Monte Carlo Tree Search

CompSci 590.2
Ron Parr
Duke University

A different view of how to plan

• So far, we have (mostly) assumed that we can compute a value function or policy in one big computation and use them for execution
• Exception: TD Gammon
  • Computes value function
  • Combines value function with search before each move

• What if we emphasized search more?
Searching before acting – “on line planning”

• Requires an accurate simulator
  • True for TD Gammon
  • Sensible assumption for most games
• Requires time to plan/search before each action - may not be practical for control problems
• Does not necessarily require planning for the entire state space, but
• Potentially wastes resources by continually replanning

Straw man

• Build a complete search tree out to depth $d$
• Alternate between action nodes and chance nodes
• Choose $d$ so that $\gamma^d R_{\text{max}}$ is small
• Solve for policy in this tree recursive from leaves to root

• Problem:
  • $b = \text{branching factor} = (\text{#of actions} \times \text{#possible next states})$
  • $b^d \text{ nodes}$
Remove dependence on #next states

- Kearns et al. introduced trajectory trees
- Instead of considering all next states, sample next states
- Still branch on all actions
- Generate multiple trees instead one fat tree
- Evaluate potential policies against trees – value of policy is average value across trees

- Replaces dependence on #of next states with:
  - Dependence on VC dimension of policy space (linear), $1/e^2$, $\log(1/\delta)$
  - # of trees needed to get good average evaluation of policies

Trajectory tree example

Kearns et al.
Trajectory tree limitations

- Main problem remains exponential dependence on $d$
- Each tree can still be very big
- Even if the number of trees isn’t as bad as you might expect, still very expensive to do in practice

A different approach: Bandits

- Bandit problem:
  - Multiple slot machines with unknown expected payoffs
  - Need strategy for playing arms so that learn which slot machine is best without too much opportunity cost of learning
- Regret: Difference between what you got and what you could have gotten if you played optimally
- Goal: Algorithms with bounded regret
UCB1

**Deterministic policy:** UCB1.

**Initialization:** Play each machine once.

**Loop:**
- Play machine $j$ that maximizes $\bar{x}_j + \sqrt{\frac{2\ln n}{n_j}}$, where $\bar{x}_j$ is the average reward obtained from machine $j$, $n_j$ is the number of times machine $j$ has been played so far, and $n$ is the overall number of plays done so far.

From Auer et al., who show that UCB1 has regret \textit{logarithmic in} $n$.

---

**Application to online planning**

- Since we are using a simulator, we don’t care so much about regret.

- BUT: Don’t still don’t want to waste time.

- Idea: What if we view each state as a sort of bandit problem when we explore a tree of possible outcomes from our current state?
Generic Monte Carlo Tree Search

1: function MonteCarloPlanning(state)
2: repeat
3:   search(state, 0)
4: until Timeout
5: return bestAction(state,0)

6: function search(state, depth)
7:   if Terminal(state) then return 0
8:   if Leaf(state, d) then return Evaluate(state)
9:   action := selectAction(state, depth)
10:  (nextstate, reward) := simulateAction(state, action)
11:  q := reward + \gamma \text{search}(nextstate, depth + 1)
12:  UpdateValue(state, action, q, depth)
13:  return q

From Kocsis & Szepesvari

Understanding UpdateValue

• Update value computes average value of descendants in the tree
• UCT includes an exploration bonus:
  • $C \sqrt{\frac{\log N(s)}{N(s,a)}}$
  • $C = \text{sqrt}(2)$ for bandits
• Issues:
  • Unlike bandits, some updates can include “stale” values from children, i.e.,
    value of a node should reflect value of acting optimally for node’s children,
    but we update as we learn, so child values may not be right
  • How do you pick $C$?
  • Memory
Staleness

• K&S show that for sufficiently large C, we will converge to the correct values and action at the root
• Intuition:
  • Eventually, the leaf values will start converging to the correct values
  • If C is big enough, then we’ll get enough samples for parents of these nodes to converge, overwhelming errors from earlier iterations
  • Apply this idea inductively

How to pick C

• Not much practical guidance here
• In practice, this will need to be very large
• Why?
  • Leaf values still matter
  • May need exponential number of steps to find leaf values with high rewards
  • No inherent way around this

• In practice:
  • Make C big enough so that you burn all the time you have
  • Works better than it should in many cases
Memory

• What if you can’t afford to maintain value estimates for every node you encounter?
• Note: On modern computers, you can run out of memory very quickly!
• When you hit a node you don’t want to store the value for:
  • “Rollout”
  • Forward simulate to the end of the horizon using the current or random policy, and use this value
  • Does this make sense?

Go

• Ancient game that involves placing black/white stones on a lattice
• 9x9, 13x13, 19x19 (standard) versions
• Surround other players stones to capture and remove from board
• Objective: Maximize number of stones of your color on the board
Why Go is hard

• ~200 moves per turn vs. ~37 in chess
• ~300 turns per game vs. ~57 in chess
• $10^{170}$ possible positions vs. $10^{47}$ in chess

• Evaluation is subtle – number of pieces on the board at any time is not in itself very predictive of outcome
• Very difficult to learn/invent a good evaluation function

MCTS for Go

• Classical approaches to Go did not do very well – nowhere close to master level play
• MCTS was a big improvement
• Tricks:
  • Parallelization
  • When/how to do rollouts
  • What policy to use for rollouts
  • Sharing information across subtrees
  • Using databases of expert moves when possible
Does this work for other games?

• Kind of, but not all

• Not a big win for chess

• What’s happening?
  • No practical way to pick D big enough to satisfy conditions for theoretical convergence to optimal behavior
  • Can’t explore the entire (remaining) tree except very close to end of game
  • Rollouts are very important for estimating the value of the truncated tree
Rollouts: Chess vs. Go speculation

- Go positions are hard to evaluate, but perhaps at a certain point, the good ones and bad ones have **wide paths** towards certain outcomes that are hard to miss with sampling.

- Chess tends to have very narrow paths, so that even towards the end of the game, getting towards a particular outcome can be like threading a needle – hard to find with sampling.