Initial Draft: A full market dynamics model

Mark Brezina

May 2025

1 Introduction

We formalize the system as follows. Let $t = 0, 1, 2, \ldots$ be discrete time steps. The environment consists of (1) a set of hierarchical **rules** governing dynamics and constraints, (2) a time-varying graph $G(t) = (V, E(t))$ whose *Vertices* represent moving objects (or "particles") in the system, (3) a discretized **geometrical surface** (e.g. a spatial lattice or mesh) on which values and fields evolve (modeled with multiple layers), and (4) a set of **agents** with states and utility functions. We describe each in turn.

- **Rules Hierarchy:** We assume a structured set of rules $\mathcal{R}$ (e.g. invariants, probabilistic transition rules, constraints) at multiple tiers. For example, Tier-1 rules might enforce conservation laws or feasibility constraints, Tier-2 rules govern probabilistic state transitions (e.g. if event E occurs, object state updates according to distribution $P(\cdot|E)$), and Tier-3 rules define outcomes of interactions among objects or agents. Such hierarchical, rule-based environments have been studied in agent-based modeling (e.g. Lotzmann & Meyer's declarative rule-based agent models). In general we let $\mathcal{R} = r_1, r_2, \ldots$ denote all rules; each rule r is a mapping from certain conditions (on agents, objects, or environment) to outcomes (possibly stochastic).
- **Graph Structure (Objects as Vertices):** We model abstract objects or particles as vertices in a directed graph. Concretely, let V be the set of graph nodes (for example, nodes can encode discrete space-time positions). An object moving from node i at time t to node j at time $t+1$ is represented by a directed vertice v from (i, t) to $(j, t+1)$. Thus each vertice encodes an object's trajectory between discrete space–time events (cf. "nodes = events, vertices = particles" in physical universe graphs). We denote the set of edges at time t by $E(t)$. Each edge (object) $e \in E(t)$ carries a state vector

  $$\Theta_e(t) = (pos_e(t), val_e(t), \tau_e, ...)$$

  where $\mathrm{pos}_e(t)$ is its spatial position (or node location), $\mathrm{val}_e(t)$ its internal value or state, τ_e its type/category, etc. As the system evolves, edges can move, be created, or be destroyed according to the rules. This is analogous to interaction networks where nodes/edges represent objects and relations
- **Discretized Geometrical Surface and Layers:** The underlying spatial domain is discretized (e.g. a lattice or triangular mesh) into nodes $i \in V_{\text{space}}$ with coordinates x_i. On this surface we define multiple "information" layers. For example:
  - **Evaluation Layer (E):** A scalar field $E_i(t)$ at each site i representing local evaluation or utility density. (This could encode resource levels, reward potential, or other signals.)

  - **Propagation Layer (P)**: Encoded by a propagation matrix or adjacency on the grid (e.g. weights P_{ij}). This governs how values diffuse or propagate between neighboring sites (analogous to a graph Laplacian)
  - **Interaction Layer (I)**: A layer that captures how agents/objects at neighboring sites interact (e.g. collision effects or communication).

  These layers together form a multilayer network on the same surface nodes, with each layer representing a different aspect (evaluation, diffusion, interaction). Multilayer networks (networks with multiple types of relations or layers) are a standard tool for complex systems

- **Agents:** Let $\mathcal{A} = 1, \ldots, N$ be the set of agents. Each agent i has a state $s_i(t)$ (which may include location on the surface, internal variables, etc.) and an action space $\mathcal{A}_i$. Agents follow: (a) General rules (e.g. physical laws or system-wide constraints) and (b) Agent-specific rules (e.g. strategic preferences or class-based behaviors). Each agent has a utility (or reward) function $U_i(t)$, which they seek to maximize (or loss L_i to minimize). Agents may act individually or form teams. The influence between agents is modeled by a time-varying interaction graph: for example, a weighted adjacency $W(t) = [w_{ij}(t)]$ between agents where $w_{ij}(t)$ is stochastic and time-varying. Such time-varying interaction networks are studied in network game theory. In particular, we can treat each pair (i, j) with weight $w_{ij}(t)$ as a (possibly random) communication or influence link (the graph of agent interactions may change each step). In summary, agents i have utility dynamics and interact on graphs as in multi-agent dynamic games.

2 Time-Evolving Equations

We now describe the discrete-time dynamics. Let the global state at time t be $X(t) = (S(t), E(t), s_i(t))$, where $S(t)$ denotes all surface layer values, $E(t)$ all object-edge states, and $s_i(t)$ agent states.

