Using Domain-specific Knowledge for Monte Carlo Tree Search in Go

Martin Müller
University of Alberta
NCTU, August 2015
Contents

- Introduction - why use domain knowledge?
- Many kinds of knowledge in Go
- How to acquire
- How to use
- Research problems
Format of Talk

- Informal talk, much is unpublished, work in progress
- I have more questions than answers...
- I use our Fuego program as an example
Many Types of Knowledge in Go

- Rules, if-then-else...
- Patterns
- Deep neural networks
- Search control knowledge
- Exact knowledge, e.g. proven wins
- And more...

Credits: sciencedaily.com
About Fuego

- Fuego is:
  - A Game-independent MCTS framework
  - A Go program
  - Open source
  - Mostly developed at University of Alberta

- Many other programs use Fuego as basis (e.g. MoHex)
- Many researchers have used Fuego for experiments
The Fuego Go Program

- Developed since 2008, based on older Go program Explorer
- Uses Monte Carlo Tree Search (MCTS), RAVE, prior knowledge
- MoGo-style rule-based simulations (+ some changes)
- Lock-free multithreading
- In 2009, won 9x9 game on even vs Chou Chun-Hsun
- Won the 2009 Computer Olympiad 9x9 and 2010 UEC Cup (19x19)
- MP-Fuego: massively parallel version (TDS-df-UCT, Yoshizoe) uses up to 2000 cores
- Strength: Fuego on good PC about 1 dan, MP-Fuego maybe 3 dan
Types of Knowledge in Fuego

- Part 1: Simulations (very short here)
- Part 2: In-tree knowledge (a lot)
  - Rules, features, “Greenpeep” patterns
- Part 3: “Slow” knowledge (some)
  - DCNN
  - Tactical search
- [Part 4: Exact knowledge - not today]
Part 1: Simulations

- **Fuego**: Rule-based, as in MoGo
  - Select move from highest-ranked rule that produces at least one move
- **Alternative**: probability-based, as in Crazy Stone
  - Weight map over all legal moves
  - Used to select the next move to play in simulation
- Speed about 1,000,000 moves/second/core
Research Questions

- What works in simulations?
  - Right now, we still mostly use trial-and-error

- How to design an effective playout policy?

- How to evaluate a policy? (without playing thousands of test games)

- What distinguishes a good from a bad policy?
Part 2: In-Tree Knowledge

- Evaluated for each node in the game tree
- Used in UCT formula to select best child in tree
- Big influence on shape of tree
- Speed goal: about 1000 nodes/second/core
Using In-Tree Knowledge

- Assume you have some knowledge. What do you do with it?
- Three main approaches in the literature
- Two are used in Fuego
  - Initialize playout statistics with “fake” wins and losses
  - Add a third term to the UCB formula: mean + exploration + knowledge
Third Way: Iterative Widening

- Consider only N best moves
- Increase N over time
- Never tried in Fuego
Fuego’s In-Tree Knowledge

1. Oldest: hand-coded rules, “fake” wins and losses

2. Next: “Greenpeep” patterns, additive knowledge

3. Recent: Feature learning using Latent Factor Ranking
1. Handcoded Rules

- Simple, crude rules (from 2008)
  - Bonus for moves in corner and on 3rd line
  - Bonus for moves in low-liberty situations (e.g., ladders)
  - Bonus for moves from the simulation policy
- Weights (number of wins/losses) tuned manually
2. “Greenpeep” Patterns

- Greenpeep was the name of a Go program by Chris Rosin
- Greenpeep used 12 point diamond-shaped patterns with extra knowledge (liberty counts)
- Chris developed a machine learning technique based on self play to train weights
- “Additive” knowledge in Fuego, about 130 Elo improvement (about 2010)
- Theory: C. Rosin, Multi-armed bandits with episode context, ISAIM 2010
3. Feature Learning Using Latent Factor Ranking

- Work on feature learning
  - Remi Coulom, Computing Elo Ratings of Move Patterns in the Game of Go, 2007
  - Later improved by Coulom and Aja Huang
  - Wistuba and Schmidt-Thieme, Move Prediction in Go – Modelling Feature Interactions Using Latent Factors, KI 2013
Main change:

Model pairwise interactions between features

Example: A and B may be OK features by themselves, but A and B together is really good
Main Ideas in Feature Learning

