Master Thesis

Mitigating dead- and live-locks in Machine Learning-based Multi-Agent Path Finding

K. de Klerk

Delft University of Technology

TU Delft

SIOUX TECHNOLOGIES

Master Thesis

Mitigating dead- and live-locks in Machine Learning-based Multi-Agent Path Finding

by

K. de Klerk

Student Name	Student Number
Kieran de Klerk	4810708

Delft, 20 June 2025

Supervisor:	Dr. C. Pek
Company Supervisor:	T. van Groeningen
Company:	Sioux Technologies
Faculty:	Faculty of Mechanical Engineering, Delft

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

SIOUX TECHNOLOGIES

Acknowledgements

I would like to thank the following people:
Dr. Chris Pek for the supervision and insightful meetings, Tom van Groeningen for the daily support, Tom Rijnsburger for feedback and code reviews, Sjoerd Nijboer for the use of his personal computer and Anna de Boer for her endless patience.

Summary

When a robot is navigating through an environment crowded by obstacles and other robots, it can get stuck while trying to go from point A to point B. If they get stuck in place, this is often called a deadlock. If they rather get stuck in a pattern of repetitive movements, this is called a livelock. Those locking situations are detrimental to the efficiency of the robot as it loses time and possibly energy. Locking situations should therefore be avoided as much as possible or be recovered from as quickly as possible.

Some planning algorithms are more prone to locking situations than others, in particular, decentralised learning-based planners such as DCC, PRIMAL and MAPPER. This is caused by their lack of a mechanism that explicitly coordinates different agents and their limited perception of the state of the environment. It would be beneficial if the tendency to get dead- or livelocked could be curbed with simple techniques that can be applied to any planner that works in largely the same manner. Hence, this research proposes and evaluates four methods: punishing actions that lead to an agent getting locked during training (Lock Punishment or LP), training on locking situations encountered during training to teach the model to recover from such a situation (Targeted Training or TT), teach an already trained model to recover from locking situation (Targeted Fine-tuning or TF), and placing obstacles that are only visible to an agent that has just entered a locking situation to encourage it to move away from that location (Phantom Obstacles or PO).

All of these methods are evaluated on the same benchmark to see which singular or combination of techniques performs best. In the end, the most effective methods turned out to be PO and a combination of TT and PO. PO demonstrates an increase of 1.6% in its mean success-rate and a decrease of 3% in its computation times over the baseline, and TT+PO's mean Sum of Costs, Sum of Fuel, and Makespan are lower than the baseline by 0.8%, 1.2% and 2.1% respectively. Although LP+PO has relatively poor performance in most metrics, it reduces the number of locks the baseline encounters by 66%, thus achieving the lowest number of locks.

Contents

1

Introduction

Imagine a scenario where you are walking on the street and you notice someone in front of you walking in the opposite direction. How many times has it happened that you attempt to avoid an imminent collision by stepping to the side, only for the other person to step in the exact same direction before mirroring your movement to the other side once again? Much like us ”sidewalk shuffling” humans, robots can get stuck in a loop, repeating the same actions instead of reaching their desired positions. These looping situations are often referred to as live- or deadlocks, depending on whether the robots are moving or not. Most of the time, this is a problem with the path finding algorithm, occurring both when planning for single agents and when planning for multiple. Locks are more frequent in the latter case, specifically when using planners that do not plan the whole path in one shot [1][2]. These include windowed planners as well as decentralised learning-based planners such as PRIMAL and its variants [3][2] [4] or DCC [5]. When robots guided by these planners lock up, it deteriorates the solution quality, or even causes a failure to return a valid solution (i.e. all agents reaching their goal). He et al. have proposed the SYLPH framework designed to avoid locking situations by training agents to dynamically select their own degree of selfishness [6]. However, this framework is quite complex and designed specifically for the rest of the network.

The aim of the research presented in this thesis is to develop and evaluate simple, drop-in methods to reduce the tendency of a decentralized learning-based solver to get caught in a deadlock or livelock situation. This research proposes four methods developed for reinforcement learning-based planners: Lock Punishments (LP), Targeted Training (TT), Targeted Fine-tuning (TF) and Phantom Obstacles (PO). These methods were submitted to an ablation study to find out which method or combination of methods are the most effective reducing the number of locks encountered and increasing the success-rate. Furthermore, insights into where locks occur most frequently was drawn from a number of heatmaps generated for each individual method.

Chapter 2 provides the necessary background information by formulating the specific problem that is addressed in this research and provides all crucial definitions along the way. From there, chapter 3 picks up the thread by explaining the implementation of the aforementioned methods. Chapter 4 walks through the experiments that were performed, their results, and provides interpretations of the latter. Finally, chapter 5 summarises the findings and provides recommendations for future research.

2

Background

Before looking into how to solve dead- and live-locks, some definitions should be laid out concerning the environment in which they occur, how solvers' performance is assessed, what solver was used, and what constitutes a lock.

2.1. Defining MAPF problems

The research presented in this thesis focuses on Multi-Agent Path Finding (MAPF) in two-dimensional grid map environments. For the purposes of this research, each agent occupies one entire cell and obstacles are represented by a number of inaccessible cells (as shown in figure 2.1). Then, at any time

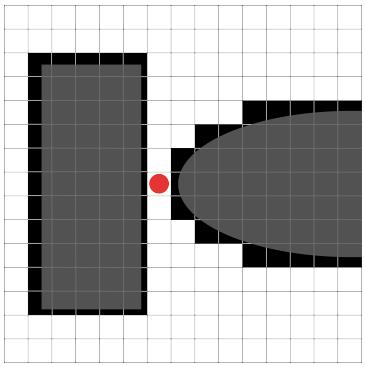
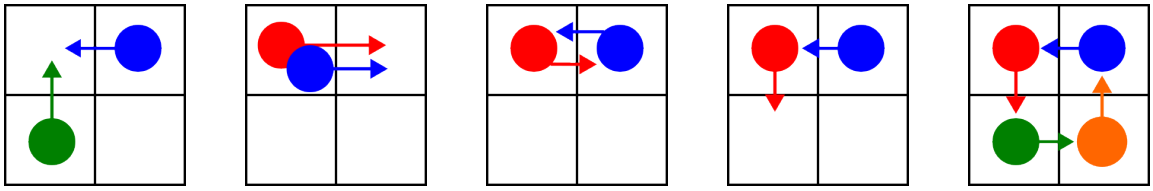


Figure 2.1: Obstacle representation in a 2D grid map. Grey shape represent continuous space obstacles, blacked out cells represent the discretised version of the grey obstacles, the red circle represents an agent

step T, each agent is allowed to move one cell in any of the four cardinal directions (i.e. *'up'*, *'down'*, *'left'*, *'right'*) or to stay in place (i.e. *'wait'*), resulting in the action space $\mathcal{A}$ with $|\mathcal{A}| = 5$. Naturally, certain rules have to be imposed on how agents can move relative to each other and to obstacles. Disallowing agents to move into obstacles is trivial, but what constitutes collisions between agents may depend on specific circumstances. Stern identifies five elemental conflict categories that might cause problems in different settings: vertex conflicts, edge conflicts, swap conflicts, follow conflicts and cycle conflicts [7] (see figure 2.2).



(a) Vertex conflict **(b)** Edge conflict **(c)** Swap conflict **(d)** Follow conflict **(e)** Cycle conflict

Figure 2.2: Illustrations of elementary conflict situations. The arrows indicate the agents' intended action

For the purpose of this research, agents are considered to occupy the majority of the surface of a cell, thus not leaving any space for other agents to enter the same cell or to cross paths with another agent within the width of a cell. Therefore, only vertex conflicts (see figure 2.2a) and swap conflicts (see figure 2.2c) will be taken into account and are defined in detail below. On the other hand, follow and cycle conflicts should not lead to collisions on the condition that all synchronous motions are correctly coordinated. Assuming agents do not start in the same cell, edge conflicts can also be disregarded as they consist, in essence, of two consecutive vertex conflicts and would therefore never occur.

The following are definitions for both vertex and swap conflicts, where π_i is the sequence of actions $(a_1, ..., a_n)$ of agent i' and $\pi_i[T]$ is the location of agent i' at time step T after taking action a_1 through a_T from its starting position [7]:

- Vertex conflict (figure 2.2a): Occurs between 2 or more agents $i, ..., j$ when there exists a time step T for which $\pi_i[T] = ... = \pi_j[T]$. In other words, the conflicting agents plan to occupy the same vertex at the same time step.
- Swap conflict (figure 2.2c): Occurs between 2 agents i and j when there exists a time step T for which $\pi_i[T+1] = \pi_j[T]$ and $\pi_i[T+1] = \pi_j[T]$. In other words, the agents plan to swap to each other's previous position.

2.2. Metrics

2.2.1. Path length metrics

Most path finding problems require solutions to be as short as possible. However, "short" is ambiguous when considering MAPF problems, which is why this research uses the following path length metrics [7].

- Sum of Costs (SoC): $\sum_{i=1}^{k} |\pi_i|$, reflecting the total uptime the solution requires of the agents.
- Sum of Fuel(SoF): $\sum_{i=1}^{k} \sum_{t=0}^{|\pi_i|} \begin{cases} 0, & \text{if } \pi_{i,t} = \text{'wait'} \\ 1, & \text{otherwise} \end{cases}$, reflecting the total amount of energy or "fuel" required to execute the planned path.
- Makespan: $\max_i(|\pi_i|)$, reflecting the time it takes for the slowest agent to reach its goal.

2.2.2. Success-rate

Success-rate is the most commonly used metric to compare different solvers and is defined as the fraction of problems out of a fixed set a solver is able to solve within a time limit. However, the definition of this time limit often differs between the literature on one-shot and sequential solvers (not to be confused with one-shot and lifelong MAPF problems). One-shot solvers return either an entire solution, or nothing, depending on whether a solution is possible and whether it can be found within the given time limit, while sequential solvers return a single action, or a sequence actions until reaching their goal. For the former, the limit is set on the allowed computation time (commonly 60 seconds), while for sequential solvers, it is common to put a limit on the simulation time (or makespan). This research follows the example set by Sartoretti et al. [3] and sets the makespan limit for maps of 40×40 cells to 256, and adds 128 for every doubling in map dimensions.

