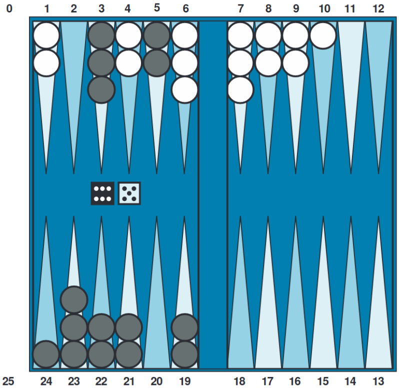
Games

CSC 548, Artificial Intelligence II

Chance/Non-determinism in Games

- Approaches such as minimax are only appropriate for deterministic games.
- Some games have a element of randomness, often imparted via dice or shuffling.
- Considering games of chance
  - more realistic in the sense that life is not deterministic
  - more complicated which allows us to examine additional search techniques

Example: Backgammon



- Basic idea: move your pieces around the board and then off; available moves are determined by rolling two dice.

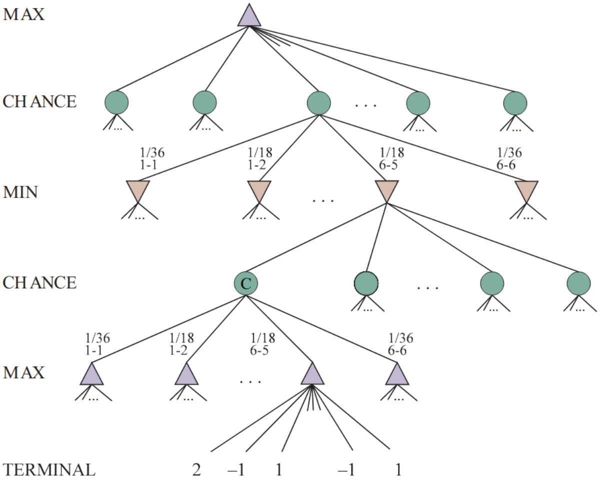
Example: Backgammon

- If we know the dice rolls, then it is straightforward to get the next states
- For example, white rolls a 5 and a 6 the possible moves are:
  - (7-2, 7-1),
  - (17-12, 17-11),
  - ...

Searching with Chance (Backgammon)

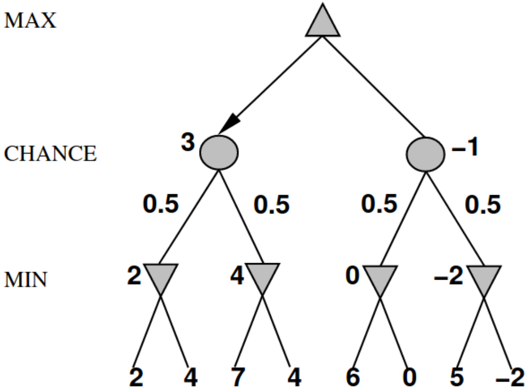
- We know there are 36 different dice rolls (21 unique)
- Idea: insert a “chance” layer between each ply with a branching factor of 21
    - Note: this drastically increases the branching factor (by a factor of 21!)
- Associate a probability with each chance branch
    - each double has a probability of $1/36$ and all others have a probability of $1/18$
- In general, the probabilities are easy to calculate

Example Search Tree

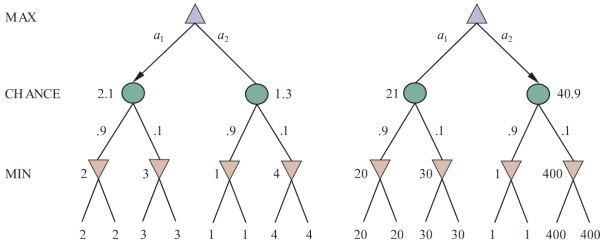


Expected Minimax Value

- Rather that the actual value, we calculate the expected value based on the probabilities
- Evaluation of a chance node: $\sum_{successors(s)} p(s) * v(s)$



