Dynamic Programming Challenge

Simone Licciardi & Samuele Cipriani
For typos: `slicciardi@student.ethz.ch`

November 27, 2025

We detail our solution report in two sections: the details of our implementation of the DPA algorithm, and the structural characteristics of the problem that we exploited. We also attached a section about **super cool but too fancy** attempts.

Problem Structure

The P matrix is **sparse** and has structure, but **cannot be written neither as a block-matrix or banded matrix** in an implementation-friendly way. Moreover, the problem has no trivial pre-conditioner[1].

Write $\mathcal{X} \ni i = (i_{YV}, i_{DH})$ for $i_{YV} \in \mathbb{Z}^2, i_{DH} \in \mathbb{Z}^{2M}$. The structure of P we used is

$$P(i,j|u) = P^{(YV)}(i_{YV}, j_{YV}|u,c)P^{(DH)}(i_{DH}, j_{DH}|u,c)P^{(coll)}(c,i,j) = \\ = P^{(YV)}(i_{YV}, j_{YV}|u)P^{(DH)}(i_{DH}, j_{DH})P^{(coll)}(i). \quad (1)$$

Namely, we used this bits of the problem's structure: that the transition in the space of DH coordinates is independent of u, that assuming no collision happened the transition in DH coordinates only depends on the DH coordinates of i, j, the transition in YV coordinates only depends on the YV coordinates, and that whether a collision happens only depends on the initial state i.

Therefore, we only computed these three matrices (in a vectorized way), and assembled P out of these using vectorized operations. We chose to compute P in `csr_matrix` format to make the downstream matrix multiplications faster.

Solver

Our numerical solver for this optimization problem is a mixed implementation of Value Iteration and Policy Iteration, which tries to get the best of the two.

Note that Policy Iteration's cost is linear only if you can solve sparse systems fast. This happens if P is also structured (block-matrix or banded), or if you have an excellent pre-conditioner (see footnote). Neither applied.

[1] *Purely algebraic approaches which simply take the numerical matrix entries as input can have merit in situations where little is known about the underlying problem they certainly have the useful property that it is usually possible to apply such a method but the generated pre-conditioners can be poor.* - A well-known review on the matter.

Instead, Value Iteration is linear if you can do sparse matrix multiplications very fast: that is the case, as it only requires the (proper) sparsity structure.

On the other hand, Policy Iteration converges very fast if your initialization is good enough, and Value Iteration's optimal value convergence is slow and the greedy policy becomes optimal with a rough estimate of the value already.

Our idea was to iterate the following: run a few Value Iterations, enough to get a rough estimate of J, compute the greedy policy and then a single Policy Evaluation step. If the estimate o J is good enough, the output of this procedure is the exact optimal cost.

An important detail of the implementation is that we do *not* take 2 Policy Iteration steps, which is the minimum number in order to apply PI's termination criterium. The reason for this is that due to the precedent considerations, a single Policy Iteration step has a time-cost equivalent to 10-20 Value Iteration steps. Therefore, our termination criterium relies on the fact that if the cost of the first policy iteration was optimal, then applying the successive round of Value Iterations would satisfy immediately the tolerance-bound exit condition. Empirically this reduced the number of Policy Evaluations (the costly part of the Policy Iteration) that we performed, and the total compute time.

Algorithm 1: Hybrid Value-Policy Iteration

Input: C (problem constants), N_{val} (PI frequency)
Output: J^* (optimal cost), μ^* (optimal policy)
Compute $P^{(u)} \in \mathbb{R}^{K\times K}$ for $u \in \{0,1,2\}$ and $Q \in \mathbb{R}^3$;
Initialize $J \leftarrow -50 \cdot \mathbf{1}_K$, $k \leftarrow 0$;
while $\|J^{(k)} - J^{(k-1)}\|_\infty > \epsilon$ **do**
  `// Value Iteration Step`
  $J^{(k+1)} \leftarrow \min_{u\in\{0,1,2\}} \left\{P^{(u)}J^{(k)} + Q_u\mathbf{1}_K\right\}$;
  $k \leftarrow k+1$;
  if $k \bmod N_{val} = 0$ **then**
    `// Policy Improvement`
    $\mu(i) \leftarrow \arg\min_u \left\{P^{(u)}J^{(k)} + Q_u\mathbf{1}_K\right\}_i$ for all $i \in \mathcal{X}$;
    `// Policy Evaluation`
    Solve $(I - P^{(\mu)})J^{(k+1)} = Q_\mu$;
  end
end
Extract $\mu^* \leftarrow \arg\min_u\{P^{(u)}J^{(k)} + Q_u\mathbf{1}_K\}$;
return $J^{(k)}, \mu^*$

Fancy Attempts

Firstly, we noted that if you initialize J low enough, say -50, it takes only a few (5, in our tests) iteration of VI to determine the *quasi-terminal* states. Namely, those with $-1, \ldots, -4$ values; these correspond to the cost of a state that is 1-timestep before certain end, and so on. We though about isolating these indexes and only iterate the solver over the remaining state space. We also though about guessing the J by setting the rest to a parameters-tailored lower value (say -1300 for the `main.py` problem). The overhead seemed well worth it for larger problems (10% compute time reduction) but made smaller problems worse. not knowing anything about the evaluation set, and since we couldn't test this enough, we preferred avoid using it in the actual submission.

Secondly, we noted that

$$P(u)J_{opt}^{(k)} = P^{(YV)}(u)(J_{opt}^{(k)}\,\mathrm{Transp}(P^{(DH)})) * P^{(coll)},$$

which promised to reduce the number of total floating-point operations by pre-computing the right parenthesis, and avoiding the cost of computing P. At the end of the day, computing P accounted for less than 1% of our cost, and the overhead of a sparse matrix multiplication (due to the different data structure) resulted in worsening time compute.