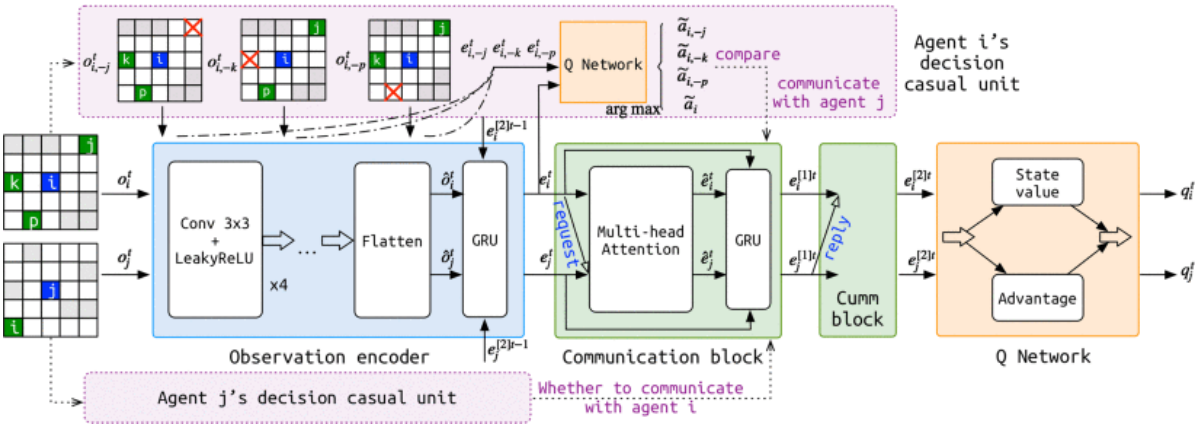


Figure 2.3: DCC system flow [5]. From left to right, the agent's limited observation of its surroundings is processed by the observation encoder in blue, before the result is passed on through the first communication block (green), which hands out its information as a request to all agents indicated by the decision causal unit (in pink). The output of the first communication block is passed on to the second communication block which sends it out to all agents that had sent it a request in the previous block, before processing all messages together with its own, into a final message that is used by the Q Network (orange) to determine the best action to take.

2.2.3. Goals found

Knowing the number of goals that were found before a problem instance ran out of time or steps can provide valuable insights. This metric is expressed as a percentage of the number of agents.

2.3. State-of-the-art solvers

In theory, even simple but effective algorithms like A* [8] are able to solve MAPF problems, however, the complexity of a problem scales exponentially with the number of agents (time complexity for A* : $\mathcal{O}(b^d)$, where d is the length of the solution and b is the branching factor that scales exponentially with k), meaning that the time and memory requirements will make it extremely impractical [9]. This is why much research has been done into better suited alternatives such as Conflict-Based Search (CBS) [10] and its many variants [11][12][13][14][15][16][17] or M* [18] and its variants. These are considered to be among the state-of-the-art in traditional (i.e. non-learning-based) MAPF solvers and most use the principle of decoupling [19]. This means that, instead of planning the actions for all agents simultaneously, these planners only consider a single agent at a time, allowing for much faster computation in most cases. In order to avoid collisions, some process still needs to look at the actions of different agents and compare their timing (e.g. the high-level search of the Constraint Tree (CT) in CBS [10]), which can scale badly with the number of agents. Distributed planning could circumvent this by letting each agent plan its own path. Most current research into distributed planning focuses on learning-based approaches. PRIMAL and its variants [3][2][4], DCC [5] and MAPPER [20] are often counted among the state of the art in this field. This research is based on DCC, as it has the most straightforward setup, thanks in part to the fact that Ma et al. made their source code publically available.

2.3.1. Decision Causal Communication

Decision Causal Communication (DCC) [5] builds forth upon *Distributed, Heuristic and Communication* (DHC) [21] by altering its communication mechanism: rather than broadcasting its information to all agents within a specified radius, an agents learns which information to communicate and with whom to do so. This was proven to be more effective than the technique applied in DHC, especially in agent-dense environments. As can be seen in figure 2.3, the model consists of four main components: the *observation encoder*, the *causal decision unit*, the *communication block*, and the *Q-network*.

Input tensor

The input of this model is a local, $6 \times l \times l$, observation tensor centred on an agent, where l is the size of the Field Of View (FOV) of the agent ($l = 9$ for this research and the results of the original work[5]). The first 2 channels represent a local map of the obstacles and a local map of other agent positions,

respectively, while the last four channels are heuristic channels which help guide the agent to its goal by indicating whether taking either action is likely to get the agent closer to its goal from each cell within its FOV (see figure 2.4).

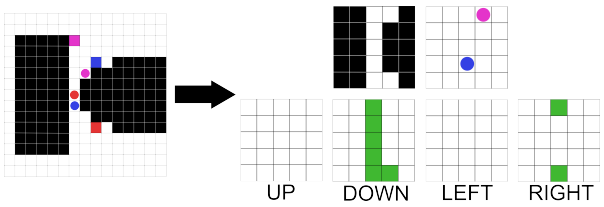


Figure 2.4: Representation of the observation tensor of the red agent. The top left grid shows the obstacle view, where each obstacle is represented by a 1 and free space by a 0. The top right shows the agent view, where each agent is represented by a 1 and the rest of the spaces are 0. The four bottom views are the heuristic channels. The green cells represent that taking the action corresponding to the channel in that cell will move tha agent closer to its goal. Each green cell is represented by a 1.

Preprocessing

The *observation encoder* takes the observation tensor and feeds it through 4 consecutive convolutional layers whose output is then flattened and fed into a Gated Recurrent Unit (GRU) [22][23], using the output of the *communication block* from the previous time step as the hidden state. The result is an intermediate message that can be fed into the *communication block*.

Communication decision

The *causal decision unit* decides which nearby agent is worth requesting for information by estimating the effect each agent within the original agent's FOV would have on its action. This is done by computing an intermediate message for each situation where a single agent is removed from the original agent's FOV and feeding those directly into the *Q-network*, bypassing the *communication block*. The resulting actions are compared to the action obtained from passing the regular intermediate message directly through the *Q-network* and a communication scope is established, where only the agents in whose respective absence the output changed are selected for communication.

Communication

The communication block takes the intermediate message output from the *observation encoder* as input, as well as the relative positions of other agents within the FOV. The communication block sends requests in the form of its intermediate message along with the relative positions, embedded as an $l \times l$ one-hot matrix fed through a fully connected layer, to each agent selected for communication by the *decision causal unit*. If the agent receives requests, they are used to update its intermediate message with a multi-headed attention mechanism. The updated message is then fed into a GRU, with the original intermediate message as hidden state, to aggregate them into a new, first round, message. The same process is then repeated using the first round message instead of the intermediate message, sending it out in reply to all agents that sent a request. The resulting second round message, $e_i^{[2]t}$ is fed to the *Q-network*.

Translation into action

This *Q-network* is made up of dueling Q-networks [24], meaning the network splits into two separate streams, one going through a fully connected layer producing the state value $V_s(e_i^{[2]t})$ for the current state s, the other going through a different fully connected layer producing the action advantage $A_a(e_i^{[2]t})$ for each action $a \in \mathcal{A}$. The Q-values for each action a at state s are then estimated as follows for agent i.

$$Q_{s,a} = V_s(e_i^{[2]t}) + \left(A_a(e_i^{[2]t}) - \frac{1}{|\mathcal{A}|} \sum_{a'} A_{a'}(e_i^{[2]t}) \right) \tag{2.1}$$

Environment

The environment keeps track of where all agents are at all times and whether they are currently standing on their goal. Agents do not dissapear upon reaching their goal and can still choose to move away from

their goal at any moment until all agents reach their goal or until the makespan limit is exceeded. At that moment, the environment is said to be finished.

Training

The model is trained using reinforcement learning, guided by a curriculum. Through the use of the Python package Ray[25], a set of actors (16 in Ma et al.'s research [5], 29 in this research) are each assigned one CPU and are tasked to execute their copy of the model in their own environment instance in order to gather experience and feed it to a buffer. The buffer ranks the experiences according to their expected learning utility which is estimated using their absolute Temporal Difference (TD) errors [26]. The learner then samples the most useful experiences to compute a multi-step TD error and subsequently applies Adam optimization to update the weights. Each update is henceforth referred to as an epoch of the learner.
The curriculum walks the actors from easier and smaller environments to increasingly complex environments. It starts at an environment of 10×10 cells with a single agent and will eventually go all the way to 40×40 maps with 16 agents, making step of 5 cells in the map size and steps of 1 agent. The actors copy the learner's update parameters periodically and randomly sample new environment settings from the curriculum every time they finish an environment. One such run through an environment is referred to as an episode. Whenever actors collectively achieve a success-rate 90% in samples with k agents and $N \times N$ maps, the model is considered to enough to move on to more complex problems. As such, the curriculum adds both $(k' = k+1, N' = N)$ and $(k' = k, N' = N+5)$ to its list of samplable settings.

Execution terminology

Execution refers to any forward pass through the model that will not be used to then calculate gradients. Thus, the actor processes execute the network in order to obtain actions that will then be fed to their environments. In order to differentiate between execution during training and after, the latter will be referred to as application.

