A Dynamic Programming-Based Analysis of Optimal Strategies for Score Maximization in TETR.IO Blitz Mode

Bryan Pratama Putra Hendra - 13524067
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10, Bandung 40132, Indonesia
bryanpratama0111@gmail.com, 13524067@std.stei.itb.ac.id

***Abstract*— TETR.IO Blitz is a two-minute score attack mode in which the player attempts to obtain as many points as possible under strict time pressure. Although the game appears to be an action-oriented stacking challenge, the decision process behind high-score play can be modeled as a sequential optimization problem: each tetromino placement changes the board state, affects future placement opportunities, and produces score events such as line clears, combos, back-to-back clears, and perfect clears. This paper proposes a dynamic programming-based framework for analyzing optimal score maximization strategies in TETR.IO Blitz Mode. The model explicitly incorporates the seven tetrominoes, the seven-piece bag randomizer, the hold mechanic, the visible queue, and Blitz scoring events. The game is formalized as a finite-horizon decision problem with a board representation, queue information, bag residue, hold piece, scoring state, and remaining time. A Bellman-style recurrence is then used to compute the best attainable value over a bounded search space. Because the full Tetris state space is computationally large, this paper also discusses state compression, dominance pruning, beam-style state retention, and heuristic evaluation as practical approximations. The resulting framework shows how dynamic programming can be used not only to select locally strong placements, but also to reason about delayed rewards such as maintaining back-to-back chains, preserving perfect clear potential, using seven-bag structure for planning, and sacrificing immediate score to improve future scoring opportunities.**

***Keywords*— Dynamic Programming, TETR.IO, Blitz Mode, Seven-Bag Randomizer, Perfect Clear, DPC, Score Maximization, Bellman Recurrence**

I. INTRODUCTION

TETR.IO is a modern online stacker inspired by the familiar falling-block structure of Tetris. Among its solo modes, Blitz challenges the player to obtain as many points as possible within two minutes. The official TETR.IO interface describes Blitz as a mode where the player should get as many points as possible within two minutes, while clearing lines increases the level, score potential, and speed of the game [1]. This combination of time limit, level progression, and scoring multipliers makes Blitz a highly strategic optimization problem rather than a simple survival challenge.

A naive way to play Blitz is to maximize the score of the current piece only. For example, a player may immediately take any available line clear or choose the placement that gives the highest short-term reward. However, high-level score attack play often requires delayed rewards. The player may postpone a line clear to prepare a Tetris, preserve a back-to-back chain, maintain a combo structure, or aim for a perfect clear. These decisions are naturally sequential: the value of a move depends not only on its immediate score, but also on the board state it leaves for subsequent pieces.

This sequential dependency motivates the use of dynamic programming. Dynamic programming is suitable for problems that exhibit optimal substructure, where an optimal solution can be constructed from optimal solutions to subproblems. In the context of TETR.IO Blitz, a subproblem can be defined as: given the current board, hold piece, piece queue, scoring context, and remaining time, what is the maximum score that can still be achieved? This formulation leads directly to a Bellman recurrence that evaluates possible placements by combining immediate reward and future value.

The main challenge is that Tetris-like games have an enormous state space. Previous theoretical work has shown that even offline Tetris optimization is computationally difficult; Demaine, Hohenberger, and Liben-Nowell proved that several natural offline Tetris objectives are NP-complete or hard to approximate [6]. Therefore, a full exact solution for real-time Blitz play is not practical. Nevertheless, dynamic programming remains valuable as an analytical and algorithmic framework when the horizon, queue depth, board representation, or candidate states are bounded.

This paper focuses on constructing a rigorous but implementable model for score maximization in TETR.IO Blitz. The objective is not to reverse-engineer every internal detail of TETR.IO, nor to encourage automated play on live leaderboards. Instead, this work aims to analyze how dynamic programming can support strategic reasoning in a bounded simulation model of Blitz scoring. The contributions of this paper are as follows:

1) Formalizing TETR.IO Blitz score attack as a finite-horizon dynamic programming problem.
2) Defining a state representation that includes board configuration, hold piece, queue prefix, combo state, back-to-back state, level, and remaining time.
3) Deriving a Bellman recurrence for maximizing expected or deterministic future score.
4) Discussing practical pruning and approximation methods for reducing the large Tetris state space.

5) Explaining how the model captures strategic trade-offs such as immediate line clears versus delayed high-value scoring events.

II. THEORETICAL FOUNDATION

A. Tetris-Like Stacking Games

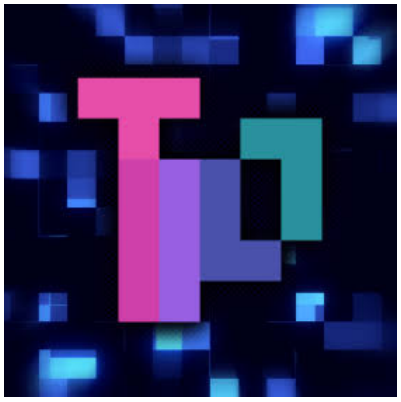

Fig. 1. Tetr.io Logo
Source: Author's Archive

Tetr.io is a Tetris-like stacking game is played on a rectangular grid, commonly represented as a matrix with a fixed width and height. Tetromino pieces fall into the board and must be positioned through horizontal movement, rotation, and dropping. When a row becomes completely filled, it is cleared and the rows above it shift downward. The player loses when new pieces can no longer enter or be placed on the board.

