Dynamic Programming for Battery Energy Storage Arbitrage Scheduling Considering Capacity Degradation Cost

Made Branenda Jordhy - 13524026
Informatics Engineering
School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Ganesha 10 Bandung 40132, Indonesia
13524026@std.stei.itb.ac.id, ethgalleryin@gmail.com

***Abstract*—A battery storage owner can earn money from price arbitrage. The owner charges the battery when electricity is cheap and discharges it when electricity is expensive. Each charge and discharge wears the battery a little, and this wear is a real cost. A simple buy low and sell high rule ignores this cost and cycles the battery too much. This paper applies dynamic programming to schedule a single battery for one day at a time. The state is the level of charge, the stages are the hourly slots, and the Bellman recurrence is solved by backward induction. The key idea is to add a linear degradation cost to the reward, which keeps the problem additive and solvable in $O(T\,N_s\,N_a)$ time. We test the method on one year of real German day ahead prices. Scored at the real wear cost, the degradation aware schedule earns a positive yearly profit, while a greedy rule and a dynamic program that ignores wear both end with a loss. The method keeps about 41 percent of the gross revenue while cutting battery throughput by about 86 percent. We also confirm the linear runtime in the horizon length.**

***Index Terms*—dynamic programming, energy storage, battery arbitrage, degradation cost, optimal scheduling**

I. Introduction

Electricity prices change a lot during the day. They are often low at night and high in the evening. A Battery Energy Storage System (BESS) can use this change to make money. The owner buys energy and stores it when the price is low. Later the owner sells the stored energy when the price is high. This is called energy arbitrage.

There is a hidden cost in this plan. Every time the battery charges or discharges, it loses a small part of its capacity. This slow loss is called capacity fade. Over many cycles the battery becomes weaker and must be replaced sooner. The cost of this wear is real money. A naive plan that only looks at the price gap will cycle the battery many times. The gross revenue looks large, but the wear cost can eat all of it and more.

This paper studies how to schedule one battery for one day. We want to maximize the net profit, which is the revenue minus the cost of energy bought minus the wear cost. We use dynamic programming (DP) because the problem has the two features that DP needs. First, it has optimal substructure. The best plan from a given hour and a given charge level depends only on the best plan for the rest of the day. Second, it has overlapping subproblems. The same charge level appears again and again across many possible paths.

Our main contribution is simple and clear. We put the degradation cost directly into the reward of the DP. We use a linear cost based on energy throughput. This keeps the reward additive and keeps the charge level as the only state, so the standard DP still works. We then show, on real market data, that this small change turns a losing strategy into a profitable one. We compare three methods. The first is our DP with the wear cost. The second is a greedy rule with no foresight. The third is a DP that ignores wear. To be fair, we score all three with the same real wear cost. Only the wear aware DP makes a profit.

The rest of the paper is organized as follows. Section II explains the theory of dynamic programming. Section III reviews related work. Section IV states the problem. Section V gives the DP formulation and the algorithm. Section VI describes the baselines. Section VII gives the data, the setup, and the results. Section VIII lists the limits, and Section IX concludes.

II. Theoretical Background

This section explains dynamic programming in plain terms, following the course lecture notes [1], [2]. Dynamic programming, or DP, is a method to solve a problem by breaking it into smaller subproblems. It solves each subproblem once and stores the answer. It then builds the answer to the full problem from the stored answers. DP works well when a problem has two features. The first is optimal substructure. The second is overlapping subproblems. We explain both, then we contrast DP with the greedy method.

A. Optimal Substructure

A problem has optimal substructure when an optimal solution contains optimal solutions to its subproblems. In our case, suppose we know the best plan for the whole day. Take any hour t and the charge level s at that hour. The part of the plan from hour t to the end must itself be the best possible plan that starts from s at hour t. If it were not, we could swap in a better tail and improve the whole plan, which is a contradiction. So the best plan for a long horizon is built from the best plans for

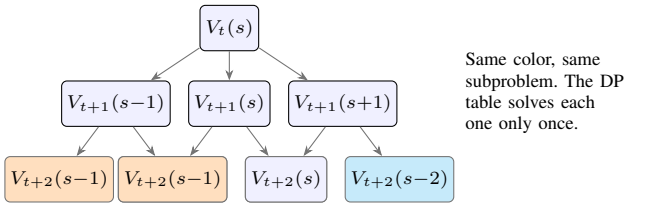

Fig. 1. Overlapping subproblems in the value recursion. A plain recursion reaches the same state at the same stage along different branches. Nodes with the same color are the same subproblem. The DP table stores each one once, so the work drops from a search over paths to a loop over states.

shorter horizons. This is what lets us define a value function and solve it step by step.

