Applications of Search-based Methods and Reinforcement Learning in 3D GridWorld Construction

Thanh Nguyen **Ryan Cooper** **Truong-Giang Pham** **Weiguo Jiang**
Juan F. Lezama **Huy Ta**
Department of Computing Science, University of Alberta
{thanh2,rdc,truonggi,weiguo,jfernan1,htta}@ualberta.ca

Abstract

Plans for large-scale construction projects itemize construction into sequential steps to be executed by the assembly crews. The order of construction must also account for scaffolding to allow above-the-ground access to the workers. Due to the complexity of the desired structures and their prerequisite scaffolding, producing these building sequences relies heavily on professional construction planners. Hand-designed solutions, the current state of the art, are expensive and can be slow to adapt to the changing nature of a construction zone. This work presents Reinforcement Learning (RL) and planning methods to find a sequence of instructions in a simple 3D Grid World construction task. We perform a brief survey of potential techniques, and show that Deep Q-Network can achieve promising solutions in building predefined structures with and without scaffolding. In addition, we show A^* based methods can achieve optimality in the scaffoldless environment.

1 Introduction

The construction industry has traditionally depended on manual planning by professional construction planners. Planning the step-by-step sequences to construct smaller substructures such as columns or beams, to larger, more complex structures such as houses and buildings, can take months. With only a blueprint as reference, a planner must carefully consider the scaffolding necessary to support worker movement and construction. Reinforcement Learning and Search-based methods are typical methodologies to automate construction plan generation in hope to reduce the time and labor spent by professional planners.

In this paper, we explore various methods, from Reinforcement Learning (RL) to Search-based approaches, to generate comprehensive plans for predefined structures with various levels of success. With RL, we show that the agent trained using deep Q learning and its variants show promising results in generating construction plans for predefined structures on the $4 \times 4 \times 4$ and $6 \times 6 \times 6$ gridworld with and without scaffolding. On search-based methods, we show that they generate optimal and near optimal plans without any overhead of training time. However, the scalability of search-based methods concerning the size of the grid can be an issue as the search time grows exponentially.

2 Background

2.1 Reinforcement Learning

Agent and Environment The interaction between the agent and the environment in a Reinforcement Learning (RL) can be formulated as a *Markov Decision Process* (MDP). At any discrete time step t = 0, 1, 2, 3, ... the agent receives observation S_t from the environment. It then takes action A_t, receiving from the environment the next observation S_{t+1} and reward signal R_{t+1}.

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

The agent chooses actions based on policy distribution π, and we define the return from state S_t as $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ where $\gamma \in [0, 1]$ is the discounting parameter. Following π, the value function is defined as $v_\pi(s) = E_\pi[G_t|S_t = s]$, and the action-value function is defined as $q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a]$. The policy π can be parameterised and learned directly, as in Actor-Critic-based methods (Sutton and Barto, 2018), or it can be derived from the agent acting ϵ-greedy with respect to the action-value function as in Q-learning.

Deep Q-Network In this approach, we indirectly obtain policy π by behaving ϵ-greedy with respect to the *q-values* as defined above. We parameterise $Q(s, a, \theta)$ where θ is a neural network (Mnih et al., 2013). At every time-step, the agent has a probability of $1 - \epsilon$ to choose the action that yields the highest Q value, or an ϵ probability of choosing a random action. The resulting transition tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ is then added to a replay buffer, as is standard in prior work (Mnih et al., 2013). θ is optimised by stochastic gradient descend to minimise $\delta = R_{t+1} + \gamma \max_a Q(S_{t+1}, a, \bar{\theta}) - Q(S_t, A_t, \theta)$. The replay buffer can either sample the training transitions uniformly (Mnih et al., 2013), or can prioritise sampling transitions with larger $|\delta|$ (as in Prioritise Replay Buffer (Schaul et al., 2015)). A pseudo-code for DQN, and the different replay buffers are provided in Appendix D.

Advantage Actor-Critic Methods Instead of deriving the policy from learned Q function, we can instead parameterise $\pi(a|s, \theta)$ (actor) and $v(s, w)$ (critic). The actor is trained by gradient ascent $\nabla_\theta \log \pi(a|s, \theta)(R_{t+1} + \gamma v(S_{t+1}, w) - v(S_t, w))$; while the critic is trained by gradient descending $(R_{t+1} + \gamma v(S_{t+1}, w) - v(S_t, w))\nabla_w v(S_t, w)$. Proximal Policy Optimisation (PPO) (Schulman et al., 2017) is an add-on to advantage actor-critic, further improving sampling efficiency by updating π multiple times for each episode rollout, while at the same time clipping the update magnitude to make sure π does not change too drastically, leading to instability.

2.2 Search Methods

$\boldsymbol{A}^*$ is an optimal search algorithm that is an extension on top of Dijkstra's algorithm which adds the use of a heuristic to guide the search. A slight modification of A^* where the nodes are expanded are the smallest heuristic cost is Greedy best-first search (GBFS). The benefit of (GBFS) is shorter search time though the solutions are not guaranteed to be optimal and directly rely on how well the heuristic represents the problem.

