Towards Uniform Sampling of Pareto-Optimal Armies in StarCraft II

Karthik Narayan (karthik@starfruit-llc.com)

Background

Heron Systems plans to construct reinforcement learning (RL)-based agents which use self-play to attain high levels of performance at StarCraft II. In particular, we are particularly interested in "micro-battles," a term used to colloquially describe two players that micro-manage their respective units in a battle engagement. At a high level, a single battle engagement will consist of (1) a learning agent and (2) a fixed agent which may either be scripted or perhaps a previous iteration of a learning agent. We will provide each agent a randomly chosen army, and the agents will micro-manage their respective armies in battle.

We hypothesize that it is likely important to ensure that both armies are of roughly similar strength to boost learning speed. For example – if a learning agent, who has a substantially stronger army, wins against a fixed agent, the learning agent may not be easily able to ascribe positive or negative value to actions, since the outcome of the battle was muddled by relative army strengths.

To generate armies, we define a fixed budget (e.g., on minerals, gas, and supply), and aim to random sample two *Pareto-optimal armies* within this space; we then assign one army to each side. A *Pareto-optimal army* is an army where no further units may be added without exceeding the given budget. The question is – how do we randomly sample a Pareto-optimal army given a number of budget constraints?

Simple Sampling and Biased Armies

Let us consider a simple sampling method. Given, `m` minerals, `g` gas, and `s` supply, we may run the following to draw a sample from all possible armies:

```
def draw_sample(m, g, s, available_units):
    army = []
    while m > 0 and g > 0 and s > 0:
        random_unit = sample_random_unit(available_units, m, g, s)
        army.append(random_unit)
        m -= random_unit.mineral_cost
        g -= random_unit.gas_cost
        s -= random_unit.supply_cost
    return army
```

Here, the function `sample_random_unit` returns a randomly selected unit which costs at most `m` minerals, `g` gas, and `s` supply. The function `draw_sample` certainly yields a

Pareto-optimal army. However, we can easily show that `draw_sample` does not draw samples uniformly at random from the set of all possible armies.

For example, consider the case where we are given 1000 minerals, 250 gas, and 20 supply, and the available units are Adepts and Marauders, which each cost an identical 100 minerals and 25 gas. There are clearly 11 possible armies; army `i` consists of `i` adepts and `10 - i` marauders, where `i` ranges from 0 to 10. However, `draw_sample` does not draw each of these armies with equal probability. Instead, the probabilities are listed below:

# of Adepts	# of Marauders	Probability
0	10	0.0977%
1	9	0.9766%
2	8	4.3945%
3	7	11.7188%
4	6	20.5078%
5	5	24.6094%
6	4	20.5078%
7	3	11.7188%
8	2	4.3945%
9	1	0.9766%
10	0	0.0977%

In this case, `draw_sample` is largely biased towards sampling armies with equal numbers of adepts and marauders; indeed, around 65% of the time, we draw either 4, 5, or 6 adepts. We very rarely draw armies purely consisting of marines or adepts.

But why is uniform sampling important? To rephrase – why is biased sampling bad? If we employ `draw_sample`, learning agents will encounter armies with equal unit compositions a lot more frequently than imbalanced unit compositions, likely leading RL controllers to perform poorly with armies consisting of imbalanced unit compositions.

To avoid this bias, we detail an algorithm that draws samples from all possible Pareto-optimal armies that satisfy a number of user-specified budget constraints, uniformly at random.

Approach

Suppose that we are given n available units that our army may comprise of. Let x_j denote the number of unit j that we pick for our army. A user may define m linear budget constraints, which may be summarized as follows:

$$\sum_{j=1}^{n} A_{ij}x_j \leq b_i$$

where $i \in \{1, 2, \cdots, m\}$. Here, the matrix A encodes details such as unit costs (minerals, gas, supply) while the vector b encodes the budget constraints.[1] The problem we aspire to solve effectively reduces to uniformly sampling at random a Pareto-optimal[2] lattice point x in the polytope defined by the region:

$$Ax \leq b, x \geq 0$$

To the best of our knowledge, we are unaware of any previously published work that accomplishes this task. The closest work we could find is Barvinok's algorithm, which for a fixed dimension d, calculates the number of lattice points in a rational polyhedron via generating functions.

Eliminating the Pareto-Optimal Constraint

For a solution to be Pareto-optimal, at least one of the constraints in A must be tight. Suppose that, without loss of generality, constraint m is tight while the other constraints may or may not be tight. We may introduce slack variables s_i for $i \in \{1, 2, 3, \cdots, m-1\}$ and reformulate our problem as follows:

$$s_i + \sum_{j=1}^{n} A_{ij}x_j = b_i$$

$$\sum_{j=1}^{n} A_{mj}x_j = b_m$$

where $i \in \{1, 2, \cdots, m\}$.

If we can uniformly sample solutions at random that satisfy this system of equations, we will have sampled a Pareto-optimal solution that in particular, makes the last constraint tight.

To sample Pareto-optimal solutions that may make any of the constraints tight, we can:

---

[1] In this paper, we assume that all entries of A and b are non-negative.

[2] As earlier, by Pareto-optimal, we mean that incrementing any x_j would cause a violation of the constraint $Ax \leq b$.