B. The Principle of Optimality and the Bellman Equation

Bellman stated this idea as the principle of optimality [3]. It says that whatever the first decision is, the remaining decisions must form an optimal plan for the state that results from the first decision. We turn this idea into an equation. Let $V_t(s)$ be the best total reward we can still earn from stage t when the state is s. The principle gives a recurrence,

$$V_t(s) = \max_a \big[\, R_t(s,a) + V_{t+1}(\,\mathrm{next}(s,a)\,)\,\big]. \tag{1}$$

The term $R_t(s,a)$ is the reward of taking action a now. The term $V_{t+1}(\cdot)$ is the best reward for the rest of the horizon. We add them and take the action that gives the most. This single equation is the heart of DP. We solve it from the last stage back to the first. This order is called backward induction. We fill a table of values, one cell per state per stage. Because we go backward, the future values we need are always ready.

C. Overlapping Subproblems

A problem has overlapping subproblems when the same subproblem appears many times. If we solved the recurrence by plain recursion, we would compute the same $V_t(s)$ again and again. Fig. 1 shows why. Starting from one state, the recursion branches into actions, and different branches soon reach the same state at the same stage. The number of distinct states is small and bounded, but the number of paths grows fast. DP avoids the waste. It stores each $V_t(s)$ in a table and reads it back when needed. So each subproblem is solved only once. This turns an exponential search over paths into a loop over a small table.

D. Why Not Greedy

A greedy method makes the choice that looks best right now and never looks back. Greedy is fast and simple. It is correct only for special problems where a local best choice is always part of a global best plan. Battery arbitrage is not such a problem. The best action in one hour depends on prices in later hours and on the wear cost over the whole day. A greedy rule that sells at the first high price may miss a higher price later, or it may cycle the battery for a small gain that the wear cost wipes out. DP looks at the whole horizon at once through the value function, so it does not fall into these traps. This is the main reason we choose DP for this problem, and the experiments in Section VII show the gap in real money.

III. Related Work

Dynamic programming is a classic method for sequential decisions [3]. Standard algorithm textbooks present it as a way to solve problems with optimal substructure and overlapping subproblems [4]. Battery arbitrage is a natural fit because the decision in each slot depends on the charge level, which is a small and bounded state.

Several works study battery scheduling in electricity markets. Jiang and Powell use approximate dynamic programming for hour ahead bidding with storage in the real time market [5]. Other works frame arbitrage as a linear or mixed integer program and add operational limits or revenue stacking [6]. Some works include market impact and solve the problem with dynamic programming for a price maker [7]. Recent work studies arbitrage profit when efficiency and degradation both change over time, and uses linear and nonlinear programs [8].

Our work is smaller in scope but clear in message. We keep the model simple and exact. We use a plain backward induction over a discrete charge level. We focus on one teaching point. A small additive wear term inside the reward is enough to fix the over cycling problem, and the method stays in the basic DP form taught in an algorithms course.

IV. Problem Statement

We schedule one battery over one day. The day has T time slots. Each slot lasts Δt hours. In this study $T = 24$ and $\Delta t = 1$ hour. The price in slot t is p_t in EUR per MWh. We assume the prices are known in advance. This is the perfect foresight case, which is common as a first model.

The battery has a usable energy capacity $E_{\max}$ in MWh. It has a charge power limit $P_{\text{chg}}^{\max}$ and a discharge power limit $P_{\text{dis}}^{\max}$, both in MW. So the most energy that can move in one slot is the power limit times Δt. The battery has a round trip efficiency η between 0 and 1. We split it into a one way efficiency $\eta_1 = \sqrt{\eta}$ for charge and the same for discharge.

The state of charge (SoC) is the energy stored in the battery. We write it as s. The action in each slot moves energy in or out. We use a simple energy rule. When charging by an SoC amount g, the grid energy bought is g/η_1. When discharging by an SoC amount h, the grid energy sold is $\eta_1 h$. The SoC update is

$$\text{charge:} \quad s_{t+1} = s_t + g_t, \tag{2}$$

$$\text{discharge:} \quad s_{t+1} = s_t - h_t, \tag{3}$$

$$\text{idle:} \quad s_{t+1} = s_t, \tag{4}$$

with the SoC kept in $[0, E_{\max}]$ at all times.

We want to maximize the net profit over the day,

$$\begin{aligned}\text{profit} = \sum_{t=0}^{T-1} \Big[\, & p_t\,(\text{energy sold})_t \\ & - p_t\,(\text{energy bought})_t - C_{\text{deg}}(a_t)\,\Big],\end{aligned} \tag{5}$$

subject to the SoC dynamics and the power limits. Here a_t is the action in slot t and C_{deg} is the wear cost of that action.

