Monte-Carlo Tree Search

Michèle Sebag

TAO: Theme Apprentissage & Optimization

Acknowledgments: Olivier Teytaud, Sylvain Gelly, Philippe Rolet, Romaric Gaudel

CP 2012
Foreword

Motivations

- CP evolves from “Model + Search” to “Model + Run”: ML needed
- Which ML problem is this?
Wanted: For any problem instance, automatically

▶ Select algorithm/heuristics in a portfolio
▶ Tune hyper-parameters

A general problem, faced by

▶ Constraint Programming
▶ Stochastic Optimization
▶ Machine Learning, too...
1. Case-based learning / Metric learning

**Input**
- Observations

**Output**
- For any new instance, retrieve the nearest case
- (but what is the metric?)
2. Supervised Learning

Input
- Observations
- Target (best alg.)

Output: Prediction
- Classification
- Regression
From decision to sequential decision

In each restart, predict the best heuristics
... it might solve the problem;
otherwise the description is refined; iterate

Can we do better: Select the heuristics which will bring us where we’ll be in good shape to select the best heuristics to solve the problem...
3. Reinforcement learning

Features

▶ An agent, temporally situated
▶ acts on its environment
▶ in order to maximize its cumulative reward

Learned output

A policy mapping each state onto an action
Formalisation

Notations

- State space $\mathcal{S}$
- Action space $\mathcal{A}$
- Transition model
  - deterministic: $s' = t(s, a)$
  - probabilistic: $P^a_{s,s'} = p(s, a, s') \in [0, 1]$.
- Reward $r(s)$ bounded
- Time horizon $H$ (finite or infinite)

Goal

- Find policy (strategy) $\pi : \mathcal{S} \mapsto \mathcal{A}$
- which maximizes cumulative reward from now to timestep $H$

$$
\pi^* = \arg\max \mathbb{E}_{s_{t+1} \sim p(s_t, \pi(s_t), s)} \left[ \sum r(s_t) \right]
$$
Reinforcement learning

Context
In an uncertain environment,
Some actions, in some states, bring (delayed) rewards [with some probability].

Goal: find the policy (state $\rightarrow$ action)
maximizing the expected cumulative reward
This talk is about sequential decision making

- Reinforcement learning:
  First learn the optimal policy; then apply it

- Monte-Carlo Tree Search:
  Any-time algorithm: learn the next move; play it; iterate.
MCTS: computer-Go as explanatory example
Not just a game: same approaches apply to optimal energy policy
MCTS for computer-Go and MineSweeper

Go: deterministic transitions
MineSweeper: probabilistic transitions
The game of Go in one slide

Rules

▶ Each player puts a stone on the goban, black first
▶ Each stone remains on the goban, except:
  - Group w/o degree freedom is killed
  - A group with two eyes can’t be killed
▶ The goal is to control the max. territory
Go as a sequential decision problem

Features

- Size of the state space $2 \times 10^{170}$
- Size of the action space 200
- No good evaluation function
- Local and global features (symmetries, freedom, ...)
- A move might make a difference some dozen plies later
Setting

- State space $S$
- Action space $A$
- Known transition model: $p(s, a, s')$
- Reward on final states: win or lose

Baseline strategies do not apply:
- Cannot grow the full tree
- Cannot safely cut branches
- Cannot be greedy

Monte-Carlo Tree Search
- An any-time algorithm
- Iteratively and asymmetrically growing a search tree
  most promising subtrees are more explored and developed
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Monte-Carlo Tree Search

Gradually grow the search tree:

- **Iterate Tree-Walk**
  - **Building Blocks**
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
  - Update information in visited nodes

- Returned solution:
  - Path visited most often

- Bandit phase
- Random phase, roll-out
- Evaluate
- Propagate
Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
    - Evaluate
  - Update information in visited nodes
    - Propagate

- Returned solution:
  - Path visited most often
Gradually grow the search tree:

▶ Iterate Tree-Walk
  ▶ Building Blocks
    ▶ Select next action
  ▶ Add a node
    Grow a leaf of the search tree
  ▶ Select next action bis
    Random phase, roll-out
  ▶ Compute instant reward
    Evaluate
  ▶ Update information in visited nodes
    Propagate

▶ Returned solution:
  ▶ Path visited most often
Gradually grow the search tree:

- **Iterate Tree-Walk**
  - **Building Blocks**
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
    - Evaluate
  - Update information in visited nodes
    - Propagate

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
    - Select next action bis
      - Random phase, roll-out
  - Compute instant reward
- Evaluate
- Update information in visited nodes
  - Propagate

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
    - Evaluate
  - Update information in visited nodes
    - Propagate

- Returned solution:
  - Path visited most often

Kocsis Szepesvári, 06
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
    - Evaluate
  - Update information in visited nodes
    - Propagate

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
  - Update information in visited nodes
    - Evaluate

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
  - Update information in visited nodes

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
  - Update information in visited nodes
- Returned solution:
  - Path visited most often

Kocsis Szepesvári, 06
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
    - Evaluate
  - Update information in visited nodes
    - Propagate

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- **Iterate Tree-Walk**
  - **Building Blocks**
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
  - Update information in visited nodes

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
    - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
  - Evaluate
  - Update information in visited nodes

- Returned solution:
  - Path visited most often
Monte-Carlo Tree Search

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
    - Evaluate
  - Update information in visited nodes
    - Propagate

- Returned solution:
  - Path visited most often
MCTS Algorithm

**Main**

**Input:** number $N$ of tree-walks

Initialize search tree $\mathcal{T} \leftarrow$ initial state

**Loop:** For $i = 1$ to $N$

\[
\text{TreeWalk}(\mathcal{T}, \text{initial state})
\]

**EndLoop**

**Return** most visited child node of root node
**Tree walk**

**Input:** search tree $\mathcal{T}$, state $s$

**Output:** reward $r$

**If** $s$ is not a leaf node

- Select $a^* = \arg\max \left\{ \hat{\mu}(s, a), tr(s, a) \in \mathcal{T} \right\}$
- $r \leftarrow \text{TreeWalk}(\mathcal{T}, tr(s, a^*))$

**Else**

- $A_s = \{ \text{admissible actions not yet visited in } s \}$
- Select $a^*$ in $A_s$
- Add $tr(s, a^*)$ as child node of $s$
- $r \leftarrow \text{RandomWalk}(tr(s, a^*))$

**End If**

Update $n_s, n_{s,a^*}$ and $\hat{\mu}_{s,a^*}$

**Return** $r$
Random walk

Input: search tree $T$, state $u$
Output: reward $r$

$A_{rnd} \leftarrow \{\} // store the set of actions visited in the random phase$

While $u$ is not final state
    Uniformly select an admissible action $a$ for $u$
    $A_{rnd} \leftarrow A_{rnd} \cup \{a\}$
    $u \leftarrow tr(u, a)$
EndWhile

$r = Evaluate(u) // reward vector of the tree-walk$
Return $r$
Monte-Carlo Tree Search

Properties of interest

- Consistency: $\Pr(\text{finding optimal path}) \to 1$ when the number of tree-walks go to infinity
- Speed of convergence; can be exponentially slow.

Coquelin Munos 07
Comparative results

2012  MoGoTW used for physiological measurements of human players
2012  7 wins out of 12 games against professional players and 9 wins out of 12 games against 6D players

2011  20 wins out of 20 games in 7x7 with minimal computer komi
2011  First win against a pro (6D), H2, 13×13
2011  First win against a pro (9P), H2.5, 13×13
2011  First win against a pro in Blind Go, 9×9
2010  Gold medal in TAAI, all categories
      19×19, 13×13, 9×9

2009  Win against a pro (5P), 9×9 (black)
2009  Win against a pro (5P), 9×9 (black)
2008  in against a pro (5P), 9×9 (white)
2007  Win against a pro (5P), 9×9 (blitz)
2009  Win against a pro (8P), 19×19 H9
2009  Win against a pro (1P), 19×19 H6
2008  Win against a pro (9P), 19×19 H7
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Action selection as a Multi-Armed Bandit problem

In a casino, one wants to maximize one’s gains *while playing*.

Lifelong learning

Exploration vs. Exploitation Dilemma

- Play the best arm so far?
- But there might exist better arms...