2.4. Dead- and live-locks

The distributed learning-based solvers discussed earlier tend to suffer from dead- and live-locks, blocking a relatively small number of agents from reaching their goals [2]. As demonstrated in figure 2.5, failure to solve a problem often comes paired with a higher number of locks than success. Therefore, reducing the number of times an agent enters into lock situations and their durations may improve the likelihood of the solver successfully guiding each agent to its goal. However, before solving these lock

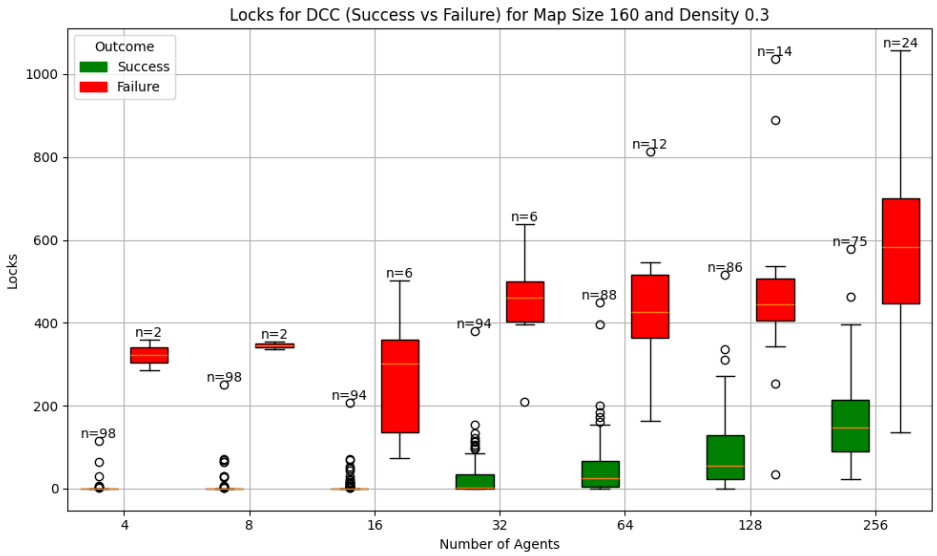


Figure 2.5: Boxplots showing the distribution of the total sums of locked steps accross all agents in different environments of 160×160 with 30% obstacles categorised by number of agents and whether or not DCC was able to solve the sample, illustrating a negative correlation between the the number of locks and the probability of success

situations, a concrete definition should be established:

- **Deadlock**
  - Collision-induced deadlock: occurs when an agent collides with the environment (i.e., obstacles or environment bounds) or enters into a vertex or swap conflict with other agents for 3 consecutive time steps. Collisions happen extremely infrequently in DCC (around 1% of randomly sampled runs has more than 0 collisions), implying that when they do happen, the agent doesn't know what to do. This is why agents are only allowed 2 consecutive collisions before being noted as locked.
  - Waiting deadlock: occurs when an agent waits in a cell that is not its goal for 10 consecutive steps. This limit is chosen arbitrarily to allow agents to wait in place for potential other agents to leave the first agent's intended path without being considered locked.
- **Livelock**
  - Short pattern livelock: occurs when an agent oscillates back and forth between 2 adjacent cells 3 consecutive times (i.e., 6 consecutive steps).
  - Long pattern livelock: occurs when an agent travels the same cyclic path 3 consecutive times.

With these concrete definitions of what this research attempts to solve established, it is time to move on to how it will do so.

3

Approach

This chapter enumerates and explains the different methods of mitigating dead- and live-locks developed and evaluated in this research. First, a system of lock-specific negative rewards is proposed to penalize undesirable behaviour. Second, data aggregation techniques are used to collect and target training on locking scenarios, either during curriculum progression or as a fine-tuning step. Lastly, the novel concept of phantom obstacles is explored as a runtime intervention method to subtly guide agents away from problematic states without affecting their broader decision-making logic.

3.1. Lock Punishment

During training, agents can be discouraged from entering locking situations by awarding a negative reward for each step where the agent is locked. As can be seen in table 3.1b, this punishment is set to -0.1 for a collision-induced or a waiting lock in an attempt to strike a balance between effectively discouraging locks and keeping training stable. Livelocks are punished by half that amount, as they are somewhat less detrimental. Previous experiments showed that higher values caused slow convergence or bad training stability as evidenced by extremely slow progress through the curriculum. Note that these values are added to the regular rewards (see table 3.1a), a collision-induced deadlock step is therefore awarded a total of -0.6.

(a) Baseline DCC reward structure

Action	Reward
move	-0.075
wait on goal	0
wait off goal	-0.075
collision	-0.5
finish	3

(b) Lock punishments structure

Lock type	Reward
collision lock	-0.1
wait lock	-0.1
short pattern	-0.05
long pattern	-0.05

Table 3.1: Tables showing the comparison between the rewards structure of DCC and the empirically determined reward structure of the lock punishments

3.2. Data aggregation

Another way to help DCC learn how to handle dead- and livelocks would be to train specifically on the situations where they occur. To achieve this, situations are stored when one or more locks are detected at certain points during execution before being sampled for training. Two distinct approaches to using this data aggregation method to improve DCC are detailed below.

3.2.1. Targeted training

When using targeted training (derived from the DAGger algorithm [27]), the data aggregation takes place simultaneously to the training. When an actor process encounters locked agents, the current

state of the environment is stored. This includes the map (i.e. obstacle positions), the positions of agents and their goals, as well as the last action taken by each agent.
Every n episodes an actor samples three lock situations ($n = 30$ for the purposes of this research) and trains on them instead of reinitialising its environment based on randomly picked settings from the current curriculum. These situations are removed from storage in order to be replaced with new ones when an actor encounters new locking situations. This process of storing and sampling is only activated once the curriculum has reached sufficient complexity (according to an arbitrary function with a parametrised threshold), with the aim of limiting the number of stored samples near the start of training and give the model a regular beginning.

3.2.2. Targeted Fine-tuning

Instead of interspersing regular randomly sampled environments with locking situations during regular training, the targeted fine-tuning process takes a pre-trained model and trains it specifically on those locking situations. In order to establish a sufficiently large and varied dataset, a data aggregation script randomly initiates environments with randomly sampled settings (number of agents, map size and obstacle density) and stores situations where the agents encounter locks. Since the time that the gradient calculations take during training scales very quickly with the number of agents and the numbers of agents that can be sampled are quite high (64 to 256) only a subset of the agents are stored when locks are encountered. The subset consists of all agents that are locked at the time step the situation is saved, as well as each agent that requests to communicate with these agents or received a communication request from these agents as, according to the decision causal unit (see section 2.3.1) these are the only ones that will affect the locked agent's actions. Any samples that end up with $k > 32$ aren't stored. This was deemed to be a sufficient limit, as the original curriculum does not exceed 16 agents and yet the model still manages to generalise to 128 agents [5].
While training on the aforementioned dataset, lock situations encountered by the actors are still detected and stored back in the dataset, this time conserving all agents. It is likely that some agents in a stored locking situation are already on their goals, in which case a randomly selected number of them will be assigned a new goal to conserve sample efficiency. Furthermore, every 100 epochs, a new batch of 500 locking situations is computed (like the batch that is computed before the fine-tuning) and added to the dataset in an effort to introduce some new maps into circulation.

3.3. Phantom obstacles

Instead of altering the training process, one could add obstacles only visible to a locked agent during application; in order to break up locking situations. These obstacles will henceforth be referred to as "phantom obstacles", as they are only meant to dissuade an agent from moving into a specific location, while it will not be registered as a collision if the agent does move there. Where to place a phantom obstacle relative to the locked agent and how long it should stay there depends on the specific lock subcategory (see 2.4) and its direct surroundings.
When an agent is in a collision-induced deadlock, if the agent's intended action would cause a conflict (i.e. excluding cases where agents would collide with the environment), the obstacle is placed in the location the agent intended to move to for one step. This phantom obstacle will never be placed on top of the agent's position, as an agent that chooses to wait will never be registered as entering a conflict. For agents in waiting deadlocks, DCC offers little insight into what a stationary agent's intentions are and why it is not moving. Therefore, it is necessary to look at the cells directly adjacent to the agent's position (i.e. directly north, east, south, or west). Define n_{obs} as the number of obstacles directly adjacent to the locked agent A_l and n_{agent} as the number of agents directly adjacent to A_l, then the following rules should be checked in order, with previous rules taking precedence over subsequent rules.

1. If $n_{obs} = 0$ and $n_{agent} = 0$, the situation is ambiguous and therefore ignored.
2. If $n_{obs} = 3$, no phantom obstacle is created, as there is either only one way and adding a phantom obstacle on top of a regular one has no effect.
3. If $n_{obs} + n_{agent} = 4$, no phantom obstacle is created, as A_l is surrounded on all sides.
4. If $n_{agent} = 1$, a phantom obstacle is created on top of the single adjacent agent with a lifespan of 2 steps, as it is the most likely source of confusion for A_l.

5. If $n_{agent} = 2$, determine whether A_l is part of a cluster. A cluster consists of at least two agents, where each agent is adjacent to at least one other agent in the cluster. Let $\vec{p}_{A_l}$ be the position of agent A_l, C a cluster such that $A_l \in C$ and $\hat{u}_{A_l,C}$ be the unit vector indicating the position of the centroid of C relative to p_{A_l}, then a phantom obstacle is placed at $\vec{p}_{PO} = \vec{p}_{A_l} + \vec{r}_{A_l,C}$, where $\vec{r}_{A_l,C}$ equals $\hat{u}_{A_l,C}$ rounded component-wise to the nearest integer. The obstacle's lifespan is set to 2. If $\vec{r}_{A_l,C} = [0, 0]$, no obstacle will be placed, otherwise, it could be placed in either one of the 4 directly adjacent locations or either of the second-order adjacent cells (i.e. half cardinals).
6. If $n_{agent} = 3$, place a phantom obstacle following the same process as the previous rule, but with the added condition that it cannot be placed in the only free directly adjacent cell. The obstacle's lifespan is set to 3.
7. If $n_{agent} = 0$ and $n_{obs} \in \{1, 2\}$, place a phantom obstacle in the cell in the agent A_l's observation radius that is furthest away from its goal (euclidean distance) and which is within the bounds of the environment and not and obstacle or phantom obstacle. Its lifespan is set to 5.

If an agent is short pattern livelocked, it suffices to place an obstacle in the next location of the pattern, on the condition that it is not the agent's goal position. Phantom obstacles for livelocks are kept for the duration of the pattern, which means two steps in the case of a short pattern.
In the case of a long pattern livelock, both the next and previous position in the pattern are taken into consideration. If either is further from the goal position, a phantom obstacle is placed there, otherwise, no obstacle is created.

4

Experiments

In order to assess the effectiveness of the methods described in chapter 3, each has been tested individually, as well as each unique combination of methods. The tests consisted of a quantitative benchmark and a qualitative assessment of the context and frequency of locks in a small set of maps.

4.1. Benchmark

During the benchmark, an array of different metrics are collected over a set of different maps. To ensure methods and combinations of methods can be compared fairly against one another, the maps are pre-generated and stored along with start and goal positions. For each combination of agents in table 4.1, 30 samples were generated with 0% and 30% obstacle density each, with randomly distributed obstacles and start and goal positions. It was empirically determined that going beyond 30% obstacle density led to the creation of an increasing number of small and narrow enclosed spaces that allow two or three agents to be initialised in them but without the possibility for them to reach their respective goals while obeying the rules layed out in section 2.1.

Table 4.1: Table showing the combinations of map size and numbers of agents used

		Agents						
		4	8	16	32	64	128	256
Size	20×20	y	y	y	y			
	40×40	y	y	y	y	y	y	
	80×80	y	y	y	y	y	y	y
	160×160	y	y	y	y	y	y	y

The metrics gathered were the three path length metrics (see section 2.2.1), the success-rate (see section 2.2.2, the computation, the number of locks, the number of agents that found their respective goals and the number of collisions.

