Adversarial Search

Tian-Li Yu

Taiwan Evolutionary Intelligence Laboratory (TEIL)
Department of Electrical Engineering
National Taiwan University
tianliyu@ntu.edu.tw

Readings: AIMA Chapter 5 with 5.6 sketched and 5.7 skipped

Outline

Types of Games

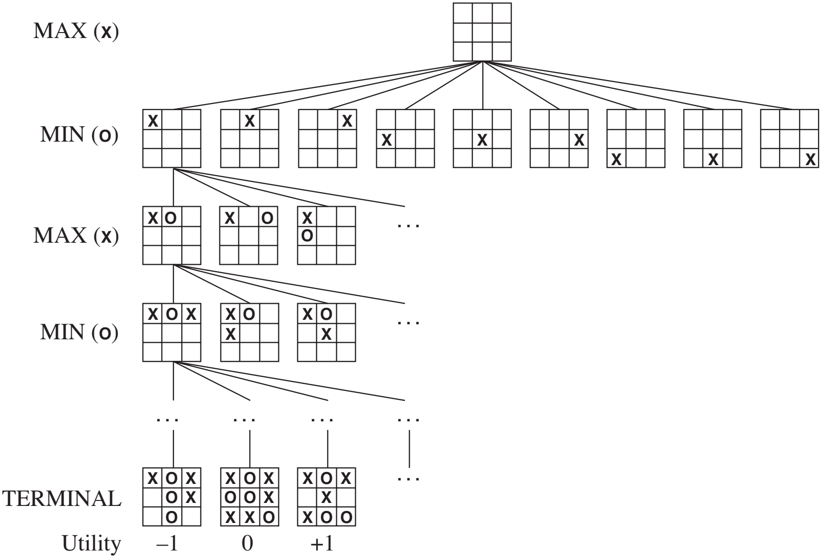
- Adversarial search considers multi-agent and competitive environments.
- Game theory consider both competitive and cooperative environments.
- Most common games are deterministic, turn-taking, two-player, zero-sum games with perfect information.
    - Let's focus on this type of games for a while until told otherwise.

	Deterministic	**Stochastic**
Perfect Information	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Imperfect Information	Battleships, Bingo	Bridge, Poker

Symbols

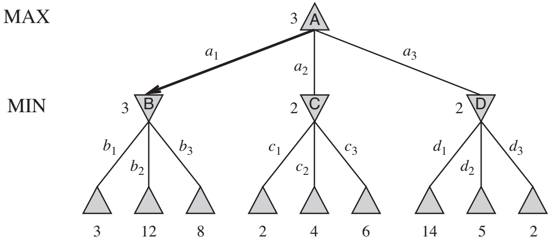
- s_0: Initial state.
- $\textsc{Player}(s)$: The player in state s.
- $\textsc{Action}(s)$: Returns the set of legal moves in state s.
- $\textsc{Result}(s, a)$: The transition model, which returns the resulting state of a move a in state s.
- $\textsc{Terminal-Test}(s)$: $\textsc{true}/\textsc{false}$. States where the game has terminated are called terminal states.
- $\textsc{Utility}(s, p)$: A utility function (also called objective or payoff).
  - $\textsc{Utility}(s)$ for 2-player, zero-sum games.
  - Reason: $\textsc{Utility}(s, p_1) = \text{-}\textsc{Utility}(s, p_2)$

Game Tree (of Tic-Tac-Toe)



Optimal Decision

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$



Minimax Search

Negamax Search

- $\min\{a_0, \cdots, a_n\} = -\max\{-a_0, \cdots, -a_n\}$
- Such simplified implementation of $\textsc{Minimax}$ is called $\textsc{NegaMax}$.
- Copying the whole state (line 5) is memory consuming. Practical implementation usually adopts $s = \textsc{BackTrack}(s', a)$.

