CS 221

Solving Box Stacking Problem Using Reinforcement Learning

Ju Zhang
juzhang@stanford.edu

Katherine Chen
kathchen@stanford.edu

1 Introduction

In modern warehouses, robots are used for stacking packages. However, the stacking rules are often hard-coded into the robot, which can be tedious when there are many different-sized packages. This project aims to simulate a box-stacking system using Markov decision processes (MDP) and reinforcement learning (RL) methods, and to learn the optimal strategy for stacking different-sized boxes as stable as possible while minimizing gaps between them.

2 Literature Review

Academic research on developing algorithms that learns optimal policy to stack items has been done. Yifang Liu et. al. [1] tried to solve a more complex problem using Q-learning to learn the best strategy to stack irregular objects. The proposed method in the article outperforms previous heuristics-based planning. Several other studies focusing on robotic stacking have proposed reinforcement learning methods to pick the next best move to stack items without breaking the balance [2] [3] .

The box stacking problem can also be viewed as a variant of the Tetris gaming problem, which is an important benchmark for research in artificial intelligence and machine learning [4]. There are multiple studies published in which the problem was solved by different algorithms, including approximated dynamic programming [5], reinforcement learning [6], policy iteration[7], and genetic algorithms [8]. In recent years, many studies have tried to solve the Tetris problem by reinforcement learning combined with neural networks. For example, a study by Patrick Thiam et al. [9] found that reinforcement learning (Q-learning) can significantly outperform agents using fixed policies.

Our project extends these related works by utilizing Q-learning and physics simulation to learn the optimal stacking strategy. But instead of irregularly shaped objects like rocks or Tetris pieces, we focus specifically on rectangular boxes with irregular dimensions as warehouse robots are most commonly faced with cuboid cardboard boxes.

3 Dataset

In this project, we simplify the problem to a discretized 2D world (x-y grid with a fixed width and infinite height) with rectangular boxes of integer dimensions. For each MDP trial, we prepare the following the following data:

- $W \in \{3, 4, 5\}$, the world width
- $N = \in \{3, 4, 5\}$, the number of boxes
- N pairs of (w, h), the dimensions of the boxes, each sampled uniformly between 1 and W

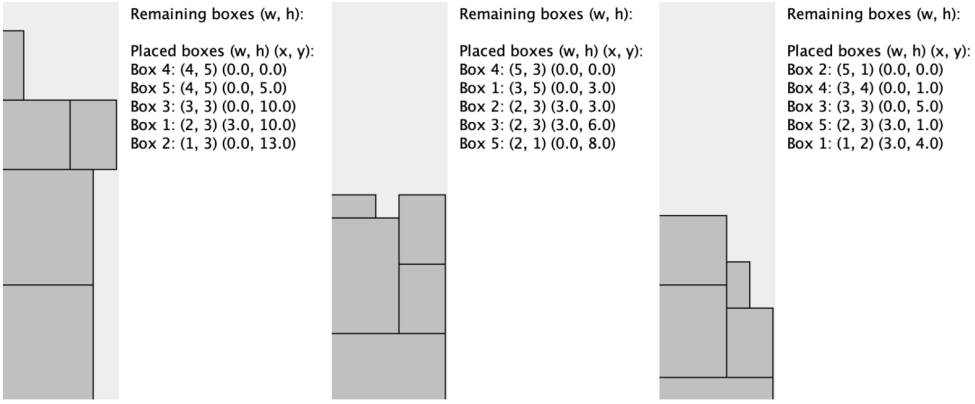
The bottom-left corner of each box is to be placed at an integer location (x, y) within the world. Our goal is to determine the best policy that describes the order in which to choose the boxes and the x-location to drop each box at. To reduce the problem to a reasonable size, we fix W, N, w, h, x, and y to integers. We will test our algorithm on different combinations of W and N.

4 Baseline

Our baseline approach follows an intuitive policy that consists of two rules defined by the team:

1. Grab the boxes in order from widest to narrowest, and tallest to shortest in case the width is a tie. This rule favors stability as it is likely for wider boxes to end up at the bottom and narrow boxes to end up at the top.
2. Drop each box at a location such that the bottom of the box ends up as close to the ground as possible (favoring stability). If there is a tie, choose the leftmost of location (favoring small gaps).

Figure 1: **Examples of end states reached by following the baseline policy**



The reward of a trial is calculated at the end state by the reward function defined in Section 6, which is based on the final stack's stability and gaps. To calculate the rewards, the baseline policy used the same reward functions as the RL policy, as detailed in section 6.

