CSCE 631 — Algorithmic Game Theory

Lecture 16: Game Abstraction II

Alan Kuhnle

Summer 2026

Texas A&M University

Today's Plan

Automated Abstraction Algorithms

Potential-Aware Abstraction

Imperfect Recall Abstractions

Action Translation

Blueprint Strategies and Warm Starting

Real-Time Subgame Solving

Nested and Depth-Limited Solving

Post-Processing and Refinement

Connection to Libratus/Pluribus

Summary

Automated Abstraction Algorithms

From Manual to Automated Abstraction

- Lecture 15: the *what* of abstraction (definitions, quality metrics, pathologies)
- This lecture: the *how* — algorithms that automatically construct and exploit abstractions

The Automated Abstraction Pipeline

1. Compute features for each information set (hand strength, equity distribution, etc.)
2. Cluster information sets using a distance metric (EMD, L_2, etc.)
3. Solve the abstract game (e.g., via CFR)
4. Translate the abstract strategy back to the original game

k-Means Clustering for Card Abstraction

- Input: equity histograms $\{F_{h_1}, F_{h_2}, \ldots, F_{h_m}\}$ for all hands at a given game point
- Distance metric: Earth Mover's Distance (from Lecture 15)
- Algorithm: k-means with k = desired number of buckets

Procedure

1. Initialize k cluster centers (e.g., uniformly spaced quantiles)
2. Assign each hand to the nearest center under EMD
3. Recompute centers as the Fréchet mean of assigned histograms
4. Repeat until convergence

Computational cost: $O(m \cdot k \cdot T \cdot B)$ where B = histogram bin count, T = iterations. Parallelizable across game points.

Hierarchical Abstraction

- In multi-round games, abstraction is applied *per round*
- The abstraction at round t depends on the abstraction at round $t-1$
- **Bottom-up**: start from the last round and propagate upward
- **Top-down**: start from the first round and refine downward

Texas Hold'em: Four-Round Hierarchy

Round	Full states	Typical buckets	Compression
Preflop	169	8–20	$\sim 10\times$
Flop	$\sim$ 1,755,000	200–5,000	$\sim 350\times$
Turn	$\sim$ 56,000,000	200–5,000	$\sim 11{,}000\times$
River	$\sim$ 2,500,000,000	200–5,000	$\sim 500{,}000\times$

Potential-Aware Abstraction

Beyond Current Hand Strength

Problem with naïve bucketing: grouping by current hand strength ignores *potential*.

- A flush draw and a made pair may have similar current equity
- But the flush draw has high *positive potential* (probability of improving to a strong hand)
- The made pair has *negative potential* (opponent may overtake on future cards)
- These hands should be played *differently*

Potential-Aware Abstraction (Gilpin et al., 2007)

Key Idea

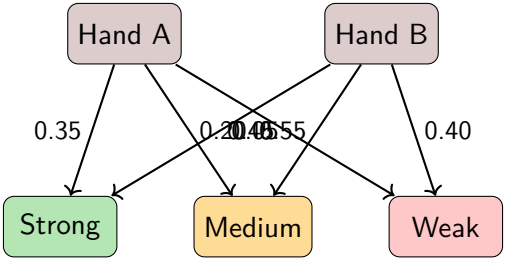
Cluster hands based on their **distribution over future buckets**, not just their current strength.

1. Compute buckets for the *next* round first (bottom-up)
2. For each hand at the current round, compute its transition distribution: $P(b' \mid h, b)$ = probability of landing in future bucket b' given current hand h in bucket b
3. Use this transition distribution as the feature vector for clustering at the current round
4. Distance: EMD between transition distributions

This separates hands with the same current strength but different *futures* — exactly

Potential-Aware: Worked Example

Current round: same bucket (similar equity)



Next round: transition probabilities

Hand A (draw): high variance transitions. Hand B (made hand): concentrated in medium. A potential-aware method places them in *different* buckets despite similar

Imperfect Recall Abstractions

Perfect vs. Imperfect Recall

Perfect Recall

A player has **perfect recall** if, at every information set, the player remembers all their own previous actions and observations. Formally: if h, h' are in the same information set, then the player's action–observation sequences leading to h and h' are identical.