- **Object/Edge Dynamics:** Each edge (object) $e \in E(t)$ evolves according to rules. For example, if e moves from site i to j, its position updates as

  $$\mathrm{pos}_e(t+1) = \mathrm{pos}_e(t) + \mathrm{v}_e(t)$$

  where $v_e(t)$ is a (possibly random) displacement determined by rules or physics. In general, the internal state $\Theta_e(t)$ follows a discrete update:

  $$\Theta_e(t+1) = f_e(\Theta_e(t), \{\Theta_{e'}(t) : e' \sim e\}, S(\cdot, t), \xi_e(t))$$

  where f_e is a (possibly nonlinear) function encoding the rules, $e' \sim e$ are neighboring edges or objects, and $\xi_e(t)$ is noise. Concretely, one might write

  $$val_e(t+1) = val_e(t) + \Delta t \cdot g_e(val_e(t), pos_e(t), S(pos_e(t), t)) + \eta_e(t)$$

capturing internal growth or decay. The appearance/disappearance of edges is likewise rule-driven: e.g., a collision rule may remove two edges and create a new one, according to a rule in $\mathcal{R}$.

- **Surface (Field) Dynamics:** Let $E_i(t)$ be the evaluation-layer value at site i, and let L be the discrete Laplacian (graph Laplacian) for the surface grid. Then a diffusion-like update is natural:

  $$E(t+1) = E(t) + \Delta t(DLE(t) + F_{source}(t))$$

Here D is a diffusion coefficient, and $F_{\text{source}}(t)$ sums contributions from objects and agents. More explicitly, for each site i one can write:

$$E_i(t+1) = E_i(t) + \Delta t \left(D \sum_{j \in \mathcal{N}(i)} (E_j(t) - E_i(t)) + \sum_{e:e \text{ } at \text{ } i} \alpha_e(t) + \sum_{k:k \in \mathcal{A}_i(t)} \beta_{i,k}(t) \right)$$

where $\mathcal{N}(i)$ are neighbors of i, each object e at i injects amount $\alpha_e(t)$ into the field (e.g. resource deposit), and each agent k at site i injects $\beta_{i,k}(t)$ (e.g. information or consumption). The term $\sum_j (E_j - E_i)$ is the discrete Laplace operator on the grid (cf. In other words, surface values diffuse and receive source terms. (Similar equations govern other layers if needed, e.g. propagation weights may adapt, etc.)

- **Agent Dynamics:** Each agent i has state $s_i(t)$ and chooses action $a_i(t) \in \mathcal{A}i$. *The agent's state then updates by*
  $s_i(t+1) = G_i(s_i(t), S(\cdot, t), \{s_j(t)\}_{j \neq i}, a_i(t), \{a_j(t)\}_{j \neq i})$,
  according to general and agent-specific rules. For example, an agent's position may change on the surface, its wealth may increase by consuming resources (captured by S), etc. Its utility $U_i(t)$ is updated by an instantaneous reward (or loss):
  $U_i(t+1) = U_i(t) + r_i(s_i(t), S(\cdot, t), a_i(t), \{a_j(t)\}_{j \neq i})$
  where r_i is the reward function (negative of a loss) as defined by the agent's objectives. This covers agent–environment interactions and agent–agent interactions. In particular, agents may share information or constraints via the interaction graph weights $wij(t)$:
  for instance the reward r_i might depend on a weighted sum of neighbors' states $\sum_j w_{ij}(t) s_j(t)$, capturing communication or influence (with $w_{ij}(t)$ stochastic and time-varying

Collecting the above, the overall state $X(t)$ evolves as a (generally stochastic) dynamical system under the rule set $\mathcal{R}$:

$$X(t+1) = T\left(X(t), a_1(t), \ldots, a_N(t), \zeta(t)\right)$$

where $\zeta(t)$ represents random disturbances. This discrete-time system can be viewed as a high-dimensional Markov chain (or game, if multiple agents choose actions).

3 Optimal Policy for a Fully Informed Agent

We now outline how a single *fully informed* agent can compute an optimal policy (maximizing cumulative utility). If agent i observes the global state $X(t)$ and knows all rules, one can formulate its decision problem as a Markov Decision Process (MDP). Let the agent's *value function* $V_i(X,t)$ be the maximum expected future utility starting from state X at time t. The Bellman optimality equation gives:

$$V_i^*(X,t) = \max_{a_i \in \mathcal{A}_i} \left\{ r_i(X, a_i) + \gamma \mathrm{E}\left[V_i^*(X', t+1) \mid X, a_i \right] \right\}$$

where $\gamma \in (0,1]$ is a discount factor, $r_i(X, a_i)$ is the immediate reward (which may depend on X and a_i), and the expectation is over the next state X' distribution given (X, a_i) (others' actions can be folded into the environment or assumed known). In the undiscounted, finite-horizon case one sets $\gamma = 1$ up to horizon T. This equation is the discrete-time Bellman optimality condition, which embodies *backward induction* for dynamic optimization (cf. Bellman's principle).

