General Game Playing and Monte Carlo Tree Search

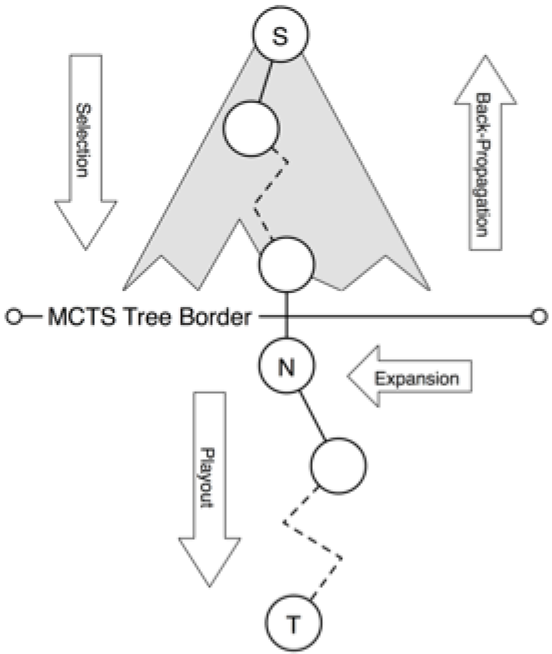
Armando Ramirez

Paper Critique

Background: Game Description Language

- *role(a)* means that *a* is a role in the game.
- *base(p)* means that *p* is a base proposition in the game.
- *input(r,a)* means that *a* is an action for role *r*.
- *init(p)* means that the proposition *p* is true in the initial state.
- *true(p)* means that the proposition *p* is true in the current state.
- *does(r,a)* means that player *r* performs action *a* in the current state.
- *next(p)* means that the proposition *p* is true in the next state.
- *legal(r,a)* means it is legal for role *r* to play action *a* in the current state.
- *goal(r,n)* means that player the current state has utility *n* for player *r*.
- *terminal* means that the current state is a terminal state.

Background: Monte-Carlo Tree Search



- *Selection* Choose amongst next-states
- *Playout* Randomly play a game to completion, noting the result
- *Expansion* The tree is typically grown one state at a time
- *Back-Propogation* Results of each state's simulation must be propogated to all of its ancestors

Paper Critique

Generalized Monte-Carlo Tree Search Extensions for General Game Playing

Summary

- Multiple extensions for MCTS are proposed
- All the extensions make no game-specific assumption so that they will be applicable to general game playing.
- *Goal Stability Early Cutoff* If a simulation seems to be reasonably trending towards a conclusion, terminate early
- *Terminal Interval Early Cutoff* If a game tends to terminate in a certain interval of turns,
- *Unexplored Action Urgency* "Exploit the fringe" by not always exploring all children

Critique

- Took great care to make all extensions use no game-specific knowledge
- Some extensions are not applicable for many games on Stanford's website (such as Tic-Tac-Toe and even Chinese Checkers)
- No assumptions about game are made, but usefulness is determined by game description.

Finnsson, H. 2012. Generalized monte-carlo tree search extensions for general game playing. In AAAI.

Paper Critique

Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search

Summary

- Identifies three factors that affect MCTS's effectiveness in imperfect information games:
- *Leaf Correlation* How much control a player has over their fate at a certain point in the game
- *Bias* Whether or not a game inherently favors a player
- *Disambiguation Factor* In imperfect information games, how much the possibilities shrink with each revelation

Critique

- Thorough testing of theory, with both synthetic game trees and actual games.
- Could be very applicable to GDL-II, which includes stochastic and imperfect information games
- MCTS can be exploited by an opponent

Long, J. R.; Sturtevant, N. R.; Buro, M.; and Furtak, T. 2010. Understanding the success of perfect information monte carlo sampling in game tree search. In AAAI.

Paper Critique

Monte-Carlo tree search and rapid action value estimation in computer Go

Summary

- Exhaustive mathematical explanation of Monte-Carlo Tree Search and related concepts and algorithms
- Proposes two extensions
- *Rapid Action Value Estimation (RAVE)* Speeds up MCTS vastly thanks to parallel tree nodes sharing information, but reduces effectiveness
- *MC-RAVE* Combines RAVE with more Monte-Carlo simulations in order to increase effectiveness while retaining speed

Critique

- Only applicable to computer Go
- Results were convincingly displayed
- While not directly applicable to my implementation (GGP), the mathematical explanation still was helpful

Gelly, S., and Silver, D. 2011. Monte-carlo tree search and rapid action value estimation in computer go. Artificial In- telligence 175(11):1856–1875.