Planning Domain Definition Language(PDDL) is a language specifically design to specify planning problems. A PDDL problem description consists of two components: the domain which describes the environment including the actions, predicates and functions (used to describe numerical values), and the problem file which is used to specify the initial state and the goal state.

Fast downward (Hoffmann, 2001) is a planner that translates PDDL into multi-valued planning tasks changing many of the implicit constraints to explicit constraints. The planner includes implementations of many search algorithms, of which we use A^* and (GBFS). Fast forward has implementation of heuristics such as the FF heuristics (Hoffmann, 2001) which decomposes the problem into smaller problems. Which solves them and uses there solution length as the heuristic. Though in this paper we also consider the implementation of ipdb (Haslum et al., 2007) which uses hill climbing to find a suitable pattern databases (Culberson and Schaeffer, 1998) for the given problem.

3 Related Work

Reinforcement Learning DQN has been shown to perform well on a wide range of tasks (Mnih et al., 2013; Young et al., 2017; Gu et al., 2023). In addition to the vanilla architecture (Mnih et al., 2013), there are various improvements such as Double DQN (van Hasselt et al., 2015) to combat overestimation, Dueling DQN (Wang et al., 2015) to help the network further distinguish between the effect of each action, and Prioritised Replay Buffer (Schaul et al., 2015) to improve the quality of each back-propagation update. We show that Double and Dueling DQN are beneficial for our specific task, while agents trained with Prioritised Replay Buffer trail behind in terms of performance. Specifically to the 3D grid world construction task, recurrent DQN also shows promising results in a similar work (Han et al., 2023) where the agent has a smaller span of view instead of global knowledge as in this work.

Beyond deriving a policy from action-value methods, policy gradient methods such as Advantage Actor-Critic (A2C) (Sutton and Barto, 2018) and its improvement PPO (Schulman et al., 2017) are also considered for our task due to their impressive performance in complicated environment (Yu et al., 2022; Zakharenkov and Makarov, 2021; Ye et al., 2020). A2C and PPO, however, are very

sensitive to hyper-parameters (Eimer et al., 2023; Zheng et al., 2023; Paul et al., 2019), and searching for an optimal set of hyper-parameters can be expensive.

Search-based Methods Minimum spanning trees combined with dynamic programming are shown to be capable of creating single-agent plans for construction (Kumar et al., 2014). Hierarchical planning with PDDL and an action-dependency graph are also viable in multi-agent construction plans (Singh et al., 2023). In addition, Planning and Mixed Integer Linear Programming have been combined to create a set of algorithms that decompose buildings into sub-structures that can be built by several agents in parallel (Srinivasan et al., 2023). However, little previous work has been done in using Planning and Search for construction with embedded scaffolding as in our use case.

4 Methodology

4.1 Environment

Due to the complexity of real-life construction, we abstracted this complex domain to an $n \times n \times n$ 3D grid world environment. Given a predefined structure, our object is to generate a valid sequence of steps of construction, abiding by constraints and physics imposed on the environment.

The virtual environment emulates a discrete construction environment. The agent is tasked to build a goal structure and the scaffolding, if required, to complete it. The agent is scored on the number of steps taken to complete the structure – the less number of steps, the better the score. At each timestep, the agent picks a single action, such as placing a block or moving around the environment. The environment consists of a goal structure, a current building zone, and the agent position. We further distinguish two types of environments: scaffoldless, and scaffolded environments.The agent observation is a tuple of construction zone, agent position and goal structure. In consideration of the state space, we decided on $4 \times 4 \times 4$ scaffoldless environment, and a $6 \times 6 \times 6$ scaffolded environment.

4.1.1 Scaffoldless Environment

The environment contains 2 types of blocks. Blue blocks are called column blocks, and can only be stacked vertically. Red blocks are versatile blocks, and can attach to any other block. Building a block without any supported attachment is considered illegal. The agent has no movement constraints, that is, it can move through blocks and fly. The goal is to build 3 predefined structures, as seen in Fig 1a, either by learning with RL or Search methods. RL approaches require a reward scheme to incentives agent behaviours. We detail the exact reward scheme used in Appendix B section 7.3.

4.1.2 Scaffolded Environment

In addition, we introduce scaffold block as a third type of block, indicated by the grey colour. To build either column or versatile blocks, they must either be (1) supported by the ground (on the floor) or (2) supported underneath by scaffolds. Fig 1b shows a 2-block-high column appropriately supported by scaffolds. The reward scheme particular to the scaffolded environment is included in Appendix B section 7.3.

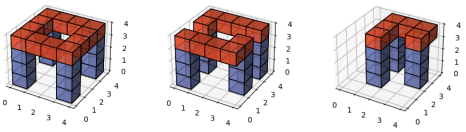

Figure 1a: 3 targets used for both RL and planning environment

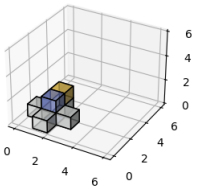

Figure 1b: Example of stacking placing a block in an environment with scaffold

4.2 Reinforcement Learning

4.2.1 DQN

Using neural networks as function approximations is appropriate for our task due to the large number of different states of the environment ($2^{4^3} + 4^3$ for the scaffoldless, and $3^{6^3} + 6^3$ for the scaffolding).

