Propositional Nets

Metagaming

Metagaming is match-independent game processing, i.e. game processing that is done independent of any particular opponent or any particular state.

Objective of metagaming - to optimize performance in playing specific matches of the game.

Usually done offline, i.e. during the startclock or between moves or in parallel with regular game play.
Examples

Boring:
Headstart on Game-Graph Search
Endgame book

Statistical:
Machine Learning

Engineering:
Compilation (machine language, fpga’s)

Structural:
Change of Framework (e.g. state machines to propnets)
Game Reformulation (e.g. game decomposition)

Benefits

Methods discussed so far take time proportional to the size of the game graph.

Propositional nets can be processed in time proportional to the size of the game graph but with much lower coefficient.

Propositional Nets can be exponentially smaller. They lend themselves to analysis and many properties can be discovered in time proportional to this smaller size instead of the size of the game graph.
Games as State Machines

States versus Features

In many cases, worlds are best thought of in terms of features, e.g. red or green, left or right, high or low. Individual actions often affect individual features or small subsets of features.

States represent all possible ways the world can be. As such, the number of states is exponential in the number of “features” of the world, and the action tables are correspondingly large.

Idea - represent features directly and describe how actions change individual features rather than entire states. (Reference: STRIPS.)
Tictactoe Example

\[
\begin{array}{c|c|c}
\text{cell}(1,1,x) & \text{cell}(1,1,x) \\
\text{cell}(1,2,b) & \text{cell}(1,2,b) \\
\text{cell}(1,3,b) & \text{cell}(1,3,o) \\
\text{cell}(2,1,b) & \text{cell}(2,1,b) \\
\text{cell}(2,2,o) & \text{cell}(2,2,o) \\
\text{cell}(2,3,b) & \text{cell}(2,3,b) \\
\text{cell}(3,1,b) & \text{cell}(3,1,b) \\
\text{cell}(3,2,b) & \text{cell}(3,2,b) \\
\text{cell}(3,3,x) & \text{cell}(3,3,x) \\
\text{control}(black) & \text{control}(white) \\
\end{array}
\]

\[
\begin{align*}
\text{black} & \rightarrow \text{mark}(1,3) \\

\text{next}(\text{cell}(M,N,o)) & : - \\
\text{does}(\text{black},\text{mark}(M,N)) & \land \\
\text{true}(\text{cell}(M,N,b)) & \\
\end{align*}
\]

Propositions

Decompose states into “propositions”.

Benefit - \( n \) propositions can represent \( 2^n \) states.
Propositional Net Components

Propositions

Connectives

Transitions

Propositional Net
Markings

A *marking* for a propositional net is a function from the propositions \( P \) to boolean values.

\[
m: P \rightarrow \{true, false\}
\]

A marking \( m \) is *partial* if and only if \( m \) is a partial function. Otherwise, it is *total*.

Think of a marking as a state of a game.

Acceptability

A marking is *acceptable* if and only if it obeys the logical properties of all connectives.

Negation with input \( x \) and output \( y \):

\[
m(x)=false \iff m(y)=true
\]

Conjunction with inputs \( x \) and \( y \) and output \( z \):

\[
m(x)=true \land m(y)=true \iff m(z)=true
\]

Disjunction with inputs \( x \) and \( y \) and output \( z \):

\[
m(x)=true \lor m(y)=true \iff m(z)=true
\]
Definitions

A transition is *enabled* by a marking \( m \) if and only if all of its inputs are marked *true*.

The *transitional marking* for \( m \) is the partial marking that assigns *true* to the outputs of all transitions enabled by \( m \) and *false* to the outputs of all others.

An *input marking* for a propositional net is a marking for the extrinsic propositions.

Update

The *update* \( m' \) of a marking \( m \) (with transitional marking \( t \)) and input marking \( e \) is any acceptable marking consistent with \( t \) and \( e \).

*Algorithm for update*: (1) Given \( m \), compute enabled transitions and form a transitional marking \( t \). (2) Take as input an input marking \( i \). (3) Erase all marks from propnet. (4) Compute acceptable marking \( m' \) from \( t \) and \( I \) using the rules of acceptability.