The strategic difficulty of Tetris arises from the fact that every placement is irreversible in the short term. A placement that produces points now may create holes, increase stack height, or reduce flexibility for future pieces. Conversely, a placement with no immediate score may be valuable because it prepares a high-reward clear later. This dependency between present action and future opportunity makes Tetris a natural domain for sequential decision-making.

B. The Seven Tetrominoes

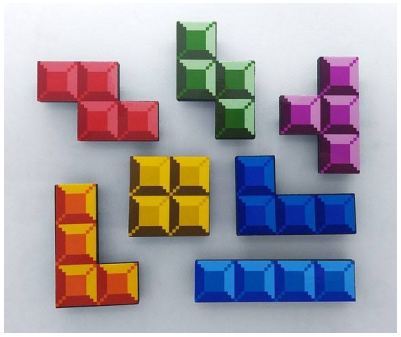

Fig. 2. Tetr.io Logo
Source: Author's Archive

The basic objects in the game are tetrominoes, each consisting of four unit blocks called minos. Modern Tetris-like games commonly use seven one-sided tetromino types: I, J, L, O, S, T, and Z [13]. These seven pieces are important not only as visual shapes, but also as strategic resources. For example, the I piece is essential for a four-line clear, the T piece enables T-Spins, the O piece has no rotational asymmetry, and the S and Z pieces often create parity or overhang constraints.

TABLE I
THE SEVEN TETROMINOES AND THEIR COMMON STRATEGIC ROLES.

Piece	Common role in score planning
I	Completes a Tetris/Quad well and is often held for high-value clears.
J	Creates left-facing structures, overhangs, and perfect clear residues.
L	Creates right-facing structures, overhangs, and perfect clear residues.
O	Stable square piece; useful in perfect clear patterns because it covers a compact 2×2 area.
S	Snake piece; can create parity constraints and specific perfect clear residues.
T	Enables T-Spins and many high-value opener structures.
Z	Snake piece mirrored from S; important in parity-sensitive patterns.

C. TETR.IO Blitz Mode

Blitz is a score-focused solo mode. The player's objective is not merely to survive or clear a fixed number of lines, but to maximize score within a two-minute time limit. TETR.IO states that Blitz gives the player two minutes to get as many points as possible and that clearing lines causes level progression, increasing both score and speed [1]. The TETR.IO community wiki also describes Blitz as a mode similar to Ultra in other stacker games, where the player must score as many points as possible within two minutes [2].

This time constraint changes the optimization objective. In a survival mode, the player may choose safe and low-risk placements to continue indefinitely. In Blitz, however, survival alone is insufficient. The player must convert limited time into score efficiently. A move is therefore evaluated by both its scoring reward and its time cost. A theoretically strong scoring pattern may be suboptimal if it requires too many placements or excessive setup time.

D. Seven-Piece Bag Randomizer

Fig. 3. Bag Representation
Source: Author's Archive

A crucial property of modern Tetris-like planning is that the piece sequence is not generated by independent uniform

randomness at every step. Instead, many modern games use a Random Generator, also known as the seven-bag system. In this system, the seven tetrominoes $\{I,J,L,O,S,T,Z\}$ are shuffled into a random order, drawn one by one until the bag is empty, and then a new shuffled bag is generated [13]. Since a single bag contains exactly one copy of each tetromino, there are $7! = 5040$ possible bag permutations.

This structure has important strategic consequences. First, the player can reason about which pieces must still appear before the current bag ends. If the player has already seen I, T, and O in the current bag, then the remaining bag must contain J, L, S, and Z in some order. Second, a single tetromino cannot appear three times in a row under a standard seven-bag system. Two identical consecutive pieces are possible only at a bag boundary, for example when an I piece is the last piece of one bag and another I piece is the first piece of the next bag. Third, long droughts are limited compared with memoryless random generation, which makes opener theory, perfect clear planning, and dynamic programming search more reliable.

For dynamic programming, the bag can be represented as a residue set

$$g_t \subseteq \mathscr{P}, \quad \mathscr{P} = \{I,J,L,O,S,T,Z\},$$

where g_t contains the pieces that have not yet appeared in the current bag. When a piece p appears, the next bag state is

$$g_{t+1} = \begin{cases} \mathscr{P} \setminus \{p\}, & \text{if } g_t = \{p\}, \\ g_t \setminus \{p\}, & \text{otherwise.} \end{cases}$$

This representation makes the probability model more accurate than assuming every next piece is independently chosen from all seven tetrominoes.

E. TETR.IO Blitz Scoring Table

TETR.IO Blitz rewards line clears, spin clears, all clears, combos, back-to-back chains, and drop distance. Public TETR.IO documentation and community-maintained TETR.IO references describe Blitz as a two-minute score attack mode where clearing lines increases level, score potential, and game speed [1], [14]. The scoring values used in this paper follow the publicly documented TETR.IO solo scoring table shown in Table II. In Blitz, these values are multiplied by the current level, while soft-drop and hard-drop points are not level-multiplied [14], [2].

TABLE II
PUBLICLY DOCUMENTED TETR.IO BLITZ SCORING VALUES.