In practice, one solves this by dynamic programming. A simple backward-induction algorithm (pseudocode) is:

```
Initialize: for all terminal states X, set V[T+1,X]=0.
For t = T down to 0 do:
    For each possible global state X:
        For each admissible action a_i of agent i:
            Compute immediate reward r = r_i(X, a_i).
            Compute expected next-state value:
                V_next = 0
                for each possible next state X':
                    V_next += P(X'|X, a_i) * V[t+1, X']
            TotalVal = r + gamma * V_next
        End for
        Choose a_i that maximizes TotalVal.
        Set V[t, X] = max_a_i TotalVal.
        Record policy pi(t,X) = argmax_a_i TotalVal.
    End for
End for
Return policy pi(.).
```

In words: at each time step t and state X, the agent evaluates each action a_i by summing immediate reward plus discounted future value (using transition probabilities $P(X'|X, a_i)$). It then picks the action maximizing expected utility. The optimal policy $\pi(t, X)$ and value function $V[t, X]$ are computed by this value-iteration (backward) procedure. In the infinite-horizon case, one can use the Bellman update iteratively until convergence. The correctness of this procedure follows from Bellman's optimality principle.

This abstract framework and algorithm are general and can incorporate any specific choices of rules, graph structure, surface discretization, and agent utilities. By clearly defining the state space and transition rules, one can systematically derive the difference equations above and compute policies, ensuring extensibility to varied applications in physics, economics, or game-theoretic domains.

4 Optimal policy

We consider a *fully informed* agent operating on a discrete **graph-based environment** (a "discretized geometrical surface" with possibly multiple layers of abstraction) that contains other agents and dynamic objects. Formally, let the environment state at time t be S_t, which encodes all agent positions, object states, and relevant features on a graph $G = (V, E)$. The fully informed agent has complete knowledge of S_t, all transition dynamics $P(S_{t+1} \mid S_t, A_t)$, and the policies or strategies of other agents. The agent's goal is to choose actions a_t to maximize its **cumulative monetary reward** minus monetary loss over time. In standard decision-theoretic terms, if R_t and L_t are the reward and loss at step t, it seeks to maximize

$$J(\pi) = \mathbf{E}\left[\sum_{t=0}^{T} \gamma^t (R_t - L_t)\right]$$

where π is the agent's policy and $0 < \gamma \leq 1$ is a discount factor (or $\gamma = 1$ for finite horizon). Because the agent is fully informed, this problem can be formulated as a **known Markov decision process (MDP)** or a leader–follower (Stackelberg) game if other agents' responses are considered. For instance, the Bellman optimality condition in the MDP framework is

$$V^*(s) = \underset{a}{max} \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') - L(s, a, s') + \gamma V^*(s')\right]$$

with an optimal policy $\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a)[R - L + \gamma V^(s')]$. In a multi-agent Stackelberg setting, the agent can be viewed as a leader who selects an action anticipating that other (follower) agents will best-respond.

5 Three-Part Decision System

The agent's policy is organized into three interacting modules: a S**trategy System**, a **Prediction System**, and a **Risk/Validation System**. These are arranged so that the Strategy System uses predictions and risk evaluations to select actions.

5.1 Strategy System

The Strategy System computes the *action* that maximizes expected net outcome. Given the current state S_t and a set of candidate actions a, it uses the outputs of the Prediction and Risk systems to score each action by its expected *monetary* return minus loss. Concretely, let $\hat{S}_{t+1}$ denote the predicted next state, and $\hat{R}(a), \hat{L}(a)$ the predicted reward and loss from taking a. The strategy computes

$$Q(S_t, a) = \mathbf{E}\left[\hat{R}(a)\right] - \mathbf{E}\left[\hat{L}(a)\right] + \gamma \mathbf{E}\left[V(S_{t+1})\right]$$

where V is the value-to-go (initialized or computed recursively). It then selects

$$a^* = \underset{a}{argmax} Q(S_t, a)$$

In other words, the agent acts to maximize the expected sum of future rewards minus losses. This is exactly the Bellman optimal policy for an MDP with reward $R - L$. In multi-agent terms, if other agents' reactions are predictable, the agent can solve a Stackelberg-style optimization: it chooses a^* knowing followers will best-respond, effectively optimizing leader payoff. Thus the strategy system reduces action selection to a known optimal-control problem (no exploration needed, since the agent is fully informed).