4.2. Qualitative assessments

Assessments on the placement and frequency of locks could be made thanks to heatmaps showing these graphically. In order to obtain these heatmaps for a specific planner (i.e. baseline DCC or any combination of methods), the planner is deployed on a set of 30 maps with different, randomly distributed, start and goal positions, but the same obstacle layout. Throughout the run, a list of all locks and their coordinates is kept. A heatmap is then constructed by calculating the number of locks occurring in each cell and averaging it over the number of samples and mapping the resulting matrix to a color range. This process is repeated for 3 different randomly distributed obstacle layouts of 40×40 and 80×80 containing 64 and 128, and 128 and 256 respectively as well as for 2 sets of 40×40 without obstacles, containing 64 and 128 agents, respectively.

4.3. Ablation study

To analyse the effects of each method, each was benchmarked individually, before moving on to combinations of 2, 3 and, finally, all 4 methods. Naturally, the models to which targeted training (TT) or lock punishment (LP) were added needed to be trained from scratch before benchmarking, taking around 48 hours for each training run. When adding targeted fine-tuning (TF), the additional training time was about 3 hours per run. As phantom obstacles (PO) are only processed during deployment, the models where this method was added could directly be benchmarked.

4.3.1. Benchmark results

Before analysing the various methods, a solid baseline had to be established. A version of DCC was trained locally, henceforth referred to as the trained baseline, and benchmarked along with ODrM*[18] and a set of DCC weights, henceforth referred to as baseline DCC, which Ma et al. provided along with their codebase [28]. ODrM* is a state-of-the-art centralised planner that is often used for comparison against learning-based planners [3]. For the purpose of these experiments, it was given a suboptimality bound of 10 times the optimal solution SoC and was subject to the customary 60 second timeout limit (see section 2.2.2).

As can be seen in figure 4.1, the trained baseline keeps a higher success-rate than baseline DCC in environments devoid of obstacles, but has a tendency to drop of faster in environments with 30% obstacle density. All the while, neither is able to outperform ODrM*. When looking at figure 4.2, it is also revealed that ODrM* needs far less time to compute a solution than both baselines.

In order to allow a quick survey of the relative performance differences over the entire benchmark set, the means of the different performance metrics were gathered into table 4.2. On top of regular mean success-rates, the mean weighted average (MWA) is also presented. For each unique set of map size and obstacle density, (N, ρ) the average of the success-rates are weighed by the number of agents, thus emphasizing the effects of higher agent density on success-rate when contrasted with the regular mean.

Overall, baseline DCC outperforms the trained baseline on success-rate by an average of 1%. However, the former only drops by 11.1% when looking at the MWA success-rate, while the latter drops by 13.8%, suggesting a systematically lower performance in more agent-dense environments. However, the trained baseline has lower mean computation time and path length metrics, albeit higher than ODrM*. Although there are some significant differences between the baselines, repeated training produced comparable results, which would indicate that the training is stable and these differences may rather be due to altered parameters. Sadly, the repository provided by Ma et al. did not have documentation on the parameters used to produce the specific weights of baseline DCC, which is why the parameters were ensured to be identical to those specified in the paper [5] and kept as is everywhere else. For this reason, the trained baseline is used for all other comparisons.

Table 4.2: Comparison table between ODrM* with an inflation factor of 10, baseline DCC and the trained baseline. The rows are made up of the mean success-rate, the mean of the success-rates averaged weighted by number of agents, the mean number of locks per agent, the mean percentage of goals found, the mean percentage of goals found for all failed samples, the mean computation time per agent, the mean sum of costs per agent, the mean sum of fuel per agent, and the mean makespan divided by the respective number of agents

Planner	ODrM ($\epsilon = 10$)	baseline DCC	trained baseline
Mean success-rate ($\%$)	97.8	92.0	91.0
MWA success-rate ($\%$)	93.5	80.9	77.2
Mean locks/step	-	0.225	0.312
Mean goals found ($\%$)	-	99.6%	99.0%
Mean GF no success ($\%$)	-	95.3	91.8
Mean computation time/agent (s)	0.032	0.248	0.197
Mean SoC/agent	62.687	67.626	66.947
Mean SoF/agent	59.321	63.090	62.609
Mean makespan/agent	7.575	8.109	8.030

A glance at the success-rate plots in figure 4.3 will show that LP has difficulty reaching all its goals when it is deployed in environments dense with obstacles and agents. A closer look will reveal that PO is the only method that outperforms the trained baseline with some consistency.

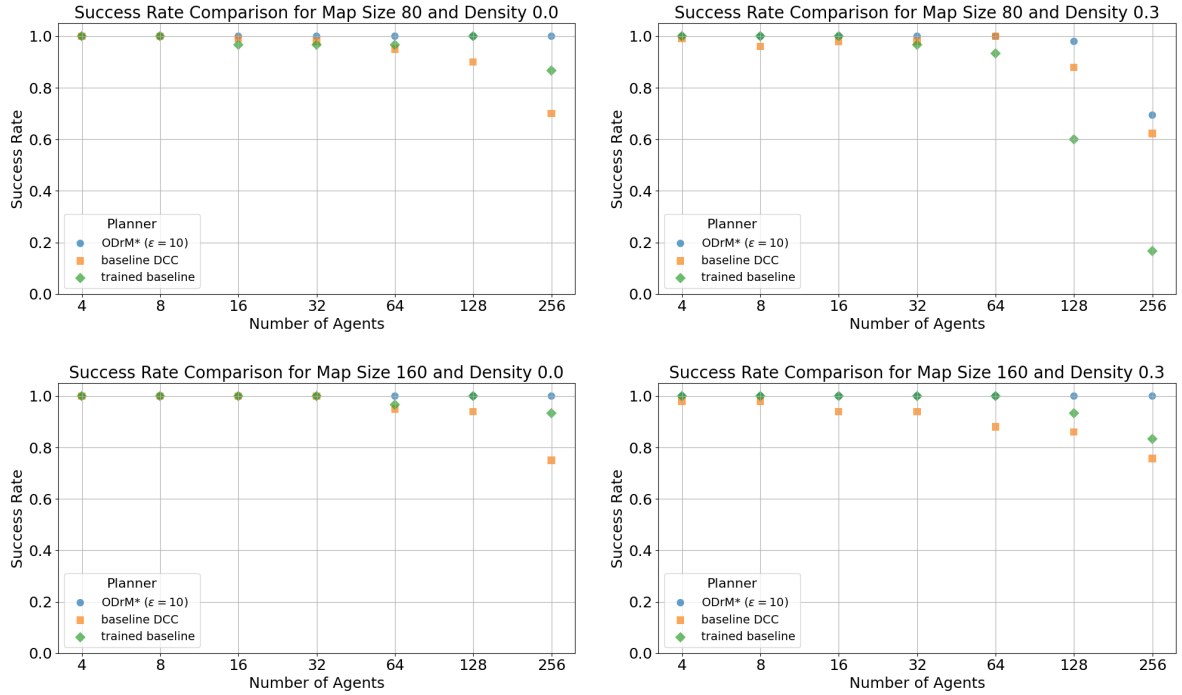


Figure 4.1: Plots comparing the success-rates of ODrM* with an inflation factor of 10, baseline DCC and trained baseline DCC

Now, taking a look at figures 4.4 and 4.5, it is evident that LP has among the lowest number of locks. This becomes even clearer in table 4.3, where it has the lowest mean number of locks per step. LP also presents the second highest path length metrics of the singular methods and the baseline, which is in contrast with TT, which has the lowest.
Furthermore, table 4.3 also shows that PO has the highest mean and MWA success-rates, as well as the lowest mean computation time per agent. PO is also first in terms of mean percentage of found goals, but loses out to both the trained baseline and TF when only failed samples are considered.

Table 4.3: Comparison table between all four singular methods and the trained baseline based on various metrics. The rows are made up of the mean success-rate, the mean of the success-rates averaged weighted by number of agents, the mean number of locks per agent, the mean percentage of goals found, the mean percentage of goals found for all failed samples, the mean computation time per agent, the mean sum of costs per agent, the mean sum of fuel per agent, and the mean makespan divided by the respective number of agents

Planner	**trained baseline**	**TT**	**LP**	**TF**	**PO**
Mean success-rate (%)	91.0	88.5	82.4	84.9	**92.6**
MWA success-rate (%)	77.2	72.0	62.9	69.5	**80.2**
Mean locks/step	0.312	0.914	**0.137**	0.419	0.234
Mean goals found (%)	99.0	98.0	97.9	98.8	**99.2**
Mean GF no success (%)	91.8	85.8	89.9	**93.3**	91.5
Mean computation time/agent (s)	0.197	0.382	0.227	0.220	**0.191**
Mean SoC/agent	66.947	**66.532**	69.117	70.541	66.909
Mean SoF/agent	62.609	**61.896**	65.434	65.849	62.571
Mean makespan/agent	8.030	**8.017**	8.553	8.881	8.020

Among the double methods (see table 4.4), none outperform their corresponding singular methods across the board, but they do manage to improve specific metrics. LP+PO improves the lowest number of locks per step, and TT+PO does the same for all three path length metrics. Furthermore, PO increases the success-rate of all three methods it is added to. There is only one case where adding TF improves a metric, and it is the computation time of TT+TF compared to TT.