Action	Base points
Single	100
Double	300
Triple	500
Quad/Tetris	800
Spin Zero	400
Spin Single	800
Spin Double	1200
Spin Triple	1600
Mini Spin Zero	100
Mini Spin Single	200
Mini Spin Double	400
All Clear/Perfect Clear	3500
Back-to-Back difficult clear	$x \times 1.5$
Combo bonus	$x \times 50$
Soft drop	1 per cell
Hard drop	2 per cell

This table explains why perfect clear strategies are attractive in Blitz. A perfect clear gives a large base reward, resets the board into a clean state, and can be combined with level multiplication. Similarly, back-to-back Tetrises or T-Spins are valuable because a difficult clear can receive a multiplier when chained. Combos add another layer of optimization because a sequence of consecutive line clears can produce additional score even when individual clears are not large.

F. Scoring Events in Modern Tetris Systems

Modern Tetris scoring systems usually reward more difficult line clears with higher points. Tetris Wiki records that recent guideline-compatible games commonly assign larger rewards to Tetrises and T-Spins, apply back-to-back multipliers to difficult clears, and include combo bonuses [4]. Although TETR.IO has its own implementation details and balancing choices, these general scoring concepts are useful for building an abstract score model.

For this paper, a scoring event is represented as

$$e = (l, \sigma, pc, d, k),$$

where l is the number of lines cleared, σ is the clear type such as Single, Double, Triple, Tetris, or T-Spin variant, pc indicates whether the placement produces a perfect clear, d is the drop distance, and k is the current combo index after the clear. The score increment is modeled by a configurable function

$$R(s,a) = G(e, \lambda, c, b),$$

where λ is the current level, c is the combo count, and b is the back-to-back state. In the TETR.IO Blitz approximation used in this paper, line-clear, spin-clear, all-clear, combo, and back-to-back points are multiplied by level, while drop points are added separately. This abstraction allows the dynamic programming model to represent Blitz-like scoring while keeping the scoring constants explicit and adjustable.

G. Dynamic Programming

Dynamic programming is an algorithmic technique for solving optimization problems by decomposing them into

overlapping subproblems. The technique relies on the principle of optimality: an optimal policy has the property that, after the first decision is made, the remaining decisions must be optimal with respect to the state resulting from that first decision. This idea is commonly expressed using a Bellman recurrence [9], [11].

For a finite-horizon deterministic decision problem, let s_t be the state at step t, a_t be an action, $T(s_t, a_t)$ be the transition function, and $R(s_t, a_t)$ be the immediate reward. The value function can be written as

$$V_t(s_t) = \max_{a_t \in A(s_t)} \left[R(s_t, a_t) + V_{t+1}(T(s_t, a_t)) \right].$$

The recurrence states that the best value at the current state is obtained by trying all legal actions and selecting the one that maximizes immediate reward plus future optimal value.

In a stochastic setting, such as when future pieces are unknown, the recurrence can be written as an expected value:

$$V_t(s_t) = \max_{a_t \in A(s_t)} \left[R(s_t, a_t) + \mathbb{E}_p \left[V_{t+1}(T(s_t, a_t, p)) \right] \right],$$

where p denotes the random future piece. If the next-piece queue is known for a limited depth, the deterministic recurrence can be applied over that visible queue prefix.

H. Optimal Substructure in Blitz Strategy

Blitz score maximization exhibits optimal substructure when the state representation contains all information needed to evaluate future rewards. For example, the future value after a placement depends on the board shape, next pieces, hold piece, combo counter, back-to-back state, level, and remaining time. If these variables are included in the state, then the best continuation from that point can be treated as a subproblem.

This property is especially important for delayed scoring patterns. A Tetris setup may require several low-reward placements before generating a high-value clear. A greedy method may reject these placements because each individual move has low immediate reward. Dynamic programming can assign value to such moves because the future reward is propagated backward through the value function.

I. State-Space Explosion

Despite its conceptual suitability, exact dynamic programming for Tetris is difficult because the number of possible board configurations is enormous. A standard board with width 10 and height 20 has a theoretical binary occupancy space of 2^{200}, before considering hold state, piece queue, combo count, back-to-back status, and time. The practical reachable state space is smaller, but still very large.

This state-space explosion motivates approximate dynamic programming and pruning. In Tetris research, approximate dynamic programming has been studied as a way to make value-based or policy-based optimization tractable. Gabillon, Ghavamzadeh, and Scherrer showed that approximate dynamic programming can perform strongly in Tetris when formulated through classification-based modified policy iteration [7]. This supports the idea that dynamic programming principles remain useful even when exact exhaustive enumeration is impossible.

III. PROBLEM FORMULATION

A. Objective Function

Let a Blitz game have a total time limit $T_{max} = 120$ seconds. The objective is to find a policy π that maps each game state to an action and maximizes the total accumulated score:

$$\max_{\pi} \sum_{t=0}^{N-1} R(s_t, \pi(s_t)),$$

subject to

$$\tau_{t+1} = \tau_t - C(s_t, \pi(s_t)), \quad \tau_t \geq 0,$$

where τ_t is the remaining time and $C(s_t, a_t)$ is the estimated time cost of executing action a_t. The number of placements N is not fixed because it depends on how quickly actions are executed and whether the player tops out before time expires.

B. State Representation

A state in the proposed model is defined as

$$s_t = (B_t, q_t, g_t, h_t, c_t, b_t, \lambda_t, \tau_t),$$

where:

TABLE III
STATE VARIABLES USED IN THE BLITZ DYNAMIC PROGRAMMING MODEL.