Implementation

Background: core.logic

Functions vs Relations

The plus function

```
REPL:> (+ 2 3)
5
```

The plus relation

```
REPL:> (run* [q] (+ 2 3 q))
(5)
REPL:> (run* [q] (+ 2 q 5))
(3)
REPL:> (run* [q] (+ q 3 5))
(2)
REPL:> (run* [q] (+ 2 3 5))
(._0) ;; SUCCESS
REPL:> (run* [q] (+ 2 3 7))
() ;; FAILURE
```

- The (run* [q] RELATION) macro means “give me all the values of q such that the relation (with q substituted in accordingly) succeeds”

Background: core.logic

More complex example

English

p is a pair of whole numbers (x,y) such that x + y = 3 or x + y = 4

Background: core.logic

More complex example

English

p is a pair of whole numbers (x,y) such that x + y = 3 or x + y = 4

core.logic

```
(defn pair-example [p]
```

Background: core.logic

More complex example

English

p is a pair of whole numbers (x,y) such that x + y = 3 or x + y = 4

core.logic

```
(defn pair-example [p]
  (fresh [x y]
    (== [x y] p)
```

Background: core.logic

More complex example

English

p is a pair of whole numbers (x,y) such that x + y = 3 or x + y = 4

core.logic

```
(defn pair-example [p]
  (fresh [x y]
    (== [x y] p)
    (in x y (interval 0 Infinity))
```

Background: core.logic

More complex example

English

p is a pair of whole numbers (x,y) such that x + y = 3 or x + y = 4

core.logic

```
(defn pair-example [p]
  (fresh [x y]
    (== [x y] p)
    (in x y (interval 0 Infinity))
    (conde
      [(+ x y 3)]
      [(+ x y 4)])))
```

Background: core.logic

More complex example

English

p is a pair of whole numbers (x,y) such that x + y = 3 or x + y = 4

core.logic

```
(defn pair-example [p]
  (fresh [x y]
    (== [x y] p)
    (in x y (interval 0 Infinity))
    (conde
      [(+ x y 3)]
      [(+ x y 4)])))
```

It works!

```
REPL:> (run* [q] (pair-example [0 3]))
(._0) ;; SUCCESS

REPL:> (run* [q] (pair-example [10 10]))
() ;; FAILURE

REPL:> (run* [q] (pair-example q))
([0 3] [0 4] [1 2]
 [1 3] [2 1] [2 2]
 [3 0] [3 1] [4 0])
```

Motivation: English->GDL->core.logic

English

It is Legal for player **w** to mark cell (**m**,**n**) if it is true that cell (**m**,**n**) is blank and it is true that it is player **w**'s turn to move. If it is X's turn to move, O can only noop. If it is O's turn to move, X can only noop. That is, they take turns.

Motivation: English->GDL->core.logic

English

It is Legal for player **w** to mark cell (**m**,**n**) if it is true that cell (**m**,**n**) is blank and it is true that it is player **w**'s turn to move. If it is X's turn to move, O can only noop. If it is O's turn to move, X can only noop. That is, they take turns.

GDL

```
(<= (legal ?w (mark ?m ?n))
    (true (cell ?m ?n b))
    (true (control ?w)))

(<= (legal X noop)
    (true (control O)))

(<= (legal O noop)
    (true (control X)))
```

Motivation: English->GDL->core.logic

English

It is Legal for player **w** to mark cell (**m**,**n**) if it is true that cell (**m**,**n**) is blank and it is true that it is player **w**'s turn to move. If it is X's turn to move, O can only noop. If it is O's turn to move, X can only noop. That is, they take turns.

GDL

```
(<= (legal ?w (mark ?m ?n))
    (true (cell ?m ?n b))
    (true (control ?w)))

(<= (legal X noop)
    (true (control O)))

(<= (legal O noop)
    (true (control X)))
```

core.logic

```
(defn legal [role action]
 (conde
  [(fresh [?w ?m ?n]
    (== role ?w)
    (== action [:mark ?m ?n])
    (true [:cell ?m ?n :b])
    (true [:control ?w]))]
  [(== role :X)
   (== action :noop)
   (true [:control :O])]
  [(== role :O)
   (== action :noop)
   (true [:control :X])]))
```

Motivation: Use of core.logic legal relation

Legality checking

```
REPL:> (run* [q] (legal :X [:mark 1 1]))
(._0) ;; SUCCESS
REPL:> (run* [q] (legal :O [:mark 1 1]))
() ;; FAILURE. O cannot mark any square
```