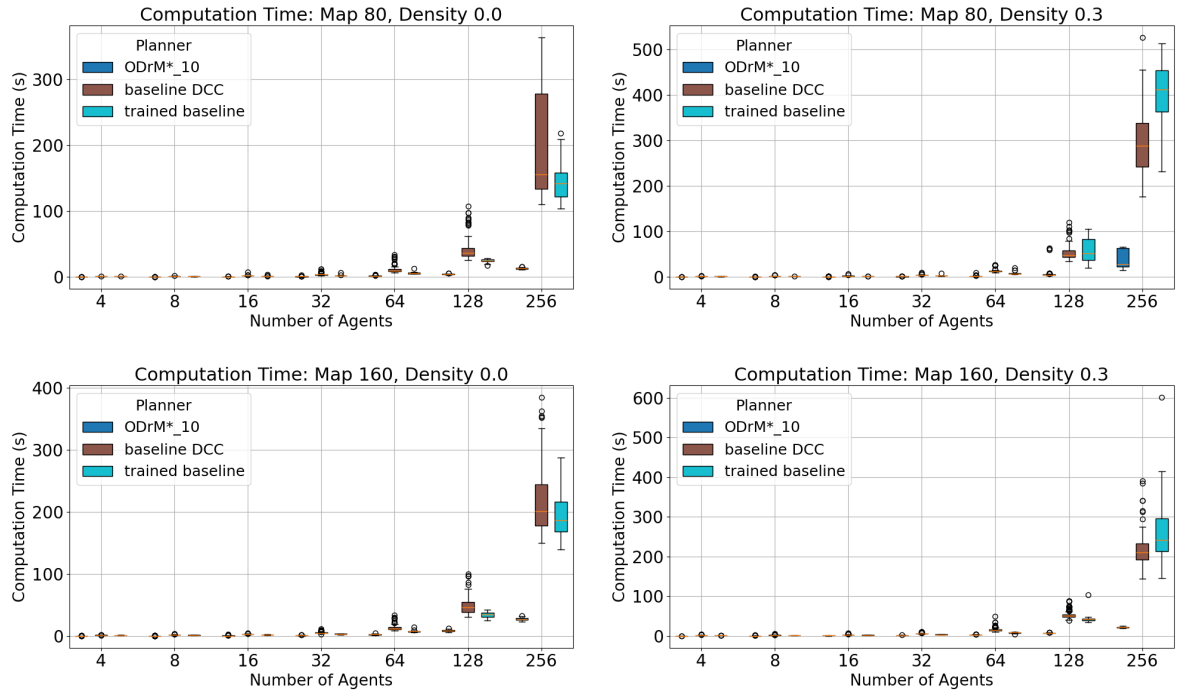


Figure 4.2: Plots comparing the computation times of ODrM* with an inflation factor of 10, baseline DCC and trained baseline DCC

Table 4.4: Comparison between the six double methods based on various metrics. The rows are made up of the mean success-rate, the mean of the success-rates averaged weighted by number of agents, the mean number of locks per agent, the mean percentage of goals found, the mean percentage of goals found for all failed samples, the mean computation time per agent, the mean sum of costs per agent, the mean sum of fuel per agent, and the mean makespan divided by the respective number of agents

Planner	**LP+TT**	**LP+TF**	**LP+PO**	**TT+TF**	**TT+PO**	**TF+PO**
Mean success-rate (%)	83.5	55.6	82.7	50.0	**89.7**	89.1
MWA success-rate (%)	64.2	32.6	63.8	21.9	73.9	**75.5**
Mean locks/step	0.555	0.699	**0.106**	1.406	0.937	0.248
Mean goals found (%)	97.8	94.7	97.9	94.4	98.1	**99.1**
Mean GF no success (%)	88.3	88.1	89.7	91.3	84.6	**92.4**
Mean CT/agent (s)	0.309	0.326	0.228	0.345	**0.204**	0.343
Mean SoC/agent	67.867	88.917	68.129	109.565	**66.385**	69.984
Mean SoF/agent	63.433	78.526	64.478	104.189	**61.841**	65.389
Mean Makespan/agent	8.286	13.519	8.458	14.274	**7.861**	8.592

None of the triple or quadruple methods provides absolute improvements of any performance metric, but an interesting detail is that the average number of goals found in failed samples for each method in table 4.5 is still relatively high, not dropping below 86.6% (LP+TF+PO). This is higher than the absolute lowest and second lowest set by TT+PO (84.6%) and TT (85.8%) respectively. On another note, each combination of methods to which PO is added, once again, sees its success-rate increase to varying degrees.

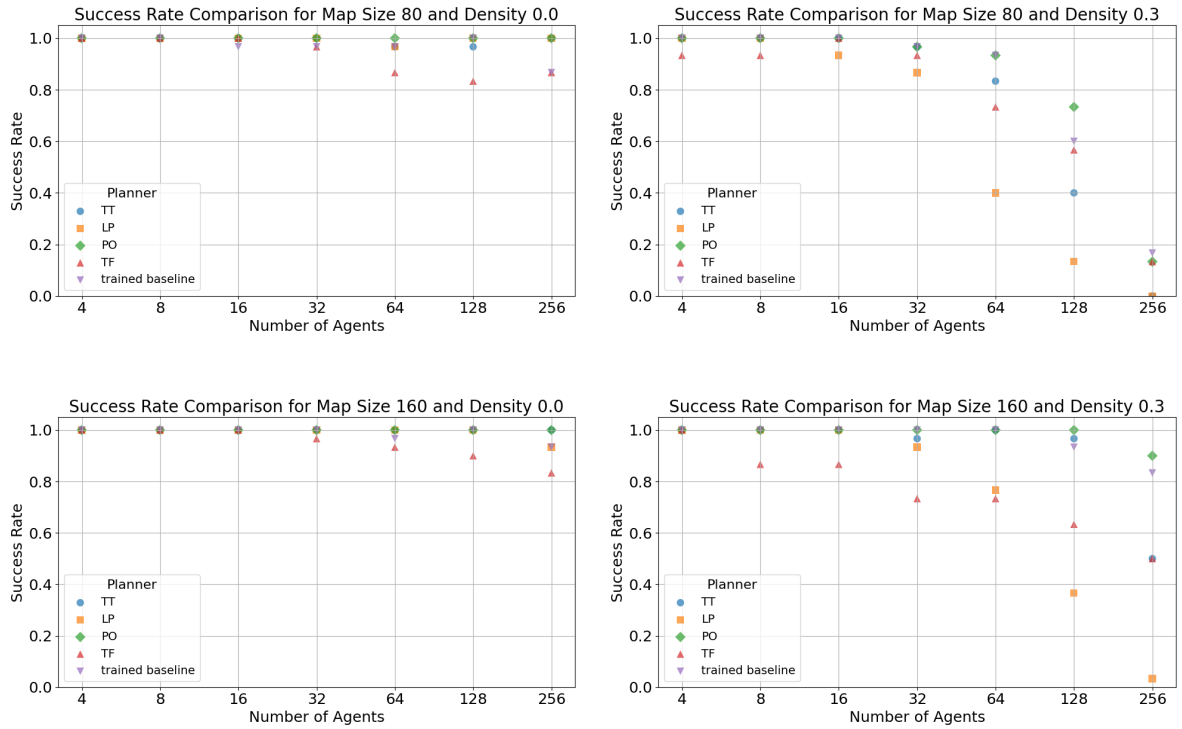


Figure 4.3: Plots comparing the success-rates of trained baseline DCC to that of Lock Punishments (LP), Targeted Training (TT), Targeted Fine-tuning (TF) and Phantom Obstacles (PO)

Table 4.5: Comparison between the four triple methods and the single quadruple method based on various metrics. The rows are made up of the mean success-rate, the mean of the success-rates averaged weighted by number of agents, the mean number of locks per agent, the mean percentage of goals found, the mean percentage of goals found for all failed samples, the mean computation time per agent, the mean sum of costs per agent, the mean sum of fuel per agent, and the mean makespan divided by the respective number of agents

Planner	LP+TT+TF	LP+TT+PO	LP+TF+PO	TT+TF+PO	LP+TT+TF+PO
Mean success-rate (%)	47.3	**84.0**	64.0	55.2	51.4
MWA success-rate (%)	29.2	**64.8**	45.7	26.3	32.4
Mean locks/step	1.183	0.510	**0.378**	1.013	0.963
Mean goals found (%)	94.9	**97.7**	95.2	95.0	95.3
Mean GF no success (%)	90.5	87.9	86.6	**91.9**	90.5
Mean CT/agent (s)	0.294	**0.224**	0.307	0.339	0.288
Mean SoC/agent	76.282	**67.557**	87.550	107.808	75.110
Mean SoF/agent	71.890	**63.055**	77.333	102.386	70.788
Mean Makespan/agent	14.922	**8.250**	13.027	13.708	14.441

4.3.2. Discussion

It turns out that each individual method outperforms all others in at least one metric: LP experiences the fewest locks, TT produces the shortest paths, TF achieves the highest rate of found goals when it fails to find all, and PO has the highest success-rates and lowest computation times. It would therefore not be unreasonable to assume that combining two or more methods would yield a well-rounded solver. However, in practice, this only resulted in two solvers that could rightly be considered to be better than the sum of their parts: TT+PO and LP+PO.

Adding PO to TT yielded even lower path length metrics, improved upon TT's mean success-rate by 1.2% (4.2% MWA success-rate) and cut down computation times by $46,6\%$. The resulting SoC, SoF and makespan improved by 0.8%, 1.2% and 2.1%, respectively, compared to the trained baseline, making TT+PO the first choice when solution quality is the primary concern (provided a decentralised learning-

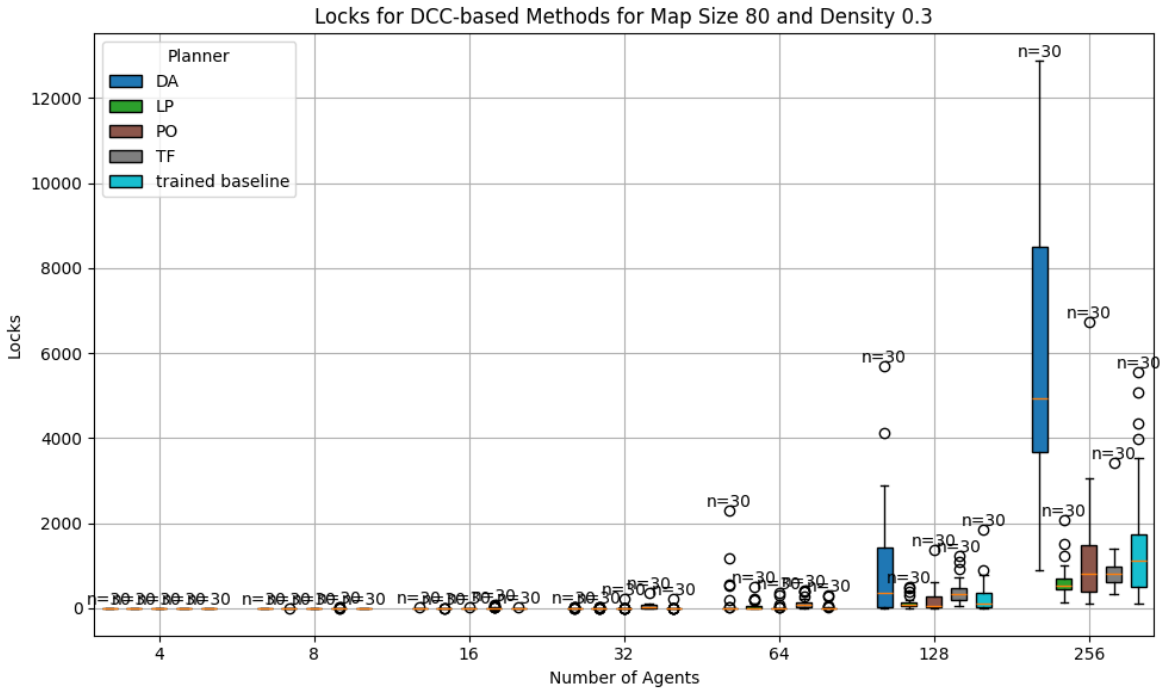


Figure 4.4: Plot comparing the number of locks encountered by the trained baseline, Lock Punishments (LP), Targeted Training (TT), Targeted Fine-tuning (TF) and Phantom Obstacles (PO) in environments of size 80

based planner is required).