Symbol	Description
B_t	Board occupancy matrix at step t.
q_t	Visible next-piece queue prefix.
g_t	Remaining pieces in the current seven-piece bag.
h_t	Hold piece, or empty if hold has not been used.
c_t	Current combo count.
b_t	Back-to-back state for difficult clears.
λ_t	Current level or score multiplier state.
τ_t	Remaining time, discretized into time units.

The bag residue g_t is optional when a complete deterministic queue is known, but it becomes important when the algorithm samples or evaluates unknown future pieces. The board B_t may be represented directly as a binary matrix or compactly as a bitboard. A bitboard representation is efficient because row clearing, collision checking, and surface feature extraction can be performed using bit operations. For a theoretical paper, the matrix representation is easier to explain; for implementation, the bitboard representation is preferable.

C. Action Space

An action represents a legal placement of the active tetromino. If hold is available, the action may also include a hold decision. Formally,

$$a_t = (u, r, x, \eta),$$

where u indicates whether the active piece or hold piece is used, r is the rotation state, x is the horizontal position, and

η represents optional execution metadata such as movement path or finesse cost.

The legal action set $A(s_t)$ consists of all placements that do not collide with occupied cells and remain within the board boundary. Each legal placement produces a new board, possibly clears lines, updates the score context, consumes time, and advances the queue.

D. Transition Function

The transition function maps a state-action pair into a new state:

$$s_{t+1} = T(s_t, a_t).$$

The transition consists of the following operations:

1) Select the piece to place, considering hold if used.
2) Drop and lock the piece at the chosen rotation and position.
3) Clear all completed rows.
4) Detect the scoring event, including line clear type and perfect clear status.
5) Update combo count and back-to-back state.
6) Update level according to the number of cleared lines.
7) Reduce remaining time by estimated execution cost.
8) Advance the next-piece queue.

A terminal state occurs when $\tau_t \leq 0$, the player tops out, or the queue horizon ends in a bounded simulation.

E. Reward Function

The reward function is designed to approximate Blitz scoring incentives. A general form is

$$R(s_t, a_t) = S_{clear} + S_{combo} + S_{b2b} + S_{pc} + S_{drop},$$

where S_{clear} is the base line-clear score, S_{combo} is the combo bonus, S_{b2b} is the back-to-back multiplier or bonus, S_{pc} is the perfect clear bonus, and S_{drop} is optional drop score. The exact constants may be adjusted to match the target scoring table.

For analytical purposes, the reward can be simplified as

$$R(s_t, a_t) = \lambda_t \cdot base(e_t) \cdot m_{b2b}(b_t, e_t) + combo(c_t, \lambda_t) + pc(e_t),$$

where e_t is the scoring event produced by the action. This formulation captures the main idea: score depends not only on the number of cleared lines, but also on context such as level, combo, back-to-back, and perfect clear status.

IV. DYNAMIC PROGRAMMING MODEL

A. Bellman Recurrence for Known Queue

If the next-piece queue is known up to a finite horizon H, the problem can be treated deterministically. Let $V_i(s)$ be the maximum additional score obtainable from state s when considering the remaining queue positions from i to H. The recurrence is

$$V_i(s) = \max_{a \in A(s)} \left[R(s,a) + V_{i+1}(T(s,a)) \right],$$

with terminal condition

$$V_H(s) = 0.$$

If a state is terminal due to top-out or time expiration, its value is also zero or a large negative penalty, depending on whether the model wants to punish early failure.

B. Bellman Recurrence with Time Discretization

Because Blitz is constrained by time rather than only by piece count, remaining time must be included in the recurrence. Let τ be the remaining time measured in discrete ticks. Then the value function becomes

$$V_i(s, \tau) = \max_{a \in A(s), C(s,a) \leq \tau} \left[R(s,a) + V_{i+1}(T(s,a), \tau - C(s,a)) \right].$$

This recurrence naturally balances score and speed. A high-scoring action may be rejected if its execution cost is too large relative to the available time.

C. Expected Value for Unknown Future Pieces

When the full future queue is unknown, the model can use expectation over possible next pieces. The seven-bag system makes this expectation conditional on the remaining bag residue rather than uniform over all seven pieces. If g is the set of pieces not yet drawn from the current bag, then

$$P(p \mid g) = \begin{cases} 1/|g|, & p \in g, \\ 0, & p \notin g. \end{cases}$$

The stochastic recurrence becomes

$$V(s, \tau) = \max_{a \in A(s)} \left[R(s,a) + \sum_{p \in g(s)} P(p \mid g(s)) V(T(s,a,p), \tau - C(s,a)) \right].$$

This is more accurate than assuming independent random pieces because it uses information about which tetrominoes have already appeared in the current bag. It also explains why perfect clear openers can be studied as pattern families: the seven-bag sequence restricts the possible residue combinations after a fixed number of pieces.

D. Strategy Recovery

After computing values, the optimal action for each state can be recovered using

$$\pi^*(s, \tau) = \arg\max_{a \in A(s)} \left[R(s,a) + V(T(s,a), \tau - C(s,a)) \right].$$

The policy π^* is optimal only with respect to the chosen model, horizon, scoring approximation, and pruning rules. Therefore, the word "optimal" in this paper refers to optimality inside a bounded and explicitly defined search space.

E. Illustration of the DP Flow

Fig. 4 illustrates the high-level dynamic programming flow. Each placement creates a new board and scoring state. Future values are propagated backward so that early decisions account for long-term score potential.

