Game Tree Search

1/6/17
Frameworks for Decision-Making

1. Goal-directed planning
   • Agents want to accomplish some goal.
   • The agent will use search to devise a plan.

2. Utility maximization
   • Agents ascribe a utility to various outcomes.
   • The agent attempts to maximize expected utility.
Advantages of Utility Modeling

• Handles uncertainty better
  • Choose actions to maximize *expected* utility.
  • We’ll take advantage of this in a few weeks.

• Simplifies modeling other agents
  • Assume all agents are utility maximizers.
    • And all agents know all other agents are utility maximizers.
  • We just have to figure out their utilities.

Sometimes this is really hard, but this week it’s easy.
Behaving Optimally with Multiple Agents

We need game theory!

If agents act sequentially:
• Extensive form games
  • Our focus this week.

If agents act simultaneously:
• Normal form games
  • We’ll come back to this at the end of the semester.
Extensive form game terminology

decision nodes (states)
Each node belongs to a specific agent (player).

actions (moves)

terminal nodes (outcomes)
Each outcome lists a utility for every player.
Example Game: Nimm

• There are initially N pieces.

• Each turn a player must remove 1, 2, or 3 pieces.

• The player who removes the last piece loses.

Let’s play a game where N=9, you go first.
Exercise: play a few games of Nimm

• Try different values of N.
  • 1, 2, 3, ..., 9, 10, ...

• Who wins under optimal play?
• How does it depend on N?
<table>
<thead>
<tr>
<th>N</th>
<th>Outcome for P1</th>
<th>First move</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>W</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>W</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>L</td>
<td>?</td>
</tr>
<tr>
<td>6</td>
<td>W</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>W</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>W</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>L</td>
<td>?</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Backward Induction

Key idea: start from outcomes and work your way up.

• At leaf nodes, return the outcome.
• At decision nodes, recursively determine the outcome of each action.
• The optimal move is the one that gives the best outcome for the current player.
function backward_induction(state, player):
    if state is terminal:
        return outcome

    initialize best_outcome, best_utility

    for each action available in state:
        ns, np = make_move(state, action)
        outcome = backward_induction(ns, np)
        if utility(outcome, player) > best_utility:
            update best_outcome, best_utility

    return best_outcome
Special Case: Zero-Sum Games

• The sum of utilities is zero for every outcome.

• In a zero-sum game, my gain is always your loss.

• We can represent one-fewer utility per outcome.

• Is Nimm zero-sum?
function min_max(state, player):
    if state is terminal:
        return none, value
    initialize best_action, best_value
    for each action available in state:
        next_state = make_move(state, action)
        act, val = min_max(next_state, other_player)

        if player is maximizer and val > best_value:
            update best_action, best_value
        if player is minimizer and val < best_value:
            update best_action, best_value
    return best_action, best_value
Alternative Min-Max Pseudocode

function max_value(state):
    if state is terminal:
        return value
    initialize best_val
    for each action available in state:
        next_state = make_move(state, action)
        best_val = max(min_value(next_state), best_val)
    return best_val

function min_value(state):
    ...
    best_val = min(max_value(next_state), best_val)
    ...

Problem: game tree size

• For most interesting games the game tree is too large to search to the end and to find optimal moves.

• In chess, the branching factor is approximately 35 and games can last for 100 moves.

• This creates a game tree of $35^{100}$ nodes which is approximately $10,154!$.

• Instead we will search to a limited depth and try to approximate the value of states.

How big is the game tree for tic-tac-toe?
Checkers?
Evaluation Function

• Look at a game state without knowing any context and try to assign it a value.

• Performance of a game playing program is highly dependent on this evaluation.

• Using a good evaluation function allows us to make informed decisions about which move now is likely to lead to good situations later.
Features of a good evaluation function

• When a terminal state is reached, score it correctly.
• Should be efficient to calculate since it will be called many, many times.
• Should reflect the actual chances of winning.
• Exactness is less important than trying to get the relative values correct.