

CS 157: Computational Logic
Fall 2007

General Game Playing

David Haley
Stanford University
Logic Group

Games

- One of the oldest human activities
- Physical, intellectual activity
- Outlet for 'safe' competition

- More recently, games on the computer
- Why not write programs to play games?
- Major motivation for early AI research.

- Surely, if a program can play Chess, it is intelligent... right?

AI and Games

- Most game programs very domain-specific
- Examples:
 - Deep Blue
 - Amazingly good at Chess. Cannot play anything else, even variants of Chess (e.g. Knight-Chess).
 - Chinook
 - recently solved Checkers, but not “tournament-style” Checkers where both players have randomized first moves.

AI and Games

- These programs only play one game, or a set (not class) of predefined games
- Are these intelligent programs or intelligent programm***ers***?

AI and Games

- Learning structures specified by humans
- Heuristic structures specified by humans
- AI agent learns in framework for very precise games developed with careful thought and insight by humans

- What structures or regularities do humans look for in a game to build this framework? Can the search for features be generalized?

General Games

- Instead of playing one kind of game, program should be able to play arbitrary games.
 - It would have to deduce (like humans) what the general physics and principles of the world are.
- Most general form: should be able to interact with arbitrary environment, observe effects, deduce physics, ... like humans.
 - For now: we focus on a formally specified, limited class of games.
- Emphasis not on learning one game; more on feature discovery and transfer learning.

General Game Playing

- Definition from games.stanford.edu:

A General Game Playing System is one that can accept a formal description of an arbitrary game and, without further human interaction, can play the game effectively.

General Game Playing

- Rules sent to player at beginning of game
- Player given time to cogitate
- Only thing known about a game beforehand is the form the rules come in (i.e. the space of possible rules)
 - “*Unknown* rules in a *known* language”
- For simplicity, we only consider a restricted class of games

Considered Games

- Completely specified.
 - Players know all rules (physics) in advance.
- Complete information.
 - Players know everything about the world's state.
- Deterministic.
 - No probabilities. Next state is defined as function of current state and player moves.
- Finite.
 - Finite number of possible game states.
 - One initial state, one or more terminal states.
- Simultaneous move.
 - Not really a restriction (w.r.t. alternating moves).

Considered Games

- Playable.
 - Every player has at least one legal move in every non-terminal state.
- Winnable.
 - Single-player games strongly winnable.
 - There must exist a solution to the puzzle.
 - Multi-player games at least weakly winnable.
 - There must be a way for all players to win (note: not a way to *force* a win)

Game Description Language

- Games are finite state machines.
 - But a finite state machine for games can be huge!
- We need a more compact representation.
- Formal logical language: GDL
 - Subset of KIF (Knowledge Interchange Format)
- Uses (subset of) Relational Logic to define games, states, and transitions.

Logic Language

- Constants: $a, b, \text{white}, \dots$
- Variables: $?x, ?y, \dots$
- Functions (see caveat): $(f\ a\ b\ c)$
- Relations: e.g. $(\text{true}\ (f\ a))$
- Connectives: e.g. $(\text{and}\ (\text{true}\ ?x)\ (\text{true}\ ?y))$
- No quantifiers! Variables are implicitly universal.
- Domain:
 - Herbrand: the domain of discourse is the set of constants. (I.e., domain closure.) Caveat: functions.
 - Unique names: constant a and constant b refer to distinct objects in the domain.

Language Vocabulary

- Game-independent relation constants
 - `(init x)`: `x` is true in initial state
 - `(true x)`: `x` is true in current state
 - `(next x)`: `x` will be true in the next state
 - `(legal x y)`: player `x` may do `y` in current state
 - `(does x y)`: player `x` performs `y` in current state
 - `terminal`: current state is terminal
 - `(goal x y)`: player `x` has obtained score `y`.