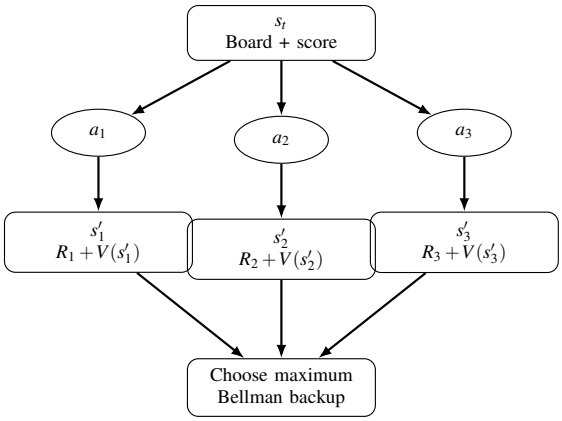

Fig. 4. Bellman backup process for selecting the best placement from a state.

V. PRACTICAL OPTIMIZATION TECHNIQUES

A. State Compression

Exact board representation provides maximum accuracy but is expensive. State compression reduces the number of distinct states by storing only strategically relevant features. Common features in Tetris agents include column heights, aggregate height, holes, bumpiness, wells, cleared lines, and landing height [8]. For Blitz score maximization, additional features are useful, such as perfect clear potential and back-to-back readiness.

Let $\phi(B)$ be a feature vector extracted from board B:

$$\phi(B) = [h_{avg}, h_{max}, holes, bump, wells, pcPot, tetrisReady].$$

The DP can then store values by compressed state

$$\hat{s}_t = (\phi(B_t), q_t, h_t, c_t, b_t, \lambda_t, \tau_t).$$

This reduces memory usage but sacrifices exactness because different boards may share the same feature vector.

B. Dominance Pruning

A state s_1 can be said to dominate another state s_2 if it has at least as much score, at least as much remaining time, and a board that is no worse according to safety and scoring features. Dominated states can be removed because they are unlikely to lead to a better final score.

A simple dominance rule is:

$$s_1 \succ s_2 \quad \text{if} \quad score(s_1) \geq score(s_2),\ \tau_1 \geq \tau_2,\ f(B_1) \geq f(B_2),$$

where $f(B)$ is a board quality function. This pruning is approximate, because a worse-looking board can sometimes enable a future perfect clear or T-Spin. Therefore, dominance pruning should be conservative.

C. Beam-Style Dynamic Programming

Beam-style DP keeps only the top K states at each depth or time layer. This method is not exact, but it makes the search tractable. Each state is ranked by a combination of accumulated score and heuristic future potential:

$$rank(s) = score(s) + \alpha \cdot H(B_s, q_s, h_s),$$

where H estimates future scoring potential. A larger beam width K improves accuracy but increases computation time.

D. Heuristic Evaluation

The heuristic function should reflect both survival and score potential. A possible evaluation function is

$$H(B) = w_1 \,\text{holes} + w_2 \,\text{bumpiness} + w_3 \,\text{maxHeight} + w_4 \,\text{tetrisWell} + w_5 \,\text{pcPotential}.$$

The signs of the weights depend on whether the feature is desirable or undesirable. Holes, bumpiness, and excessive height are usually penalized, while Tetris well readiness and perfect clear potential may be rewarded.

E. Time-Cost Modeling

In Blitz, a scoring plan is only useful if it can be executed quickly. Therefore, the model includes action cost. A simple time-cost approximation is

$$C(s,a) = t_{base} + \gamma_1 \cdot moves(a) + \gamma_2 \cdot rotations(a) + \gamma_3 \cdot hold(a),$$

where $moves(a)$ is the number of horizontal movement inputs, $rotations(a)$ is the number of rotations, and $hold(a)$ indicates whether the hold action is used. This cost function encourages efficient placements and penalizes unnecessarily complex movement sequences.

VI. IMPLEMENTATION DESIGN

The proposed method can be implemented as a simulator rather than a live game-playing bot. The simulator receives an initial board, a known or sampled piece queue, and scoring parameters. It then enumerates legal placements, updates states, and applies dynamic programming to find a high-scoring plan.

A. Main Data Structures

The implementation uses the following data structures:

- **Board**: a binary matrix or bitboard representing occupied cells.
- **Piece Queue**: a list of upcoming tetrominoes.
- **State**: board, hold, combo, back-to-back, level, score, and remaining time.
- **Action**: selected piece source, rotation, x-position, and estimated execution path.
- **Parent Pointer**: used to reconstruct the final optimal sequence of placements.

B. Legal Placement Generation

For each active piece, the generator enumerates all unique rotations and horizontal positions. A piece is dropped until it collides with the board, then locked if the final position is valid. The result is a set of candidate actions.

```
function GenerateActions(state):
    actions = empty list
    candidatePieces = {activePiece}
    if hold is available:
        candidatePieces.add(holdOrNextPiece)

    for piece in candidatePieces:
        for rotation in UniqueRotations(piece):
            for x in ValidHorizontalRange(piece,
                rotation):
```

```
                y = DropToSurface(board, piece,
                    rotation, x)
                if IsValidPlacement(board, piece,
                    rotation, x, y):
                    actions.add(Action(piece,
                        rotation, x, y))
    return actions
```

Listing 1. Legal placement generation.