For our baseline algorithm, if the AI system can see all the boxes to be stacked from the beginning, the whole process of box stacking is deterministic and each starting state corresponds to a determined reward. This could be unrealistic in real-world situation. Therefore we made the AI can only see the k boxes in the queue, grab boxes from it and replenish the 'seen' box list with one random box from the 'unseen' boxes. In the final result, we used $k = 3$.

5 Main Approach

5.1 Modeling the MDP

Our MDP stores the current box stack, the boxes that still need to be dropped, and updates states through an alternating sequence of pick $\rightarrow$ drop $\rightarrow$ pick $\rightarrow$ drop $\rightarrow$ ... actions. It is defined as:

States: Each state is represented as a 3-element tuple: `(world, boxList, boxedPicked)`.

- `world` is a 2D tuple representing the current box stack having W columns and a variable number of rows, where row 0 corresponds to the ground. Upon initialization of the MDP, the number of rows is set to a specified height. While stacking the boxes, we increment the number of rows as we run out of vertical space. The value of `world[row][col]` is 1 if the cell is occupied by a box, and 0 if it is free.
- `boxList` is a list of (w, h) pairs representing the remaining boxes to be picked and dropped.

- `boxedPicked` is the box that has just been picked and ready to be dropped. If the previous action is a pick, then `boxedPicked` is the (w, h) tuple of the box. Otherwise, `boxedPicked` is set to be `None`.

Start state: In the start state, `world` is a 2D tuple of all 0s, `boxList` contains N pairs of (w, h) boxes representing all boxes generated for the trial, and `boxedPicked` is `None`.

Actions(s): If `boxPicked` is `None`, then Actions(s) = {'pick 0', 'pick 1', ...,}, where 'pick i' represents picking the box at index i of `boxList`. Otherwise, Action(s) = {'drop 0', 'drop 1', ...}, where 'drop x' represents dropping `boxPicked` at available horizontal location x.

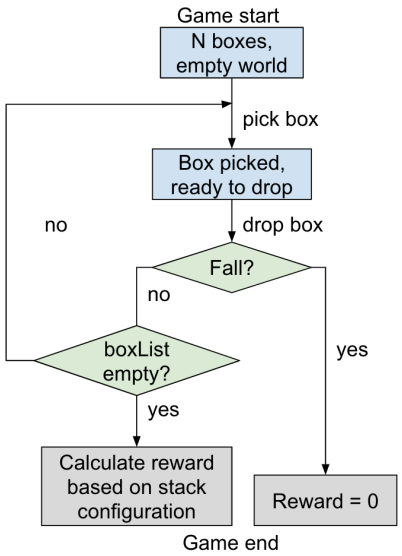
SuccAndProbReward(s, a):

- If action a is in the form of 'pick i', then state s = `(world, boxList, None)` transitions to its successor s' = `(world, boxList` with box at index i removed, removed box`)` with probability = 1 and reward = 0.
- If a is in the form of 'drop x' and s = `(world, boxList, boxPicked)`, try to update world to a `newWorld` with `boxPicked` dropped at location x. If the stack falls (determined by whether or not the center of gravity of the newly dropped box is supported), then s transitions to s' = `(None, boxList, None)` with probability = 1 and reward = -999. Otherwise, transition to s' = `(newWorld, boxList, None)` with probability = 1, and reward = [`reward(world)` if `boxList` is empty else 0]. Note that the reward function is only ever called at the end of the trial, provided that the boxes have been stacked successfully.

IsEnd(s): Returns true when `world` is `None` (meaning that the box stack has fallen) or when `boxList` has been depleted, and false otherwise.

Discount factor: $\gamma = 1.0$

Figure 2: **Flowchart for one trial simulated on the MDP**



5.2 Reinforcement Learning with Q-Learning

Based on the MDP defined, our goal is to determine the optimal policy, which is a map from each state s to an action a in Action(s) that yields the highest expected reward. Stacking boxes is an online task. In the real world, the robot does not know how the world works, for example, whether or not dropping a box at a certain location would cause the current stack to fall. With reinforcement learning, the robot learns the optimal policy by performing actions in the world to find out and collect rewards. Our reinforcement learning algorithm utilizes epsilon-greedy and Q-learning, outlined below:

```
For each iteration t = 1, 2, 3,...
        Choose action a current state s # epsilon-greedy
        Receive reward r, observe new state s'
        Update parameters # Q-learning with function approximation
```

Define the following:

- A feature extractor $\phi(s, a)$ that maps a (state, action) pair to feature values. We are currently using the identity feature extractor with all feature values equal to 1.
- Weight w for each (state, action) pair
- $\hat{Q}_{opt}(s, a) = w \cdot \phi(s, a)$

At each state s, epsilon-greedy chooses an action a (picks a box or drops the current box at a location) with exploration probability $\epsilon = 0.2$:

$$\pi_{act}(s) = \begin{cases} \text{argmax}_{a \in Actions(s)} \hat{Q}_{opt}(s, a) & \text{with probability } 1 - \epsilon \\ \text{random from Actions}(s) & \text{with probability } \epsilon \end{cases}$$

Then the algorithm takes action a, receive a reward r, and observes a new state s'. Q-learning updates the weight of (s, a) by $w \leftarrow w - \eta[\hat{Q}_{opt}(s, a) - (r + \gamma \hat{V}_{opt}(s'))]\phi(s, a)$, where $\eta = \sqrt{\frac{1}{\text{numIterations}}}$. Running this algorithm many times gives us a learned policy: $\pi_{act}(s) = \text{argmax}_{a \in Actions(s)} \hat{Q}_{opt}(s, a)$.

5.3 Example

Figure 3: **An example of a MDP trial**

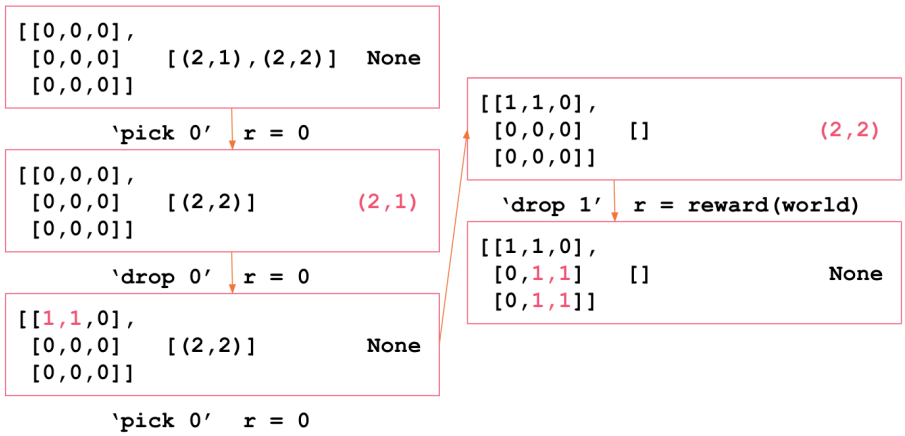


Figure 3 shows a concrete sequence of states and actions that could happen when simulating Q-learning on the MDP. For simplification purposes, consider a world with $W = 3$ and $N = 2$. At the beginning, the algorithm hasn't seen any reward, so it doesn't have a preference over which box should be picked and where each box should be dropped. It could pick the 2 by 1 box, drop it at x-location 0, pick the 2 by 2 box, then drop it at x-location 1. At this point we've reached an end state and evaluate the reward based on the ending stack configuration as described in the next section. In the next trial, the algorithm would have updated weights and an update policy, and it may exploit nonzero $\hat{Q}_{opt}$'s when choosing actions.

6 Evaluation Metric

To evaluate the performance of the baseline and main approaches, we defined a reward function based on the final state of the box stack, and compared the rewards from the simulations in which the AI either follows the baseline policy or the policy learned by the reinforcement learning algorithm.

The reward function considers two aspects of the final box stack, $instability$ and $misalignment$. Figure 4 shows a sample space where **unstable** and **misaligned** pixels are labeled.

Representing the number of boxes to drop as n, the pixel world in the end state as $space$, the $instability$ is the total number of gaps (pixels not occupied by the box) of the final stack on vertical direction, as

