UIT2504 Artificial Intelligence

Imperfect Decisions in Games

C. Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

September 11, 2024

Game Formulation

- States and the initial state S_0

Game Formulation

- States and the initial state S_0
- TO_MOVE(s): The player whose turn it is to move in state s

Game Formulation

- States and the initial state S_0
- TO_MOVE(s): The player whose turn it is to move in state s
- ACTIONS(s): set of legal moves in a state s

Game Formulation

- States and the initial state S_0
- TO_MOVE(s): The player whose turn it is to move in state s
- ACTIONS(s): set of legal moves in a state s
- RESULT(s,a): the transition model that defines the result of a move a in a state s

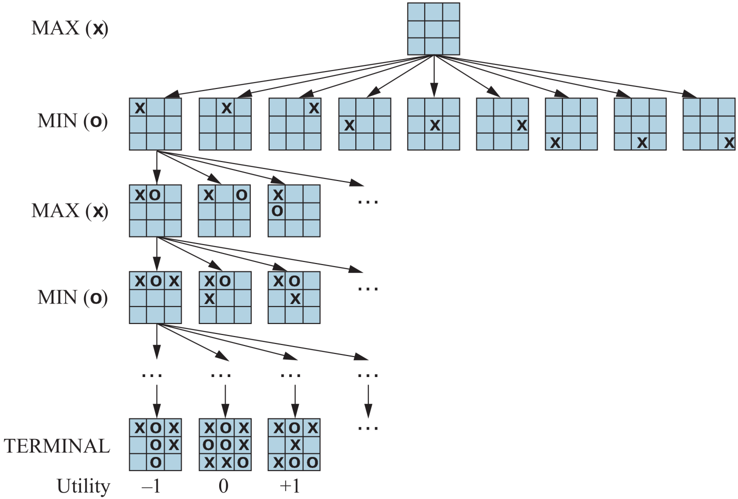
Game Formulation

- States and the initial state S_0
- TO_MOVE(s): The player whose turn it is to move in state s
- ACTIONS(s): set of legal moves in a state s
- RESULT(s,a): the transition model that defines the result of a move a in a state s
- IS_TERMINAL(s): Terminal test — 'true' when the game is over — terminal states

Game Formulation

- States and the initial state S_0
- TO_MOVE(s): The player whose turn it is to move in state s
- ACTIONS(s): set of legal moves in a state s
- RESULT(s,a): the transition model that defines the result of a move a in a state s
- IS_TERMINAL(s): Terminal test — 'true' when the game is over — terminal states
- UTILITY(s,p): Utility function — an objective function that defines the final numeric value to a player p when the game ends in a terminal state s — in chess, outcome is a win, loss, or draw, with values $+1, 0, \frac{1}{2}$

Search Strategies for Playing Games



Minimax Decision

- Optimal strategy for deterministic games
- Utility values percolate up from terminal states
- At MAX level, choose successor with highest utility
- At MIN level, choose successor with lowest utility (note that utility is from MAX's point of view)

Minimax Decision

- Optimal strategy for deterministic games
- Utility values percolate up from terminal states
- At MAX level, choose successor with highest utility
- At MIN level, choose successor with lowest utility (note that utility is from MAX's point of view)

MINIMAX(s) =

$$\begin{cases} Utility(s, MAX) & \text{if IS\_TERMINAL}(s) \\ max_{a \in ACTIONS(s)} MINIMAX(RESULT(s, a)) & \text{if TO\_MOVE}(s) = \text{MAX} \\ min_{a \in ACTIONS(s)} MINIMAX(RESULT(s, a)) & \text{if TO\_MOVE}(s) = \text{MIN} \end{cases}$$