Language Vocabulary

- Game-specific constants:
 - Player objects, e.g. `white`, `black`, `wumpus...`
 - Action objects, e.g. `(mark 1 2)`
 - State objects, e.g. `(cell 1 2 x)`
 - aka: transients, fluents, volatiles...
 - View relations, e.g. `(row 1 x)`
 - Type relations, e.g. `(number 1)`

Function Caveat

- Be careful about functions vs. relations
- Consider the function: `(cell 1 2 b)`
 - This is *not* a relation
 - It maps the arguments to the state object saying that cell 1;2 has a blank
 - These 'state objects' exist for a technical reason but are not part of the domain.
- `(true (cell 1 2 b))`
 - The state object is in the true relation.
- Functions not used otherwise, e.g. we do not define $s(1) = 2$.
 - Instead, relations: `(successor 1 2)`

Playing the Game

- Game fully described in logic.
- To reason about the game, need a prover.
- Two main kinds of knowledge:
 - Static information. Basically, the game description. Includes base relations, rules and so forth.
 - Volatile (state) information. What is true in the current state.
 - Implementation trick: split into separate databases for e.g. indexing efficiency

Static Knowledge

- Describes basic facts about game
 - `(player white)`
 - `(init (cell 1 1 b))`
 - `(successor 1 2)`
- Gives rules of the game
 - `(less-than ?x ?y) <= (and (successor ?z ?x)`
`(less-than ?z ?y))`
 - `(next (true (cell 1 1 b)))`
`<= (true (cell 1 1 b))`
 - `(legal ?x (mark ?y ?z))`
`<= (and (true (cell ?y ?z b))`
`(true (control ?x)))`
 - etc.

Volatile Knowledge

- List of things true in current state.
 - (true (cell 1 1 b))
 - (true (cell 1 2 x))
 - (true (cell 1 3 o))
 - (true (cell 2 1 b))
 - (true (cell 1 2 x))
 - (true (cell 1 3 b))
 - ...

No Lexical Assumptions

- Except for built-in relations, vocabulary has *no* semantic meaning.
- For instance: “cell” is simply a string of characters and the player must make no assumptions that “cell” is actually talking about cells on a board.
 - Besides, during competition, names are scrambled before being sent to the players:

```
(next (welcoul ?m ?n himenoing)) <=  
  (and (does poontron (dukepse(?m ?n)))  
    (true (welcoul ?m ?n lorenchise)) )
```

Example

- Tic-Tac-Toe
- Notes on size:
 - Naively, search tree has $9! = 362,880$ states
 - Taking advantage of e.g. symmetry can bring it down considerably
 - Instead of having 9 choices for the first move, there are only three distinct moves modulo board rotation
 - But note compactness of description
- Consider Chess with more than 10^{30} !

Putting it Together

- How does one actually reason about the game?
- Everything (can be) done with a logic prover.
- Standard techniques used:
 - backward-chaining
 - unification
 - answer extraction

Proving Things

- Example proof: is player *white* in control?
 - `(control white)?`
- Basic algorithm:
 - Is it in our knowledge base?
 - Yes: return “proved”
 - Can we unify it with the head of any rule?
 - Yes?
 - Then for every rule whose head we unified with:
 - Prove: unified-rule-body?
 - No?
 - return “can't prove”

Sample Proof Trace

- Find: Rule body: (diagonal o)
Find: (diagonal o)
Rule head: (diagonal o)
Find: Rule body: (true (cell 1 1 o)) &
(true (cell 2 2 o)) &
(true (cell 3 3 o))
Find: (true (cell 1 1 o))
Exit: true
Find: (true (cell 2 2 o))
Exit: true
Find: (true (cell 3 3 o))
Exit: true
Exit: true
Exit: true
Exit: true

Proof Algorithm Sketch

- Prove: φ
 - If $\varphi \in \text{KB.Facts}$ then return true
 - For each rule ρ in KB.Rules :
 - $\sigma = \text{mgu}(\varphi, \text{head}(\rho))$
 - if $\sigma = \text{null}$ then continue
 - prove: $\text{body}(\rho).\text{apply}(\sigma)$
 - if success then return true
 - Return false