5.2 Prediction System

The Prediction System is a **modular forecasting module** that estimates future environment dynamics and other agents' behavior. It takes the current state S_t (and possibly a planned action a) and outputs predictions such as $\hat{S}_{t+1}$ or expected future rewards/losses. Importantly, this system is *model-agnostic*: it may use machine learning models, statistical estimators, simulators, or expert rules. For example, one could include a learned world model or time-series forecaster for market prices, or game-theoretic predictors of other agents' policies. The Strategy System interfaces with the Prediction System by querying it for expected reward $\hat{R}$ and loss $\hat{L}$ given a candidate action. We do not prescribe a specific predictive model here; rather, the design is *philosophically modular* so that any state-of-the-art predictor can be plugged in to forecast the consequences of each action sequence.

5.3 Risk and Validation System

The Risk and Validation System evaluates the monetary outcomes of proposed plans but uses a strictly binary utility structure: only the expected **monetary reward** and **monetary loss** of a plan are considered. In other words, the agent cares only about the difference $R - L$. Traditional risk metrics (e.g. variance, conditional value-at-risk, Sharpe ratio, etc.) are explicitly excluded. The system essentially performs a final check that an action's expected profit exceeds its expected cost. This ensures decisions are driven purely by net gain. In practice, one may set thresholds or guards (e.g. discarding plans with expected loss above a tolerance), but no extra risk penalty is added beyond the loss itself. This reflects a risk-neutral criterion in terms of statistical moments, focusing solely on maximizing expected utility $R - L$.

6 Graph-Based Memory and Recall

The agent maintains a **graph-structured episodic memory** of past decision paths. Each stored "path" corresponds to a sequence of events $(s_0, a_0, r_0), (s_1, a_1, r_1), \dots$ that the agent has experienced. We encode memory as a directed graph $\mathcal{M}$ whose nodes represent state or state-action events and whose directed edges represent transitions in recorded trajectories. Concretely, one can let each node correspond to a visited state s (or a state-action pair (s, a)), and add a directed edge from node s_t to s_{t+1} (or from (s_t, a_t) to (s_{t+1}, a_{t+1})) for each time step of each trajectory. Each edge (or node) carries a weight such as a visit count or value. This is similar to the *Levelled Graph Episodic Memory (LeGEM)* approach, which represents each agent's past trajectories by a sequence of graphs. For example, one can define each subgraph ϕ_t^i containing nodes Ψ and edges Ξ for timestep t, and collect all $\phi_0^i, \dots, \phi_{T-1}^i$ as the memory structure. In practice, we may compress all episodes into a single growing graph where multiple trajectories share nodes/edges; the key point is that the memory graph encodes how the agent moved through states.
Memory evolves over time. Each new experience is appended to the memory graph: if a trajectory $(s_0, a_0, \dots)$ is observed, its sequence of nodes/edges is added or has its visit counts incremented. Simultaneously, unused parts of memory are allowed to **decay**. For example, each edge weight w may be multiplied by a factor $\alpha < 1$ per time-step of non-use (or by $\exp(-\lambda\Delta t)$), so that older paths fade away. This implements the intuition of "forgetting": paths that have not been reinforced by recent experience gradually lose influence. Neuroscience-inspired work supports this: agents that forget old episodic memories can actually perform better, since it "reduces the influence of outdated information and infrequently visited states". In practice, one may also impose a memory capacity: when the number of stored states exceeds a limit, one can remove the least recently used entries. That is, memories not accessed for a long time are overwritten by new data, mimicking an LRU cache (this was done, e.g., by replacing the oldest dictionary entry in an episodic RL controller). The combination of decay and/or explicit forgetting ensures that irrelevant past paths do not clutter decision-making.
Memory **recall** is triggered when the current situation aligns with a stored experience. At each decision point, the agent may search the memory graph for paths relevant to S_t. For instance, one can match the current state S_t against stored state-nodes in the graph: if a node (or sequence of nodes) closely matches S_t (or the recent history), the agent can retrieve the continuation of that path. Concretely, an episodic controller might compute a similarity (e.g. Euclidean or graph distance) between S_t's representation and all memory keys, and select the entry with minimal distance. Then the associated action–return values from memory are used to form a candidate policy for the present state. In effect, if memory contains a successful trajectory starting in a state similar to S_t, the agent can "jump" to follow the rest of that trajectory. This is a form of case-based or experience-based planning. As illustrated in Figure 2 below, one simple scheme is

to treat each visited state as a key with an array of action-value returns (appended or updated on each visit), and on recall use a softmax over the stored returns to generate action probabilities. Figure: *An episodic memory controller. In (A) "storage" mode, visited states are stored as keys in a dictionary, with each key mapping to a value-array of action returns. New experiences append or update these key–value entries (colored bars). In (B) "retrieval" mode, the current state (orange) is compared to stored keys, and the closest key's action–return array is retrieved. A softmax over that return array yields the policy (shown as action probabilities).* The graph-based memory thus allows rapid recall of past strategies: once the current context matches a stored path, the agent can effectively switch to that known action sequence without replanning from scratch.

