Whether these improvements were not implemented simply due to a lack of time, or simply not prioritised, is another story.

We've chosen to partition possible improvements into one of two categories: Increasing the performance of the current

approach, using more complex algorithms, and taking an entirely new approach to a sub-problem.

7.1 Improving the Current Approach

Some improvements that could be to our solution, are primarily changing the algorithms and data structures that makes up the foundation of our project, to more advanced and efficient ones.

Distance Maps As we saw in our benchmarks, there *is* a considerable time penalty for using the Floyd-Warshall distance map on the `SAGroup42F13` level. Originally we chose this particular algorithm, due to it being considerably easier to understand and implement, than other *slightly* more efficient algorithms. We estimated Floyd-Warshall to be the best benefit/resource ratio.

Having said this, if we had had more time, this is definitely a source of improvement. One improvement is using Johnson's Algorithm (Chaudhary, Sablok, and Gupta) instead, when the level is considered sparse. This effectively reduces the running time from $O(|V|^3)$ to $O(|V|^2 log|V| + |V||E|)$, where $|V|$ is the number of vertices and $|E|$ the number of edges. But this is far from the best that *could* be done. Mikkel Thorup proposes an algorithm in (Thorup 1999) which can calculate all shortest paths from a single source in $O(|E|)$ time. This can then be repeated for all sources – or vertices – to calculate *all* shortest paths in $O(|E||V|)$ time. Having said that, the algorithm proposed by Thorup is *far* more complicated than the Floyd-Warshall algorithm.

Searching the Nodes During the development of the code, we saw that the standard Java PriorityQueue was not optimal: The `.contains(..)` method – which is invoked a considerable number of times – was causing a significant performance issue. This is caused by it simply iterating over the queue in $O(n)$ time, cf. (Oracle 2015), each time it is invoked. To avoid this, we added a secondary data structure: A hash-set. Using the hash-set we can invoke `.contains(..)` in expected $O(1)$ time.

However, this comes at a disadvantage: We have to keep *two* data structures in memory, while also maintaining them. To avoid this issue, another approach could be to use a Fibonacci Heap (Boyer 1997). This data structure would allow us to perform most operations at expected time $O(1)$, while in linear space, resulting in a *theoretical* improvement.

7.2 New Approaches

There is a limit to what can be improved, by using the existing approach and optimising on it. It can therefore be necessary negate some disadvantages by using a completely new approach.

Coordination We had problems with several agents' lack of coordination. This often caused agents to hit each other, even though there would be plenty of space for them to navigate around each other. Additionally, it also resulted in agents performing more `NoOps` than necessary, when resolving a conflict.

To mitigate these issues, all agents need to have some idea of where the remaining agents are, at all times. This can be achieved by using a blackboard approach. This lets all agents validate and modify their plans, before executing them.

Backtracking To solutions of a higher quality, backtracking could be improved. This could be done by performing the search completely offline, and thereby allowing the search to backtrack to before the current goal. This will also allow the system to recover from situations, where the agent would otherwise get stuck, due solving the goal in the wrong order.

Goal Decomposition To increase the performance on single-agent levels, a more guided approach could be used. This could be done by grouping goals together, applying the same assumption as used for the multi-agent clustering. The difference from the multi-agent approach would be that more groupings than agents exists. A more advanced approach would be to apply the techniques from (Han Yu and Siegel 2004) to find the ordering of the goals and solve them in that order. This approach could also be applied in combination with clustering for multi-agents, to determine the way goals are grouped.

7.3 Generalisations of our Solution

It could be interesting to apply the system on other domains. We will therefore in this section be looking at what it would require to exactly that.

Sokoban One of the most obvious generalisations would be Sokoban, or any variants of it. Due to it being a very similar domain, it will therefore be simple to generalise to it. The only concern would be that Sokoban does have some irreversible actions/situations, due to the fact that there is no pull action. The concern is that a single error can make a level unsolvable and it would not necessarily be obvious at the time. This means that online replanning can be problematic and backtracking is far more important in this domain.

Controlling NPCs A not quite-so-obvious generalisation, could be to employ our solution for controlling the behaviour of NPCs *(Non-Player-Character)*, in open-world sandbox games.

Imagine such a game, where a player is allowed to roam freely around. In these games NPCs usually stroll about, minding their own business, until their wonderful day is disrupted by a maniac with a Gatling gun. These walking, driving, cycling, or some other form of transportation patterns, could easily be described using our solution.

What makes our solution excellent for this purpose, is that we employ online planning. The player – an outside agent – comes in and disrupts the agents' view of the world, by completely eradicating the car in front, using aforementioned Gatling gun. Had we used offline planning, this would be a huge issue. However, with *our* approach a simple replanning – or invoking our conflict resolution – suffices to figure out how the NPCs will react.

One aspect where modification would be required is the conflict resolution. As it stands now, NPC-NPC conflicts are

solvable, but a more detailed description of Player-NPC conflict would definitely need to be modelled. This, however, could rather easily be achieved by expanding on how we analyse and resolve conflicts.

References

[Boyer 1997] Boyer, J. 1997. The fibonacci heap. *Dr. Dobb's Journal* 22(1):106–8, 114–15.

[Chaudhary, Sablok, and Gupta] Chaudhary, R.; Sablok, A.; and Gupta, D. -. All pairs shortest paths algorithms. Technical report, Bhagwan Parshuram Insittie of Technology, Delhi, India, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.5554.

[Han Yu and Siegel 2004] Han Yu, Dan C. Marinescu, A. S. W., and Siegel, H. J. 2004. Planning with recursive subgoals.

[Likas, Vlassis, and Verbeek 2003] Likas, A.; Vlassis, N.; and Verbeek, J. J. 2003. The global k-menas clusterins algorithm. *The Journal of the Pattern Reconition Society*.

[Oracle 2015] Oracle. 2015. Class priorityqueue¡e¿. `https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html`.

[Tao Zhang and Ueno 2009] Tao Zhang, L. Z., and Ueno, H. 2009. A multi-agent planning approach integrated with learning mechanism. *Robotics and Biomimetics, 2008. ROBIO 2008. IEEE International Conference on*.

[Thorup 1999] Thorup, M. 1999. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM* 46(3):362–394.

[Zubaran and Ritt 2011] Zubaran, T., and Ritt, M. 2011. Agent motion planning with pull and push moves. *Encontro Nacional de Inteligncia Artificial (ENIA)* 8.