1. Create a list of m separate systems of equations, $S_1, S_2, \cdots, S_m$; the system of equations S_i will make a constraint i tight.
2. To draw a sample:
    a. The counter: pick a random S_i where the probability of drawing S_i is proportional to the total number of possible solutions to S_i for each i.
    b. The sampler: given S_i, draw a sample uniformly at random which satisfies S_i.

For ease in exposition, we address a number of edge-cases to make this algorithm precisely correct in the Appendix. We now detail algorithms for the counter and the sampler.

The Counter and The Sampler

We need to be able to count the number of positive integral solutions that satisfy the linear system of equations:

$$s_i + \sum_{j=1}^{n} A_{ij} x_j = b_i$$

$$\sum_{j=1}^{n} A_{mj} x_j = b_m$$

We propose a dynamic programming-based solution. For ease, we walk through an example, which may be more relatable than math. Consider the system:

$$\begin{bmatrix} 3 & 2 & 1 & 1 & 0 & 0 \\ 1 & 7 & 2 & 1 & 1 & 0 \\ 2 & 5 & 6 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 10 \\ 17 \\ 12 \end{bmatrix}$$

Notice here that the first constraint is set to be tight while the second and third constraints have slack variables. We consider subproblems of the form $T(c, b_1, b_2, b_3)$, where $T(c, b_1, b_2, b_3)$ counts the number of solutions to the original problem, keeping only columns c and to the right of the constraint matrix, and replacing the entries of b with $b_{1\ldots3}$. For example, $T(3, b_1, b_2, b_3)$ counts the number of solutions to the problem:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 2 & 1 & 1 & 0 \\ 6 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \\ s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

We can show that $T(c, \vec{b}) = \sum_{t=1}^{\min_k b_k / A_{k,c}} T(c+1, \vec{b} - t \cdot A_{\cdot,c})$. We initially start by setting c to the rightmost column of the constraints matrix, obtaining one solution for the slack variable. By moving left, we can build the total number of solutions found by subtracting away increasingly larger column contributions from the current column. This allows us to determine the total number of solutions for the overall system.

For a more detailed explanation for a simpler case, see this StackExchange post; the method we detailed here is a generalization of this post to more constraints.

Sampling is now easy given that we can count – we simply need to maintain a data structure that stores backtracking information at each column. In particular, to sample, we start with the first column, $c = 1$. We obtain the counts associated with each value of t that we iterated over for $c = 1$. We draw a weighted sample for the value at $c = 1$ proportional to the counts, and then enter the subproblem for $c = 2$ and repeat until all variables have values.

Appendix

Here, we detail a number of edge cases in the sampling mechanism described in "Eliminating the Pareto-Optimal Constraint."

Exact Tightness May Yield No Solutions

The astute reader may notice that there may be an issue with our tightness constraint. For example – what happens if the user specifies a mineral constraint of 101 minerals? Because StarCraft II units cost a multiple of 5 minerals, it is impossible to craft an army that achieves perfect tightness for this minerals constraint.

Assume that we are making constraint m tight. To work with this issue, we split the problem into further systems of equations $\{S_m^k\}_{k=1,2,\ldots}, \cdots$ where each system of equations S_k is identical to S_m other than the fact that b_m in system S_k is replaced with a value that may be obtained with positive linear combinations of values in the row A_m that is strictly greater than $b_m - \min A_m$ and at most b_m. Doing this maintains Pareto-optimality. After running this step, we simply replace S_m with $\{S_m^k\}_{k=1,2,\ldots}$ in the list of systems of equations in step (1).

Zero-Valued Constraints May Yield non-Pareto-Optimal Solutions

After generating all systems of equations in (1) in the above sampling algorithm, it is crucial to discard systems of equations for which $\boldsymbol{b}$ is zero. Otherwise, we may end up with peculiar scenarios where we generate non-Pareto-optimal solutions.

For example, suppose that we have a mineral budget of 50 and a gas budget of 0 and are trying to generate a Zerg army. Zerglings are the only constructible unit with this budget, where a single Zergling costs 25 minerals and 0 gas. Here, there are two tightness cases:

1. In the case where we enforce tightness on the mineral budget and keep slackness on the gas budget, we end up with a single valid army of 2 Zerglings.
2. In the case where we enforce tightness on the gas budget and keep slackness on the mineral budget, we end up with 2 valid armies: (1) a single Zergling, and (2) two Zerglings.

Eliminating systems of equations for which $\boldsymbol{b}$ is zero solves this issue.

Armies that Satisfy Multiple Systems of Equations Tightly may be Oversampled

The example in the previous case highlights another issue with our sampling approach; armies that tightly satisfy multiple systems of equations may be oversampled. In particular, the same army composition of two Zerglings appears as a solution to two systems of equations. While this was the only valid Pareto-optimal army in the above case, this may not necessarily be the case more generally.

To avoid oversampling armies that tightly satisfy multiple equations, we first enumerate all subsets of budget constraints[3]; for each subset, we count the number of solutions for which that subset of budget constraints can be made tight while the other budget constraints are slack. Using the Principle of Inclusion and Exclusion, we can precisely count the number of solutions that makes each constraint tight or not-tight. Once we have the solution count per region, we can then (1) pick a region at random proportional to the number of solutions in that region, (2) construct a system of equations as described in the above section where we additionally enforce the slack variables to have value at least 1, and (3) sample a solution from within this region.

[3] The number of budget constraints is usually small (< 5), so this is not a problem despite the exponential growth of the number of subsets.