Semantics

- Language semantics
 - Basically, normal semantics except for N.A.F.
 - Atoms:
 - $p(x_1, \dots, x_n)$ true iff the tuple x_1, \dots, x_n is in the interpretation for p
 - i.e. if that row is in database table for p
 - Sentences:
 - Normal behavior for conjunction, disjunction, implication
 - Negation as Failure:
 - $\sim\phi$ true if we fail to prove ϕ
 - More later

Equality

- For efficiency, we do not have normal equality.
 - No paramodulation! No demodulation! Fast!
- Instead, we have “distinct”:
 - (distinct a b) iff a and b refer to different objects
- Further restriction: distinct can only take object constants as terms, not functions or variables
- “Fake” equality: (not (distinct a b))

Rule Restrictions

- To guarantee that proofs are decidable and relatively short, we impose two syntactic restrictions on rules:
 - Safety
 - Stratified negation
- Safety:
 - Every variable in the rule head must appear positively in the body
 - Otherwise, we could be returning very large sets of objects!

Safety

- Safe rule:
 - $r(?x, ?y) \leq p(?x) \wedge q(?y)$
- Unsafe rule:
 - $r(?x) \leq p(?y)$
 - This would mean that if p is true of a single object, r is true of everything! Computationally bad.
- Safe rules are *very important*.

Stratified Negation

- A negation is said to be stratified if there is no recursion involved in the negation.
- In other words, you cannot have p depend on $\sim r$, which depends (possibly via other relations) on p again.
- Otherwise, at least these bad consequences:
 - Opens the door to subtle infinite recursion
 - Computationally expensive (like safety)
- All negation must be stratified.
- Having stratified negation is *very important*.

Negation as Failure

- In “classical negation”, we consider $\sim\varphi$ to be true iff $\sim\varphi$ is in our knowledge base (that is, we *know* that $\sim\varphi$ is true);
- In Negation as Failure, we consider $\sim\varphi$ to be true if we fail to prove φ .
 - Prove: $\sim\varphi$
 - If φ in KB then return false
 - For all rule ρ in KB.Rules:
 - $\sigma = \text{mgu}(\varphi, \text{head}(\rho))$
 - If $\sigma = \text{null}$ then continue
 - Prove: $\text{body}(\rho).\text{apply}(\sigma)$
 - If success then return false
 - Return true

More on Proving

- Existential vs. “universal” proofs
 - Is it the case that: `(true (cell 1 1 b))`
 - Give me all `?x` such that: `(true ?x)`
- For existential, we must try all bindings of variables until we find just one that matches.
- For universal, we must try all possible bindings.
- Proofs very efficient to compute: e.g. `(p ?x)`
 - Enumerate all ways to make it true:
 - KB
 - Rules
 - We don't actually have to iterate over the domain!
 - If not there, must be false (NAF)

Answer Extraction

- Very common questions:
 - what is true now? (`true ?x`)
 - what is true in the next state? (`next ?x`)
- Perform a universal proof. Along the way, record all successful variable bindings for the expression you tried to prove.
- This way, we find all possible answers to the question.
 - Note that we still can only report answers actually entailed, not possible solutions to a problem e.g. map coloring.

Answer Extraction

- It turns out that most of the time, we are doing answer extraction.
 - What's true now? What are my moves? If these are the moves, what's true next round?
 - So, very important to be efficient.
- Implementation note:
 - Keeping around and applying all these substitutions and unifying so often is *expensive*.
 - Profiling results: most time is spent allocating/copying/applying substitutions.

Answer Extraction Example

- Consider this this proof:
 - `(true ?x)`
- We might have a rule saying:
 - `(true ?x) <= (q ?x)`
- Assume that `q` is true of constants `a,b,c,d`, and no rules have `q` as their head.
- We can bind `x` to `a,b,c,d` to make `(q ?x)` true. No other way to make `(q ?x)` true.
- Record all proofs of rule body. List of successful substitutions: `?x <- a`, `?x <- b`, etc.