$\textsc{NegaMax}(s)$

1 **if** $\textsc{Terminal-Test}(s)$
2   **return** $\textsc{Utility}(s, p)$
3 $result = -\infty$
4 **for** each $a \in \textsc{Action}(s)$
5     $s' = \textsc{Result}(s, a)$
6     $result = \max(result, -\textsc{NegaMax}(s'))$
7 **return** $result$

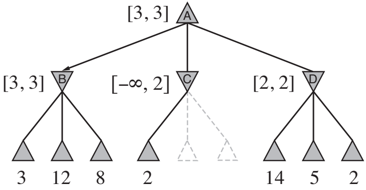
Properties of Minimax (Negamax) Search

- Completeness: Yes, if tree is finite.
- Optimality: Yes, against an optimal opponent. Otherwise?
  - Risky moves that leads to complicated variations might be better to revert unfavored situations.
- Time Complexity: $O(b^m)$.
- Space Complexity: $O(bm)$ (DFS); or $O(m)$ if the algorithm generates actions one at a time.

For chess, $b \simeq 35$, $m \simeq 100 \Rightarrow$ optimal decision is practically intractable.
Do we need to explore every path?

$\alpha - \beta$ Pruning

- Not every node needs to be evaluated.

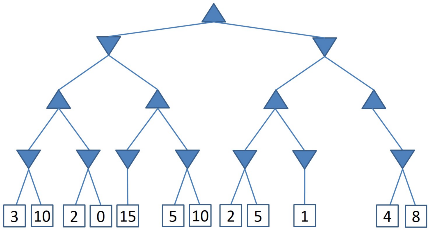


The value of *root* is independent of two unknown nodes x and y

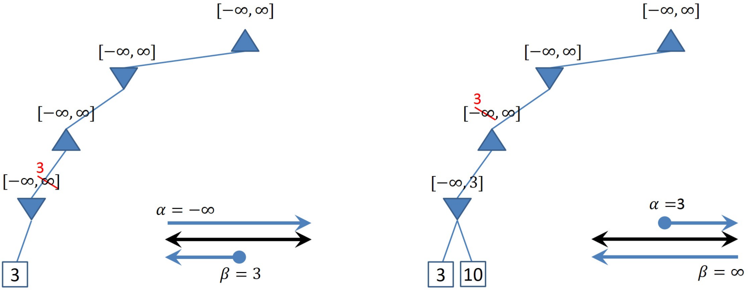
$$\begin{aligned}\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\ &= \max(3, \min(2,x,y), 2)) \\ &= 3\end{aligned}$$

$\alpha - \beta$ Pruning

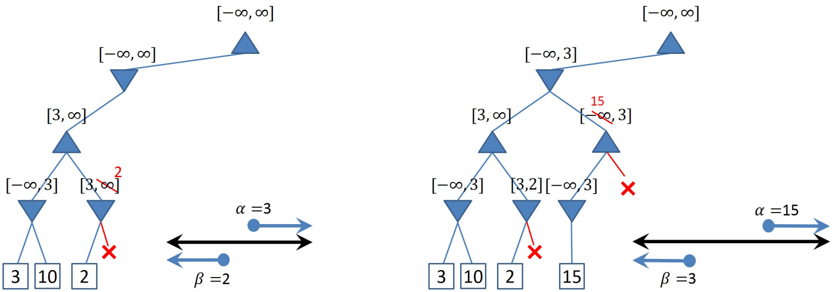
- Keeping α (maximum lower bound) for the maximum utility for player MAX, initialized to $-\infty$.
- Keeping β (minimum upper bound) for the minimum utility for player MIN, initialized to ∞.
- Only the moves within the $[\alpha, \beta]$ window are expanded; otherwise its branches are pruned.
- The pruning does NOT compromise solution quality.