To demonstrate the importance of the neural net architecture, we train two different DQN agents, a fully connected (FC) and a 3D convolutional (CNN) neural net on the same scaffoldless $4 \times 4 \times 4$ environment. The two agents have approximately the same number of floating point parameters, and are both trained on a normal replay buffer of size 4096 with the same set of hyper-parameters (refer to Appendix C). We observe that CNN consistently outperforms FC over 10 independent runs (Fig 3.1), and move on with CNN in all of our future DQN agents. Fig 2 shows the architecture of the CNN agent, while Fig X in Appendix C shows the architecture of the FC agent.

Scaffoldless Environment: With the CNN architecture, we train Vanilla DQN, Double DQN, Dueling DQN and Double Dueling DQN on a normal replay buffer of size 4096. Each agent is trained for 1000 episodes, where each episode either ends either when the agent finishes building the desired structure, or when the number of steps taken exceeds 850. All agents are implemented in Pytorch, and each train takes less than 2 hours on an NVIDIA 4090 GPU. Different seeds for network initialisation can affect the performance (Andrychowicz et al., 2020), and to measure this effect, we train 10 independent agents for each algorithm mentioned above. Gathering the results, we observe certain advantages to Double Dueling DQN, and we apply this algorithm to the larger scaffolding environment later on.

While Prioritised Replay Buffer (Schaul et al., 2015; Wang et al., 2015, 2016) and N-step methods (Sutton and Barto, 2018) have been shown to be improvements to the DQN algorithm, we find these additions harmful for our agent. We train Vanilla, Double, Dueling and Double Dueling DQN with and without N-step with $N = 3$ using a Prioritised Replay Buffer. All agents under these configurations cannot finish building the structure within the 850 steps limit, and while they can reliably build the columns in blue, they often become stuck and cannot build the red blocks on top.

Scaffolded Environment: Scaffolding requires extra space around each column, and thus a bigger environment $6 \times 6 \times 6$ is more appropriate. Since the environment is much more complex with the introduction of scaffolding, we modify the agent architecture by adding an extra LSTM layer in between the Convolution and Linear Layer. We hope that this recurrence addition will allow the agent to keep an internal history of past actions and transitions, which will help with building and removing scaffolding. We employ Double Dueling DQN with a normal replay buffer of size 30000, and increase the timestep cap from 850 to 3000. A table of all hyper-parameters used is included in Appendix C. This agent is trained over 10000 episodes, for approximately 4 days on an NVIDIA L4 GPU on Google Cloud. Due to the long and expensive training process, we cannot experiment with different architectures and different set of hyper-parameters, which can have a major role in improving the agent performance.

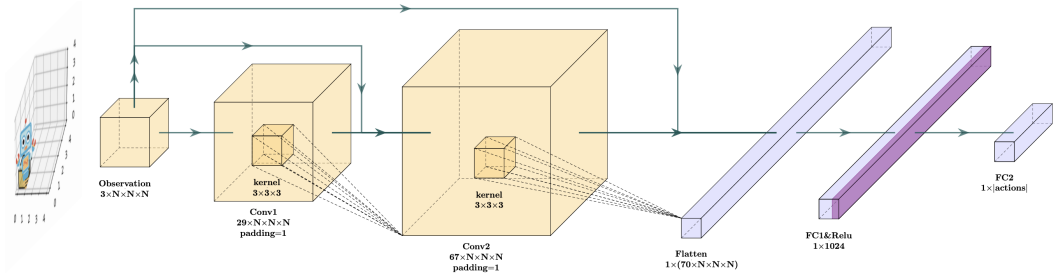

Figure 2: Convolutional Architecture

4.2.2 Policy Gradient Methods

We applied the same CNN architecture for our A2C agent, but instead with 2 different output heads, one head estimates value function $v(s)$ and the other head outputs a policy $\pi(a|s)$. The agent is trained with RMSProp optimizer, with step-size 0.00005, value coefficient $\in [0.05, 0.15, 0.2, 0.5]$, policy coefficient $\in [0.2, 0.3, 0.5, 1]$ and entropy coefficient $\in [0.01, 0.02, 0.3, 0.5]$. Unfortunately, in this limited hyper-parameter search, no A2C agents converged to a reasonable policy. Looking at the losses, we hypothesise the magnitude of each backpropagation update is too large, pushing the policy in extreme directions and making it difficult to converge. However, decreasing the step size does not improve the situation.

Another idea to prevent the policy from changing too rapidly and leading to instability is to make sure each new policy is reasonably close to the old one from the last iteration as suggested by Schulman

et al. (2017), in which PPO changes the optimizer to Adam, and introduces two extra hyperparameters: batchsize and epoch numbers, as it updates the policy multiple times per rollout. Both of our PPO agents, scaffoldless and scaffolding, converged prematurely to a local optimum: They can only build the first few column blocks and resort to moving actions only from that point since moving is the cheapest action.

