Project 2-1

Backgammon
Group 19

Boris Borisov, Patrick Geyer, Anna Górska, Alexandre Martens, Stijn Overwater, Daniël Tossaint

Randomness AI

- Chooses a move from set of possible moves
- Fully random
- No learning/ analyzing
- Tool for comparison
- O(n)

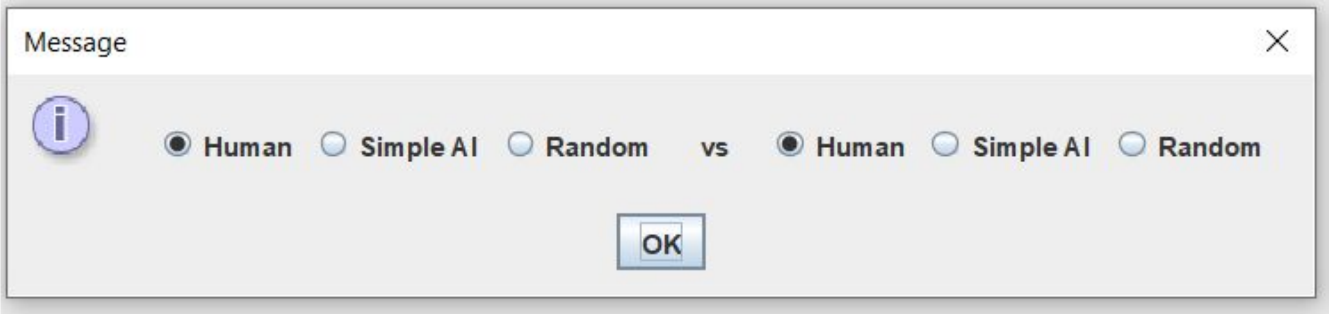
Simple AI

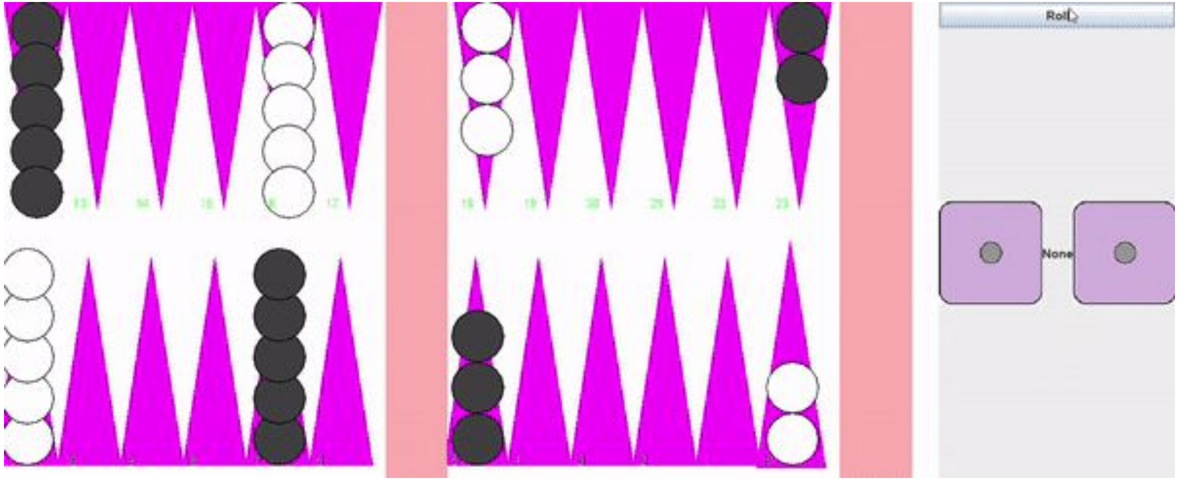
- Evaluates moves using Evaluation Function
- Chooses move with the highest value
- Basic analysis
- O(n)