Complexities

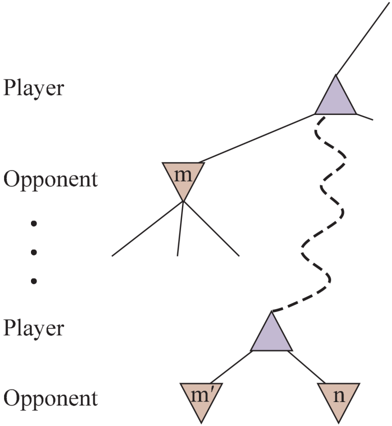
- Complete? — Yes, if the tree is finite — complete depth-first search
- Optimal? — Yes, if the opponent is optimal
- Space Complexity? — $O(bm)$ — may be reduced to $O(m)$ if successors are generated one at a time
- Time Complexity? — $O(b^m)$
- Impractical for non-trivial games such as chess — 35^{100}

Alpha-Beta Pruning

- α : Value of the best choice we have found so far along a path for MAX — $\alpha =$ "at least" — lower bound for MAX
- β : Value of the best choice we have found so far along a path for MIN — $\beta =$ "at most" — upper bound for MAX

Alpha-Beta Pruning

- α : Value of the best choice we have found so far along a path for MAX — α = "at least" — lower bound for MAX
- β : Value of the best choice we have found so far along a path for MIN — β = "at most" — upper bound for MAX



Move Ordering

- If best move ordering can be achieved, time taken may be halved

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$ — which is same as $O\left(\left(\sqrt{b}\right)^{m}\right)$ —

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$ — which is same as $O\left(\left(\sqrt{b}\right)^m\right)$ — effective branching factor is reduced to $\sqrt{b}$

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$ — which is same as $O\left(\left(\sqrt{b}\right)^{m}\right)$ — effective branching factor is reduced to $\sqrt{b}$
- If the successors are examined in a random order, time complexity will be roughly $O(b^{3m/4})$

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$ — which is same as $O\left(\left(\sqrt{b}\right)^m\right)$ — effective branching factor is reduced to $\sqrt{b}$
- If the successors are examined in a random order, time complexity will be roughly $O(b^{3m/4})$
- Heuristics may be used to order the moves

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$ — which is same as $O\left(\left(\sqrt{b}\right)^m\right)$ — effective branching factor is reduced to $\sqrt{b}$
- If the successors are examined in a random order, time complexity will be roughly $O(b^{3m/4})$
- Heuristics may be used to order the moves — in chess, captures first, then threats, then forward moves, and then backward moves

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$ — which is same as $O\left(\left(\sqrt{b}\right)^m\right)$ — effective branching factor is reduced to $\sqrt{b}$
- If the successors are examined in a random order, time complexity will be roughly $O(b^{3m/4})$
- Heuristics may be used to order the moves — in chess, captures first, then threats, then forward moves, and then backward moves
- Dynamic move ordering — moves found to be best in the past (learning) — or, iterative-deepening

Move Ordering

- If best move ordering can be achieved, time taken may be halved — $O(b^{m/2})$ — which is same as $O\left(\left(\sqrt{b}\right)^m\right)$ — effective branching factor is reduced to $\sqrt{b}$
- If the successors are examined in a random order, time complexity will be roughly $O(b^{3m/4})$
- Heuristics may be used to order the moves — in chess, captures first, then threats, then forward moves, and then backward moves
- Dynamic move ordering — moves found to be best in the past (learning) — or, iterative-deepening
- Transposition table — hash the α–β values for future use

Type A and Type B Strategies

- Even with alpha-beta pruning and other techniques such as move ordering, minimax algorithm may not work for games such as Chess and Go

Type A and Type B Strategies

- Even with alpha-beta pruning and other techniques such as move ordering, minimax algorithm may not work for games such as Chess and Go
- Claude Shannon proposed two additional strategies:
- Type A strategy: consider all possible moves to certain depth and then use a heuristic evaluation function to estimate the utilities of the states at that depth (wide but shallow strategy)

Type A and Type B Strategies

- Even with alpha-beta pruning and other techniques such as move ordering, minimax algorithm may not work for games such as Chess and Go
- Claude Shannon proposed two additional strategies:
- Type A strategy: consider all possible moves to certain depth and then use a heuristic evaluation function to estimate the utilities of the states at that depth (wide but shallow strategy)
- Type B Strategy: ignore moves that look bad, and follow promising paths as far as possible (deep but narrow strategy)

Questions?