$$instability(space) == \sum_i \sum_j I(pixel(i,j) = 0)$$

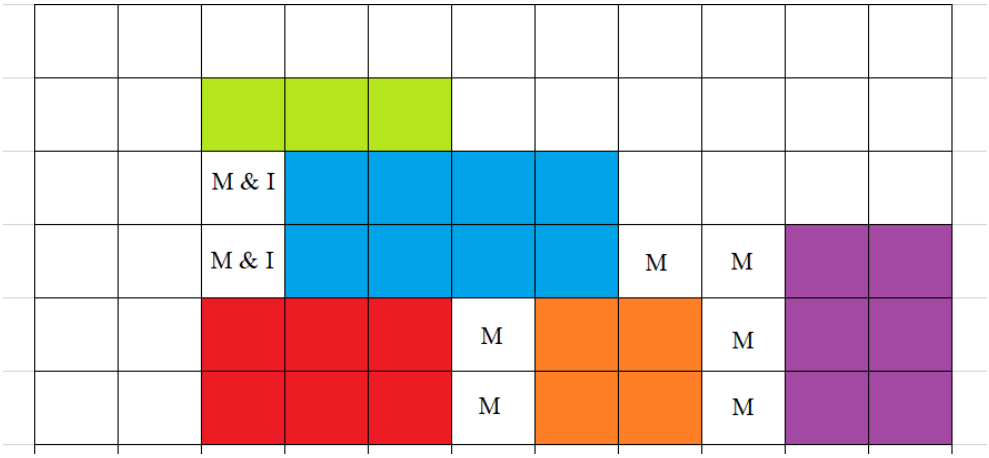
for all i locations of x-axis on which not all pixels = 0

and $misalignment$ is the total number of gaps of the final stack on horizontal direction

$$misalignment(space) = \sum_i \sum_j I(pixel(i,j) = 0)$$

for all j locations of y-axis on which not all pixels = 0

Figure 4: **Diagram showing the evaluation of instability and misalignment on the final stack. M: pixels that count towards misalignment; I: pixels that count towards instability**



The reward function is defined as follows to penalize both the instability and misalignment after all boxes are stacked. In summary, it is the negative of total counts of the pixels of gaps in both vertical and horizontal direction:

$$Reward_1(s) = \begin{cases} -(instability(space) + misalignment(space)) \\ -999 \quad \text{If the stack falls} \end{cases}$$

Alternatively, we can define the reward function of the second version to take the size of box stack into consideration:

$$Reward_2(s) = \begin{cases} \frac{-(instability(space)+misalignment(space))}{\sum_i \sum_j I(pixel(i,j)=1)} \\ -999 \quad \text{If the stack falls} \end{cases}$$

In the result analysis, we tested the performance of baseline and RL-learned policy based on both rewards functions.

7 Results & Analysis

We have generated random box list containing 3, 4, or 5 boxes with random width and height between 1 and 3. Besides the box list size, it is also interesting to test the algorithms in space with different width (world size). Using a world width of 3, 4, 5, we have 3*3=9 total situations in combination.

For one round of simulation, the RL policy was learned first, using 30,000 trials with a maximum of 2,000 iteration. Then the AI follows either the learned RL policy or the baseline policy to stack the boxes. After the end state was reached, the rewards were calculated by either of the two reward functions described above. We chose to run 30 rounds of simulations with the considerations of both the statistical power and time for running. Finally, The mean rewards from all simulations were calculated and the performance of the baseline approach and RL algorithm were compared (table 1).

Table 1. Comparison of average reward scores collected from the RL policy and the baseline policy

Simulation Settings		Average Reward					
		$Reward_1(s)$			$Reward_2(s)$		
Width of the world	N of Boxes	Baseline	RL	Diff	Baseline	RL	Diff
3	3	-0.15	-65.56	-65.41	-0.0146	-53.275	-53.2604
3	4	-0.3	-79.65	-79.35	-0.0122	-56.485	-56.4728
3	5	-0.1	-82.345	-82.245	-0.0102	-72.51	-72.4998
4	3	-0.1	-82.585	-82.485	-0.0045	-92.18	-92.1755
4	4	-0.25	-109.125	-108.875	-0.0568	-115.175	-115.118
4	5	-0.25	-114.35	-114.1	-0.0234	-111.81	-111.787
5	3	-50.1	-143.495	-93.395	-49.968	-110.245	-60.277
5	4	-0.45	-153.57	-153.12	-0.0472	-163.935	-163.888
5	5	-0.15	-174.9	-174.75	-49.9725	-159.025	-109.053

On average, the performance of baseline policy is inferior to that of RL algorithm. This is within our expectation, because intuitively the baseline approach should work pretty well other than it may cause a move that lead to the falling of the stack and return 0 reward. But in our small simulated world, this situation is very rare, since often gravity center of a box usually can get some supports. We have occasionally observed that a -999 reward case from the baseline approach in our simulation. On the other hand, our current RL algorithm needs to explore all state spaces to learn the optimal strategy, and this is a challenge when the number of states can exceed tens of thousands. As the result, the learned RL algorithm often makes a decision that cause the stack to fall and harvested negative -999 rewards. With regard to different reward functions, using $Reward_2(s)$ seems not giving the RL algorthim a better performance compared to $Reward_1(s)$.