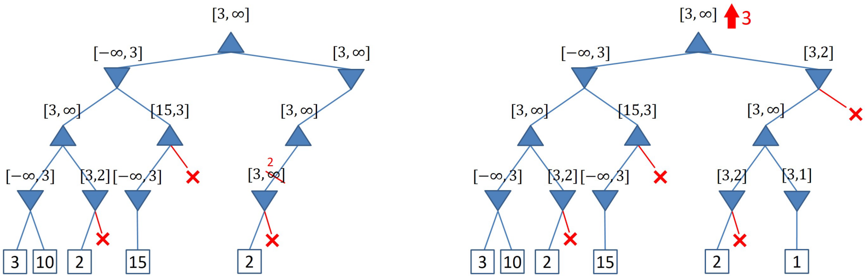
$\alpha - \beta$ Pruning Example



$\alpha - \beta$ Pruning Example (contd.)



$\alpha - \beta$ Pruning Example (contd.)



Implementation of $\alpha - \beta$ Pruning

AlphaBeta(s, α, β)

```
if Terminal-Test(s)
    return Utility(s)
if Player(s) == MAX
    v = −∞
    for each a ∈ Action(s)
        v = max(v, AlphaBeta(Result(s, a), α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
else
    v = ∞
    for each a ∈ Action(s)
        v = min(v, AlphaBeta(Result(s, a), α, β))
        if α ≥ v return v
        β = min(β, v)
    return v
```

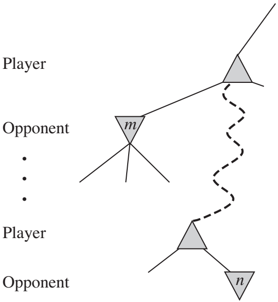
Implementation of $\alpha - \beta$ Pruning

- $\text{NegaMax} + \text{AlphaBeta} = \text{AB-NegaMax}$.

$\text{AB-NegaMax}(s, \alpha, \beta)$

if $\text{Terminal-Test}(s)$
    return $\text{Utility}(s, p)$
$v = -\infty$
for each $a \in \text{Action}(s)$
    $s' = \text{Result}(s, a)$
    $v = \max(v, -\text{AB-NegaMax}(s', -\beta, -\alpha))$
    if $v \geq \beta$ **return** v
    $\alpha = \max(\alpha, v)$
return v

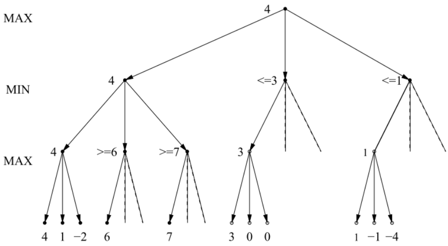
Main Idea of $\alpha - \beta$ Pruning



- If m is better than n for Player, n will never be reached in actual play.
- Once we have found enough about n to reach this conclusion, we can prune it.

Efficiency of $\alpha - \beta$ Pruning

- Highly depends on the order of moves.
- Worst case: no pruning $\rightarrow O(b^m)$.
- Best case: Always check the best move first.
  - Need to check every child for the first move.
  - Only need to check first child for other moves.
  - $O(b \cdot 1 \cdot b \cdot 1 \cdots) = O(b^{m/2})$



- Average case: $O(b^{3m/4})$
- Very simple ordering usually achieves $O(b^{m/2})$
  - Another good reason to adopt iterative deepening.
  - Reduce the effective branch factor to $\sqrt{b}$.
  - Make the search twice as deep as before.

Aspiration Windows

- Assume UTILITY is quite consistent.
- Then the old score means something.
- Original ALPHABETA searches at $[-\infty, \infty]$.
- We can try something like $[\text{OLD\_SCORE} - 30, \text{OLD\_SCORE} + 30]$.
- If a move lies outside the window, we need to re-search with a wider window.
- Some fine tuning is needed to ensure speed-up.

NegaScout

- Assume UTILITY is integral.
- Aspiration search can be taken to extreme.
- If our ordering is quite good, it's highly possible that the first move is the best move.
- We then can make the search window from $[\alpha, \beta]$ to $[\alpha, \alpha + 1]$ (a null window).
- If our assumption is correct, the search speeds up.
- If our assumption is incorrect, hopefully the cutoff occurs very soon (α can actually be improved).
- We then search it properly (using the original $[\alpha, \beta]$ window).