Cutoff Test

- It is still impractical to search down till terminal nodes and propagate utility values up

Cutoff Test

- It is still impractical to search down till terminal nodes and propagate utility values up
- It is prudent to cutoff the search at some pre-determined depth (ply) [Either Type A or Type B strategy]

Cutoff Test

- It is still impractical to search down till terminal nodes and propagate utility values up
- It is prudent to cutoff the search at some pre-determined depth (ply) [Either Type A or Type B strategy]
- A simple cutoff test replaces the terminal test — cutoff may be based on just the depth or other simple logic

Cutoff Test

- It is still impractical to search down till terminal nodes and propagate utility values up
- It is prudent to cutoff the search at some pre-determined depth (ply) [Either Type A or Type B strategy]
- A simple cutoff test replaces the terminal test — cutoff may be based on just the depth or other simple logic
- But, what is the utility of a non-terminal cutoff node?

Cutoff Test

- It is still impractical to search down till terminal nodes and propagate utility values up
- It is prudent to cutoff the search at some pre-determined depth (ply) [Either Type A or Type B strategy]
- A simple cutoff test replaces the terminal test — cutoff may be based on just the depth or other simple logic
- But, what is the utility of a non-terminal cutoff node? — use a heuristic evaluation function

Cutoff Test

- It is still impractical to search down till terminal nodes and propagate utility values up
- It is prudent to cutoff the search at some pre-determined depth (ply) [Either Type A or Type B strategy]
- A simple cutoff test replaces the terminal test — cutoff may be based on just the depth or other simple logic
- But, what is the utility of a non-terminal cutoff node? — use a heuristic evaluation function

$$\text{H-MINIMAX}(s, d) =$$

$$\begin{cases} Eval(s, MAX) & \text{if IS-CUTOFF}(s, d) \\ max_{a \in ACTIONS(s)} \text{H-MM}(RESULT(s, a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ min_{a \in ACTIONS(s)} \text{H-MM}(RESULT(s, a), d+1) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

Heuristic Evaluation Function

- An evaluation function returns an estimate of the expected utility of a state s to player p — quality of this estimation is very important

Heuristic Evaluation Function

- An evaluation function returns an estimate of the expected utility of a state s to player p — quality of this estimation is very important
- Should identify terminal states — order them the same way as the true utility function
- Value should be somewhere between a "loss" and a "win"
- Should be easy to compute!
- Strongly correlated with the actual chances of winning

Heuristic Evaluation Function

- Evaluation functions may be designed based on certain features of a state

Heuristic Evaluation Function

- Evaluation functions may be designed based on certain features of a state
- With features, we may abstract a set of states as a category (or equivalence classes) and estimate the expected value from experience

Heuristic Evaluation Function

- Evaluation functions may be designed based on certain features of a state
- With features, we may abstract a set of states as a category (or equivalence classes) and estimate the expected value from experience
- For example, 82% of states encountered in the two-pawns vs one-pawn category lead to a win; 2% to a loss; and 16% to a draw — expected value may be $(0.82 \times 1) + (0.02 \times 0) + (0.16 \times \frac{1}{2}) = 0.90$

Heuristic Evaluation Function

- Evaluation functions may be designed based on certain features of a state
- With features, we may abstract a set of states as a category (or equivalence classes) and estimate the expected value from experience
- For example, 82% of states encountered in the two-pawns vs one-pawn category lead to a win; 2% to a loss; and 16% to a draw — expected value may be $(0.82 \times 1) + (0.02 \times 0) + (0.16 \times \frac{1}{2}) = 0.90$
- Another simple idea: Consider a weighted combination of the features (each feature is assumed to contribute a numeric value)