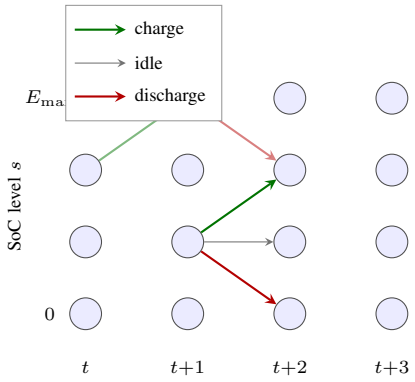

Fig. 2. The DP state and transition lattice. Stages run left to right. Discrete SoC levels run bottom to top. From one state, charging moves up, discharging moves down, and idle stays level. The power limit bounds how far a single step can move.

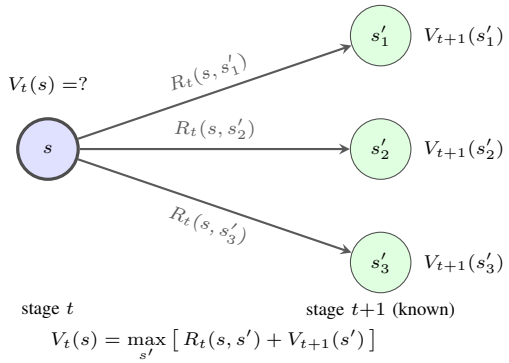

Fig. 3. One backward induction step. The values at stage $t{+}1$ are already known. The value at stage t is the best over the feasible actions of the slot reward plus the known future value.

V. Dynamic Programming Formulation

A. State, Stage, and Action

The stage is the time index t, from 0 to T. The state is the SoC. We make the SoC discrete. We split the range $[0, E_{\max}]$ into N_s equal levels. The step size is $\delta = E_{\max}/(N_s - 1)$. A state is the index of a level. The action in slot t is the choice of the next SoC level. The change in level fixes the charge or discharge energy. The power limits set how many levels we can move up or down in one slot. So the action set in each state is a small window of nearby levels. We call its size N_a. Because we pick the next level on the grid, every transition lands exactly on a grid point. We do not need any interpolation. Fig. 2 shows the idea.

B. Reward

The reward of a transition is the money it earns in that slot. Let the change in SoC be $\Delta = s' - s$. If $\Delta > 0$ the battery charges, so it buys Δ/η_1 units of energy and the price effect is $-p_t\,\Delta/\eta_1$. If $\Delta < 0$ the battery discharges by $|\Delta|$, so it sells $\eta_1|\Delta|$ units and the price effect is $+p_t\,\eta_1|\Delta|$. We then subtract the wear cost. The reward is

$$R_t(s, s') = (\text{price effect}) - c_{\text{deg}}\,\big(g + h\big), \tag{6}$$

where g and h are the charge and discharge amounts on the SoC side, and at most one of them is nonzero in a slot.

C. Degradation Cost

The wear cost is the heart of this paper. We use a linear throughput model,

$$C_{\text{deg}}(a) = c_{\text{deg}}\,(g + h). \tag{7}$$

The term $g + h$ is the energy that moves through the battery in that slot on the SoC side. The factor c_{deg} is the marginal wear cost per MWh of throughput. We get it from the battery capital cost divided by the total energy the battery can move over its life. This model has two good properties. It is additive over slots, and it depends only on the current action. So it fits the DP with the SoC as the only state, and it does not break the Markov property.

D. Bellman Recurrence and Backward Induction

Let $V_t(s)$ be the best profit we can still earn from stage t with SoC level s. The recurrence is

$$V_t(s) = \max_{s' \in \mathcal{A}(s)} \big[\, R_t(s, s') + V_{t+1}(s') \,\big], \tag{8}$$

where $\mathcal{A}(s)$ is the set of feasible next levels under the power limits. The terminal value is

$$V_T(s) = \begin{cases} 0 & \text{if } s \geq s^{\text{end}}_{\min}, \\ -\infty & \text{otherwise.} \end{cases} \tag{9}$$

We require the end SoC to be at least a set minimum. This stops the solver from draining the battery for free at the end of the day. We solve the recurrence by backward induction. We fill the table from $t = T$ down to $t = 0$. Fig. 3 shows one step. We store the best next level in a policy table. Then we walk forward from the start SoC and read the policy table to rebuild the schedule. Algorithm 1 gives the full method.