Playing the Game

- Now that we have a proof system, we have to use it to do something useful *vis-à-vis* actually playing the game!
- With infinite resources, we could construct the entire game graph
- Unfortunately, our resources are *very* finite

Player Skeleton

- Skeleton of a player:
 - Obtain rules
 - Think about rules (“start clock”)
 - Initialize game state
 - Main loop:
 - Make a move
 - Wait for other players' moves
 - Update state
 - (next ?x)
 - If game over, stop; otherwise, continue loop

Search Complexity

- Exhaustive search is usually not possible.
 - Tic Tac Toe, naive search tree has roughly ~362k nodes. Big search space but still feasible.
 - But other games (like Chess) have $>10^{30}$ states!
- Need to aggressively optimize search.
- Heuristics!
 - A heuristic is a function that gives an approximate utility of a state. A good heuristic is one that returns a value very close to the actual utility of a state.

Heuristics

- Without heuristics, searching a very large tree is basically random.
 - Read: hopeless.
- In “specific game playing”, programmers write the heuristics.
 - Programmers know what the game is and can think about it (at great length).
 - For Deep Blue, much of the work was in finding a very, very good heuristic framework.
- But general game players don’t know the game beforehand.
 - How to make a heuristic?

Heuristic Generation

- Very important open topic in GGP research.
- Examples:
 - Board detection (dangerous)
 - Mobility (or inverse mobility) (dangerous)
 - Novelty (how much board changed) (dangerous)
 - Goal proximity (not necessarily useful)
 - Goal proximity bis
 - Probabilistic methods

Other Strategies

- Re-writing rules
 - Common DBMS technique.
- Reformulate game.
 - Missionaries and Cannibals.
- Factor game states.
 - Hodgepodge.
 - Step counters.
- Randomly simulate game (i.e. “guess”) and learn from simulation
- Rule of thumb:
 - smarts about the search space worth much more than smarts about constant-time efficiency

Some Practical Results

- Introducing:
Team Camembert



- CS 227b game player
 - Written by Pierre-Yves Lalgand and yours truly

Game Demonstration

- Demonstration: CS227b Final Match
 - Checkers (captures not forced)
- Correctly guessed that pieces were important.
 - Having more pieces is good, having fewer is bad.
- Did not understand that having *even more* pieces is better than having more pieces.

Game Demonstration

- Well, oops.
- Similar story for the demise of players in the annual GGP competition (e.g., guessing the wrong sign on a heuristic...)
- Mobility:
 - More moves for me = better;
less moves for them = better.
 - Right? Wrong...
- Take-home message: heuristics are dangerous

Timing Results

- Time to find first legal moves (legal me ?x)
 - Chess
 - With caching: 0.7 s Without: 7.6 s
 - Mini-chess
 - With caching: 39 ms Without: 53 ms
 - Tic-Tac-Toe
 - With caching: 3 ms Without: 3 ms
 - Othello
 - With caching: 0.3 s Without: 1.3 s
- If it takes almost a second just to find your moves, how are you going to meaningfully search?

Caching

- Caching always helpful? No. Metric: Finding complete game solution using Alpha-Beta.
 - Tic-Tac-Toe
 - With caching: 16 s. Without: 14.7 s.
 - Mini-chess
 - With caching: 570 ms. Without: 287 ms. (Ouch.)
- Caching benefits depend a great deal on the nature of the game rules. Big, hard negations are a good thing to cache. Small, easy rules constructed from KB are not good to cache.
 - As cache size increases, cache lookup time increases.

More GGP

- Lots of open areas of research.
 - Heuristics (of course)
 - Transfer learning
 - Surely if I am very good at Checkers, I will be at least half-decent when playing Checkers with forced captures
 - Opponent modeling
 - Play the game from opponent's perspective
 - But what is “rational” behavior? Are players even rational, given how little of the search tree they might see?
 - What about cooperative games?

More GGP

- Stanford CS227B
 - Project-based course: implement a GGP player!
- Annual GGP competition at AAAI
 - see <http://games.stanford.edu>
- Shameless plug: 'Jocular' GGP player framework available for download:
 - Implements a basic but fully functional player. Uses a minimax search; no heuristics.
 - Written in Java.
 - <http://games.stanford.edu/players/player-jocular.html>