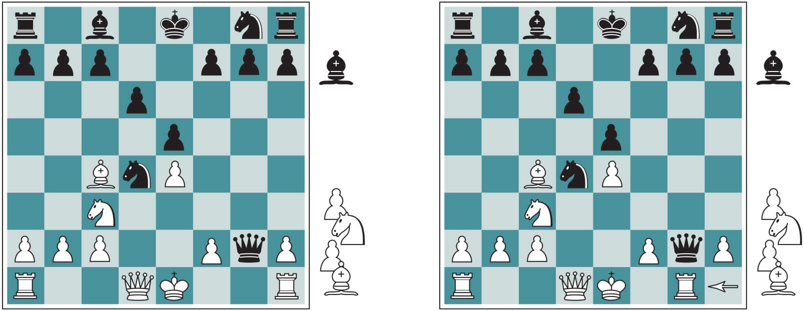
Heuristic Evaluation Function

- Evaluation functions may be designed based on certain features of a state
- With features, we may abstract a set of states as a category (or equivalence classes) and estimate the expected value from experience
- For example, 82% of states encountered in the two-pawns vs one-pawn category lead to a win; 2% to a loss; and 16% to a draw — expected value may be $(0.82 \times 1) + (0.02 \times 0) + (0.16 \times \frac{1}{2}) = 0.90$
- Another simple idea: Consider a weighted combination of the features (each feature is assumed to contribute a numeric value)
- Neural networks basically compute weighted sum of the features and apply an non-linear activation function

Heuristic Evaluation Function

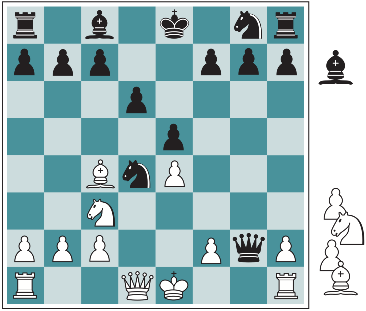
- Evaluation functions may be designed based on certain features of a state
- With features, we may abstract a set of states as a category (or equivalence classes) and estimate the expected value from experience
- For example, 82% of states encountered in the two-pawns vs one-pawn category lead to a win; 2% to a loss; and 16% to a draw — expected value may be $(0.82 \times 1) + (0.02 \times 0) + (0.16 \times \frac{1}{2}) = 0.90$
- Another simple idea: Consider a weighted combination of the features (each feature is assumed to contribute a numeric value)
- Neural networks basically compute weighted sum of the features and apply an non-linear activation function
- Neural networks and deep learning based chess engines are common today!