E. Complexity

For each stage we loop over N_s states. For each state we loop over N_a feasible next levels. So one stage costs $O(N_s\,N_a)$. Over T stages the total cost is

$$O(T\,N_s\,N_a). \tag{10}$$

This is linear in the horizon length and linear in the number of states. The window size N_a comes from the power limit. If the power limit is a fixed fraction of the capacity, then N_a grows with N_s, so the cost grows like $O(T\,N_s^2)$ as the grid gets finer. We check both effects in the experiments.

F. A Small Worked Example

A tiny example shows how the table fills. Take a battery with $E_{\max} = 1$ MWh and three SoC levels, so the levels are 0, 0.5, and 1.0 MWh. Let the power limit allow a move of one level per slot. Set the efficiency to one and the wear cost to zero, to keep the numbers clean. Take three slots with prices $p_0 = 10$, $p_1 = 50$, and $p_2 = 20$ EUR per MWh. The terminal value is $V_3(s) = 0$ for every level.

Algorithm 1 Battery arbitrage by dynamic programming

Input: prices $p_{0..T-1}$, battery params, N_s, start level s_0, end minimum $s_{\min}^{\text{end}}$
Output: optimal schedule and net profit
for $s = 0$ to $N_s - 1$ **do**
    $V_T(s) \leftarrow 0$ if $s \geq s_{\min}^{\text{end}}$ else $-\infty$
end for
for $t = T - 1$ down to 0 **do**
    for $s = 0$ to $N_s - 1$ **do**
        $V_t(s) \leftarrow -\infty$
        for s' in feasible window of s **do**
            $v \leftarrow R_t(s, s') + V_{t+1}(s')$
            if $v > V_t(s)$ **then**
                $V_t(s) \leftarrow v, \quad P_t(s) \leftarrow s'$
            end if
        end for
    end for
end for
$s \leftarrow s_0$
for $t = 0$ to $T - 1$ **do**
    record action from s to $P_t(s)$, $\quad s \leftarrow P_t(s)$
end for

We work backward. In slot $t = 2$ at price 20, from the middle level the best move is to hold, since selling at 20 first needs the battery to have charged earlier. The values become $V_2(0) = 0$, $V_2(0.5) = 0$, and $V_2(1.0) = 0$ for the hold action, while a sell at this low price is not yet useful on its own. In slot $t = 1$ at the high price 50, selling is very good. From the full level we can sell 0.5 MWh and earn $50 \times 0.5 = 25$. So $V_1(1.0) = 25$ and the best action is to discharge. From the middle level we can also sell and reach $V_1(0.5) = 25$ if it was reached with stored energy. In slot $t = 0$ at the low price 10, buying is cheap. From the empty level we can charge 0.5 MWh at a cost of $10 \times 0.5 = 5$, move to the middle level, and then collect the future value. The recurrence in (8) picks the buy now and sell later path. The policy table stores these moves. The forward walk then reads the table and returns the schedule. This is the same buy low and sell high logic, but the DP finds it by global search, not by a local rule.

Fig. 4 shows the real value table for one day from our solver. Each column is a stage and each row is an SoC level. The color is the value to go $V_t(s)$, the best profit still possible from that cell. The black line is the optimal path that the forward walk rebuilds from the policy table. The path charges to a high SoC in the cheap morning, holds, then discharges into the expensive hours. This is the DP table filled with real numbers.

VI. Baseline Methods

We compare the DP against two baselines.

The first baseline is a greedy threshold rule. It looks at the price in the current slot only. If the price is below a low threshold, it charges as much as the power limit and the free space allow. If the price is above a high threshold, it discharges as much as allowed. Otherwise it stays idle. The two thresholds

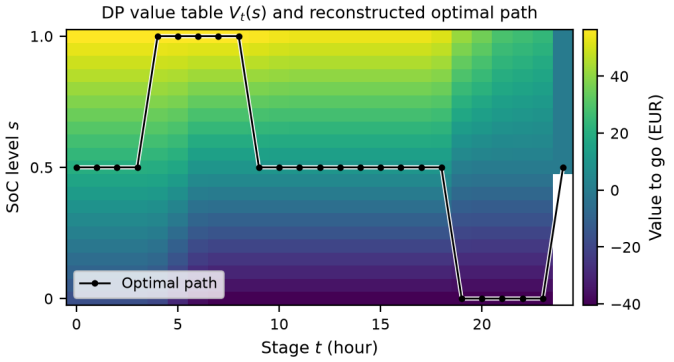

Fig. 4. The DP value table for one example day. The horizontal axis is the stage t. The vertical axis is the SoC level. The color is the value to go $V_t(s)$. The black line is the optimal path rebuilt from the policy table. It charges in the cheap hours and discharges in the costly ones.

are price quantiles of the same day. We use the 30th and the 70th percentile. This rule is myopic. It has no foresight and it ignores the wear cost when it decides.