7 Optimization Objective and Formal Policy

Given the above components, the agent's **optimization problem** is to maximize net monetary utility under full information. Formally, it seeks a policy π^* satisfying

$$\pi^* = \underset{\pi}{argmax}\mathbf{E}\left[\sum_{t=0}^{T} \gamma^t (R(S_t, \pi(S_t)) - L(S_t, \pi(S_t)))\right]$$

Equivalently, one can solve the Bellman optimality equation for the net reward $R - L$ as shown above. Because the agent knows other agents' states and transitions, this expectation is computed exactly (no exploration uncertainty). If other agents' actions are modelled, one can incorporate that into the transition probabilities or best-response function in a leader–follower model. The resulting optimal policy fully aligns with traditional optimal control: at each state it picks the action with the highest expected net return. All performance is measured in cumulative monetary terms (reward vs. loss); the agent does not consider higher moments or risk metrics beyond these scalar outcomes.

8 Algorithmic Outline

- State Observation. At time t, observe the full environment state S_t (including positions of all agents and objects). Because the agent is fully informed, this state is exact.
- Prediction. For each feasible action $a \in A(S_t)$, use the Prediction System to estimate the next state $\hat{S}t + 1$ and immediate outcomes. Compute the predicted reward $\hat{R}(a) = R(S_t, a, \hat{S}t + 1)$ and loss $\hat{L}(a) = L(S_t, a, \hat{S}t + 1)$. More generally, one could roll out $\hat{S}t + 1 : t + H$ for a horizon H if needed.
- Risk Evaluation. For each candidate action a, compute its expected net utility $\mathbf{E}[\hat{R}(a) - \hat{L}(a)]$. Discard or penalize any action that leads to unacceptable loss. (No variance or CVaR is computed; only the difference of expected reward and expected loss matters.)
- Strategy Selection. Compute $Q(S_t, a) = \mathbf{E}[\hat{R}(a) - \hat{L}(a)] + \gamma\mathbf{E}[V(S_{t+1})]$. Select $a^* = \arg\max_a Q(S_t, a)$, as per the Bellman equation. If modelling other agents, this step effectively solves a Stackelberg (leader–follower) optimization, treating followers' responses as known.
- Act and Observe. Execute a^*, observe the actual next state S_{t+1} and realized reward R_t and loss L_t. Update any relevant environment values.
- Memory Update. Append the experienced transition (S_t, a^*, R_t) to the memory graph. Concretely, add or reinforce the edge representing $S_t \to a^* S_{t+1}$ (or node sequence). Increment its

visit count or weight. Apply decay to all other graph weights (e.g. multiply by $\alpha < 1$). If memory capacity is limited, remove or overwrite the least-recently-used entries. This implements forgetting of stale paths.

- Memory Recall (if applicable). Check if the updated state S_{t+1} matches any starting point of stored trajectories. For example, search memory for nodes within small distance of S_{t+1}. If a good match is found, retrieve that path's next actions and values. The agent may then choose to follow the recalled trajectory rather than continuing its original plan. (Effectively, the policy can "jump" to the past sequence if it yields higher reward, replacing the current sub-trajectory.)
- Loop. Set $t \leftarrow t + 1$ and repeat from step 1 until the horizon or termination.

In summary, the agent iteratively predicts outcomes, evaluates net utility, and selects the best action (Maximize expected $R - L$). It uses a memory graph to store and recall useful past experiences, with old paths forgotten over time. All components are defined abstractly so that the Prediction System can use any advanced learning model, and the overall framework remains flexible and extensible for future algorithmic improvements arxiv.org openreview.net . Sources: We draw on standard MDP theory gibberblot.github.io and Stackelberg game concepts proceedings.mlr.press for the strategy logic. The graph-memory design is inspired by recent episodic-memory methods researchgate.net openreview.net . The benefit of memory decay is supported by findings that forgetting irrelevant experiences can improve RL performance frontiersin.org frontiersin.org . Throughout, we restrict to direct monetary outcomes (reward/loss) as specified, avoiding secondary risk measures.