8 Error Analysis

In most of the time, our RL algorithms learned the 'correct' action, i.e., a similar action humans will do, for a given state. But it still makes decisions that is not consistent with human choice when facing some situations.

For example, given the following state, which basically is the stack occupying the first row and the first element of the second row, and a box with a width of 1 and height of 2 on hand (the world is upside down in the program):

$$\begin{aligned} State : (((1,1,1),\\ (0,0,1),\\ (0,0,0),\\ (0,0,0)),\\ (),(1,2)); \end{aligned}$$

The learned RL policy returns $drop2$, which is to drop the boxes at the third column. As humans, we may choose drop it at column 0 or 1, to make the stack not that tall. The reason of this discrepancy is that our reward function does not penalize how tall the stack is. Dropping the box at column 2 still

generates a stack that is perfect in the program's definition of stability and alignment. In future, we may think of a more sophisticated way of evaluating the final box stack, as discussed in section 9.

9 Future Work

Treating the box stacking problem as an MDP seems overkill for our current simplified problem in a pixel world, for we observed that the learned RL policy often generate results inferior to that of baseline approach, especially when we do not force the AI conducting the baseline policy to see only k boxes in advance.

This is due to in the current simulation, picking a box of choice and dropping a box at a specific location are both deterministic processes. In this sense, a search algorithm that does not handle uncertainty may be sufficient to solve the problem. Nevertheless, modeling the box stacking as an MDP is advantageous as we can expand it to a situation with uncertainty. In real-world situations, randomness may occur during the box stacking process. For example, a box may have some probability of being crushed and damaged when another box stacked over it. It would be very interesting to add complexity to the problem by introducing uncertainty, such as a probability of a box being crushed proportional to the size of boxes above it, in the future.

One of the problems the project team encountered was the huge time complexity when trying to expand the problem to a bigger world with more boxes. The number of states increased exponentially with the size of the world and the size of box list For instance, with 7 boxes and a world of 5 pixel width, the number start states can be easily over 200,000. We were considering using a better feature extractor instead of the current identity feature extractor to generalize the RL learning algorithm and improve the efficiency. However we did not have a chance to successfully implement it given the time constraint. It would be highly desirable to complete that as the next step in future work.

One of other futures works is to re-define the reward function. A more comprehensive reward function may also take into consideration of other dimensions of the box stack, such as the height and the position of the gravity center. Additionally, we can explored more in the definition of instability and misalignment, which will in turn change the reward function. The current definition of instability and misalignment is based on the size gaps on vertical direction or horizontal direction. A more sophisticated definition should be able to define the two metrics in exact physical sense, and make the learning result closer to real world choice.

References

[1] Y. Liu, S. M. Shamsi, L. Fang, C. Chen, and N. Napp. Deep q-learning for dry stacking irregular objects. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1569–1576, 2018.

[2] Yifang Liu, Jiwon Choi, and Nils Napp. Planning for robotic dry stacking with irregular stones. *12th Conference on Field and Service Robotics (FSR19)*, Aug 2019.

[3] F. Furrer, M. Wermelinger, H. Yoshida, F. Gramazio, M. Kohler, R. Siegwart, and M. Hutter. Autonomous robotic stone stacking with online next best object target pose planning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2350–2356, 2017.

[4] Simón Algorta and Özgür Şimşek. The game of tetris in machine learning, 2019.

[5] Victor Gabillon, Mohammad Ghavamzadeh, and Bruno Scherrer. Approximate dynamic programming finally performs well in the game of tetris. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1754–1762. Curran Associates, Inc., 2013.

[6] Alexander Groß, Jan Friedland, and Friedhelm Schwenker. Learning to play tetris applying reinforcement learning methods. In *ESANN*, pages 131–136, 2008.

[7] Michail G Lagoudakis, Ronald Parr, and Michael L Littman. Least-squares methods in reinforcement learning for control. In *Hellenic conference on artificial intelligence*, pages 249–260. Springer, 2002.

[8] Niko Böhm, Gabriella Kókai, and Stefan Mandl. An evolutionary approach to tetris. In *The Sixth Metaheuristics International Conference (MIC2005)*, page 5, 2005.

[9] Patrick Thiam, Viktor Kessler, and Friedhelm Schwenker. A reinforcement learning algorithm to train a tetris playing agent. In *IAPR Workshop on Artificial Neural Networks in Pattern Recognition*, pages 165–170. Springer, 2014.