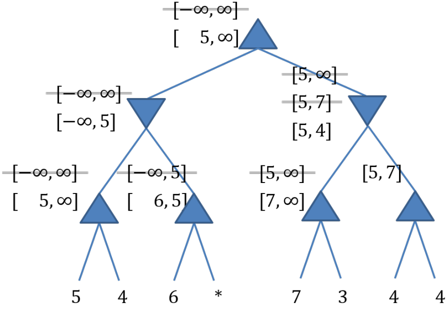
NegaScout

NegaScout(s, α, β)

if Terminal-Test(s)
    return Utility(s, p)
$b = \beta$ //initial window is $[-\beta, -\alpha]$
$v = -\infty$
for each $a \in$ Action(s)
    $s' =$ Result(s, a)
    $result = -$NegaScout$(s', -b, -\alpha)$
    if $\alpha < result < \beta$ and a is not first action
        $result = -$NegaScout$(s', -\beta, -\alpha)$ //full re-search
    $v = \max(v, result)$
    if $v \geq \beta$ **return** v
    $\alpha = \max(\alpha, v)$
    $b = \alpha + 1$ //set new null window
return v

Performance of NegaScout

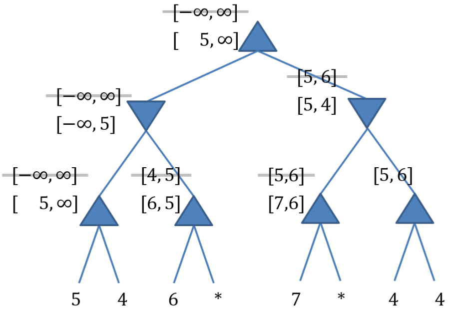
- Performance of NegaScout highly depends on the search order.
- For random order, NegaScout is usually worse than original alpha-beta.
- NegaScout outperforms original alpha-beta for good move ordering.



Example of α-β pruning.

Performance of NegaScout

- NegaScout tentatively tries $[5,6]$ which prunes the node 3 (wasn't pruned in α-β).
- The returned value 4, which is outside the true window $[5,\infty]$, validates the pruning.



Example of NegaScout.

MTD-f

- If we memorize the results from the previous search, not much harm to do more searches.
- Iteratively guessing the correct value.

```
function MTDF(root, f, d)
      g := f
      upperBound := +∞
      lowerBound := −∞
      while lowerBound < upperBound
         β := max(g, lowerBound+1)
         g := AlphaBetaWithMemory(root, β−1, β, d)
         if g < β then
              upperBound := g
         else
              lowerBound := g
     return g
```

Imperfect Real-Time Decisions

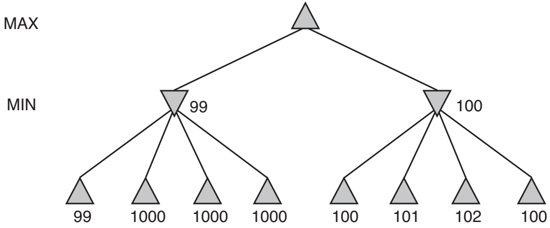
Use CUTOFF-TEST instead of TERMINAL-TEST

- Can be as simple as depth limit.
- Can adopt quiescence search to conquer the horizon effect.
- Yet another good reason to adopt iterative deepening. Return the current best move when time's up.

Use EVAL instead of UTILITY

- Usually maps state s into feature space f_i
- Typically use linear combination of features: $\text{EVAL}(s) = \Sigma_i w_i f_i(s)$
- Need to fine tune weights for strong play.

Heuristic Where Minimax May Go Wrong



- MINIMAX chooses the right branch.
- EVAL with an error with zero mean and standard deviation σ.
- $\sigma = 2$, the left branch is better 58% of the time.
- $\sigma = 5$, the left branch is better 71% of the time.