Issues in cutoff

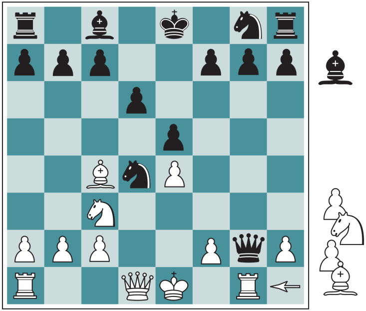


(a) White to move (b) White to move

Issues in cutoff



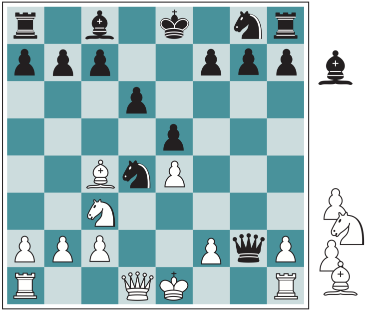
(a) White to move



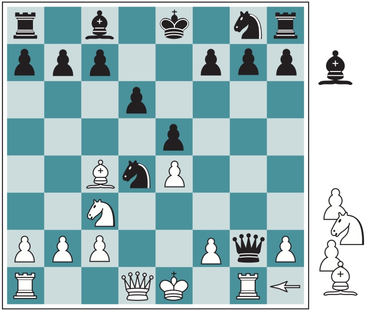
(b) White to move

- Evaluation should be applied only to positions that are quiescent

Issues in cutoff



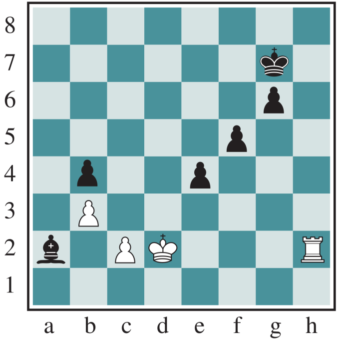
(a) White to move



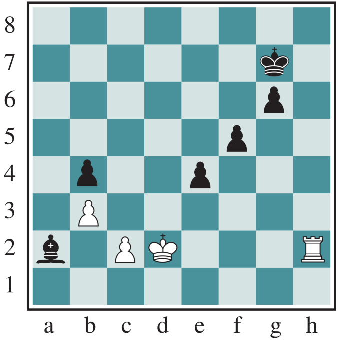
(b) White to move

- Evaluation should be applied only to positions that are quiescent
- Quiescence search: Perform extra search to confirm that there are no wild swings in the evaluation

Issues in cutoff



Issues in cutoff



- Horizon effect
- Keep a collection of singular extensions — allow moves that are "clearly better" than all other moves in a given position, even after cut-off

Forward Pruning

- Alpha-Beta pruning is perfect — does not affect the completeness or optimality of the algorithm

Forward Pruning

- Alpha-Beta pruning is perfect — does not affect the completeness or optimality of the algorithm
- It is also possible to think of imperfect pruning

Forward Pruning

- Alpha-Beta pruning is perfect — does not affect the completeness or optimality of the algorithm
- It is also possible to think of imperfect pruning
- Forward Pruning considers only a few valid moves in a state (Type B Strategy!)

Forward Pruning

- Alpha-Beta pruning is perfect — does not affect the completeness or optimality of the algorithm
- It is also possible to think of imperfect pruning
- Forward Pruning considers only a few valid moves in a state (Type B Strategy!)
- Beam search is an approach to forward pruning where only a “beam” of the n best moves (according to some heuristics) are considered

Forward Pruning

- Alpha-Beta pruning is perfect — does not affect the completeness or optimality of the algorithm
- It is also possible to think of imperfect pruning
- Forward Pruning considers only a few valid moves in a state (Type B Strategy!)
- Beam search is an approach to forward pruning where only a "beam" of the *n* best moves (according to some heuristics) are considered
- ProbCut by Buro (1995) prunes nodes that are probably outside the current (α, β) window (based on statistics gathered from experience)

Forward Pruning

- Alpha-Beta pruning is perfect — does not affect the completeness or optimality of the algorithm
- It is also possible to think of imperfect pruning
- Forward Pruning considers only a few valid moves in a state (Type B Strategy!)
- Beam search is an approach to forward pruning where only a "beam" of the *n* best moves (according to some heuristics) are considered
- ProbCut by Buro (1995) prunes nodes that are probably outside the current (α, β) window (based on statistics gathered from experience)
- Late move reduction — if the moves are ordered, probably moves that appear late in the sequence are not good, and so depth may be reduced for them

Search Vs Lookup

- A judicious combination of all these techniques has proved to be useful in practice — good heuristic evaluation, cutoff with quiescence search, a large transposition table, learning from a large collection of game database

Search Vs Lookup

- A judicious combination of all these techniques has proved to be useful in practice — good heuristic evaluation, cutoff with quiescence search, a large transposition table, learning from a large collection of game database — a 5-ply minimax search can be extended to about 14 ply search on a given hardware and time