- Moves are described by a set of features, e.g. pattern, tactics, location, distance
- Assign Weights to features to maximize “move prediction”:
- Try to guess which move was played by a strong human player
Feature Details

features_move 03
FE_EXTENSION_NOT_LADDER
FE_LINE_3
FE_DIST_PREV_3
FE_GOUCT_ATARI_DEFEND
FE_GOUCT_PATTERN
FE_POS_6
FE_GAME_PHASE_3
FE_CFG_DISTANCE_LAST_2
FE_CFG_DISTANCE_LASTOWN_4_OR_MORE
FE_SAVE_STONES_1
WB
EEE
BBB

features_move K2
FE_ATARI_LADDER
FE_LINE_2
FE_DIST_PREV_10
FE_POS_10
FE_GAME_PHASE_3
FE_CFG_DISTANCE_LAST_4_OR_MORE
FE_CFG_DISTANCE_LASTOWN_4_OR_MORE
FE_KILL_STONES_2
EEE
EEE
BWW
Example in Fuego

- Simple features
  + 3x3 patterns
- Trained weights with 20000 master games
- blue = good
- green = bad
Current Work on Features in Fuego

- By Chenjun Xiao
- Add large patterns, not just 3x3
  - Almost done…
- New algorithm for training
  - (Slightly) better results than Wistuba
  - Produces probabilities for moves being best, not just “some numbers”
Part 3: Slow Knowledge

- Too slow to compute at every node in the search
- Can still be useful
- Two Examples:
  - Deep neural network
  - Tactical search
Deep Convolutional Neural Networks (DCNN)

- Introduced for Go in two recent publications
  - Clark and Storkey, JMLR 2015
  - Maddison, Huang, Sutskever and Silver, ICLR 2015
- Very strong move prediction rates, 55.2% (Maddison et al)
- Slow to train and use (even with GPU)
DCNN in Fuego

- We use networks trained by Storkey and Henrion (Storkey’s new student)
- Integrated in Fuego by Andrew Jacobsen (my student)
Features vs DCNN

Feature Knowledge

DCNN Evaluation
Some Examples of Bad DCNN Moves
Research Questions

- How to learn when:
  - Move is *usually* bad, but good here (e.g. empty triangle example)
  - Move is *usually* good, but bad here (e.g. cut example)

- Training based on statistics of “similar” examples cannot help - unless definition of “similar” is *extremely* good

- How to catch these cases by exploration in MCTS
How to use Slow Knowledge?

- Solution in Fuego
  - Threshold N, e.g. N=200
  - Call slow knowledge for all nodes that reach N simulations
  - For large N, this is a very small percentage of all nodes
  - Can do something expensive
Problem: knowledge is only called after many simulations

MCTS may not be changed much

How to balance?

Better call right away? But for which nodes?

Our DCNN-Fuego prototype calls DCNN first, but only at root
Tactical Search

- Observation: Fuego often makes simple tactical mistakes
  - Example: “geta”, capture by net
- Can be solved by a small tactical search
- Our old program Explorer contains such a search
- Use as slow knowledge, give bonus to moves that save or capture
- About 70-80 Elo improvement for simple implementation
Other Ideas for Knowledge

(not implemented in Fuego)

- Local Life and Death search
- Semeai (capturing races)
- Prove safety, or invade/defend territories
- Local searches to filter which moves make sense locally
Discussion

- Many kinds of knowledge used in Go
- Old programs were mostly about encoding knowledge
- First MCTS programs used very little, but it is all coming back
- Want to use machine learning to deal with large amounts of knowledge
- Self-play or learn from human master games
Simulation policies are still “magic”

Probably the biggest differences between top programs and open source programs are in this area

Need scientific principles to design better policies
Discussion (3)

- Integrating “slow” knowledge is a big challenge
- How to “mix” it with a MCTS?
- We have only crude solutions (threshold, root-only)
- Can we predict which nodes are important, so we can call slow knowledge immediately?
Summary

- Reviewed knowledge in MCTS Go programs, especially Fuego
- Many imperfect, incomplete solutions
- Many different but overlapping approaches
- Can we unify them based on a good theory?
- Still much work to be done to understand and improve
- What we do in Go can help other applications