4.3 PDDL Planning

We represented our problem using PDDL and used general solvers to search through the state space to look for optimal or sub-optimal solutions. One issue we found is the difficulty in formulating our problem in PDDL. The main reason is the lack of planning solvers that support *forall* axioms inside the precondition. We attempted to use this feature to make sure all required scaffolding was placed.

Implementation details: We wrote a program that would create plans using a graphical interface then convert them to PDDL files. The PDDL files would be split into two stages one for placing beams and one for placing columns.

We used PDDL to define the domain, which in our case, is the rules of our grid world and the problems, which corresponds to the targets whose building sequence we wish to obtain. These definitions were subsequently passed to generalized solvers such as fast downward which implement different search algorithms (A^*, greedy best first search) and many different general heuristics such as pattern databases(Culberson and Schaeffer, 1998) or lmcut(Helmert and Domshlak, 2011). We also looked at numerical planning but the planners did not contain the features needed to properly specify are problem. The specific feature missing with the numeric planners is the lack of support for quantifier (*forall*, *exists*) in preconditions, which is used to make sure we are not placing a floating block, in other words, it is on the floor or there is no supporting block.

4.4 Heuristic Search

We have came up with certain heuristics specified for are problem and also denote some useful optimization for this problem.

Some Optimizations In order to store the states in a efficient manner we were planning to use bitsets to represent the current state each block is either empty or full. Then use a integer to represent the agents current location. We can use the goal vector representation to determine if we can place beams and column at given locations . For further planned implementation detail can be found in the implementation repository.

Heuristics First we present a simple heuristic that does not account for not being able to float floating blocks or the order in which to do so.

Heuristic 1

Require: A to be the set of all blocks left to place
Require: Loc to be the current agent location
Require: $h \leftarrow 0$
  while A is not empty **do**
    Use BFS to find the nearest block in A from Loc $d = \min_{a \in A} |\text{Loc} - a|$
    $h \leftarrow h + 1 + d$ ▷ Add one to account for the cost to place the block
    $A \leftarrow A \setminus a_{\text{min}}$
  end while
  return h

We now present a modified version of the heuristic that both will account for floating blocks.

Heuristic 2

Require: A to be the set of all blocks left to place
Require: B to be the set of all blocks that can be placed
Require: Loc to be the current agent location
  while A is not empty **do**
    Use BFS to find the nearest block in A from Loc $d = \min_{b \in B} |\text{Loc} - b|$
    $h \leftarrow h + 1 + d$ ▷ Add one to account for the cost to place the block
    $b \leftarrow b \setminus b_{\text{min}}$
    $A \leftarrow A \setminus b_{\text{min}}$

add blocks from A to B that we can place now ▷ This could be neighboring columns or beams once all beams have be placed
end while
return h

It is worth noting that this heuristic is perfect and can be computed in polynomial time, we have also shown that finding the shortest path can be done in polynomial time without scaffolding. To do this just have the algorithm greedily expand the nodes with smallest heuristic value. Since the heuristic perfectly compute the cost it will only expand nodes along solution path. Leading to a polynomial time algorithm to find the shortest sequence of actions for the domain without scaffolding.

Though for the scaffolding problem we are unsure that the problem is solvable in polynomial time but believe the presented heuristic will drastically quicken the search time for the scaffolding domain. Though as stated before the empirical study of these techniques is left for future work.