C. Dynamic Programming with Beam Pruning

The following pseudocode describes a practical DP algorithm with beam pruning. The algorithm is exact only when K is large enough to retain all reachable states.

```
function BlitzDP(initialState, queue, timeLimit,
    beamWidth):
    frontier = {initialState}
    parent = empty map

    for i from 0 to length(queue)-1:
        candidates = empty list

        for state in frontier:
            if state.remainingTime <= 0 or IsTopOut
                (state.board):
                continue

            actions = GenerateActions(state)

            for action in actions:
                newState = ApplyAction(state,
                    action, queue[i])
                reward = ScoreEvent(state, action,
                    newState)
                newState.totalScore = state.
                    totalScore + reward
                newState.remainingTime -=
                    EstimateCost(state, action)

                if newState.remainingTime >= 0:
                    candidates.add(newState)
                    parent[newState] = (state,
                        action)

        candidates = RemoveDominatedStates(
            candidates)
        frontier = KeepTopK(candidates, beamWidth,
            RankingFunction)

    bestState = argmax(frontier, key = totalScore)
    return ReconstructPlan(bestState, parent)
```

Listing 2. Beam-pruned dynamic programming for Blitz planning.

D. Complexity Analysis

Let H be the planning horizon, K be the beam width, and A be the maximum number of legal placements per state. Without pruning, the number of explored states can grow as $O(A^H)$. With beam pruning, each layer keeps at most K states, so the approximate complexity becomes

$$O(H \cdot K \cdot A \cdot C_T),$$

where C_T is the cost of applying a transition, clearing lines, and evaluating the resulting state. Memory usage is approximately

$$O(K \cdot H)$$

if parent pointers are stored for strategy reconstruction. This complexity is much more practical than exhaustive search, but the result is no longer globally exact.

E. Parameter Choices

Several parameters influence the quality and speed of the algorithm:

TABLE IV
IMPORTANT IMPLEMENTATION PARAMETERS.

Parameter	Effect
Queue horizon H	Longer horizon improves planning but increases cost.
Beam width K	Larger beam retains more candidate strategies.
Time tick Δt	Smaller tick gives more accurate time modeling.
Heuristic weights w_i	Determine preference for clean board, Tetris readiness, or perfect clear potential.
Dominance threshold	Controls how aggressively states are pruned.

VII. STRATEGIC ANALYSIS

A. Greedy Strategy Versus DP Strategy

A greedy strategy selects the action with the highest immediate reward. This may work when a line clear is obviously available, but it fails when the best scoring opportunity requires setup. For example, taking an immediate Single may break a back-to-back chain or destroy a Tetris well. Dynamic programming can avoid this by evaluating the future value of preserving the setup.

In simplified form, suppose action a_1 gives an immediate reward of 300, while action a_2 gives an immediate reward of 0 but enables a future reward of 1200. A greedy strategy selects a_1, while DP selects a_2 if

$$0 + V(T(s,a_2)) > 300 + V(T(s,a_1)).$$

This captures the core advantage of dynamic programming: it propagates delayed reward backward to earlier decisions.

B. Back-to-Back Preservation

Back-to-back chains reward consecutive difficult clears. In many modern Tetris scoring systems, Tetrises and T-Spins are treated as difficult clears and may receive bonus multipliers when performed consecutively [4]. A DP strategy can assign value to placements that preserve the ability to continue difficult clears. This may cause the algorithm to avoid low-value line clears that would break the chain.

C. Perfect Clear Potential

A perfect clear occurs when the board becomes completely empty after a line clear. Tetris Wiki notes that several recent games award perfect clear bonuses, with additional scoring depending on the type of clear [4]. In Blitz, perfect clear strategies are attractive because they combine high score with a clean board reset. However, perfect clear setups are fragile.

A locally reasonable placement can ruin the exact parity or residue structure needed for a perfect clear.

The DP model can include a feature $pcPotential(B)$ that estimates whether the board remains compatible with a perfect clear route. This feature helps the search retain states that may have low immediate score but high future perfect clear value.

D. Perfect Clear Openers and DPC Loops

The seven-bag system makes perfect clear planning more structured than it first appears. A standard four-line perfect clear uses ten minos, or two and a half tetrominoes in terms of block count. Because two bags contain fourteen tetrominoes, repeated perfect clear planning can be described in terms of bag residues and leftover pieces. Public opener resources such as Hard Drop's Perfect Clear Opener documentation report that common first-PC setups have success rates that depend on the first piece of the second bag under hold and seven-bag assumptions [15]. This supports the idea that perfect clear planning is not arbitrary memorization: it is constrained by bag order, hold, and board parity.

One advanced family of perfect clear strategy is DPC. FOUR.lol describes DPC as a set of T-Spin Double $\rightarrow$ Perfect Clear setups used after an eight-height perfect clear with the first three bags [17]. An eight-height perfect clear requires twenty minos, equal to two full bags and six minos, leaving one tetromino as residue. With that leftover piece, the player can build a TSD $\rightarrow$ PC pattern and return to an empty field with a fresh bag. In Blitz, this matters because a repeated PC-oriented loop can combine high perfect clear score, T-Spin value, back-to-back preservation, and clean board resets.