Search Vs Lookup

- A judicious combination of all these techniques has proved to be useful in practice — good heuristic evaluation, cutoff with quiescence search, a large transposition table, learning from a large collection of game database — a 5-ply minimax search can be extended to about 14 ply search on a given hardware and time
- For games like chess, table lookup for opening and end games have found to be very effective! — search techniques are required only for the middle game

Search Vs Lookup

- A judicious combination of all these techniques has proved to be useful in practice — good heuristic evaluation, cutoff with quiescence search, a large transposition table, learning from a large collection of game database — a 5-ply minimax search can be extended to about 14 ply search on a given hardware and time
- For games like chess, table lookup for opening and end games have found to be very effective! — search techniques are required only for the middle game
- Reading Exercise: Study open source chess playing engines such as crafty and stockfish (`https://stockfishchess.org/`)

Search Vs Lookup

- A judicious combination of all these techniques has proved to be useful in practice — good heuristic evaluation, cutoff with quiescence search, a large transposition table, learning from a large collection of game database — a 5-ply minimax search can be extended to about 14 ply search on a given hardware and time
- For games like chess, table lookup for opening and end games have found to be very effective! — search techniques are required only for the middle game
- Reading Exercise: Study open source chess playing engines such as crafty and stockfish (`https://stockfishchess.org/`)
- Reading Exercise: You may also optionally read about the recent developments based on neural networks and deep learning, such as chess engine Leela Chess Zero (`https://lczero.org/`)

Questions?

- There are games where the branching factor is quite large and it is not easy to come up with an evaluation function — for example, Go

Monte Carlo Tree Search

- There are games where the branching factor is quite large and it is not easy to come up with an evaluation function — for example, Go
- Go has a branching factor that starts with 361 and most of the states are in flux until the endgame

Monte Carlo Tree Search

- There are games where the branching factor is quite large and it is not easy to come up with an evaluation function — for example, Go
- Go has a branching factor that starts with 361 and most of the states are in flux until the endgame
- So a different strategy called Monte Carlo Tree Seach (MCTS) has been evolved

Monte Carlo Tree Search

- MCTS evaluates a state s by conducting simulations of complete games (one line of play) starting from s

Monte Carlo Tree Search

- MCTS evaluates a state s by conducting simulations of complete games (one line of play) starting from s
- Utility of s is estimated as the average utility over a number of simulations

Monte Carlo Tree Search

- MCTS evaluates a state s by conducting simulations of complete games (one line of play) starting from s
- Utility of s is estimated as the average utility over a number of simulations
- A simulation (also called as a playout or rollout) chooses moves for each player until a terminal position is reached (utility is known)

Monte Carlo Tree Search

- MCTS evaluates a state *s* by conducting simulations of complete games (one line of play) starting from *s*
- Utility of *s* is estimated as the average utility over a number of simulations
- A simulation (also called as a playout or rollout) chooses moves for each player until a terminal position is reached (utility is known)
- Utility of *s* will be average utility or win percentage (whicher makes sense) for all the simulations conducted so far

Monte Carlo Tree Search

- MCTS evaluates a state s by conducting simulations of complete games (one line of play) starting from s
- Utility of s is estimated as the average utility over a number of simulations
- A simulation (also called as a playout or rollout) chooses moves for each player until a terminal position is reached (utility is known)
- Utility of s will be average utility or win percentage (whicher makes sense) for all the simulations conducted so far
- For example, utility of s may look like $\frac{22}{39}$, which means MAX won the game 22 times among the 39 playouts

Monte Carlo Tree Search

- MCTS evaluates a state s by conducting simulations of complete games (one line of play) starting from s
- Utility of s is estimated as the average utility over a number of simulations
- A simulation (also called as a playout or rollout) chooses moves for each player until a terminal position is reached (utility is known)
- Utility of s will be average utility or win percentage (whicher makes sense) for all the simulations conducted so far
- For example, utility of s may look like $\frac{22}{39}$, which means MAX won the game 22 times among the 39 playouts
- If one more simulation is conducted from s, and MAX looses in that playout, then the utility of s is revised as $\frac{22}{40}$