5 Results

5.1 RL results

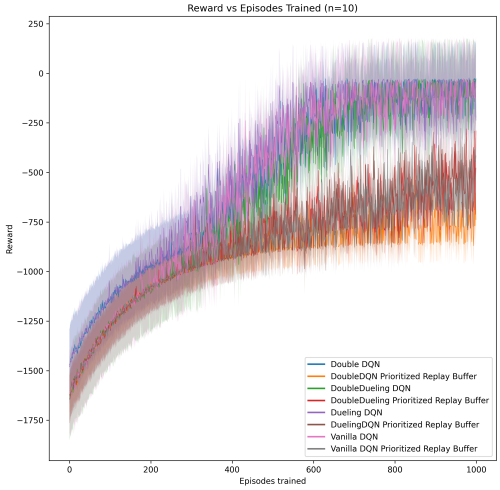

Figure 3: Rewards averaged over 10 episodes

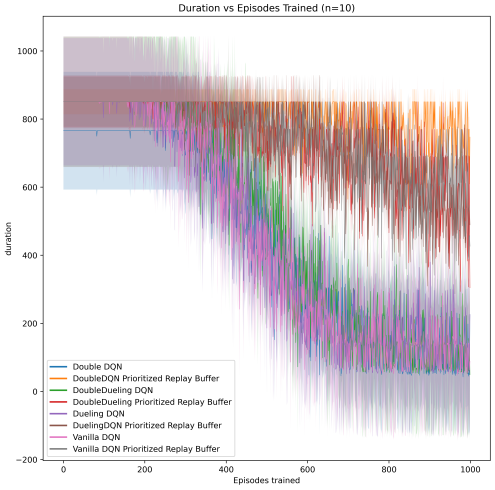

Figure 4: Duration's averaged over 10 episodes

Across 10 independent trials, agents trained with normal replay buffer consistently outperform those trained with Prioritised replay buffer. Specifically, Fig 4 shows Vanilla, Double, Dueling and Doubl Dueling DQN with normal replay buffer improve drastically at around 400 episodes, with all of them reaching maximum reward at about 600 episodes. Similarly, Fig 5. shows the clear trend of the above agents to require less time to finish building a structure. Also shown in the two figures, Prioritised replay buffer agents cannot reach optimality, as the number steps taken to finish an episode is still around 600.

The effect of different initialisation seed also presents in our experiments. Each plot describes 10 independent agents trained with the associated algorithm. For each agent, we run it 10 times and allow the agent to be ϵ-greedy with $\epsilon = 0.1$. Even when they are trained under the same algorithm and same set of hyperparameters, Fig 5 shows a small percentage of Double, Double Dueling and Vanilla DQN agents fail to finish the task. Note that Vanilla Linear DQN episode duration is capped at 850, our timestep limit.

Prioritized Replay buffer: All trials exceeded the maximum step length of 850 and got truncated(fig 6), unable to finish building the structure. Additionally, all architectures spent around 97% of the steps doing move actions. See appendix C for the graph of metrics.

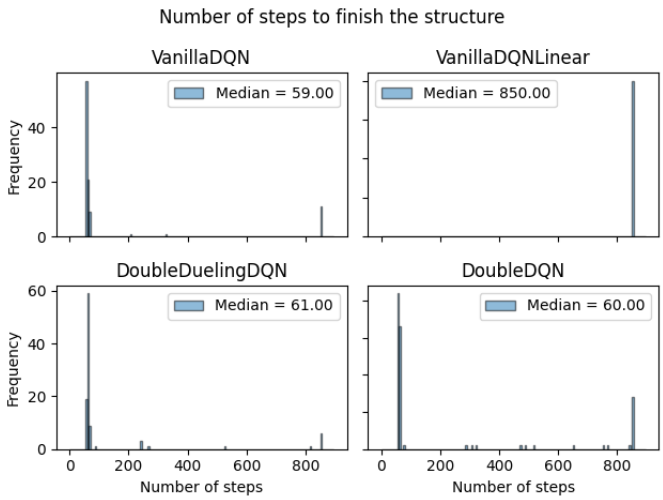

Figure 5: Normal replay buffer: Number of steps

Architectures that utilise prioritized replay buffer struggles with convergence during training. The normal replay buffer consistently converges after 400 episodes, decreasing the duration (the number of steps) require to complete the structure(fig 4). In contrast, prioritized replay buffer curves struggle to reduce the duration build the structure. Furthermore, the average reward curves for prioritized replay buffer scheme are consistently below normal replay buffer curves as the training goes.

Scaffolding: Although the agent is trained on 3000 timestep cap, we capped each trial to run in 1600 steps as we observed that agent stopped doing meaningful actions and keep getting stuck in much earlier steps. Even with 1600 step cap, all trials are truncated after 1600 steps as seen in Appendix B figure 14, unable to finish the structure. Visually, there are some promising runs with the agent finishing all columns and it starts to build beam blocks. We also observe that the agent does not complete the structure because it struggles to remove scaffold. Visualization of a more successful trial can be found in Appendix B figure 13.

5.2 PDDL and Heuristic Search Results

The results are from tests ran on an M1 pro using a single-core. Note cea stands for context additive heuristics (Eyerich et al., 2012), lm-cut stands for landmark cut (Helmert and Domshlak, 2011), blind is just uniformed search (no heuristic), ipdb (Haslum et al., 2007). The optimal search uses A^* where the suboptimal search uses (GBFS). Also note target 1 refers to the shape with the columns in the core and target 2 refers to the target with L shaped beams.

	States generated	Solution Length	Total Time
lmcut (*)	7636	59	0.125094
ipdb (*)	201	59	0.00061
blind (*)	374493	59	0.3229
ff	823	84	0.007926
cea	867	102	0.014642

Figure 6: Target 1 4x4 No scaffolding (* means search is optimal)

	States generated	Solution Length	Total Time
lmcut (*)	10325	55	0.15301
ipdb (*)	201	55	2.103691
blind (*)	330582	55	0.10377
ff	618	76	0.007287
cea	825	89	0.00875

Figure 7: Target 2 4x4 No scaffolding (* means search is optimal)

	States generated	Solution Length	Total Time
lmcut (*)	103424	103	7.470
ipdb (*)	1595	103	22.378
blind(*)	60570953	103	18.3600
ff	2779	170	0.024809
cea	4231	250	0.038748

Figure 8: 6x6 Target 1 no scaffolding (* means search is optimal)

	States generated	Solution Length	Total Time
lmcut (*)	578725	147	133.6296
ipdb (*)	62769	147	20.618
blind (*)	N/A	N/A	N/A
ff	5894	313	0.065047
cea	6274 + 4637	434	0.158907

Figure 9: 8x8 Target 1 no scaffolding (* means search is optimal)

	States generated	Solution Length	Total Time
lmcut (*)	N/A	N/A	N/A
ipdb (*)	795608	191	27.7817
blind (*)	N/A	N/A	N/A
ff	15602	509	0.182863
cea	23857	738	0.592207

Figure 10: 10x10 Target 1 no scaffolding (* means search is optimal)

Based on these tables we have greedy search leads to very suboptimal search and that ipdb seems to scale the best with increasing solution. As it was the only successful optimal heuristic to solve the problem in a reasonable time.