From a dynamic programming perspective, DPC-like strategies are a strong example of delayed reward. The immediate value of a placement may be low, but it preserves a residue class that enables a later TSD and perfect clear. Therefore, the state representation should not only record the board surface, but also bag residue, hold piece, and perfect clear compatibility. A simplified DP feature can be written as

$$pcPotential(B,g,h) = \mathbf{1}\{\text{a known PC pattern is reachable from } (B,g,h)\}.$$

In a full implementation, this indicator can be replaced by a table of known opener templates, a solver over remaining pieces, or a learned heuristic trained from perfect clear solution data.

E. Worked DP Case Study

To make the dynamic programming calculation more concrete, consider a bounded search state s in which the solver compares three families of first moves: an immediate line clear, a Quad-oriented setup, and a DPC/Perfect-Clear-oriented setup. The three candidates may have very different immediate rewards, but DP evaluates each one using the same Bellman backup:

$$Value(a) = r(s,a) + V_{H-1}(T(s,a)).$$

Thus, the selected move is not necessarily the move with the largest immediate score. It is the move with the largest sum of current reward and future continuation value.

Suppose the current board and seven-bag residue allow a DPC continuation. A greedy move a_g may clear a Double immediately, while a DPC setup move a_d may score zero on the first placement. However, if a_d preserves the residue needed to reach a T-Spin Double followed by a Perfect Clear, its continuation value can dominate the greedy move. In the full Blitz scoring abstraction, the future reward may include a T-Spin Double, a perfect clear bonus, combo value, back-to-back value, and the clean-board terminal value. In the simplified implementation, the same idea is represented mainly through the perfect-clear bonus and the PC-focused terminal evaluator.

The comparison can be written as

$$\begin{aligned} Value(a_g) &= r_g + V_{H-1}(s_g), \\ Value(a_q) &= r_q + V_{H-1}(s_q), \\ Value(a_d) &= r_d + V_{H-1}(s_d), \end{aligned}$$

where s_g, s_q, and s_d are the states produced by the greedy, Quad-setup, and DPC-setup moves. Even when $r_d = 0$, the DPC route is preferred if

$$V_{H-1}(s_d) > r_g + V_{H-1}(s_g)$$

and

$$V_{H-1}(s_d) > r_q + V_{H-1}(s_q).$$

This inequality is the key reason dynamic programming is useful for DPC analysis: the value of the setup is carried backward from a later perfect clear.

TABLE V
ILLUSTRATIVE DP COMPARISON BETWEEN STRATEGY FAMILIES.

Strategy family	$r(s,a)$	$V(T(s,a))$	Total
Immediate Double	300	420	720
Quad setup	0	980	980
DPC/PC route	0	3500+	Highest

Table V is illustrative rather than a universal result. The exact numbers depend on the board, queue, beam width, search horizon, and scoring model. Nevertheless, the structure of the calculation is important: DP can rationally select a zero-score setup move when the future value of a Perfect Clear or DPC loop is higher than the value of an immediate clear. Therefore, the conclusion should be stated conditionally: within a bounded search space and a compatible seven-bag residue, the best DP-evaluated route can be a Perfect Clear loop or a DPC continuation, not a greedy line clear.

F. Time Efficiency

Because Blitz lasts only two minutes, optimal score does not necessarily come from the theoretically largest score event per board. It comes from maximizing score per unit time. The DP model can account for this by including remaining time in the state and movement cost in the transition. This allows the algorithm to prefer a slightly lower-scoring

but faster placement if it enables more total placements before time expires.

G. Risk and Robustness

A high-scoring plan may be risky if it depends on a narrow sequence of future pieces. In a known-queue simulation, DP can exploit exact future information. In real play, the player only sees a limited queue and must handle uncertainty. Therefore, a robust strategy should optimize not only expected score but also board flexibility. This can be modeled by adding a flexibility bonus to the heuristic function:

$$H_{robust}(B) = H(B) + w_f \cdot flexibility(B),$$

where $flexibility(B)$ measures how many legal placements remain available for likely future pieces.

VIII. DISCUSSION

The proposed framework demonstrates that TETR.IO Blitz can be interpreted as a finite-horizon optimization problem. Dynamic programming is appropriate because each placement creates a subproblem whose future value can be computed or approximated. The model is especially useful for explaining why high-score strategies often avoid greedy decisions.

However, several limitations must be acknowledged. First, exact DP over the full Tetris board is computationally infeasible due to state-space explosion. Second, scoring in TETR.IO may include implementation-specific details that are not captured by a simplified scoring function. Third, human execution introduces uncertainty: even if a plan is mathematically strong, it may be impractical if it requires difficult movement or extremely fast inputs. Fourth, using such a model to automate live leaderboard play would be inappropriate; the intended use of this paper is strategy analysis and algorithmic study.

Despite these limitations, the framework is still valuable. It can be used to compare strategies, evaluate the importance of perfect clear potential, analyze the trade-off between score and time, and build offline tools for understanding Tetris-like decision-making. The added DP case study also clarifies why a Perfect Clear or DPC route may be selected even when its first placement has no immediate reward: the later perfect clear value is propagated backward through the Bellman recurrence. The model also connects game strategy with core algorithmic concepts, making it suitable for a paper in algorithm strategy.

IX. CONCLUSION

This paper presented a dynamic programming-based analysis of optimal strategies for score maximization in TETR.IO Blitz Mode. The main idea is to model Blitz as a finite-horizon sequential decision problem where each state contains the board configuration, piece queue, seven-bag residue, hold piece, combo state, back-to-back state, level, and remaining time. A Bellman recurrence is then used to evaluate legal placements based on immediate score and future value.