Evaluation Function

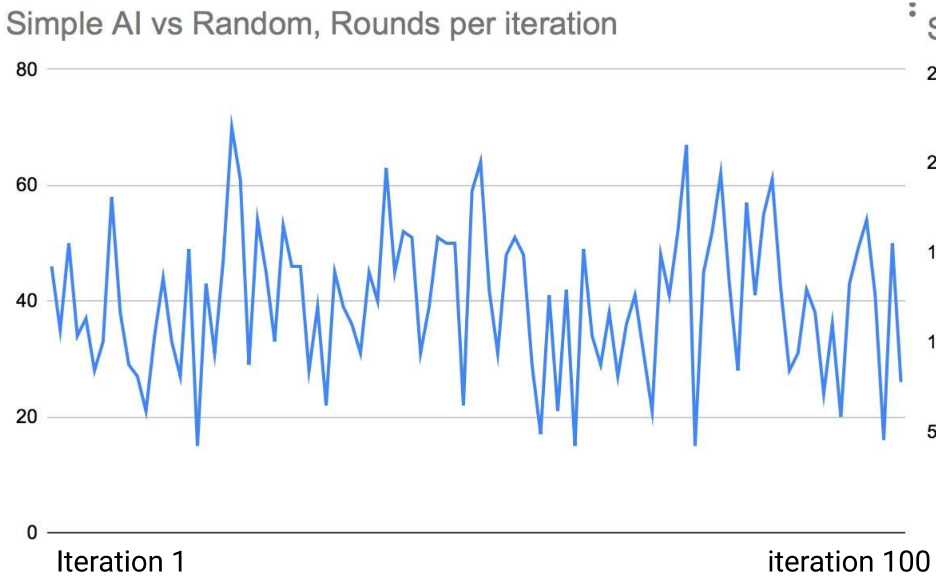
$$\frac{|distance|}{6} - soloStones \times probability + \frac{gates}{3} + hitStone + takenStones$$