6 Conclusion, Discussion & Future Works

6.1 RL Discussion

Linear vs Convolution: DQN with only fully connected (Linear) layers performed the worst, with the sequence of action being almost random. We hypothesise that the CNN layer in our RL agents can learn to detect certain features of the goal structures, such as columns and beams. This also shows that the FC agent fails to extract spatial relationship about the building, as it behaves randomly after the training procedure.

Prioritized replay buffer: In theory, prioritized replay buffer samples transition tuples (S_t, A_t, R_t, S_{t+1}) with the highest magnitude of TD error for the update. Therefore, it will replay more important transitions, bettering the quality of the update Hessel et al. (2017). However, fig 4 shows no sign of convergence to the optimal solution, with the agent mostly performing move actions. We hypothesise that the Prioritise Replay Buffer continues to present to the agents seen states of building columns, instead of valuable states that achieve building versatile (red) blocks. Without this knowledge, the agent cannot build red blocks. For normal replay buffer, all states are sampled randomly, including the usage of versatile blocks.

Scaffolding: Ultimately, scaffolding is is a much more complex problem. As the number of actions and blocks within the environment grows, so does the space, exponentially expanding. Moreover, the added task of learning to dismantle scaffolds post-structure completion raises the difficulty of the problem, prolonging both the training time and step count

6.2 Search

PDDL Planning The greedy based search methods find solution that are far from optimal yet with very few generated states leading a small running time. For are given task purely greedy search does not make since but in future works would be worth while to look into weighted A^* (WA^*) where you give the heuristic more weight. Which leads to usually a quicker search time but since it will make the heuristic no longer admissible the solutions will not be optimal.

Heuristic Search We believe a custom implementation of A^* and IDA^* would perform quite well for the given problem. The details of how this would be implemented are talk about in the Heuristic search subsection of methedology.

6.3 Future Works

While we obtained promising results from both RL and planning, several approaches and improvements should be tried in the future.

For RL, we manually tuned hyperparameters for our models which does not give the best result possible, an efficient way to search for optimal hyperparameters is needed. Most training episodes we have are cut off at 1000 timesteps, models could be trained longer to improve the policy and reduce variance. Work in generalizing to different structures is possible but generalization in deep Q learning and policy gradient methods are challenging as it requires large amount of computational resources (Ghosh et al., 2021). Other RL approaches like model-based methods and multi-agent RL are yet to be explored.

For planning, more work can be done in numerical planning to allow the heuristic to have a notion of distance. Classical planners use general heuristics, we would like to design a better heuristic

specifically for our problem. Lastly we hope that by combining RL and planning, we will be able to achieve better results than each individual approach.

References

Andrychowicz, M., Raichuk, A., Stańczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., and Bachem, O. (2020). What matters in on-policy reinforcement learning? a large-scale empirical study.

Culberson, J. C. and Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, 14:318–334.

Eimer, T., Lindauer, M., and Raileanu, R. (2023). Hyperparameters in reinforcement learning and how to tune them. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J., editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 9104–9149. PMLR.

Eyerich, P., Mattmüller, R., and Röger, G. (2012). Using the context-enhanced additive heuristic for temporal and numeric planning. In *Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments*, pages 49–64. Springer.

Ghosh, D., Rahme, J., Kumar, A., Zhang, A., Adams, R. P., and Levine, S. (2021). Why generalization in rl is difficult: Epistemic pomdps and implicit partial observability. *Advances in Neural Information Processing Systems*, 31:25502–25515.

Gu, J., Guo, Y., Lam, Y., and Pu, Z. B. (2023). Flappy bird game based on the deep q learning neural network. *Highlights in Science, Engineering and Technology*, 34:191–195.

Han, W., Wu, H., Hirota, E., Gao, A., Pinto, L., Righetti, L., and Feng, C. (2023). Learning simultaneous navigation and construction in grid worlds. In *The Eleventh International Conference on Learning Representations*.

Haslum, P., Botea, A., Helmert, M., Bonet, B., Koenig, S., et al. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, volume 7, pages 1007–1012.

Helmert, M. and Domshlak, C. (2011). Lm-cut: Optimal planning with the landmark-cut heuristic. *Seventh international planning competition (IPC 2011), deterministic part*, pages 103–105.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning.