Lai, Robbins 85
The multi-armed bandit (MAB) problem

- $K$ arms
- Each arm gives reward 1 with probability $\mu_i$, 0 otherwise
- Let $\mu^* = \arg\max\{\mu_1, \ldots, \mu_K\}$, with $\Delta_i = \mu^* - \mu_i$
- In each time $t$, one selects an arm $i_t^*$ and gets a reward $r_t$

$$
\begin{align*}
n_{i,t} &= \sum_{u=1}^{t} \mathbb{1}_{i_u^* = i} \quad \text{number of times } i \text{ has been selected} \\
\hat{\mu}_{i,t} &= \frac{1}{n_{i,t}} \sum_{i_u^* = i} r_u \quad \text{average reward of arm } i
\end{align*}
$$

Goal: Maximize $\sum_{u=1}^{t} r_u$

$\Leftrightarrow$

Minimize Regret ($t$) = $\sum_{u=1}^{t} (\mu^* - r_u) = t\mu^* - \sum_{i=1}^{K} n_{i,t} \hat{\mu}_{i,t} \approx \sum_{i=1}^{K} n_{i,t} \Delta_i$
The simplest approach: $\epsilon$-greedy selection

At each time $t$,

- With probability $1 - \epsilon$
  select the arm with best empirical reward
  
  $$i_t^* = \arg\max \{\hat{\mu}_{1,t}, \ldots, \hat{\mu}_{K,t}\}$$

- Otherwise, select $i_t^*$ uniformly in $\{1 \ldots K\}$

\[
\text{Regret}\ (t) > \epsilon t \frac{1}{K} \sum_i \Delta_i
\]

Optimal regret rate: $\log(t)$

Lai Robbins 85
Select $i^*_t = \arg\max \left\{ \hat{\mu}_{i,t} + \sqrt{C \frac{\log(\sum n_{j,t})}{n_{i,t}}} \right\}$

Decision: Optimism in front of unknown!
Upper Confidence bound, followed

UCB achieves the optimal regret rate \( \log(t) \)

\[
\text{Select } i_t^* = \arg\max \left\{ \hat{\mu}_{i,t} + \sqrt{c_e \frac{\log(\sum n_{j,t})}{n_{i,t}}} \right\}
\]

Extensions and variants

- Tune \( c_e \) control the exploration/exploitation trade-off
- UCB-tuned: take into account the standard deviation of \( \hat{\mu}_i \):
  
  \[
  \text{Select } i_t^* = \arg\max \left\{ \hat{\mu}_{i,t} + \sqrt{c_e \frac{\log(\sum n_{j,t})}{n_{i,t}}} + \min \left( \frac{1}{4}, \hat{\sigma}_{i,t}^2 + \sqrt{c_e \frac{\log(\sum n_{j,t})}{n_{i,t}}} \right) \right\}
  \]

- Many-armed bandit strategies
- Extension of UCB to trees: UCT Kocsis & Szepesvári, 06
Monte-Carlo Tree Search. Random phase

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action
  - Add a node
    - Grow a leaf of the search tree
  - Select next action bis
    - Random phase, roll-out
  - Compute instant reward
  - Update information in visited nodes

- Returned solution:
  - Path visited most often
Random phase — Roll-out policy

Monte-Carlo-based Brügman 93

1. Until the goban is filled, add a stone (black or white in turn) at a uniformly selected empty position
2. Compute $r = \text{Win(black)}$
3. The outcome of the tree-walk is $r$
Random phase — Roll-out policy

Monte-Carlo-based

1. Until the goban is filled, add a stone (black or white in turn) at a uniformly selected empty position
2. Compute $r = \text{Win(}\text{black})$
3. The outcome of the tree-walk is $r$

Improvements?

- Put stones randomly in the neighborhood of a previous stone
- Put stones matching patterns prior knowledge
- Put stones optimizing a value function

Silver et al. 07
Evaluation and Propagation

The tree-walk returns an evaluation \( r \) \( \text{win(black)} \)

Propagate

- For each node \((s, a)\) in the tree-walk

\[
\begin{align*}
\hat{n}_{s,a} & \leftarrow \hat{n}_{s,a} + 1 \\
\hat{\mu}_{s,a} & \leftarrow \hat{\mu}_{s,a} + \frac{1}{\hat{n}_{s,a}}(r - \mu_{s,a})
\end{align*}
\]
Evaluation and Propagation

The tree-walk returns an evaluation \( r \) and \( \text{win(black)} \)