Imperfect Recall

Under **imperfect recall**, a player may "forget" their own past actions or observations. Information sets may group histories with different player action sequences.

- Standard game theory assumes perfect recall (Kuhn's theorem)
- But imperfect recall abstractions can be dramatically smaller

Imperfect Recall in Practice

- **Example**: in round 2 of poker, forget which specific preflop action sequence led to the current state — only remember the pot size
- This can reduce the game tree by orders of magnitude
- But it breaks standard equilibrium guarantees:

Theorem (Piccione & Rubinstein, 1997)
In games with imperfect recall, a Nash equilibrium in behavioral strategies may not exist. The equivalence between behavioral and mixed strategies (Kuhn's theorem) fails.

- Despite this, imperfect recall abstractions are widely used in practice (e.g., in Libratus)
- They work well empirically, even though theoretical guarantees are weaker

Dealing with Imperfect Recall

Approaches to handle imperfect recall safely:

1. **Well-formed imperfect recall** (Lanctot et al., 2012): restrict to abstractions where CFR still converges
2. **Skew-aware abstraction**: weight the contribution of merged histories by their probability of being reached
3. **Post-hoc correction**: solve with imperfect recall, then apply a correction step to recover approximate equilibrium properties
4. **Hybrid**: use imperfect recall for early rounds (where the tree is largest) and perfect recall for later rounds

In practice, approach 4 dominates: coarse early, fine late.

Action Translation

The Action Translation Problem

Recall from Lecture 15: when the opponent plays an action not in the abstraction, we need a principled response.

Formal Problem

Given:

- Abstract game Γ' with action set A' and equilibrium σ'
- Real action $a \notin A'$ played by the opponent

Find: the best response conditioned on a, using only information from σ'.

This is a *real-time* problem: must be solved during play, not precomputed.

Deterministic vs. Randomized Translation

Deterministic (nearest action):

- Map a to $\arg\min_{a' \in A'} |a - a'|$
- Fast but exploitable: opponent can find bets that systematically distort the abstracted response

Randomized (pseudo-harmonic):

- Map a to a *distribution* over neighboring abstract actions
- Preserve the pot odds that the abstract strategy was designed for
- Let $a_L < a < a_R$ be the neighboring abstract actions
- Play a_L's strategy with probability p and a_R's strategy with probability $1 - p$, where:

$$p = \frac{a_R - a}{a_R - a_L}$$

Action Translation: Exploitability Comparison

Translation method	Relative exploitability
No translation (ignore off-tree actions)	Very high
Nearest action (deterministic)	High
Linear interpolation	Medium
Pseudo-harmonic mapping	Low
Real-time subgame solving	Lowest

- Pseudo-harmonic mapping is the best "cheap" option
- Real-time subgame solving (discussed next) eliminates the translation problem entirely, at the cost of computation

Blueprint Strategies and Warm Starting

Blueprint Strategies

Definition: Blueprint Strategy

A **blueprint strategy** is the equilibrium computed for the *full abstracted game* Γ' offline, before play begins. It serves as the default strategy and as the starting point for real-time refinement.

- Computed via CFR+ or MCCFR over Γ' (billions of iterations)
- Stored as a lookup table: information set $\rightarrow$ action probabilities
- The blueprint is *approximate* because Γ' is lossy
- Can be improved in real time via subgame solving

Warm Starting

Warm Starting

When solving a subgame in real time, initialize the strategy to the **blueprint strategy** restricted to that subgame, rather than starting from a uniform strategy.

Benefits:

- Faster convergence: already close to equilibrium
- Better use of limited computation time during play
- The blueprint captures the "shape" of the equilibrium; subgame solving refines the details

Caveat: warm starting with the wrong blueprint can lead CFR to a different equilibrium than cold-starting. In practice, this rarely causes problems because the blueprint is close to optimal.