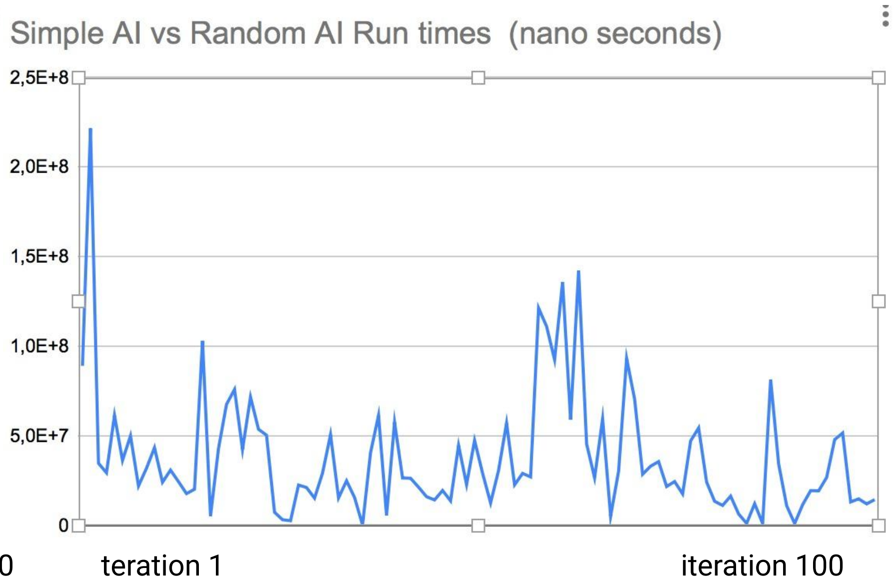
Integration with UI





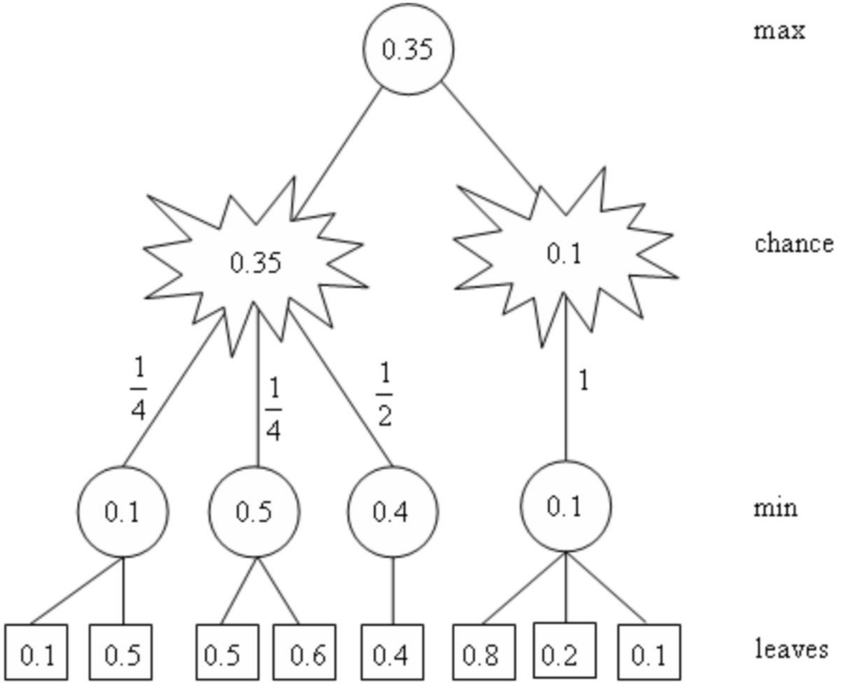
Comparison


Simple AI vs Random, Rounds per iteration
80
60
40
20
0
Iteration 1
iteration 100


Simple AI vs Random AI Run times (nano seconds)
2,5E+8
2,0E+8
1,5E+8
1,0E+8
5,0E+7
0
teration 1
iteration 100

Expectimax AI

- Chooses move depending on opponent's optimal move
- Root in max -> optimal value of children
- Root in min -> worst value of children
- Chance layer -> Add value * probability
- Depth first search
- O(n^2)



Monte Carlo Search Tree

- Tree of nodes
- Each node contains:
  - Boardstate
  - Win ratio
- Evaluation by Playouts
- Weight that relies on the long-term effect of moves

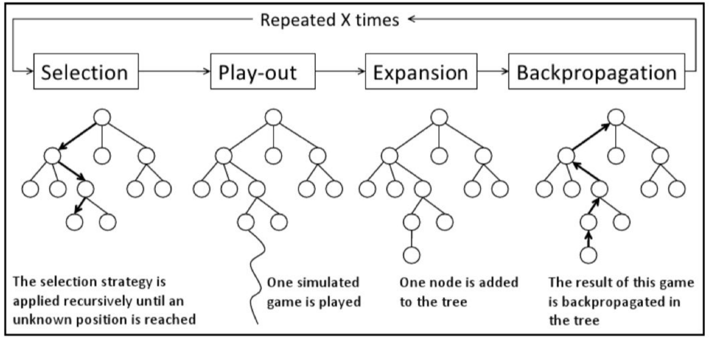


Figure 2: *Outline of the Monte-Carlo Tree Search [19].*

- Four phases
  - Selection
  - Play-out
  - Expansion
  - Backpropagation

Time complexity of monte carlo tree search

It does four steps of n iterations

O(n*(selection+expansion+simulation+backpropagation))

Expansion add a node which we can do in constant time so

O(n*(selection+simulation+backpropagation))

Given the branching factor `b`, and `d` as the depth of our tree, the selection phase runs in O(b*d),so

O(n*(b*d+simulation+backpropagation))

Backpropagation and simulation takes time proportional to the depth of the tree as well, so that becomes:

O(n*(b*d+d+d))

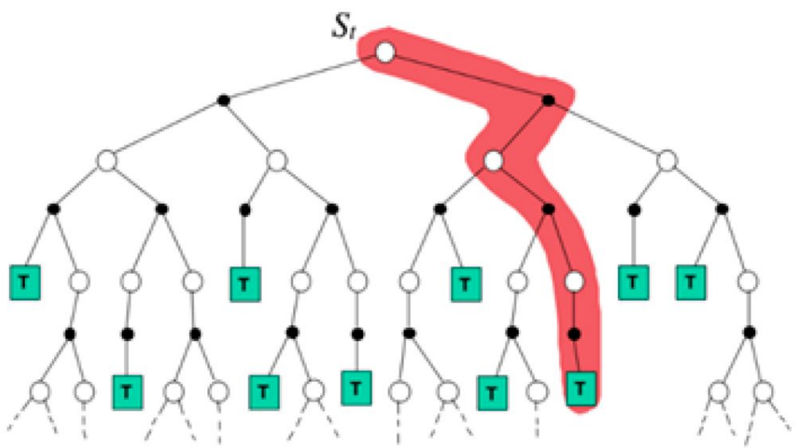
TD-Gammon

- Trains a 3 layer feed-forward network to compute utility of a given board configuration
- Backpropagate expected utility from next step as “expected”/ground truth output from current step
- At end of game send 1 as reward for a win, -1 for a loss