The case of LP, and therefore also LP+PO, is slightly more complicated. The intuitions about the inverse relation between the number of locks and the success-rate established in section 2.4 state that higher numbers of locks are an indicator of lower success-rate. However, LP's and LP+PO's relatively low success-rates and low number of locks seem to contradict this notion. This suggests that the punishments did, in fact, discourage the planner from sending agents into locking situations, but also, at the same time, sending them wandering aimlessly around the environment. This is supported by the fact that the path length metrics for both LP and LP+PO are relatively high compared to that of the trained baseline. A likely cause is that the punishments were too high compared to the rest of the rewards structure. This does not take away from the fact that LP+PO has the lowest number of locks, improving from the trained baseline by 66% and 22.6% compared to regular LP. Adding PO also increased LP's mean success-rate by 0.1% (0.9% MWA success-rate). This means that LP+PO is the best choice in the unlikely event that the number of locks is far more important than success-rate and path length.

Throughout all results, it is clear that PO is quite effective in improving success-rates, path length metrics, number of locks, as well as computation times (in most cases) for the various planners to which it is applied. Seen as the effectiveness of PO largely depends on the behaviour of the solver to which it is applied, it could be expected that the highest success-rate would be achieved by applying it to a solver that already had a high success-rate before application of PO. It is therefore no surprise that singular PO, applied to the trained baseline, has the highest mean and MWA success-rate, improving over the solver in question by 1.6% and 3%, respectively. Furthermore, it has the lowest mean computation time (3% lower than baseline) and is relatively close the best tested path length metrics (third after TT and TT+PO). It can therefore be argued that PO is the best all-round add-on to DCC, especially since success-rate is often considered to be the most important of these metrics. Added to this is the fact that PO requires the least time and resources to execute as their is no specific training involved and the amount of computational overhead that is needed to compute the phantom obstacles is evidently not so large that the computation time suffers under it.

TF can be considered to be the most mediocre among the four proposed methods. Its mean success-rate is not particularly high, and most methods that go through the fine-tuning process come out worse on all metrics. A possible cause for this could be the onset of catastrophic forgetting [29], which occurs when the training samples shift far enough from the distribution of earlier samples that the model

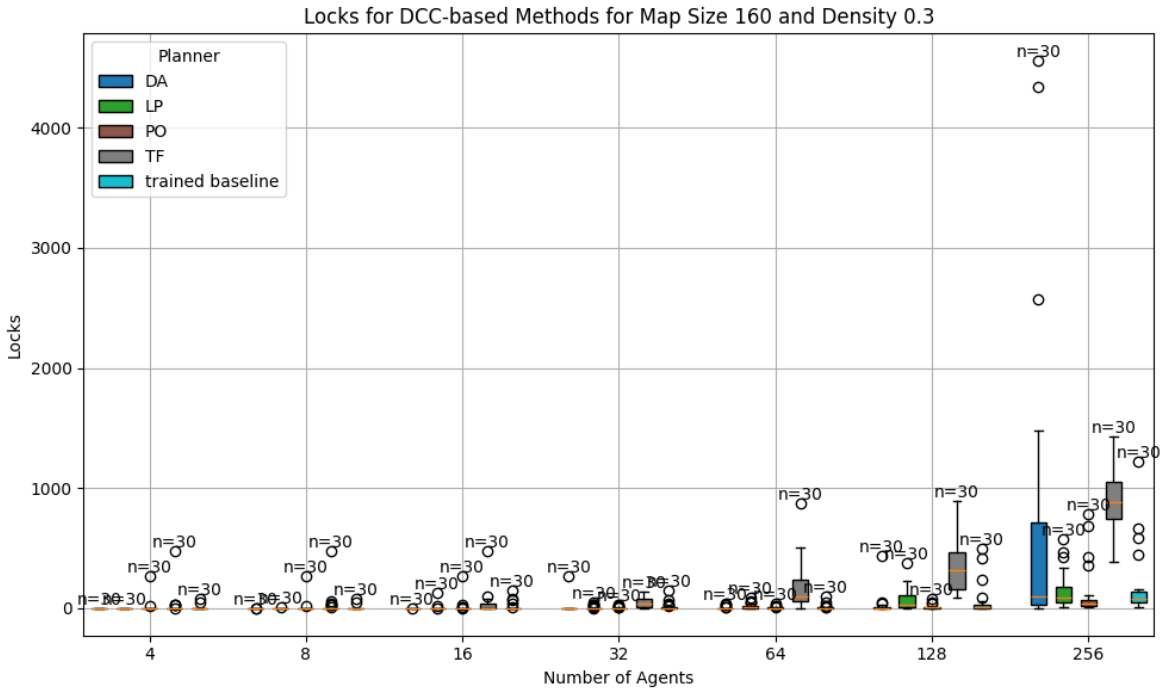


Figure 4.5: Plot comparing the number of locks encountered by the trained baseline, Lock Punishments (LP), Targeted Training (TT), Targeted Fine-tuning (TF) and Phantom Obstacles (PO) in environments of size 160

effectively ”forgets” how to complete the tasks associated with those earlier samples. Seen as during fine-tuning, the pre-trained model will receive a large number of samples that have quantities of agents up to 32, while the curriculum only goes up to 16.
As noted earlier, the number of agents that do not reach their goals in time remain fairly low, even when mean success-rates drop close to below 50%: when LP+TT+TF fails, it still manages to guide an average of 90.5% of the agents to their goals. This effect is similar to what Damani et al. remarked in their paper on PRIMAL$_2$ [2], where they equally noticed that it was a relatively small number of agents that kept the success-rates down. It is fair to assume that in the case of TT+TF, a large number of them are locked (see table 4.4), while in the case of LP+PO, this cannot be the case.

Finally, when ODrM* is added to the comparison, it is evident that DCC and all methods and combinations thereof cannot compete with it in terms of any metric, in particular computation time. However, each benchmark sample was run on a single CPU, but DCC is a decentralised planner and should, in theory, be able to make use of parallel architecture to run faster. The current implementation of DCC cannot make optimal use of multiple CPU’s, but it could likely be adapted to do so and, thus the absolute computation time would be reduced. ODrM* would be more complex to parallelise efficiently and would therefore not see a proportional decrease in computation time.

4.4. Heatmap results

Heatmaps were generated for the trained baseline and for the four singular methods, all five using the same color scale.
When comparing the trained baseline and PO in figures 4.6 and 4.7, it is clear that both encounter locks in mostly the same cells. This behaviour is to be expected, as the network is exactly the same and the only thing that changes is how the environment reacts to an agent entering into a locked status. Note also how PO attenuates some of the hotspots in the heatmap of the trained baseline, reflecting the data from figures 4.4 4.5 and table 4.3.
The contrasting behaviours of TT and TF between figure 4.6 and figure 4.7 are also striking: in the empty map, TF causes the most locks and TT only sees relatively few, while the roles are reversed in

the environment with obstacles. Do note that TF's heatmap in 4.6 shows a peculiar pattern made up of a clear, two cell wide, band along the left of the map, containing the most intense hotspots. This observation contradicts the expectation that most locks would occur nearer to the centre of the map. The random distribution of start and goal positions would make it unlikely that such a large number of agents would have to move along the left wall as to cause so many locks.

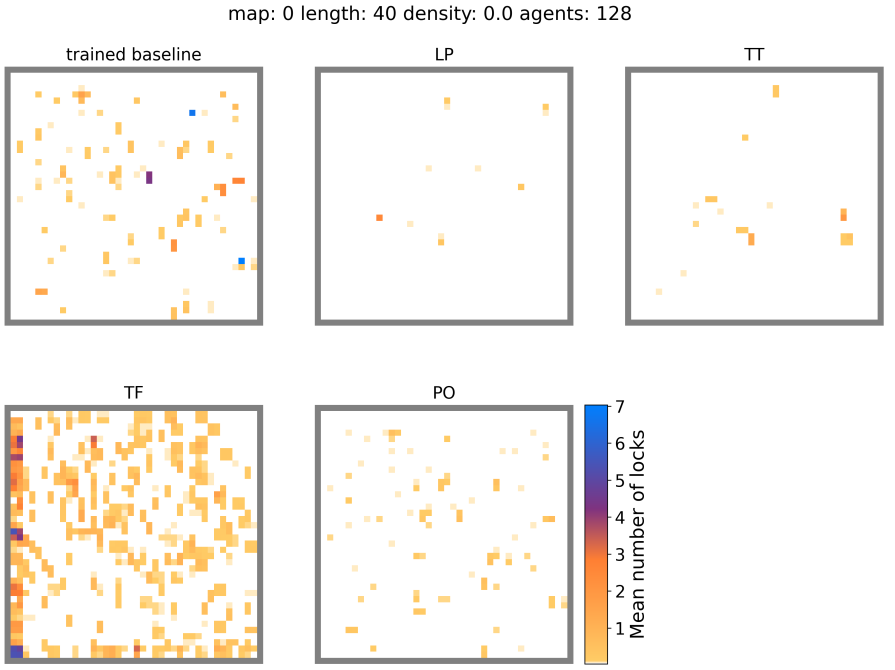


Figure 4.6: Lock heatmaps of the trained baseline and all 4 singular methods in an empty 40×40 environment with 128 agents

In the larger maps (see figures 4.8, 4.9, 4.10, and 4.11), one can see that most locks do occur closer to the centre. It especially clear that LP, PO and the trained baseline all share some locations where larger hotspots are formed. TT's heatmaps also show hotspots in those locations, but their sizes and intensity are often exaggerated in comparison. These hotspots are often located around breaks in relatively long uninterrupted walls or in long, mostly enclosed corridors. In fact, these are the exact locations where locks are expected to be more frequent, as those features are likely to act as funnels.
TF does have some hotspots in similar locations as the other four, but they are smaller and do not match as closely. Furthermore, doubling the number of agents from figure 4.8 to figure 4.9 seems to have concentrated some of the small scattered spots to a few slightly larger spots in the latter figure. This contrasts with how the spots created by the other solvers all seem to stay in place and merely grow in size and intensity.