Real-Time Subgame Solving

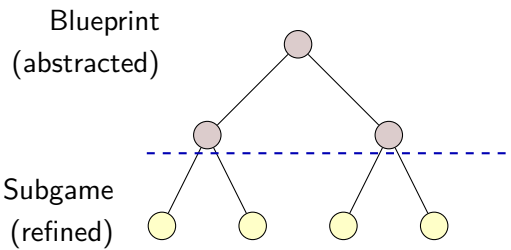
Why Solve Subgames in Real Time?

- The blueprint is computed for an *abstracted* game
- During play, the actual game state may not align with the abstraction (e.g., opponent bets an off-abstraction amount)
- Real-time subgame solving: given the actual game state, solve the remaining subgame more accurately

Key Insight

The abstraction compresses the *full game*, but at any point during play, the remaining subgame is much smaller. We can afford to solve it at higher resolution or with no abstraction at all.

Subgame Solving: Setup



- Above the dashed line: played according to blueprint
- Below: solve with finer (or no) abstraction in real time
- The boundary condition: what would the opponent have done at the root of the subgame?

Naïve Subgame Solving: Unsafe

Naïve approach: solve the subgame independently, ignoring the trunk strategy.

Problem: this can *increase* exploitability.

Why?
The blueprint strategy in the trunk "promises" certain behavior at each information set. If the subgame solution changes the strategy at information sets reachable from the trunk, the opponent can exploit the inconsistency.

Analogy: a negotiator who makes a threat in round 1 but changes their mind in round 2 — the opponent can exploit the inconsistency.

Safe Subgame Solving

Burch et al. (2014); Brown & Sandholm (2017)

A subgame solution is **safe** if it does not increase the overall exploitability compared to the blueprint.

Approach: constrain the subgame solution so that the opponent's *counterfactual value* at the subgame root does not increase (for any opponent hand).

Formally, for each opponent information set I_{-i} at the subgame boundary:

$$v_{-i}^{\text{subgame}}(I_{-i}) \leq v_{-i}^{\text{blueprint}}(I_{-i})$$

- This ensures the opponent is no better off than under the blueprint
- The constraint is applied *per information set*, not on average
- Solving with these constraints uses a modified CFR or LP

Nested and Depth-Limited Solving

Nested Safe Subgame Solving

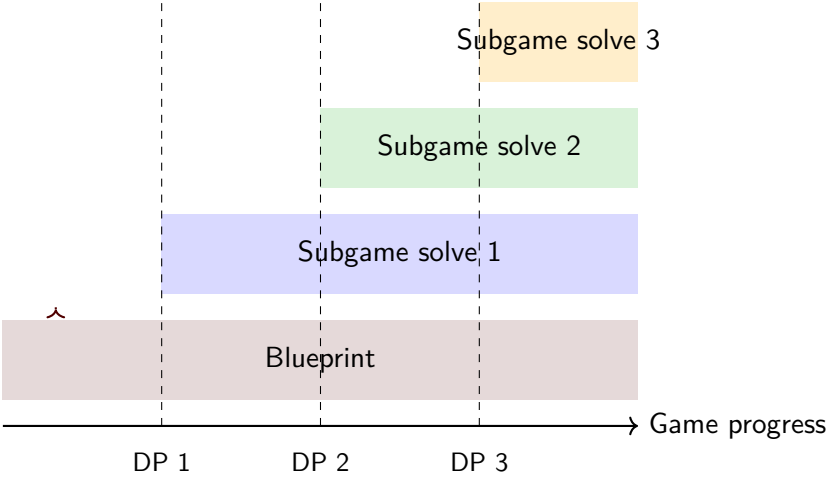
Brown & Sandholm (2017)

Apply safe subgame solving **recursively**: each time a new decision point is reached, re-solve a new subgame rooted at that point.

