Monte Carlo Tree Search

CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Logistics & Recap
2. Plain MCTS
3. Refinements: Child selection

Logistics

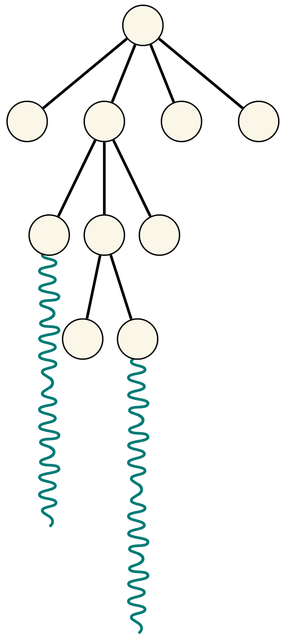
- **Practice questions #4** are available
  - Solutions will be released tomorrow (Mar 10)
- **Quiz 4** is **Friday** (Mar 13)
  - *Coverage:* up to and including **Mar 6** (last Friday's lecture)
  - Bring your student ID!
  - No calculators or other devices
  - The quiz will be run by 3 TAs (I will be in the hospital)
- **TA Office hours:** every Thursday 1pm-2pm in **UCOMM-3-136**

Minimax Search on Large State Spaces

- So far, all of our algorithms have been variations of minimax:
  - Basic minimax
  - Negamax
  - Alpha-beta
- All of these searches must search **every possible move** at every stage
  - **Question:** what is the **exception**?
- Must **complete** the search before results are meaningful (**why?**)
- But most actual games have trees that are **far too large** to search completely
  - Chess, checkers, non-trivial Go, etc.
- **Question:** How can we apply alpha-beta search to these trees?

Monte Carlo Tree Search (MCTS)

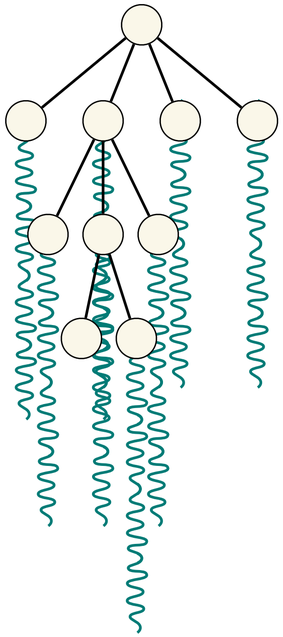
1. **Monte Carlo:** Estimate the quality of a state by **random rollouts**
    - Starting from the state, simulate play by choosing moves randomly
    - Continue until you get to a terminal node
    - Multiple rollouts can be averaged together
2. **Tree search:** Do a certain amount of non-random search before random evaluation
    - Do non-random search in a search tree starting from the current state
    - Leaf values are estimated by random rollouts
    - Tree is **expanded** on every iteration



Basic MCTS operations

- MCTS maintains a **search tree** in memory which grows as the search progresses
- Basic operations:

1. **Traverse:** Starting from the root, choose a "promising" child until a **leaf** is reached
    - Leaf of the search tree, not necessarily terminal state
2. **Expand:** Add all of the selected leaf's **children** to the search tree
3. **Rollout:** Select one of the new children, and **simulate** play until terminal state
    - The states encountered during simulation are *not* added to the search tree
4. **Backpropagate:** Update **performance statistics** on all nodes back to root
    - Each search node tracks **number of visits** and **simulated outcomes** (e.g., number of wins)



Basic MCTS Pseudocode

```
def monte_carlo_tree_search(root, time_budget):
    deadline = now() + time_budget
    while now() < deadline:
        leaf = traverse_and_expand(root)
        result = rollout(leaf)
        backpropagate(leaf, result)
    return best_move(root)

def traverse_and_expand(node):
    while not is_leaf(node):
        node = best_child(node)
    if node.visits > 0 and node.moves > 0:
        expand(node)
        node = random_choice(node.children)
    return node
```

```
def backpropagate(node, result):
    while node is not None:
        node.results.append(result)
        node.visits += 1
        result = not result # swap ptm
        node = node.parent
```

Question: Why are there two different cases for `traverse_and_expand`?

Monte Carlo Tree Search (MCTS)

- MCTS is a **very different** approach from minimax/alpha-beta
- **Best-first** traversal:
  - Tries "promising" directions first
  - Can lead to a very unbalanced search tree (narrow but deep)
- **No guarantees:**
  - Alpha-beta returns a **provably optimal** move
  - MCTS returns the move that it **estimates** to be best
- **Anytime** algorithm:
  - Solution gets better the longer you run it
  - You can stop and get a meaningful answer **at any point**