Hoffmann, J. (2001). Ff: The fast-forward planning system. *AI magazine*, 22(3):57–57.

Kumar, T., Jung, S., and Koenig, S. (2014). A tree-based algorithm for construction robots. *Proceedings of the International Conference on Automated Planning and Scheduling*, 2014:481–489.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

Paul, S., Kurin, V., and Whiteson, S. (2019). Fast efficient hyperparameter tuning for policy gradients.

Schaul, T., Quan, J., Antonoglou, I., Silver, D., and Deepmind, G. (2015). Prioritized experience replay. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pages 1–21.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Openai, O. K. (2017). Proximal policy optimization algorithms.

Singh, S., Gutow, G., Srinivasan, A. K., Vundurthy, B., and Choset, H. (2023). Hierarchical propositional logic planning for multi-agent collective construction. In *Construction Robotics Workshop*.

Srinivasan, A. K., Singh, S., Gutow, G., Choset, H., and Vundurthy, B. (2023). Multi-agent collective construction using 3d decomposition.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning.

Wang, Z., Mnih, V., Bapst, V., Munos, R., Heess, N., Kavukcuoglu, K., and Freitas, N. D. (2016). Sample efficient actor-critic with experience replay. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H. V., Lanctot, M., and Frcitas, N. D. (2015). Dueling network architectures for deep reinforcement learning. *33rd International Conference on Machine Learning, ICML 2016*, 4:2939–2947.

Ye, D., Liu, Z., Sun, M., Shi, B., Zhao, P., Wu, H., Yu, H., Yang, S., Wu, X., Guo, Q., Chen, Q., Yin, Y., Zhang, H., Shi, T., Wang, L., Fu, Q., Yang, W., and Huang, L. (2020). Mastering complex control in moba games with deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34:6672–6679.

Young, K., Vasan, G., and Hayward, R. (2017). Neurohex: A deep q-learning hex agent. *Communications in Computer and Information Science*, 705:3–18.

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., and Wu, Y. (2022). The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624.

Zakharenkov, A. and Makarov, I. (2021). Deep reinforcement learning with dqn vs. ppo in vizdoom. In *2021 IEEE 21st International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000131–000136.

Zheng, R., Dou, S., Gao, S., Hua, Y., Shen, W., Wang, B., Liu, Y., Jin, S., Liu, Q., Zhou, Y., Xiong, L., Chen, L., Xi, Z., Xu, N., Lai, W., Zhu, M., Chang, C., Yin, Z., Weng, R., Cheng, W., Huang, H., Sun, T., Yan, H., Gui, T., Zhang, Q., Qiu, X., and Huang, X. (2023). Secrets of rlhf in large language models part i: Ppo.

7 Appendix A

7.1 Acknowledgments

We would like to thank Brian Gue for his subject matter expert. We would also like to thank Professor Russell Greiner for giving feedback through the term and suggesting methods to try such as planning. We would also like to thank Steven Tang for his guidance and feedback on our progress throughout the term.

Appendix B

7.2 Environment

Table 1: Reward scheme for the agent

Condition	Reward
Correct construction	10
Per time step	-1
Exceeds maximum time steps	-2
Per block placed in construction zone	-0.25

Table 2: Scaffold environment Reward scheme for the agent

Condition	Reward
Move	-0.2
Correct block placement	0.9
Incorrect block placement	-0.5
Place scaffold	-0.3
Remove scaffold	-0.3/0.2
illegal placement	-1
finish structure	1