*Theorem*: This algorithm produces exactly one update for each initial marking and input marking.
Example

Buttons and Lights

Pressing button $a$ toggles $p$.
Pressing button $b$ interchanges $p$ and $q$. 
Propositional Net for Buttons and Lights

Tic-Tac-Toe

X

O

X
Partial Propositional Net for Tic-Tac-Toe

Propositional Net Fragment
Propositional GDL

row1o :-
  true(11o),
  true(12o),
  true(13o)
row1x :-
  true(11x),
  true(12x),
  true(13x)
row2o :-
  true(21o),
  true(22o),
  true(23o)
row2x :-
  true(21x),
  true(22x),
  true(23x)
row3o :-
  true(31o),
  true(32o),
  true(33o)
row3x :-
  true(31x),
  true(32x),
  true(33x)

Performance

? (time (genwinnerp tttrel))
  5,478 states
  130,444 milliseconds.
  142,685,136 bytes of memory allocated.
  NIL

? (time (genwinnerp tttprop))
  5,478 states
  594,555 milliseconds
  117,281,008 bytes of memory allocated.
  NIL

? (time (propwinnerp tttprop))
  5,478 states
  10,390 milliseconds.
  5,900,904 bytes of memory allocated.
  NIL
Compilation

Specialized Data Structures for state
e.g. Boolean propositional net
  1 bit per proposition
  Convert rules to bit operations
e.g. Relational net
  vector of relations

Compile the Game Description into machine code
subroutine to compute view prop/rel from state
update routine to compute base prop/rel from state

Propositional GDL

\[
\begin{align*}
(\leq & (\text{next } 11x) \\
& (\text{does white mark11})) \\
(\leq & (\text{next } 11x) \\
& (\text{true } 11x)) \\
(\leq & (\text{next } 11o) \\
& (\text{does black mark11})) \\
(\leq & (\text{next } 11o) \\
& (\text{true } 11o)) \\
(\leq & (\text{next } 11b) \\
& (\text{true } 11b) \\
& (\text{not } (\text{does white mark11})) \\
& (\text{not } (\text{does black mark11})))
\end{align*}
\]
Example

(defun tttinit ()
  (vector nil nil t nil nil t nil nil t
         nil nil t nil nil t nil nil t
         nil nil t nil nil t nil nil t
         t nil))

(defun true11x (state) (elt state 1))
(defun true11o (state) (elt state 2))
(defun true11b (state) (elt state 3))

(defun next11x (input state)
  (or (does 'white 'mark11 input) (true11x state)))

(defun next11o (input state)
  (or (does 'black 'mark11 input) (true11o state)))

(defun next11b (input state)
  (and (true11b state)
       (not (does 'white 'mark11 input))
       (not (does 'black 'mark11 input))))

Performance

? (time (genwinnerp tttprop))
  5,478 states
  594,555 milliseconds
  117,281,008 bytes of memory allocated.
  NIL

? (time (propwinnerp tttprop))
  5,478 states
  10,390 milliseconds.
  5,900,904 bytes of memory allocated.

? (time (compwinnerp tttprop))
  5,478 states
  855 milliseconds
  3,475,432 bytes of memory allocated.
  NIL
  Compilation time: 234 milliseconds
Example

(defun pbvinit () #b000000000000000000)
(defun x11 () #b100000000000000000)
(defun true11x (state)
  (= (logand state #b110000000000000000)
      #b100000000000000000))
(defun true11o (state)
  (= (logand state #b110000000000000000)
      #b010000000000000000))
(defun true11b (state)
  (= (logand state #b110000000000000000)
      #b000000000000000000))
(defun newsimulate (input state)
  (logior input state))

Performance

? (time (genwinnerp tttprop))
  5,478 states
  594,555 milliseconds
  117,281,008 bytes of memory allocated.
  NIL

? (time (compwinnerp tttprop))
  5,478 states
  855 milliseconds
  3,475,432 bytes of memory allocated.
  NIL

? (time (pbvwinnerp tttprop))
  5,478 states
  234 milliseconds
  64 bytes of memory allocated.
  NIL