1. Start with blueprint strategy for the full game
2. At the first decision point: solve the subgame from this point onward (with safety constraint)
3. At the next decision point: solve a new, smaller subgame (with updated safety constraint from step 2)
4. Continue until the game ends

Key property: each nesting level can only *decrease* exploitability (by the safety guarantee). Deeper nesting $\Rightarrow$ better strategy.

Nested Solving: Diagram



Each subgame solve refines the strategy for the remaining game. The safety constraint

Depth-Limited Solving

Problem: even subgames can be too large to solve to the end.

Depth-Limited Solving (Brown et al., 2018)
Solve the subgame only to a limited depth d. At the leaves (depth d), use the **blueprint strategy's values** as terminal payoffs.

- The blueprint provides an estimate of "what happens from here"
- The subgame is solved at full resolution but only for d moves ahead
- When the game reaches depth d, re-solve a new depth-limited subgame (nested depth-limited solving)

Analogy: like chess engines that search to depth d with an evaluation function at the leaves — but for imperfect-information games.

Depth-Limited Solving: Trade-offs

Approach	Depth	Resolution	Speed
Blueprint only	Full	Low	Instant
Full subgame solve	Full	High	Slow
Depth-limited (d)	d moves	High near root	Fast
Nested depth-limited	d per step	High everywhere	Moderate

- Libratus: nested safe subgame solving (no depth limit — possible because heads-up has only 2 players)
- Pluribus: depth-limited solving (necessary because 6-player subgames are too large to solve completely)

Post-Processing and Refinement

Post-Processing the Blueprint

After computing the blueprint, several refinement techniques improve quality:

1. **Purification**: convert mixed strategies to approximately pure strategies (in parts of the tree where mixing is near-deterministic)
2. **Action pruning**: remove actions that are played with very low probability ($< \epsilon$) to speed up real-time solving
3. **Thresholding**: set probabilities below a threshold to zero and renormalize (reduces noise from CFR convergence artifacts)
4. **Grafting**: replace subtrees with hand-crafted strategies for edge cases (e.g., extreme stack sizes)

Self-Improvement Module (Brown & Sandholm, 2018)

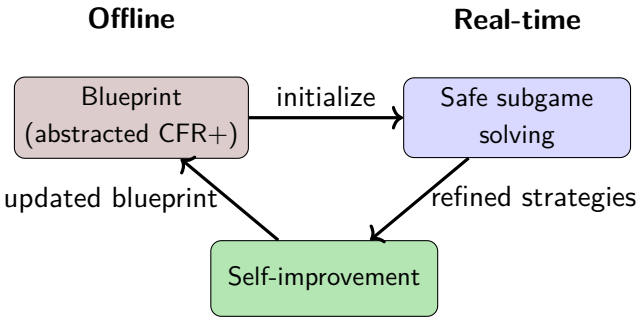
After deploying the blueprint, record the actual strategies played during matches and use them to improve the blueprint.

1. Play matches using the blueprint + subgame solving
2. Record the refined subgame strategies
3. Incorporate these into the blueprint for future games
4. The blueprint improves over time without changing the algorithm

This was used in Libratus: after each day of play against human professionals, the blueprint was updated overnight.

Connection to Libratus/Pluribus

The Libratus/Pluribus Architecture (Preview)



- This lecture covered each component
- Next lecture: detailed case study of Libratus and Pluribus
- The architecture is general — not specific to poker

Summary

Summary

1. **Automated abstraction**: k-means clustering with EMD, hierarchical per-round abstraction, bottom-up computation
2. **Potential-aware abstraction**: cluster by transition distributions to future buckets, not just current strength
3. **Imperfect recall**: dramatic compression, but breaks Kuhn's theorem; handled via well-formed restrictions or hybrid approaches
4. **Action translation**: pseudo-harmonic mapping preserves pot odds; subgame solving eliminates the problem entirely
5. **Blueprint + warm starting**: precomputed approximate equilibrium as starting point for real-time refinement
6. **Safe subgame solving**: constrain so opponent's counterfactual values don't increase — guarantees no exploitability increase
7. **Nested and depth-limited solving**: recursive refinement with bounded