Propagate

- For each node \((s, a)\) in the tree-walk

\[
\begin{align*}
    n_{s,a} &\leftarrow n_{s,a} + 1 \\
    \hat{\mu}_{s,a} &\leftarrow \hat{\mu}_{s,a} + \frac{1}{n_{s,a}}(r - \mu_{s,a})
\end{align*}
\]

Variants

Kocsis & Szepesvári, 06

\[
\hat{\mu}_{s,a} \leftarrow \begin{cases} 
    \min\{\hat{\mu}_x, x \text{ child of } (s, a)\} & \text{if } (s, a) \text{ is a black node} \\
    \max\{\hat{\mu}_x, x \text{ child of } (s, a)\} & \text{if } (s, a) \text{ is a white node}
\end{cases}
\]
Dilemma

- smarter roll-out policy →
  more computationally expensive →
  less tree-walks on a budget

- frugal roll-out →
  more tree-walks →
  more confident evaluations
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Action selection revisited

Select \( a^* = \text{argmax} \left\{ \hat{\mu}_{s,a} + \sqrt{c_e \log(n_s)} \right\} \)

- Asymptotically optimal
- But visits the tree infinitely often!

Being greedy is excluded

Frugal and consistent

Select \( a^* = \text{argmax} \frac{\text{Nb win}(s,a) + 1}{\text{Nb loss}(s,a) + 2} \)  

Berthier et al. 2010

Further directions

- Optimizing the action selection rule

Maes et al., 11
Controlling the branching factor

What if many arms degenerates into exploration?

- **Continuous heuristics**
  Use a small exploration constant $c_e$

- **Discrete heuristics**
  Progressive Widening
  Coulom 06; Rolet et al. 09

Limit the number of considered actions to $\lfloor b\sqrt{n(s)} \rfloor$
(usually $b = 2$ or $4$)

Introduce a new action when $\lfloor b\sqrt{n(s)} + 1 \rfloor > \lfloor b\sqrt{n(s)} \rfloor$
(which one? See RAVE, below).
RAVE: Rapid Action Value Estimate

Motivation

- It needs some time to decrease the variance of $\hat{\mu}_{s,a}$
- Generalizing across the tree?

$$RAVE(s, a) = \text{average } \{\hat{\mu}(s', a), s \text{ parent of } s'\}$$

global RAVE
local RAVE
Using RAVE for action selection

In the action selection rule, replace \( \hat{\mu}_{s,a} \) by

\[
\alpha \hat{\mu}_{s,a} + (1 - \alpha) \left( \beta RAVE_\ell(s, a) + (1 - \beta) RAVE_g(s, a) \right)
\]

\[
\alpha = \frac{n_{s,a}}{n_{s,a} + c_1}
\]

\[
\beta = \frac{n_{\text{parent}(s)}}{n_{\text{parent}(s)} + c_2}
\]

Using RAVE with Progressive Widening

- PW: introduce a new action if \( \lceil \sqrt[n]{n(s)} \rceil + 1 > \lfloor \sqrt[n]{n(s)} \rfloor \)
- Select promising actions: it takes time to recover from bad ones
- Select \( \text{argmax } RAVE_\ell(\text{parent}(s)) \).
A limit of RAVE

- Brings information from bottom to top of tree
- Sometimes harmful:

B2 is the only good move for white
B2 only makes sense as first move (not in subtrees)
⇒ RAVE rejects B2.
Improving the roll-out policy $\pi$

$\pi_0$ Put stones uniformly in empty positions

$\pi_{\text{random}}$ Put stones uniformly in the neighborhood of a previous stone

$\pi_{\text{MoGo}}$ Put stones matching patterns prior knowledge

$\pi_{\text{RLGO}}$ Put stones optimizing a value function Silver et al. 07

Beware! Gelly Silver 07

$\pi$ better $\pi'$ $\not\Rightarrow$ $MCTS(\pi)$ better $MCTS(\pi')$
Improving the roll-out policy $\pi$, followed

$\pi_{RLGO}$ against $\pi_{random}$

$\pi_{RLGO}$ against $\pi_{MoGo}$

Evaluation error on 200 test cases
Interpretation

What matters:

- Being **biased** is more harmful than being weak...
- Introducing a stronger but biased rollout policy $\pi$ is detrimental.