The second baseline is a DP that ignores wear. It is the same solver as in Section V, but with $c_{\text{deg}} = 0$ in the planning reward. This isolates the effect of the wear term. It shows what happens when we plan only for gross revenue.

To compare the three methods in a fair way, we score all of them with the same real wear cost. Each method picks its own actions. Then we charge every schedule the same real cost per MWh of throughput. This gives a true net profit for each method, even for the method that ignored wear when it planned.

VII. Experiments and Results

A. Data

We use real day ahead electricity prices from Open Power System Data [9]. We take the German and Luxembourg bidding zone for the calendar year 2019. The prices are hourly and in EUR per MWh. We keep only full days with all 24 hours present, which gives 364 days. The mean price is about 37.7 EUR per MWh. The mean daily spread between the highest and lowest hour is about 30.2 EUR per MWh. Some hours have negative prices, which is a real feature of this market when supply is high. A Python script downloads the data and reshapes it into one row per day. The C++ solver reads this file. Fig. 5 shows the data. The left panel is the mean price for each hour of the day. It has the typical shape, low at night and two peaks, one in the morning and a larger one in the evening. This shape is what arbitrage exploits. The right panel is the price for every day and hour of the year. The red bands at the morning and evening hours are the expensive times. The few blue marks are the negative price hours.

B. Setup

We use a normalized battery of $E_{\max} = 1$ MWh. The charge and discharge power limits are both 0.5 MW, which is a 0.5 C rate. The round trip efficiency is $\eta = 0.86$. The real wear cost is $c_{\text{deg}} = 12$ EUR per MWh of throughput, which is in the

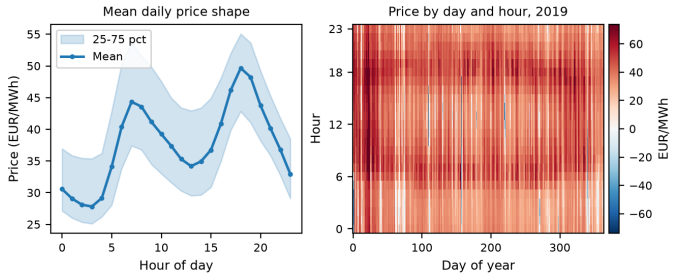

Fig. 5. The price data for 2019. Left: the mean price by hour of day, with a band for the 25th to 75th percentile. The double peak shape is what arbitrage uses. Right: a heatmap of price by day and hour over the year. Red is expensive, blue is the rare negative price.

TABLE I
E1: YEARLY TOTALS OVER 364 DAYS, SCORED AT THE REAL WEAR COST $c_{\text{DEG}} = 12$ EUR/MWH. BEST TRUE NET PROFIT IN BOLD.

Method	True net (EUR)	Gross (EUR)	Wear cost (EUR)	Cycles
DP with wear (ours)	**1389.6**	3555.6	2166.0	90.3
DP without wear	−6924.7	8645.3	15570.0	648.8
Greedy threshold	−4996.6	6397.4	11394.0	474.8

range used in storage studies. The battery starts each day half full and must end at least half full. The DP uses $N_s = 101$ SoC levels in the main runs. The solver is written in C++17. We run four experiments.

C. E1: Main Comparison

Table I shows the yearly totals for the three methods, all scored at the real wear cost. The result is clear. The wear aware DP earns a positive yearly true net profit of about 1390 EUR. The greedy rule and the wear blind DP both end with a loss. The wear blind DP earns the largest gross revenue, about 8645 EUR, but it cycles the battery 648 times in the year. The wear cost of all that cycling is larger than the revenue, so the true net is about −6925 EUR. The greedy rule sits in the middle and still loses money.

The trade is easy to read. The wear aware DP keeps about 41 percent of the gross revenue of the wear blind DP. At the same time it cuts the battery throughput by about 86 percent. It gives up some gross revenue but saves much more in wear, so the net profit is higher. This is the central result of the paper.

Fig. 6 shows why foresight matters, using the greedy rule and the DP on the same day. The greedy rule sells at the first high prices in the morning, so the battery is empty by the time the largest evening peak arrives. The DP holds its energy and sells into that evening peak instead. Both use one full cycle on this day, but the DP earns more because it places the same energy at a better time. The plan that looks ahead wins.