TD-Gammon

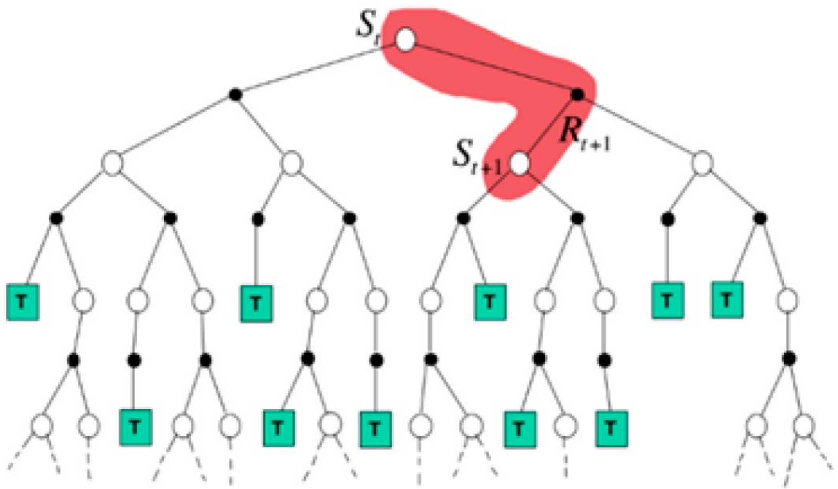
Monte Carlo

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]$$

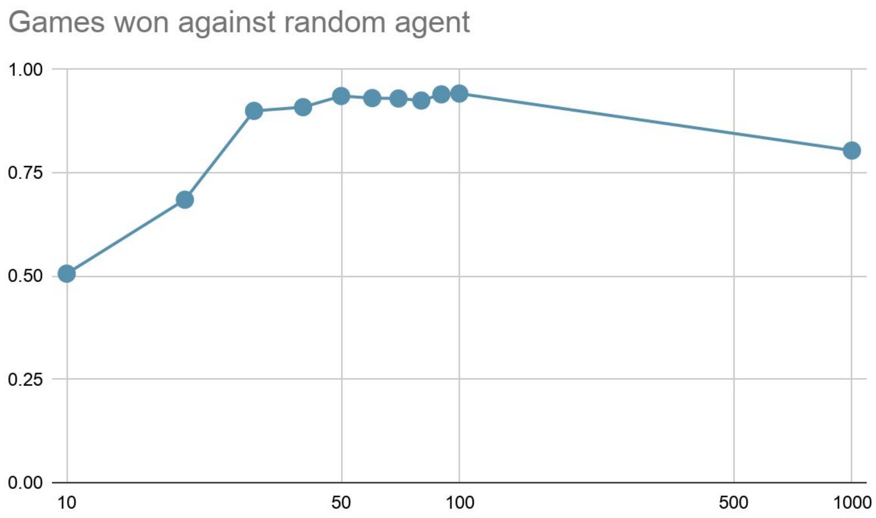


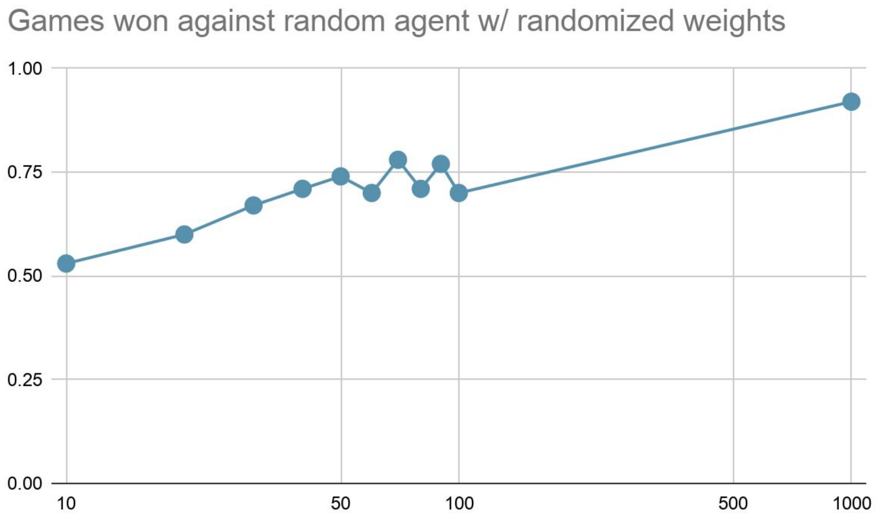
Simplest TD Method

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$



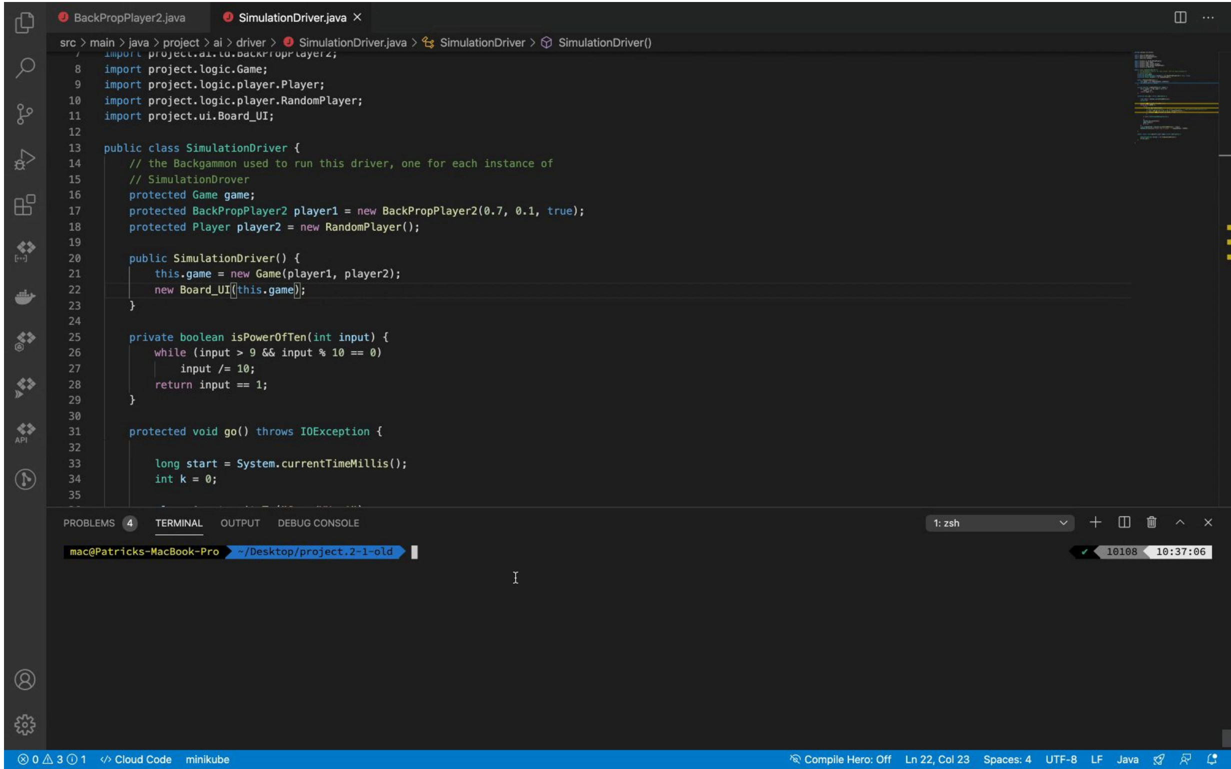
TD-Gammon


Games won against random agent
1.00
0.75
0.50
0.25
0.00
10
50
100
500
1000


Games won against random agent w/ randomized weights
1.00
0.75
0.50
0.25
0.00
10
50
100
500
1000

TD-Gammon Improvements

- Plateau at ~93% win-rate against random agent
- Train for longer
- Alter depth of network



Time per move: 1.17ms
Regardless of “search depth”

Python TD-Gammon

TD-gammon vs human

- TD gammon in python
- Neural network did not have time to train (acting randomly).
- 198 input vector, 50 hidden units, 4 output vector.
- All sigmoid functions (hidden/output)
- Output: white win or black win, gammon or normal win

Conclusion

- Monte Carlo Tree Search outperforms TD-Gammon in skill
- TD-Gammon requires training, but in-game execution is faster than multi-level search algorithms
- TD gammon implementation shows 93% win-rate against random agents.
- ExpectiMax algorithm including chance nodes
- Backgammon implemented both in Java and Python