if there exist situations where you (wrongly) think you are in good shape then you go there and you are in bad shape...
Using prior knowledge

Assume a value function $Q_{prior}(s, a)$

Then when action $a$ is first considered in state $s$, initialize

$$n_{s,a} = n_{prior}(s, a) \quad \text{equivalent experience / confidence of priors}$$

$$\mu_{s,a} = Q_{prior}(s, a)$$

The best of both worlds

- Speed-up discovery of good moves
- Does not prevent from identifying their weaknesses
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Parallelization. 1 Distributing the roll-outs

Distributing roll-outs on different computational nodes does not work.
Parallelization. With shared memory

- Launch tree-walks in parallel on the same MCTS
- (micro) lock the indicators during each tree-walk update.

Use virtual updates to enforce the diversity of tree walks.
Parallelization. 3. Without shared memory

Launch one MCTS per computational node

\( k \) times per second

- Select nodes with sufficient number of simulations
  \[ > 0.05 \times \# \text{ total simulations} \]

- Aggregate indicators

Good news
Parallelization with and without shared memory can be combined.
It works!

<table>
<thead>
<tr>
<th>32 cores against</th>
<th>Winning rate on $9 \times 9$</th>
<th>Winning rate on $19 \times 19$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$75.8 \pm 2.5$</td>
<td>$95.1 \pm 1.4$</td>
</tr>
<tr>
<td>2</td>
<td>$66.3 \pm 2.8$</td>
<td>$82.4 \pm 2.7$</td>
</tr>
<tr>
<td>4</td>
<td>$62.6 \pm 2.9$</td>
<td>$73.5 \pm 3.4$</td>
</tr>
<tr>
<td>8</td>
<td>$59.6 \pm 2.9$</td>
<td>$63.1 \pm 4.2$</td>
</tr>
<tr>
<td>16</td>
<td>$52 \pm 3.$</td>
<td>$63 \pm 5.6$</td>
</tr>
<tr>
<td>32</td>
<td>$48.9 \pm 3.$</td>
<td>$48 \pm 10$</td>
</tr>
</tbody>
</table>

Then:

- Try with a bigger machine! and win against top professional players!
- Not so simple... there are diminishing returns.
Increasing the number $N$ of tree-walks

<table>
<thead>
<tr>
<th>$N$</th>
<th>2$N$ against $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Winning rate on 9 × 9</td>
</tr>
<tr>
<td>1,000</td>
<td>71.1 ± 0.1</td>
</tr>
<tr>
<td>4,000</td>
<td>68.7 ± 0.2</td>
</tr>
<tr>
<td>16,000</td>
<td>66.5 ± 0.9</td>
</tr>
<tr>
<td>256,000</td>
<td>61 ± 0.2</td>
</tr>
</tbody>
</table>
The limits of parallelization

R. Coulom

Improvement in terms of performance against humans

≪

Improvement in terms of performance against computers

≪

Improvements in terms of self-play
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Failure: Semeai
Failure: Semeai
Failure: Semeai
Failure: Semeai
Failure: Semeai
Failure: Semeai
Failure: Semeai
Failure: Semeai
Why does it fail

- First simulation gives 50%
- Following simulations give 100% or 0%
- But MCTS tries other moves: doesn’t see all moves on the black side are equivalent.
Implication 1

MCTS does not detect invariance $\rightarrow$ too short-sighted and parallelization does not help.
Implication 2

MCTS does not build abstractions → too short-sighted and parallelization does not help.
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
MCTS for one-player game

- The MineSweeper problem
- Combining CSP and MCTS
Motivation

▶ All locations have same probability of death \( \frac{1}{3} \)
▶ Are then all moves equivalent?
Motivation

- All locations have same probability of death $\frac{1}{3}$
- Are then all moves equivalent? NO!
Motivation

- All locations have same probability of death $\frac{1}{3}$
- Are then all moves equivalent? NO!
- Top, Bottom: Win with probability $\frac{2}{3}$
Motivation

▶ All locations have same probability of death $1/3$
▶ Are then all moves equivalent? NO!
▶ Top, Bottom: Win with probability $2/3$
▶ MYOPIC approaches LOSE.
MineSweeper, State of the art