Legal move generation

```
REPL:> (run* [q] (legal :X q))
([:mark 3 3] [:mark 3 2] [:mark 3 1]
 [:mark 2 3] [:mark 2 2] [:mark 2 1]
 [:mark 1 3] [:mark 1 2] [:mark 1 1])
REPL:> (run* [q] (legal :O q))
(:noop)
```

Implementation: GDL->core.logic

The Environment

- Runs through arbitrary GDL description and generates an *environment* that represents game rules and state
- Each relation now takes an extra argument of the environment so that it can call other relations in the GDL translation:

```
;; :legal
(fn [env role action]
 (conde
  [(fresh [?w ?m ?n]
    (== role ?w)
    (== action [:mark ?m ?n])
    ((get-relation env :true) [:cell ?m ?n :b])
    ((get-relation env :true) [:control ?w]))]
  [(== role :X)
   (== action :noop)
   ((get-relation env :true) [:control :O])]
  [(== role :O)
   (== action :noop)
   ((get-relation env :true) [:control :X])]))

(defn get-relation [env r]
  (partial (env r) env))
```

Implementation: GDL->core.logic

Pre-processing

```
(collect-relations gdl ;;-->

{:legal
  {:args [env G__6139 G__6140]
   :body (quote ((<= (legal ?w (mark ?m ?n))
                     (true (cell ?m ?n b))
                     (true (control ?w)))
                 (<= (legal X noop)
                     (true (control O)))
                 (<= (legal O noop)
                     (true (control X)))))}
 ;; etc
}
```

Implementation: GDL->core.logic

Translation rules

Reflexive head calls turn into unifications of args

```
;; Transforming Legal
(legal ?w (mark ?x ?y))
;; -~->
(== role ?w)
(== move [:mark ?x ?y])
```

All other relations reference the environment

```
(true (cell 1 1 b))
;; -~->
((get-relation env :true) [:cell 1 1 :b])
```

not turns into negation as failure constraint (nafc)

```
(not (line X))
;; -~->
(nafc (env :line) env :X)
```

Implementation: GDL->core.logic

Translation rules

Relation are joined in a fresh block

```
(<= (head & args)
    & tail)
;; -~->
(fresh [fresh-vars]
 (transformed head)
 (transformed tail))
```

Multiple related relations are joined by a disjunction (conde)

```
(relation1)
(relation2)
;; ...
(relationN)
;; -~->
(conde
 [(transformed relation1)]
 [(transformed relation2)]
 ;; ...
 [(transformed relationN)])
```

Implementation: MCTS

Algorithm skeleton

```
(defn mcts-iteration
  "Performs one iteration of MCTS on state env, using initial
  statistics stats, playing as player"
  [env stats player]
  (loop [env env, stats stats, path (list env)]
    (if (terminal? env)
      (mcts-backprop path (get-scores env) stats)
      (if (not-empty (mcts-unexplored env stats))
        (mcts-grow stats path)
        (let [ch (mcts-select env stats player)]
          (recur ch stats (cons ch path)))))))
```

Implementation: Goal-Stability Early Cutoff

Pseudo-code from the article

Algorithm 1 Pseudo-code for deciding cuts for the Early Cutoff extension

```
if not useEarlyCutoff then
    return false
end if
if playoutSteps < minimumSteps then
    return false
end if
if IsGoalStable() then
    // Cutoff point has been calculated as:
    // cut ← firstGoalChange + numPlayers
    return playoutSteps ≥ cut
end if
if hasTerminalInterval() then
    // Cutoff point has been calculated as:
    // cut ← firstTerminal + 0.33 * terminalInterval
    return playoutSteps ≥ cut
end if
```

Snippet of Clojure code

```clojure
(if (or (terminal? env)
        (and use-early-cutoff?
             (is-goal-stable? goals threshold)
             (>= steps cut)))
     scores
     (recur (rand-nth (gen-children env)) next-goals (inc steps)))
```

Future Work

- Thoroughly test and refactor some of the design in order to publish code
- core.typed could be a helpful option for more rigor
- Support GDL-II
- Implement interactive game engines backed by GDL->core.logic
- Implement a variety of other extensions to GGP MCTS
- Optimize my MCTS
- Implement a "timeout" options for MCTS

Conclusion

Today I have presented

- **An overview of General Game Playing and Monte-Carlo Tree Search**
- **Critiques of three recent papers on Monte-Carlo Tree Search**
- **A simple introduction to logic programming with Clojure core.logic**
- **A functional Game Description Language to Clojure core.logic translator**
- **A functional Monte-Carlo Tree Search implementation in Clojure using the output of my GDL->core.logic translator**
- **An implemented extension to MCTS from a critiqued paper**