The analysis shows that dynamic programming is more suitable than greedy decision-making for Blitz strategy because it can capture delayed rewards. Maintaining a back-to-back chain, preparing a Tetris well, tracking seven-bag residue, preserving perfect clear potential, and selecting faster placements are all decisions whose value may only appear several moves later. The worked case study demonstrates this effect directly: under a compatible board and bag residue, the DP value of a DPC or Perfect Clear route can exceed a greedy immediate clear because the future all-clear reward dominates the short-term score. By propagating future rewards backward, dynamic programming provides a principled way to reason about these trade-offs.

Because the complete Tetris state space is extremely large, exact optimization is not practical for full-scale Blitz. Therefore, this paper also discussed state compression, dominance pruning, beam-style DP, heuristic evaluation, and time-cost modeling as practical approximations. With these techniques, the proposed framework becomes a realistic foundation for offline analysis, simulation, and strategy comparison.

Overall, TETR.IO Blitz is not only a fast-paced game mode but also a rich example of algorithmic optimization. It demonstrates how dynamic programming can be applied beyond classical textbook problems to analyze real interactive systems where local decisions, future consequences, and limited resources are deeply connected.

X. APPENDIX

The complete source code that is used in this paper will be available on my Github Repository, Click Here!

There will be a short video explaining this paper on my youtube channel, Click Here!

XI. ACKNOWLEDGMENT

First and foremost, the author is grateful to God Almighty for the strength and guidance given throughout the completion of this paper. The author also expresses sincere appreciation to the lecturers and teaching assistants of IF2211 Algorithm Strategy for providing the theoretical foundation needed to understand algorithmic problem solving, optimization, and dynamic programming. Appreciation is also extended to classmates and friends for discussions, feedback, and encouragement during the writing process. Finally, the author thanks the Tetris and TETR.IO communities for maintaining public documentation and strategic discussions that help make this topic accessible for academic exploration.

XII. REFERENCES

REFERENCES

[1] osk, "TETR.IO," *TETR.IO Official Website*, 2026. [Online]. Available: https://tetr.io/. [Accessed: Jun. 17, 2026].

[2] TETR.IO Wiki contributors, "BLITZ," *Wiki for TETR.IO*, 2026. [Online]. Available: https://tetrio.wiki.gg/wiki/BLITZ. [Accessed: Jun. 17, 2026].

[3] TETR.IO Wiki contributors, "Mechanics," *Wiki for TETR.IO*, 2026. [Online]. Available: https://tetrio.wiki.gg/wiki/Mechanics. [Accessed: Jun. 17, 2026].

[4] Tetris Wiki contributors, "Scoring," *Tetris Wiki*, 2025. [Online]. Available: https://tetris.wiki/Scoring. [Accessed: Jun. 17, 2026].
[5] Tetris Wiki contributors, "Tetris Guideline," *Tetris Wiki*, 2025. [Online]. Available: https://tetris.wiki/Tetris_Guideline. [Accessed: Jun. 17, 2026].
[6] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell, "Tetris is hard, even to approximate," in *Proceedings of the 9th Annual International Conference on Computing and Combinatorics (COCOON 2003)*, Lecture Notes in Computer Science, vol. 2697, Springer, 2003, pp. 351–363.
[7] V. Gabillon, M. Ghavamzadeh, and B. Scherrer, "Approximate dynamic programming finally performs well in the game of Tetris," in *Advances in Neural Information Processing Systems 26*, 2013.
[8] S. Algorta and O. Simsek, "The game of Tetris in machine learning," *arXiv preprint arXiv:1905.01652*, 2019.
[9] R. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1957.
[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.
[11] B. Light, "The principle of optimality in dynamic programming: A pedagogical note," *arXiv preprint arXiv:2302.08467*, 2023.
[12] Hard Drop Wiki contributors, "T-Spin Guide," *Hard Drop Tetris Wiki*, 2023. [Online]. Available: https://harddrop.com/wiki/T-Spin_Guide. [Accessed: Jun. 17, 2026].
[13] Tetris Wiki contributors, "Random Generator," *Tetris Wiki*, 2025. [Online]. Available: https://tetris.wiki/Random_Generator. [Accessed: Jun. 19, 2026].
[14] Hard Drop Wiki contributors, "TETR.IO," *Hard Drop Tetris Wiki*, 2025. [Online]. Available: https://harddrop.com/wiki/TETR.IO. [Accessed: Jun. 19, 2026].
[15] Hard Drop Wiki contributors, "Perfect Clear Opener," *Hard Drop Tetris Wiki*, 2026. [Online]. Available: https://harddrop.com/wiki/Perfect_Clear_Opener. [Accessed: Jun. 19, 2026].
[16] FOUR.lol contributors, "2nd PC Patterns," *FOUR.lol*, 2022. [Online]. Available: https://four.lol/perfect-clears/2nd/. [Accessed: Jun. 19, 2026].
[17] FOUR.lol contributors, "DPC (3rd PC after opener template)," *FOUR.lol*, 2022. [Online]. Available: https://four.lol/perfect-clears/dpc/. [Accessed: Jun. 19, 2026].

STATEMENT

Hereby, I declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not a product of plagiarism.

Bandung, June 17th 2026

Bryan Pratama Putra Hendra
13524067