Fig. 7 shows one example day. The DP charges in the cheap early morning hours and discharges into the two evening price peaks. It does not cycle on small price gaps because the wear cost is not worth it. The SoC path stays smooth and uses only a few clean cycles.

TABLE II
E3: EFFECT OF THE SOC GRID SIZE ON YEARLY NET PROFIT AND RUNTIME.

N_s	True net (EUR)	Runtime (ms/day)
11	1389.6	0.005
21	1389.6	0.012
41	1389.6	0.048
101	1389.6	0.260
201	1389.6	0.916
401	1389.6	3.439

D. E2: Sensitivity to the Wear Cost

In this experiment we keep the real wear cost fixed at 12 EUR per MWh. We only change the wear cost that the DP plans with. We then score the schedule at the real cost. Fig. 8 shows the result. When the planned cost is too low, the DP cycles too much and the true net is low or negative. When the planned cost is too high, the DP is too shy and gives up good trades. The best true net is reached when the planned cost equals the real cost of 12 EUR per MWh. This matches the theory. The number of yearly cycles falls in a smooth way as the planned cost rises. This shows that the wear term is a clean knob that trades revenue for fewer cycles.

E. E3: Discretization Granularity

We vary the number of SoC levels N_s from 11 to 401 and measure the yearly true net and the runtime per day. Table II shows the result. In this setup the net profit does not change with the grid size. The reason is that the power limit and the start and end levels all line up with the grid at every resolution we test, so the optimal actions are already reachable on the coarse grid. The runtime grows fast as the grid gets finer, in line with the $O(N_s^2)$ growth from Section V. So a coarse grid is enough here, and it is also the fastest.

F. E4: Empirical Runtime

We confirm the complexity claim. We build longer horizons by joining days end to end. Fig. 9 shows two plots. In plot (a) we fix the grid at $N_s = 101$ and grow the horizon T. The runtime rises in a straight line with T, as the $O(T\,N_s\,N_a)$ formula predicts. In plot (b) we fix $T = 168$ and grow N_s. The runtime rises faster than linear, because the action window N_a grows together with N_s under a fixed power limit. This matches the $O(N_s^2)$ growth. Both plots agree with the theory.

G. Discussion

The experiments support the thesis. A small additive wear term inside the DP reward turns a losing arbitrage strategy into a profitable one. The greedy rule fails because it has no foresight and ignores wear. The wear blind DP fails because it chases gross revenue and cycles too much. The wear aware DP wins because it values each cycle correctly. The method is also fast. One day is solved in well under a millisecond on

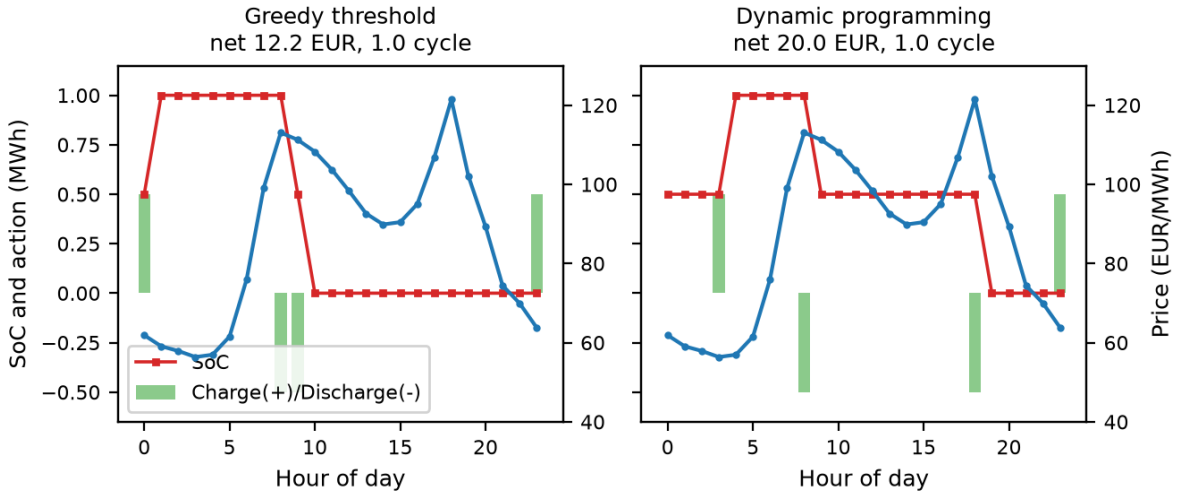

Fig. 6. Greedy rule versus dynamic programming on the same day. Blue is the price, red is the SoC, green bars are the net action. The greedy rule sells too early and misses the largest evening peak. The DP holds its energy and sells into that peak, so it earns more from the same single cycle.