Forward Pruning

- Forward pruning does compromise solution quality (so is using EVAL)
  - Some moves are pruned immediately without further consideration.
- Beam search is one way to forward pruning. Dangerous since the best move might be pruned.
- PROBCUT uses the scores from previous searches to estimate the probability that a node is outside the $[\alpha, \beta]$ window.

Search vs. Table Lookup

- For many games, deep search usually helps little at the beginning.
- Instead, fast table looking up, huge databases, and statistical analysis help more.
- Table lookup also helps a lot toward the end of games.
  - Moscov state university solves all 7-piece endgames in 2012.
  - Total number of 5×10^{12} positions.
  - Longest mate takes 549 moves!

- Finally, early exchange favors computers than humans $\rightarrow$ deeper search and more probable falls in lookup.

Handling Repeated States

- We desire some BackTrack implementation instead of copying the whole board state.
- We desire to be able to identify repeated state very fast.
- We desire to store the scores of some recently-calculated states (transposition table).

Zobrist Hashing

Randomly generate Zobrist's keys (fixed) for each piece at each position. The hash key for a board state is then the XOR of these keys.

Zobrist Hashing

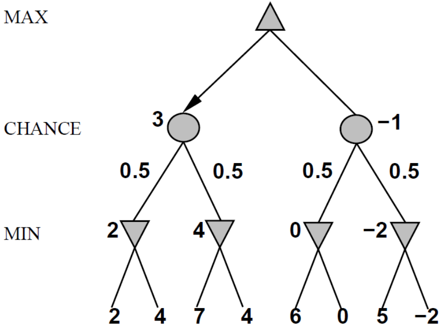
- The use of XOR makes BACKTRACK as easy as making a move (RESULT).
- $h(s)$ are stored in a large hash table to make future search faster.
- For the same state, the heuristic estimation from a deeper search replace that from a shallower one. Similar idea (also like LRTA*) can be used to improve h.

- Length of keys
  - Shorter keys are faster.
  - Longer keys reduce the chance of collision.

  - For chess, roughly $12 \times 64 \leq 800$ keys are needed.
  - For Go, roughly $2 \times 361 \leq 800$ keys are needed.
  - Nowadays, 64-bit key implementation is standard. Collision is not severe, but still needs to be handled properly.

Stochastic Games

- Introduce CHANCE nodes into the minimax tree.
- Instead of searching for maximum/minimum values, we now search for expected maximum/minimum values.



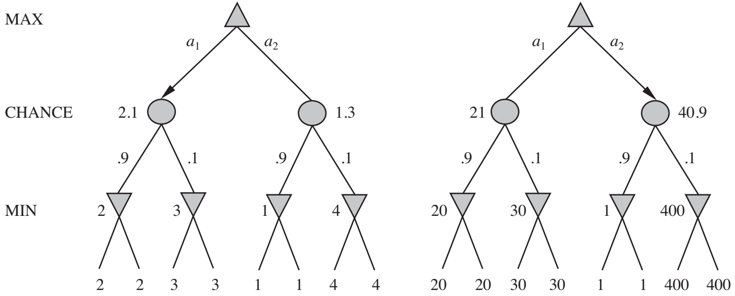
ExpectiMiniMax

- ExpectiMiniMax gives perfect play. (in what sense?)
- Similar to MiniMax, except we must also handle chance nodes.

$$\text{ExpectiMiniMax}(s) = \begin{cases} \text{Utility}(s) & \text{if Terminal-Test}(s) \\ \max_a \text{ExpectiMiniMax}(\text{Result}(s,a)) & \text{if Player}(s) == \text{max} \\ \min_a \text{ExpectiMiniMax}(\text{Result}(s,a)) & \text{if Player}(s) == \text{min} \\ \Sigma_a P(a)\text{ExpectiMiniMax}(\text{Result}(s,a)) & \text{if Player}(s) == \text{chance} \end{cases}$$

Sensitivity to Heuristic

- As mentioned before, we rarely can actually use UTILITY in EXPECTIMINIMAX.
- Instead, we use heuristic.
- However, unlike in MINIMAX, actual values of heuristic matter now. (In MINIMAX, only relative order matters.)