Chance and Evaluation Functions



- In the case of expected minimax value the magnitude of value matters, not just the ordering.
- That is, the behavior is only preserved by a positive linear transformation of the evaluation function

Games with Chance

- Given a branching factor b and a chance factor n, the search runtime becomes $\mathcal{O}((nb)^m)$
- For this reason many games of chance do not use much search
  - Example: backgammon frequently only looks ahead 3-ply
- Instead, evaluation functions play a more important roll
  - Example: TD-Gammon learned an evaluation function by playing itself over a million times

Partially Observable Games

- In many games we do not have all the information about the world
  - poker
  - bridge
  - scrabble
  - Kreigspiel
- Challenges
  - The state space can be huge
  - The minimax assumption is probably not true
  - May make move just to explore

Modern Heuristic Search Components

- Search algorithm
- Evaluation function, heuristic
- Simulation
- Combining all three is relatively new

Example: Go

- The minimax algorithm is not effective for the game of Go.
- Reasons:
  - Huge state space
    - average branching factor approximately 250
    - average game length (tree depth) greater than 250
  - No good evaluation function (until recently)

Monte Carlo Simulation

- Do not need an evaluation function
- Process:
  - Simulate the game using random moves
  - Score the game at the end
  - Use that as the evaluation
- Making random moves appears bad, but tends to work for some games
  - Random moves often preserve some difference between a good position and a bad one

Basic (Pure) Monte Carlo Search

1. Play many random games starting with each possible move
2. Keep winning statistics for each move
3. Play the move with the best winning percentage

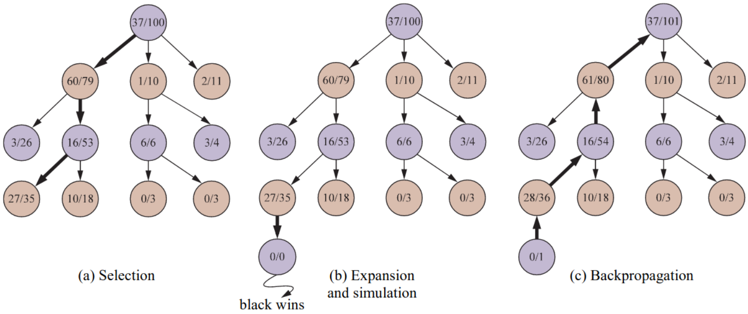
Monte Carlo Tree Search

- Idea: use results of simulations to guide the growth of the game tree
- Exploitation: focus on promising moves
- Exploration: focus on moves where uncertainty about evaluation is high

Monte Carlo Tree Search

- Monte Carlo Tree Search (MCTS) builds a search tree node-by-node with the following steps:
  1. Selection: select a leaf node starting from the root node that has a potential child from which no simulation has yet been initiated
  2. Expansion: if the selected node is not a terminal node, then create one or more child nodes and select one
  3. Simulation (rollout): run a simulated playout from the selected child node until a result is achieved
  4. Backpropagation: Update the current move sequence with the simulation result

Monte Carlo Tree Search Example



Monte Carlo Tree Search Algorithm

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Upper Confidence Bound

- An effective selection policy is called "upper confidence bounds applied to trees" which ranks each possible move based on the formula

$$UCB1(n) = \underbrace{\frac{U(n)}{N(n)}}_{exploitation} + C \underbrace{\sqrt{\frac{ln\ N(parent(n))}{N(n)}}}_{exploration}$$

where $U(n)$ is the utility of node n, $N(n)$ is the number of playouts through node n and C is a constant that balances exploration and exploitation (often set to $\sqrt{2}$)

Monte Carlo Tree Search Comments

- Successful in games and in probabilistic planning
  - Backgammon, Go, General Game Playing, ...
  - Similar methods work in multiplayer games, planning, energy resource allocation, ...
- Scales to parallel machines
- Still poorly understood as to why it works so well