Markov Decision Process
Very expensive; $4 \times 4$ is solved

Single Point Strategy (SPS)
local solver

CSP
- Each unknown location $j$, a variable $x[j]$
- Each visible location, a constraint, e.g. $\text{loc}(15) = 4 \rightarrow$


- Find all $N$ solutions
- $P(\text{mine in } j) = \frac{\text{number of solutions with mine in } j}{N}$
- Play $j$ with minimal $P(\text{mine in } j)$
Constraint Satisfaction for MineSweeper

State of the art

- 80% success beginner \((9\times9, 10 \text{ mines})\)
- 45% success intermediate \((16\times16, 40 \text{ mines})\)
- 34% success expert \((30\times40, 99 \text{ mines})\)

**PROS**

- Very fast

**CONS**

- Not optimal
- Beware of first move
  (opening book)
Upper Confidence Tree for MineSweeper

Couetoux Teytaud 11

- Cannot compete with CSP in terms of speed
- But consistent (find the optimal solution if given enough time)

Lesson learned
- Initial move matters
- UCT improves on CSP

- 3x3, 7 mines
- Optimal winning rate: 25%
- Optimal winning rate if uniform initial move: 17/72
- UCT improves on CSP by 1/72
UCT for MineSweeper

Another example

- 5x5, 15 mines
- GnoMine rule (first move gets 0)
- if 1st move is center, optimal winning rate is 100 %
- UCT finds it; CSP does not.
The best of both worlds

CSP
- Fast
- Suboptimal (myopic)

UCT
- Needs a generative model
- Asymptotic optimal

Hybrid
- UCT with generative model based on CSP
UCT needs a generative model

Given
- A state, an action
- Simulate possible transitions

Initial state, play top left

Simulating transitions
- Using rejection (draw mines and check if consistent) SLOW
- Using CSP FAST
The algorithm: Belief State Sampler UCT

- One node created per simulation/tree-walk
- Progressive widening
- Evaluation by Monte-Carlo simulation
- Action selection: UCB tuned (with variance)
- Monte-Carlo moves
  - If possible, Single Point Strategy (can propose riskless moves if any)
  - Otherwise, move with null probability of mines (CSP-based)
  - Otherwise, with probability .7, move with minimal probability of mines (CSP-based)
  - Otherwise, draw a hidden state compatible with current observation (CSP-based) and play a safe move.
The results

- **BSSUCT**: Belief State Sampler UCT
- **CSP-PGMS**: CSP + initial moves in the corners

<table>
<thead>
<tr>
<th>Format</th>
<th>CSP-PGMS</th>
<th>BSSUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 mines on 4x4</td>
<td>64.7 %</td>
<td>70.0% ± 0.6% (2000 games)</td>
</tr>
<tr>
<td>1 mine on 1x3</td>
<td>100 %</td>
<td>100%</td>
</tr>
<tr>
<td>3 mines on 2x5</td>
<td>22.6%</td>
<td>25.4 % ± 1.0%</td>
</tr>
<tr>
<td>10 mines on 5x5</td>
<td>8.20%</td>
<td>9% (p-value: 0.14)</td>
</tr>
<tr>
<td>5 mines on 1x10</td>
<td>12.93%</td>
<td>18.9% ± 0.2%</td>
</tr>
<tr>
<td>10 mines on 3x7</td>
<td>4.50%</td>
<td>5.96% ± 0.16%</td>
</tr>
<tr>
<td>15 mines on 5x5</td>
<td>0.63%</td>
<td>0.9% ± 0.1%</td>
</tr>
</tbody>
</table>
Partial conclusion

Given a myopic solver

- It can be combined with MCTS / UCT:
  - Significant (costly) improvements
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Active Learning, position of the problem

Supervised learning, the setting

- Target hypothesis $h^*$
- Training set $\mathcal{E} = \{(x_i, y_i), i = 1 \ldots n\}$
- Learn $h_n$ from $\mathcal{E}$

Criteria

- Consistency: $h_n \to h^*$ when $n \to \infty$.
- Sample complexity: number of examples needed to reach the target with precision $\epsilon$

$$\epsilon \to n_\epsilon \text{ s.t. } \|h_n - h^*\| < \epsilon$$
Active Learning, definition

Passive learning

\[ \mathcal{E} = \{(x_i, y_i), i = 1 \ldots n\} \]