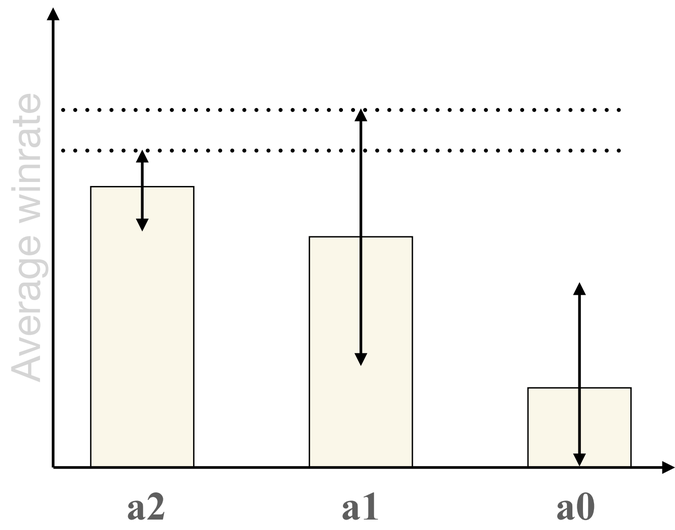
Basic MCTS Drawbacks

1. Simulations are **noisy**
    - What if a good action's simulation gets a poor value by luck?
    - **Q:** Does completely random move selection accurately measure performance?
2. "Best move" selection can get **stuck**
    - Suppose $\mathbf{a1}$ is a winning first move, and $\mathbf{a2}$ is a losing first move
    - But on first simulations, $\mathbf{a1}$ rolls out to a loss, and $\mathbf{a2}$ rolls out to a win (**how?**)
    - **Q:** Next time, which first action will we select?
    - **Q:** How could we ever learn that $\mathbf{a1}$ is actually better?

UCT for Child Selection

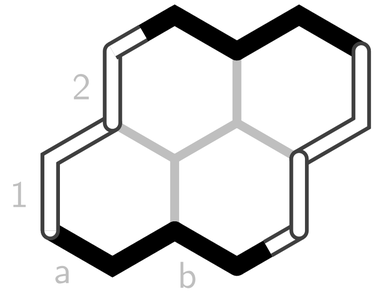
UCT incorporates **Upper Confidence Bounds** into child selection

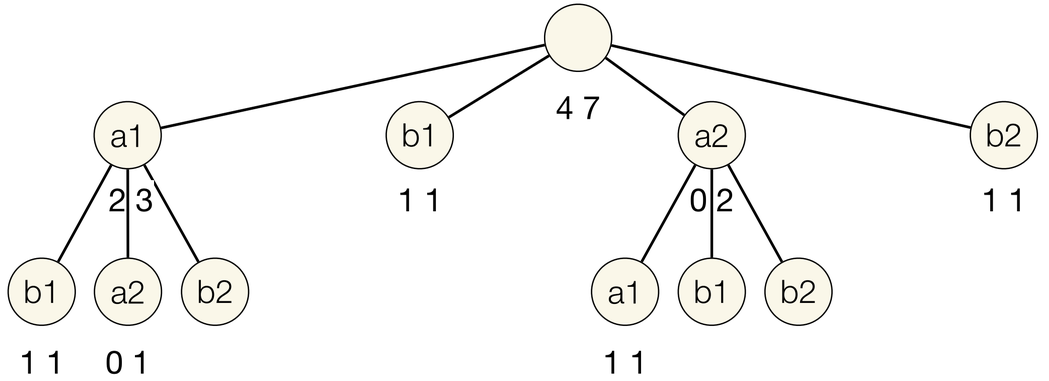
- After some simulations, we have a predicted quality for an action
- The more simulations we have, the more confident we are that this is the true average value (**why?**)
- For every action the average is really "average ± error bar"
  - More simulations makes the error bar smaller
- **Idea:** Instead of choosing the action with the highest average, choose the action with the highest average+error bar
- Actions with fewer simulations have bigger error bars, so they get an "exploration bonus"
- We will tend to explore actions that have higher average and actions that have less simulations



MCTS + UCT Example

- Expand root
- Choose random child to simulate: **b1**
  - back-propagate to root
- Traverse: **a1** has no visits, simulate & backprop
- Traverse: **a2** has no visits, simulate & backprop
- Traverse: **b2** has no visits, simulate & backprop
- Traverse: **a1** has best UCB
  - Expand **a1**
  - Choose random child: **b1**
  - Simulate & backprop
- Traverse: **a2** has best UCB
  - Expand **a2**
  - Choose random child: **a1**
  - Simulate & backprop
- Traverse: **a1** has best UCB
  - a2 has no visits, simulate & backprop





Further Child Selection Refinements

- After search deadline is reached, we choose the **best move from root** and play it
    - When it's time for the next move, we **search again** from scratch
    - New root is the **new current state**
    - **Question:** Why is this a good idea?
- Different ways to choose the best move:
    1. **Highest winrate** / average score (*not* upper confidence bound; **why?**)
    2. **Most visits** (most promising moves will have been visited the most)
- **Question:** Which is the best approach?
- **Idea:**
    - If they **agree**, then the decision is easy!
    - If they disagree, **search for longer** (10% of budget?)
    - Then choose **most visits**

Summary

- **Alpha-beta** search gives guaranteed minimax values, but search must
  - Complete
  - see entire tree (minus provable pruning)
- **Monte Carlo Tree Search** searches for likely good moves rather than proven
  - Pursues **promising actions** progressively deeper
  - **Random rollouts** to evaluate the quality of leaf nodes
  - Selected leaf node **expanded** on every iteration
  - **Exploration** and **exploitation** balanced by **UCT** to traverse search tree