A Dynamic Programming Approach To Optimal Block Configuration Building

Morris Fung

August 17, 2021

Abstract

This paper presents an optimal solution plan using dynamic programming for the following problem: There are two separate areas A and B of equal lengths l and widths w. Area A is filled with n number of blocks in a starting configuration. Area B, meanwhile, is empty. The goal is simple: move the blocks from area A to B such that:

1. We minimize the number of moves used throughout the entire process.
2. We end up with a given specified configuration in area B.
3. Blocks must, at all times, be located in either area A or B.
4. One may not stack above the height of either area.
5. It is possible to move blocks from area A to area B and within area B but it is not possible to move blocks from area B to area A or within area A.
6. At any given time, no block may "float" in midair.
7. A blocks may also occupy only one space in an area at a time, i.e you can't have two blocks in the same space.
8. Only blocks on the top row of any area may be moved, i.e you can't move a block stuck below another block.

1 Motivation

Building blocks to a certain configuration is a common operations problem found in industry. In logistics, one sees this often within the North American transportation industry after huge cargo ships come from overseas and must unload containers at the docks onto freight trains. The problem there is similar to the one attempted here. The crane operator must move each container from dock area onto the train according to an end configuration that takes into account key constraints such as weight, maximum height of stacking, offloading ease (often built into the end configuration), rail car capacities, etc.

2 Building the Dynamic Programming Solution

Dynamic Programming is a solution that requires:

1. States $s \in S$.
2. Actions $a \in A$.
3. Transitions represented by $p(s'|s, a)\ \forall s, a$.
4. Rewards
5. Policy

1	2	5	6
3	4	7	8

Table 1: Starting Area A configuration corresponding to S_0.

6			

Table 2: $S_{1,1}$.

2.1 States

Our state space consists of every possible configuration of n blocks in area B. For illustration and analysis purposes, let us use the following notation:

$$s_{i,j} \in S$$

Where $i \leq n$, i refers to the number of blocks currently in area B, and j, an optional subscript, references a specific state with i blocks in area B. If j is omitted, then we refer to the entire subset s_i.

Let us consider the following example. We have a total of n=8 blocks. Our starting configuration of blocks in Area A is as shown in Table 1:

This means that one possible state for s_1 would be as described in Table 2.

Already, one can see that we might have an explosion of states both in each subset s_i and as a result, in the entire state S universe. Although it is outside the scope of this problem to calculate the exact number of states, it is worth noting that this complex computation is the sum of many individual permutations. Is it necessary to simply guess each permutation? Perhaps.

But if one considers the following sub-problem, one can usually calculate the total number of permutations in a much more efficient manner:

Note that Area B's dimensions are l length (number of columns) and w width (number of rows). If one proposes all possible solutions to:

$$\sum_{i}^{w} x_i = n \tag{1}$$

such that

$$x_1 \geq x_2 \geq x_3 \cdots \geq x_w \tag{2}$$

and

$$x_i \leq l \, \forall i_{\leq w} \tag{3}$$

and

$$x \in Z^+ \tag{4}$$

One can then use those solutions in the following manner to obtain the total number of possible states given n:

$$\begin{cases} \sum_x P(l,x) \, if n \geq l \\ \sum_x P(x,l) \, if n < l \end{cases}$$

Needless to say, the number of states grows exponentially as n increases.

2.2 Actions

For all states $s \in S_i$, the set of actions includes the movement of any top row block from:

1. States $s \in S_i$.
2. The configuration from which the starting state in Area A transitioned and which now corresponds to S_i

2.3 Transitions

This problem is a deterministic one. As such, we don't have the added complexity of stochastic transitions. However, we must note that each state may transition from $s_{i,j}$ to either $s_{i+1,j}$ or $s_{i,k}$ where j and k refer to any random feasible state transitioned from $s_{i,j}$.

2.4 Rewards

Each feasible action is awarded 1. $r(a) = 1$. Since the goal is to minimize these rewards, it is not necessary to include a negative reward as punishment for any specific action in any specific state.

2.5 Policy

This algorithm is an on-policy one. We start with a policy initialized randomly. Over the course of the many runs, the policy will change for each state such that only the optimal action is chosen.

3 Algorithm

3.1 Bellman Equation

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r + \gamma v_\pi(s')], \text{ for all } s \in S \tag{5}$$

We utilize the Bellman Equation above for all states until convergence of values for each state. We give $\gamma = .95$.

3.2 Curse of Dimensionality

In order to avoid having the algorithm take too long to converge, we shall make use of the facts that:

1. Each reward is the same.
2. It is not possible to transition from $s_{i,j}$ to $s_{i-1,k}$.

If our initial policy were such that we choose only feasible actions that increase i as well as directly result in a state similar to the end terminal goal state, then we can use this policy until there are no more direct moves to make. At that point, we can then start our dynamic programming equation.

4 Analysis and Closing Remarks

The algorithm works surprisingly quickly for $n \leq 10$. Afterwards, it gets a tad slow. In future iterations of this algorithm, I'll change the algorithm to be reinforcement-learning based.