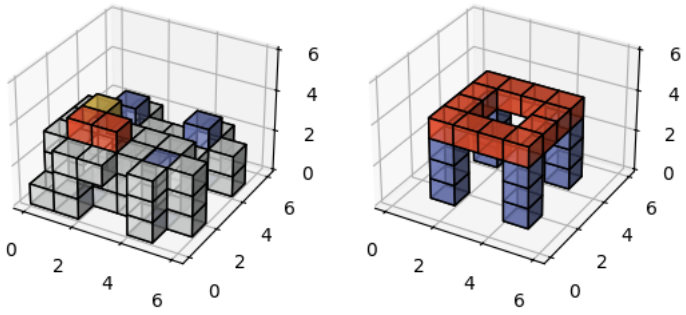

Figure 11: successful scaffold trial

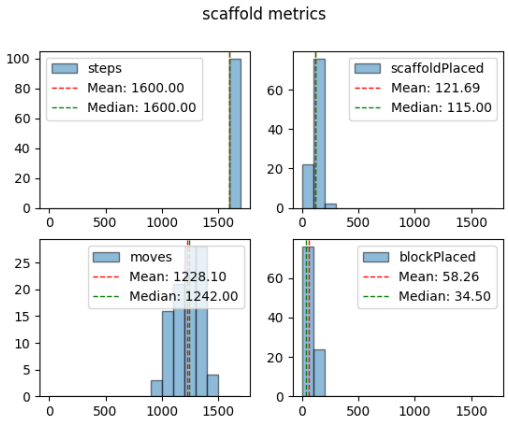

Figure 12: Agent trained on Double Dueling on 6x6x6 environment

Appendix C

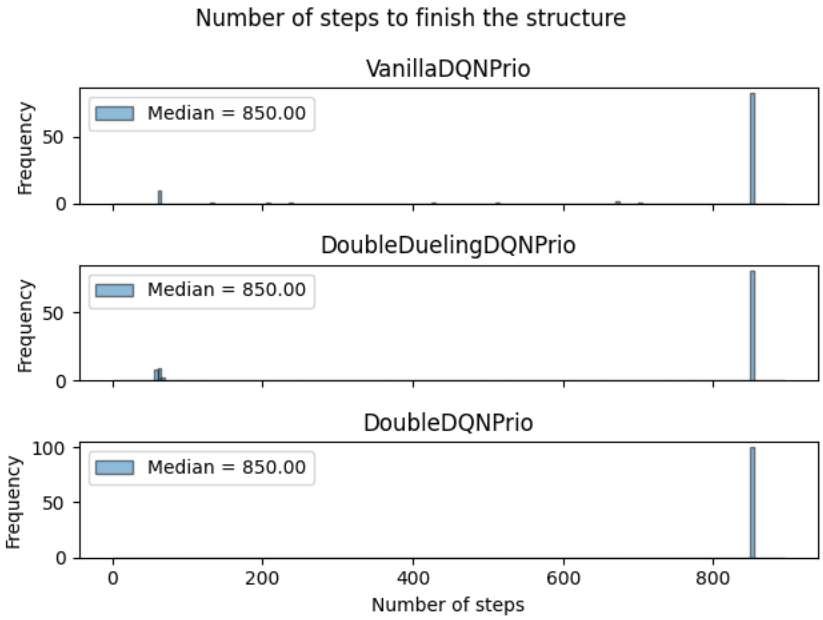

Figure 13: Number of steps with prioritized replay buffer

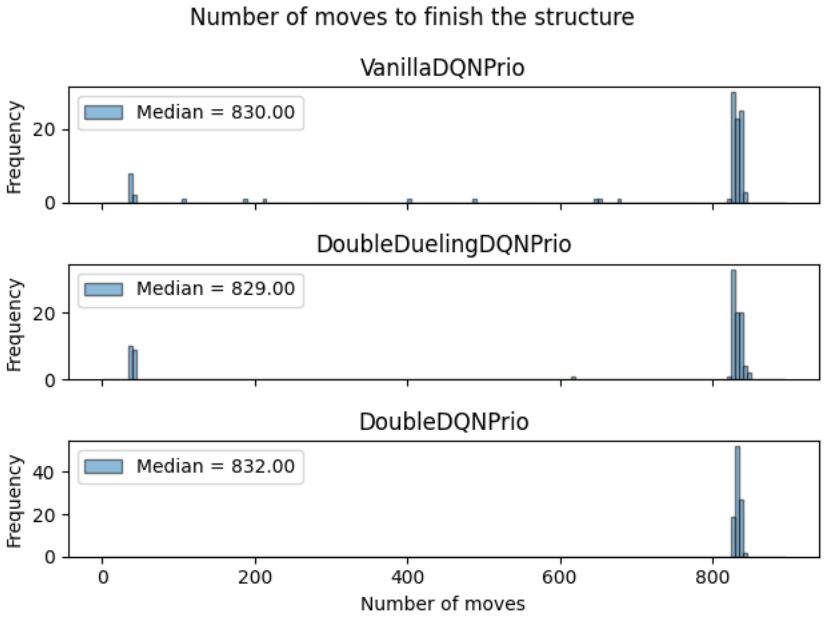

Figure 14: Number of moves with prioritized replay buffer

Appendix D

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 15: DQN with replay buffer pseudo code taken from Mnih et al. (2013)

Algorithm 1: Double DQN Algorithm.

input : $\mathcal{D}$ – empty replay buffer; θ – initial network parameters, θ^- – copy of θ
input : N_r – replay buffer maximum size; N_b – training batch size; N^- – target network replacement freq.
for *episode* $e \in \{1, 2, \ldots, M\}$ **do**
  Initialize frame sequence $\mathbf{x} \leftarrow ()$
  for $t \in \{0, 1, \ldots\}$ **do**
    Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
    Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to $\mathbf{x}$
    if $|\mathbf{x}| > N_f$ **then** delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$ **end**
    Set $s' \leftarrow \mathbf{x}$, and add transition tuple (s, a, r, s') to $\mathcal{D}$,
      replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
    Sample a minibatch of N_b tuples $(s, a, r, s') \sim \text{Unif}(\mathcal{D})$
    Construct target values, one for each of the N_b tuples:
    Define $a^{\max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
    $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{\max}(s'; \theta); \theta^-), & \text{otherwise.} \end{cases}$
    Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
    Replace target parameters $\theta^- \leftarrow \theta$ every N^- steps
  end
end

Figure 16: Double DQN with replay buffer pseudo code taken from van Hasselt et al. (2015)

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Figure 17: Double DQN with prioritized replay buffer pseudo code taken from Schaul et al. (2015)