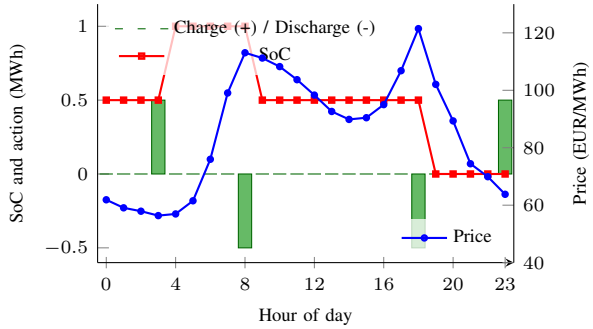

Fig. 7. One example day with the wear aware DP. The blue line is the price on the left axis. The red squares are the SoC on the right axis. The green bars are the net action, positive for charge and negative for discharge. The battery charges in the cheap trough and discharges into the price peaks.

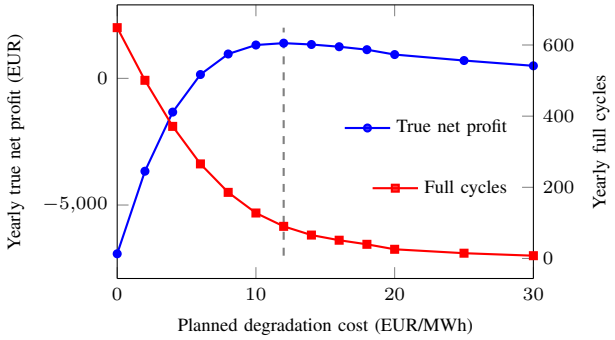

Fig. 8. E2: effect of the planned wear cost. The blue line is the yearly true net profit on the left axis. The red line is the yearly number of full cycles on the right axis. The dashed vertical line marks the real wear cost of 12 EUR per MWh, where the true net is highest.

a coarse grid, so a full year of days is solved in a fraction of a second.

The per day picture is just as clear. Fig. 10 shows the spread of daily profit for the three methods over the 364 days. The

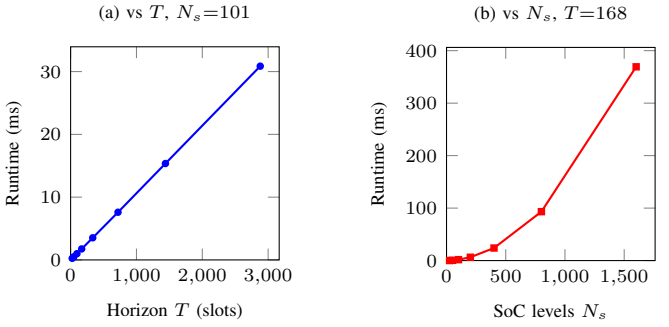

Fig. 9. E4: measured runtime of one solve. (a) Against the horizon T at a fixed grid, the runtime is linear. (b) Against the grid size N_s at a fixed horizon, the runtime grows faster than linear because the action window grows with the grid.

wear aware DP never loses money on a single day. On about half of the days it simply chooses to stay idle, because the price spread is too small to pay for the wear, so its profit on those days is zero rather than negative. The wear blind DP is profitable on only about 7 percent of the days, and the greedy rule on about 25 percent. So the gain is not driven by a few lucky days.

It comes from a steady refusal to trade when the spread does not cover the wear. This is exactly the behavior we want from a careful objective. It also matches a simple rule of thumb. A round trip is only worth it when the price spread is larger than the round trip efficiency loss plus twice the marginal wear cost. The DP applies this test in every slot and over the whole day at once.

VIII. Limitations and Future Work

The model has clear limits. First, we assume perfect foresight of prices. A real operator only has a forecast. A stochastic version would use expected future value in the recurrence. Second, we treat the operator as a price taker. Large batteries

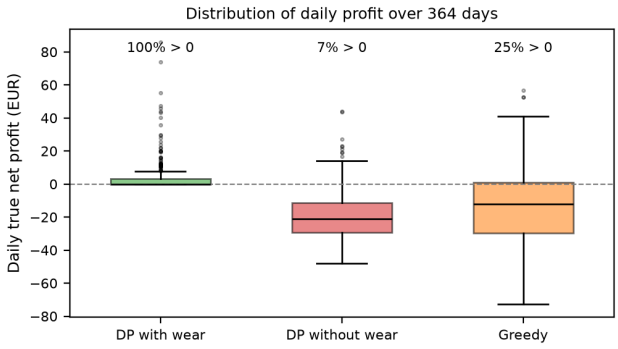

Fig. 10. Spread of daily true net profit over 364 days. The box is the middle half of the days and the line inside is the median. The wear aware DP stays at or above zero on every day. The other two methods sit mostly below zero once real wear is counted.

can shift the market price, which would need a market model. Third, the wear model is linear in throughput. A more exact model would use the depth of each cycle, for example a rainflow count. That model is path dependent, so it breaks the simple Markov state. One can approach it by adding more state, but that grows the state space. Fourth, we study one battery and one day at a time, with no network or market bidding limits. These extensions are good future work.