MCTS Playout

- A playout policy is used to select a good move for each player during a playout

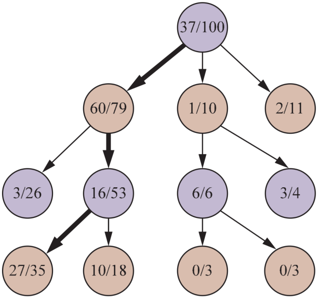
MCTS Playout

- A playout policy is used to select a good move for each player during a playout
- When the game is in progress and a search tree is kept in memory, a selection policy is used to select a child at each step and decide where to start the playout from

MCTS Playout

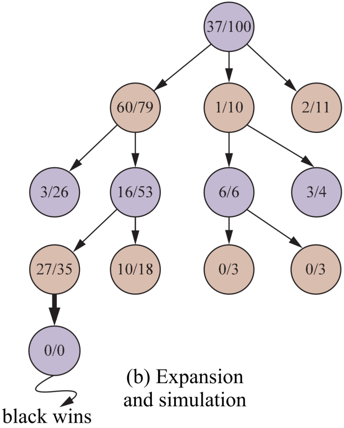
- A playout policy is used to select a good move for each player during a playout
- When the game is in progress and a search tree is kept in memory, a selection policy is used to select a child at each step and decide where to start the playout from
- Selection policy may have two strategies: Exploration of states that have had few playouts, and exploitation of states that have done well in the past playouts

MCTS: Selection



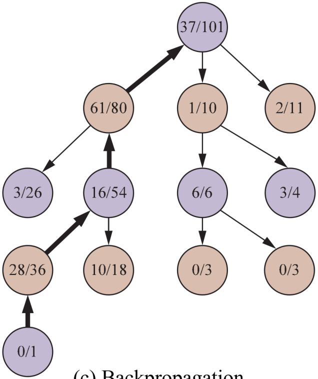
(a) Selection

MCTS: Expansion and Simulation



(b) Expansion and simulation

MCTS: Back-propagation



(c) Backpropagation

MCTS 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 selection strategy

- We have seen that the selection policy needs to balance between exploitation and exploration

Upper confidence bound selection strategy

- We have seen that the selection policy needs to balance between exploitation and exploration
- One way of achieving it will be through an Upper confidence bound formula called UCB1. For a node n, the formula is given by

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(Parent(n))}{N(n)}}$$

Upper confidence bound selection strategy

- We have seen that the selection policy needs to balance between exploitation and exploration
- One way of achieving it will be through an Upper confidence bound formula called UCB1. For a node n, the formula is given by

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(Parent(n))}{N(n)}}$$

- The value of constant C is usually taken as $\sqrt{2}$

Upper confidence bound selection strategy

- We have seen that the selection policy needs to balance between exploitation and exploration
- One way of achieving it will be through an Upper confidence bound formula called UCB1. For a node n, the formula is given by

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(Parent(n))}{N(n)}}$$

- The value of constant C is usually taken as $\sqrt{2}$
- It is possible to use early playout termination, where a non-terminal is evaluated by a heuristic function

MCTS

- It is possible to use MCTS for new games, in which there is no body of experience or database of games
- Knowledge of the game can be encoded into selection and playout policies

MCTS

- It is possible to use MCTS for new games, in which there is no body of experience or database of games
- Knowledge of the game can be encoded into selection and playout policies
- MCTS predominantly uses Type B strategy and so may completely miss a good line of play
- Even “obvious” states may need several playouts to estimate the utility

MCTS

- It is possible to use MCTS for new games, in which there is no body of experience or database of games
- Knowledge of the game can be encoded into selection and playout policies
- MCTS predominantly uses Type B strategy and so may completely miss a good line of play
- Even "obvious" states may need several playouts to estimate the utility
- MCTS is a kind of reinforcement learning

Questions?