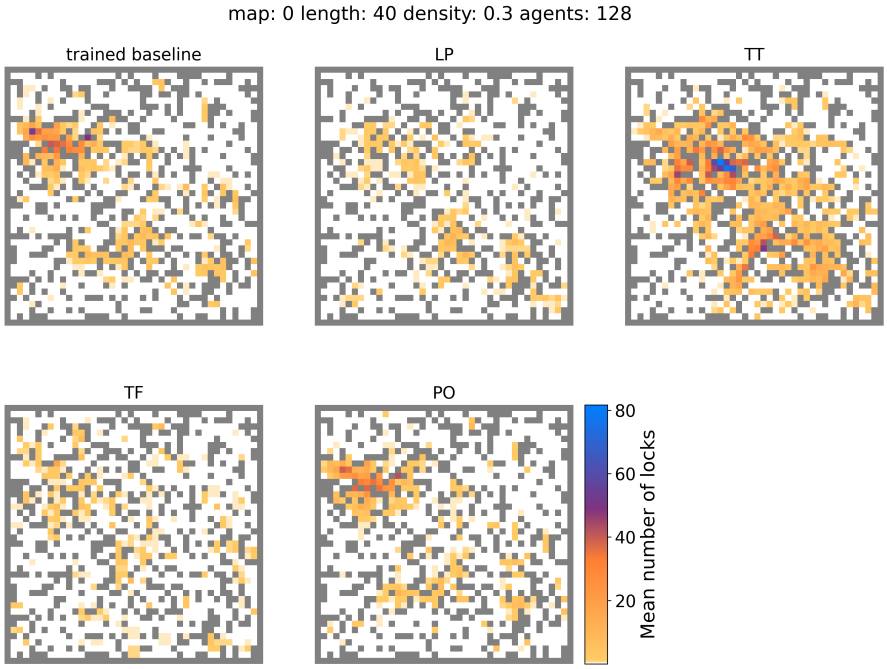


Figure 4.7: Lock heatmaps of the trained baseline and all 4 singular methods in 40×40 with 30% environment with 128 agents

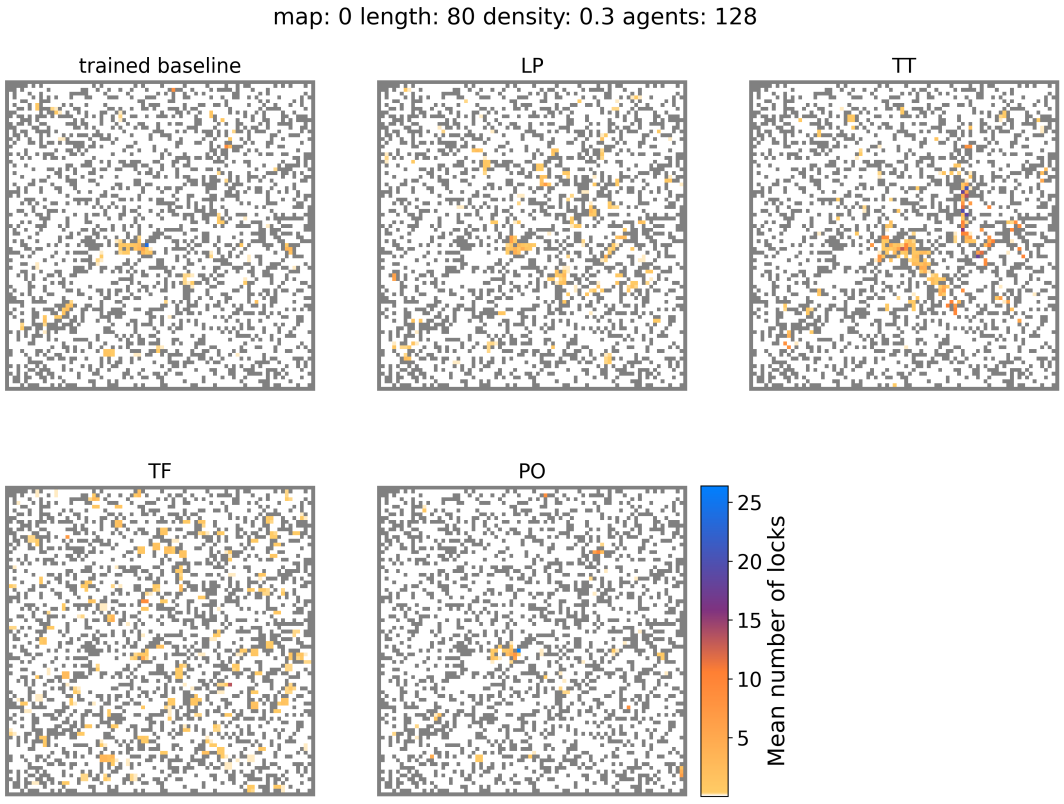


Figure 4.8: Lock heatmaps of the trained baseline and all 4 singular methods in 80×80 with 30% environment with 128 agents

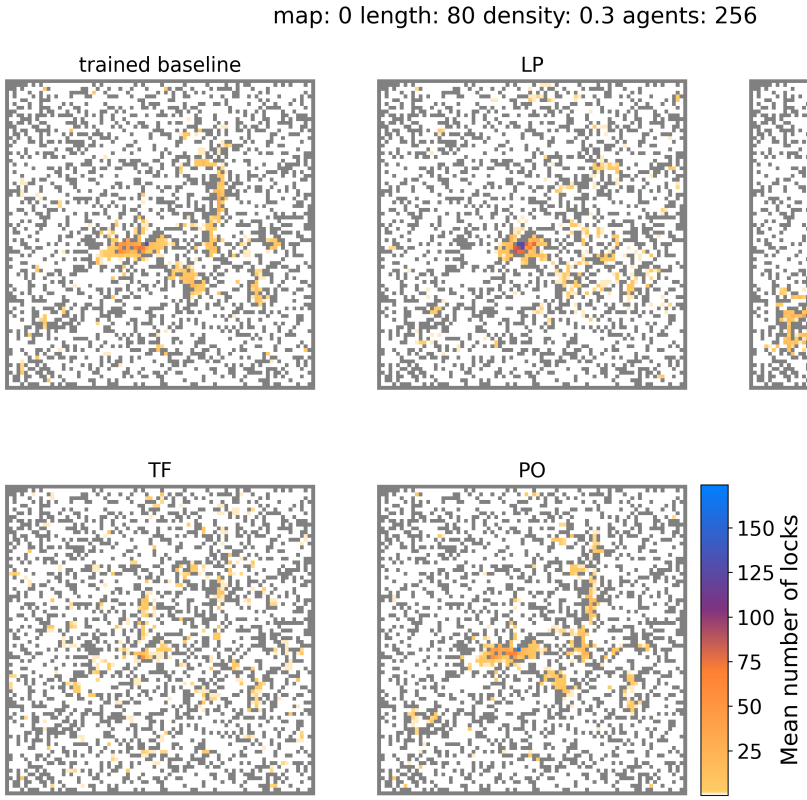


Figure 4.9: Lock heatmaps of the trained baseline and all 4 singular methods in 80×80 with 30% environment with 256 agents

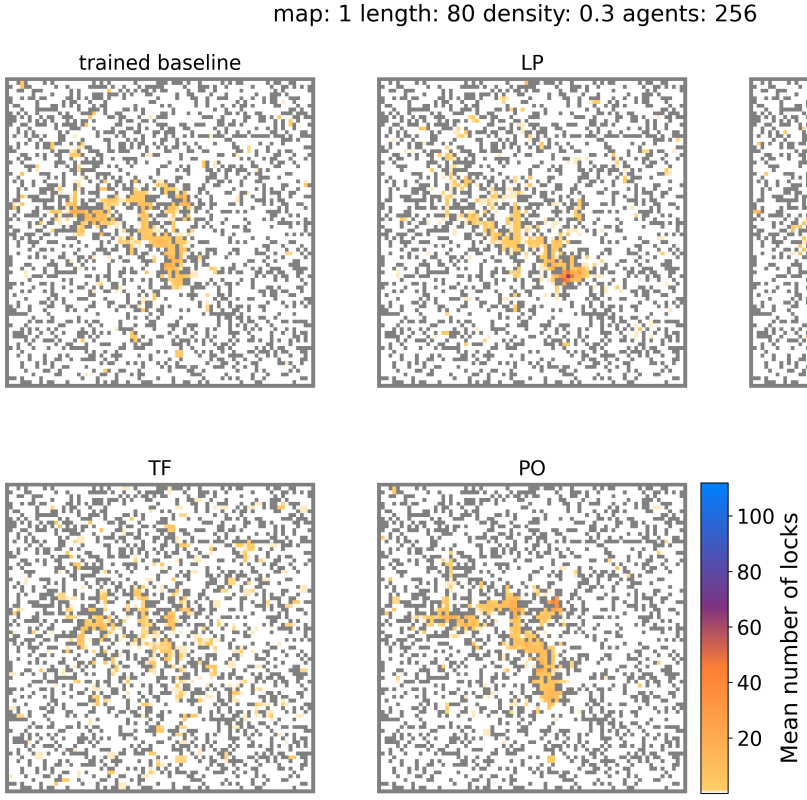


Figure 4.10: Lock heatmaps of the trained baseline and all 4 singular methods in 80×80 with 30% environment with 256 agents