Active learning

\( x_{n+1} \) selected depending on \( \{(x_i, y_i), i = 1 \ldots n\} \)
In the best case, exponential improvement:

- **PASSIVE:**
- **ACTIVE:**
A motivating application

Numerical Engineering

- Large codes
- Computationally heavy \(\sim\) days
- not fool-proof

Inertial Confinement Fusion, ICF
Goal

Simplified models

- Approximate answer
- ... for a fraction of the computational cost
- Speed-up the design cycle
- Optimal design

*More is Different*

Alternative scheme: spherical target with a gold cone*

* Kodama et al. Nature 412 798 (2001); 418 933 (2002);
Active Learning as a Game

Optimization problem

\[ F^* = \arg\min_{E_{h \sim \mathcal{A}(\mathcal{E}, \sigma, T)}} \text{Err}(h, \sigma, T) \]

\( \mathcal{E} \): Training data set
\( \mathcal{A} \): Machine Learning algorithm
\( \mathcal{Z} \): Set of instances
\( \sigma : \mathcal{E} \mapsto \mathcal{Z} \) sampling strategy
\( T \): Time horizon
\( \text{Err} \): Generalization error

Bottlenecks

- Combinatorial optimization problem
- Generalization error unknown
Where is the game?

- Wanted: a good strategy to find, as accurately as possible, the true target concept.
- If this is a game, you play it only once!
- But you can train...

**Training game: Iterate**

- Draw a possible goal (fake target concept $h^*$); use it as oracle.
- Try a policy (sequence of instances $E_{h^*,T} = \{(x_1, h^*(x_1)), \ldots (x_T, h^*(x_T))\}$)
- Evaluate: Learn $h$ from $E_{h^*,T}$. Reward $= ||h - h^*||$
Overview

Motivations

Monte-Carlo Tree Search
  Multi-Armed Bandits
  Random phase
  Evaluation and Propagation

Advanced MCTS
  Rapid Action Value Estimate
  Improving the rollout policy
  Using prior knowledge
  Parallelization

Open problems

MCTS and 1-player games
  MCTS and CP
  Optimization in expectation

Conclusion and perspectives
Conclusion

Take-home message: MCTS/UCT

▶ enables any-time smart look-ahead for better sequential decisions in front of uncertainty.
▶ is an integrated system involving two main ingredients:
  ▶ Exploration vs Exploitation rule        UCB, UCBtuned, others
  ▶ Roll-out policy
▶ can take advantage of prior knowledge

Caveat

▶ The UCB rule was not an essential ingredient of MoGo
▶ Refining the roll-out policy ≠ refining the system
   Many tree-walks might be better than smarter (biased) ones.
On-going, future, call to arms

Extensions

- Continuous bandits: action ranges in a $\mathbb{R}$  
  Bubeck et al. 11
- Contextual bandits: state ranges in $\mathbb{R}^d$  
  Langford et al. 11
- Multi-objective sequential optimization  
  Wang Sebag 12

Controlling the size of the search space

- Building abstractions
- Considering nested MCTS (partially observable settings, e.g. poker)
- Multi-scale reasoning
Bibliography

- Vincent Berthier, Hassen Doghmen, Olivier Teytaud: Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search. LION 2010: 111-124
- Romaric Gaudel, Michèle Sebag: Feature Selection as a One-Player Game. ICML 2010: 359-366
Sylvain Gelly, David Silver: Combining online and offline knowledge in UCT. ICML 2007: 273-280

Levente Kocsis, Csaba Szepesvári: Bandit Based Monte-Carlo Planning. ECML 2006: 282-293

Francis Maes, Louis Wehenkel, Damien Ernst: Automatic Discovery of Ranking Formulas for Playing with Multi-armed Bandits. EWRL 2011: 5-17

Arpad Rimmel, Fabien Teytaud, Olivier Teytaud: Biasing Monte-Carlo Simulations through RAVE Values. Computers and Games 2010: 59-68

David Silver, Richard S. Sutton, Martin Müller: Reinforcement Learning of Local Shape in the Game of Go. IJCAI 2007: 1053-1058

Olivier Teytaud, Michèle Sebag: Combining Myopic Optimization and Tree Search: Application to MineSweeper, LION 2012.