IX. Conclusion

We applied dynamic programming to schedule a single battery for energy arbitrage. The novel point is to put a linear capacity degradation cost directly into the reward. This keeps the problem additive and Markovian, so it is solved by plain backward induction in $O(T\,N_s\,N_a)$ time. On one year of real German day ahead prices, the wear aware schedule earns a positive yearly profit, while a greedy rule and a wear blind DP both lose money once real wear is counted. The method keeps about 41 percent of the gross revenue while cutting throughput by about 86 percent. A sensitivity study confirms that the best planning cost equals the real wear cost, and a runtime study confirms the linear scaling in the horizon. The result is a small and clean example of how a careful reward design lets a standard algorithm beat a naive strategy.

Appendix A
Source Code and Video

The full source code for this paper is open. It includes the Python data pipeline, the C++ dynamic programming solver, the baselines, and the LaTeX source. The repository is available at:

https://github.com/ethj0r/stima-paper

A short video that explains this paper is available at:

https://youtu.be/yUWVGqBm5hs

Acknowledgment

The author thanks Ir. Rila Mandala, M.Eng., Ph.D. and Prof. Dr. Ir. Rinaldi Munir, M.T., lecturers of IF2211 Strategi Algoritma for their guidance. The electricity price data used in this study comes from an external public source. It is the day ahead price data package of Open Power System Data, which collects the same hourly market prices that are also shared on open data platforms such as Kaggle. The author thanks these open data providers. The use of AI in this paper was limited to code scaffolding, LaTeX formatting, and language checking. The experiments and results are the author's own.

References

[1] R. Munir, "Program Dinamis (Bagian 1)," lecture notes, IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/25-Program-Dinamis-(2026)-Bagian1.pdf. Accessed: Jun. 15, 2026.

[2] R. Munir, "Program Dinamis (Bagian 2)," lecture notes, IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/26-Program-Dinamis-(2026)-Bagian2.pdf. Accessed: Jun. 15, 2026.

[3] R. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton Univ. Press, 1957. [Online]. Available: https://press.princeton.edu/books/paperback/9780691146683/dynamic-programming. Accessed: Jun. 15, 2026.

[4] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA: Pearson, 2006, ch. 6. [Online]. Available: https://www.pearson.com/en-us/subject-catalog/p/algorithm-design/P200000003259. Accessed: Jun. 15, 2026.

[5] D. R. Jiang and W. B. Powell, "Optimal hour ahead bidding in the real time electricity market with battery storage using approximate dynamic programming," *INFORMS J. Comput.*, vol. 27, no. 3, pp. 525–543, 2015. [Online]. Available: https://doi.org/10.1287/ijoc.2015.0640. Accessed: Jun. 8, 2026.

[6] B. Cheng and W. B. Powell, "Optimal battery charge scheduling for revenue stacking under operational constraints via energy arbitrage," arXiv:2305.04566, 2023. [Online]. Available: https://arxiv.org/abs/2305.04566. Accessed: Jun. 16, 2026.

[7] Y. Zhou et al., "Optimal scheduling for profit maximization of energy storage merchants considering market impact based on dynamic programming," *Energy*, vol. 222, art. 119910, 2021. [Online]. Available: https://doi.org/10.1016/j.energy.2021.119910. Accessed: Jun. 17, 2026.

[8] Authors, "Profitability of energy arbitrage for grid scale BESS considering dynamic efficiency and degradation," *J. Energy Storage*, 2024. [Online]. Available: https://doi.org/10.1016/j.est.2024.112345. Accessed: Jun. 14, 2026.

[9] Open Power System Data, "Time series" data package, version 2020-10-06. [Online]. Available: https://data.open-power-system-data.org/time_series/. Accessed: Jun. 18, 2026.

Statement

I hereby declare that the paper I have written is entirely my own work. It is not an adaptation, a translation of another person's work, nor a product of plagiarism.

Bandung, 19 June 2026

Author,

Made Branenda Jordhy

NIM: 13524026