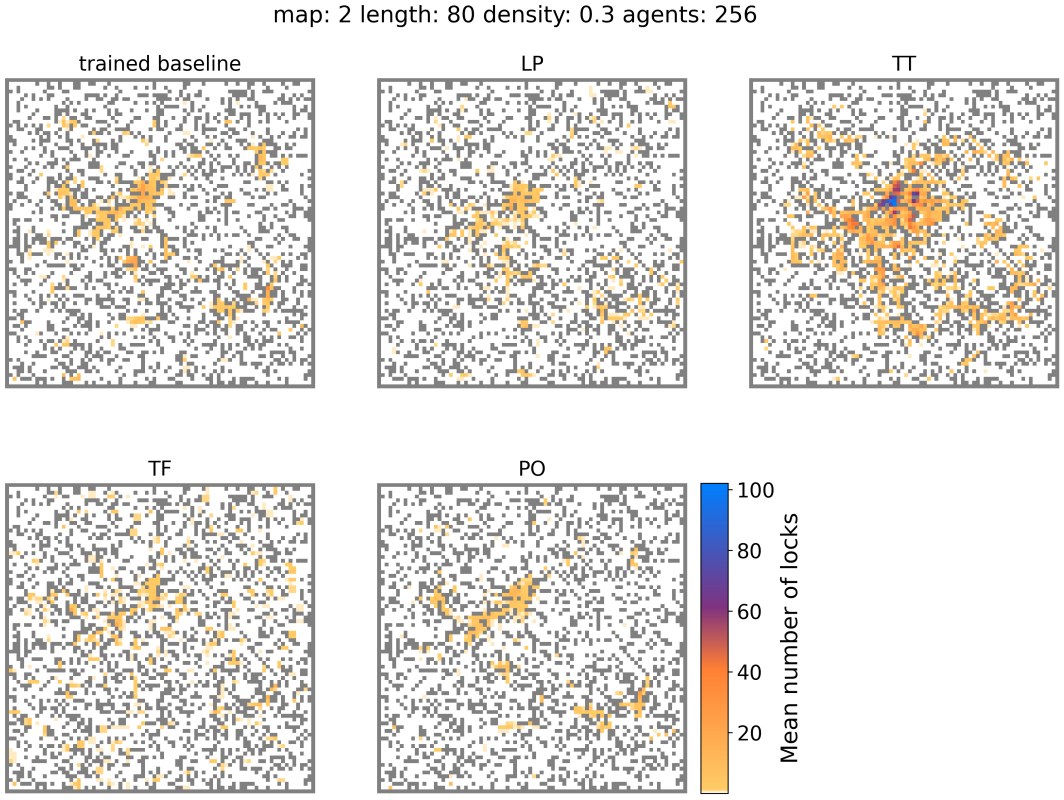


Figure 4.11: Lock heatmaps of the trained baseline and all 4 singular methods in 80×80 with 30% environment with 256 agents

5

Conclusion and Recommendations

The aim of this thesis was to present simple, drop-in methods that reduce the tendency of decentralised learning-based solvers to get caught in a deadlock or a livelock. The two most effective methods that have been proposed and evaluated are Phantom Obstacles (PO) and Targeted Training + Phantom Obstacles (TT+PO). The former has the highest success-rate and lowest computation time of all evaluated methods despite adding a small amount of computational overhead. Furthermore, it is the simplest method to implement as it doesn't require any training time beyond what was needed for the model it is applied to. The second method, TT+PO, produces the shortest paths by all three metrics discussed in section 2.2.1. Both outperform the trained baseline in these categories (see table 5.1)
Although Lock Punishments (LP) and its improved variant, LP+PO, are very effective at avoiding locks, it sacrifices a significant amount of success-rate compared to the aforementioned methods. This contradicts the previous assumption that purely reducing the number of locks will improve overall performance.

Table 5.1: Table comparing metrics between ODrM* with inflation factor of 10, the trained baseline, Phantom Obstacles (PO), Targeted Training + Phantom Obstacles (TT+PO) and Lock Punishments + Phantom Obstacles (LP +PO)

	ODrM* ($\epsilon = 10$)	trained baseline	PO	TT+PO	LP+PO
Mean success-rate (%)	97.8	91.0	**92.6**	89.7	82.7
MWA success-rate (%)	93.5	77.2	**80.2**	73.9	63.8
Mean locks/step	-	0.312	0.234	0.937	**0.106**
Mean computation time/agent (s)	0.032	0.197	**0.191**	0.204	0.228
Mean SoC/agent	62.687	66.947	66.909	**66.385**	68.129
Mean SoF/agent	59.321	62.609	62.571	**61.841**	64.478
Mean makespan/agent	7.575	8.030	8.020	**7.861**	8.478

For future research, it would be valuable to determine whether the methods proposed here indeed function as drop-in modules by adding them onto other similar solvers, such as PRIMAL [3] or MAPPER [20]. It is expected that LP would make less of an impact on the number of locks encountered by PRIMAL, as it is trained only partially through reinforcement learning. Applying the the punishments to the imitation portion of training may require some more significant adaptations. The effects of DA and TF are also likely to be affected by the PRIMAL's hybrid approach to training. In light of adapting DCC to parallelise its application on several CPUs or even separate machines for each agent, PO would likely have to undergo significant changes. Currently the phantom obstacles are administered by a central process. Furthermore, additional carefull tuning of PO's parameters (obstacle lifetime, treated cases, etc.) may yield an even more effective method and different planners might requires different profiles for optimal improvements. Further research into the reward structure of LP may just yield a very effective planner, seen as it is able to reduce the number of locks by so much. Finally, looking deeper into catastrophic forgetting and techniques to avoid it could result in insightful conclusions about how to reiterate the design of TF.

References

[1] Rishi Veerapaneni et al. “Windowed MAPF with Completeness Guarantees”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Apr. 2025. DOI: 10.1609. URL: https://doi.org/10.1609/aaai.v39i22.34499.

[2] Mehul Damani et al. “PRIMAL2: Pathfinding via Reinforcement and Imitation Multi-Agent Learning – Lifelong”. In: (Oct. 2020). DOI: 10.1109/LRA.2021.3062803. URL: http://arxiv.org/abs/2010.08184%20http://dx.doi.org/10.1109/LRA.2021.3062803.

[3] Guillaume Sartoretti et al. “PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning”. In: *IEEE Robotics and Automation Letters* 4.3 (July 2019), pp. 2378–2385. ISSN: 23773766. DOI: 10.1109/LRA.2019.2903261.

[4] Yutong Wang et al. “SCRIMP: Scalable Communication for Reinforcement- and Imitation-Learning-Based Multi-Agent Pathfinding”. In: (Mar. 2023). URL: http://arxiv.org/abs/2303.00605.

[5] Ziyuan Ma, Yudong Luo, and Jia Pan. “Learning Selective Communication for Multi-Agent Path Finding”. In: (Sept. 2021). URL: http://arxiv.org/abs/2109.05413.

[6] Chengyang He et al. “Social Behavior as a Key to Learning-based Multi-Agent Pathfinding Dilemmas”. In: (Aug. 2024). URL: http://arxiv.org/abs/2408.03063.

[7] Roni Stern et al. “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks”. In: *Proceedings of the International Symposium on Combinatorial Search* 10.1 (Sept. 2021), pp. 151–158. ISSN: 2832-9163. DOI: 10.1609/socs.v10i1.18510. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18510.

[8] Peter Hart, Nils Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136. URL: http://ieeexplore.ieee.org/document/4082128/.

[9] Stuart J. Russell and Peter Norvig. “Artificial Intelligence, A Modern Approach”. In: ed. by Peter Norvig and Stuart J. Russell. 2nd Edition. Prentice Hall, 2003. Chap. 4.

[10] Guni Sharon et al. “Conflict-based search for optimal multi-agent pathfinding”. In: *Artificial Intelligence* 219 (Feb. 2015), pp. 40–66. ISSN: 00043702. DOI: 10.1016/j.artint.2014.11.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S0004370214001386.

[11] Max Barer et al. “Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem”. In: *Symposium on Combinatorial Search*. 2014. URL: www.aaai.org.

[12] Eli Boyarski et al. “ICBS: The Improved Conflict-based Search algorithm for Multi-Agent Pathfinding: Extended Abstract”. In: *Proceedings of the Eighth International Symposium on Combinatorial Search*. 2015. URL: www.aaai.org.

[13] Ariel Felner et al. “Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 28. June 2018, pp. 83–87. DOI: 10.1609/icaps.v28i1.13883. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/13883.

[14] Jiaoyang Li et al. *Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search*. Tech. rep. 2019.

[15] Jiaoyang Li et al. “Symmetry-Breaking Constraints for Grid-Based Multi-Agent Path Finding”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 6087–6095. ISSN: 2374-3468. DOI: 10.1609/aaai.v33i01.33016087. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4565.

[16] Jiaoyang Li et al. “New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding”. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 30 (June 2020), pp. 193–201. ISSN: 2334-0843. DOI: 10.1609/icaps.v30i1.6661. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/6661.

[17] Liron Cohen, Tansel Uras, and Sven Koenig. “Feasibility Study: Using Highways for Bounded-Suboptimal Multi-Agent Path Finding”. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 6. 1. Sept. 2021, pp. 2–8. DOI: 10.1609/socs.v6i1.18363. URL: https://ojs.aaai.org/index.php/SOCS/article/view/18363.

[18] Glenn Wagner and Howie Choset. “Subdimensional expansion for multirobot path planning”. In: *Artificial Intelligence* 219 (Feb. 2015), pp. 1–24. ISSN: 00043702. DOI: 10.1016/j.artint.2014.11.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0004370214001271.

[19] Roni Stern. “Multi-Agent Path Finding – An Overview”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11866 LNAI. Springer, 2019, pp. 96–115. DOI: 10.1007/978-3-030-33274-7{_}6. URL: http://link.springer.com/10.1007/978-3-030-33274-7_6.

[20] Zuxin Liu et al. “MAPPER: Multi-Agent Path Planning with Evolutionary Reinforcement Learning in Mixed Dynamic Environments”. In: (July 2020). URL: http://arxiv.org/abs/2007.15724.

[21] Ziyuan Ma, Yudong Luo, and Hang Ma. “Distributed Heuristic Multi-Agent Path Finding with Communication”. In: (June 2021). URL: http://arxiv.org/abs/2106.11365.

[22] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: (June 2014). URL: http://arxiv.org/abs/1406.1078.

[23] Junyoung Chung et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. In: (Dec. 2014). URL: http://arxiv.org/abs/1412.3555.

[24] Ziyu Wang et al. “Dueling Network Architectures for Deep Reinforcement Learning Hado van Hasselt”. In: *Proceedings of The 33rd International Conference on Machine Learning*. New York: PMLR, 2016.

[25] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging {AI} Applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Oct. 2018, pp. 561–577.

[26] Richard S Sutton. “Learning to Predict by the Methods of Temporal Differences”. In: *Machine Learning* 3 (1988).

[27] Stéphane Ross, Geoffrey J Gordon, and J Andrew Bagnell. “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR, 2011, pp. 627–635.

[28] Ziyuan Ma. *DCC*. Dec. 2021. URL: https://github.com/ZiyuanMa/DCC.

[29] Michael McCloskey and Neal J. Cohen. “Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem”. In: 1989, pp. 109–165. DOI: 10.1016/S0079-7421(08)60536-8. URL: https://linkinghub.elsevier.com/retrieve/pii/S0079742108605368.