$$\{1, 2, 3, 4\} \rightarrow \{1, 20, 30, 400\}$$

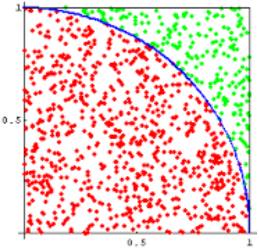
Performance of ExpectiMiniMax

- $\alpha - \beta$ pruning now does not apply to MAX/MIN nodes (why?).
- $\alpha - \beta$ pruning now still applies to CHANCE nodes (why?).
- Time Complexity: $O(b^m) \to O(b^m n^m)$, where n is the number of distinct dice rolls.
- Causes ExpectiMiniMax impractical in many cases.

- Solution: Instead of checking every MAX/MIN node, adopts Monte Carlo simulation at CHANCE nodes.
- Using random dice rolls to check only a certain number (decided by quality/time limit) of paths.

Monte-Carlo Simulation

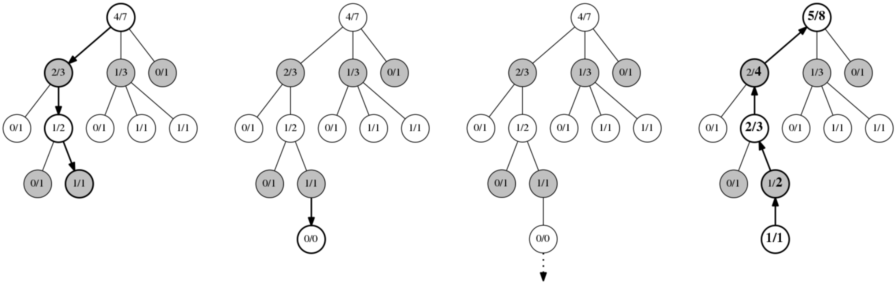
- Calculating π:
  - Uniformly random:
    $x \sim [0, 1], y \sim [0, 1]$.
  - Check whether $x^2 + y^2 \leq 1$.
  - The probability is $\pi/4$.



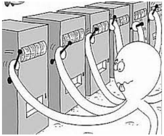
- Opinion polls.
- Mark and recapture.

Monte-Carlo Tree Search

- Selection: select the **most promising** leaf L.
- Expansion: create a node C from L.
- Simulation: random playout (or called rollout) from C.
- Backpropagation: back propagate the result from C to the root.



Multi-Armed Bandit Problem



- K arms (random variables) with distributions: $B = \{A_1, A_2, \ldots, A_K\}$, where A_i are of unknown independent distributions (not necessarily identical).
- Unknown means: $\mu_i = E[A_i]$.
- Regret ρ_T after T rounds: $T \cdot \mu^* - \sum_{t=1}^{T} \hat{r}_t$, where $\mu^* = \max_i \mu_i$ and $\hat{r}_t$ is the reward at round t.
- The goal is to minimize the regret.
- Zero-regret strategy: $\lim_{T\to\infty} \frac{\rho_T}{T} = 0$.

Zero-Regret Conditions

- Unbounded number of trials for each arm, given unbounded time.
- As time goes to infinity, the probability of choosing the current best arm tends to 1.

MAB Strategies

- ϵ-greedy (simple, not zero-regret):

  With probability of ϵ, pull an arm at random; otherwise, pull the current best arm.
- ϵ-decreasing (may be zero-regret):

  Same as ϵ-greedy, but ϵ decreases with time.
- UCB (upper confidence bound, zero-regret)

  Pull the arm with the highest UCB score.
  - UCB1: $\bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}$
  - UCB1-tuned: $\bar{x}_i + \sqrt{\frac{\ln n}{n_i} \min\left(\frac{1}{4}, \sigma_i^2 + \sqrt{\frac{2 \ln n}{n_i}}\right)}$

Partially Observable Games

- Different from stochastic games, unobservable parts are usually controlled by opponents, not probability.
- Examples: Cards held by other player in bridge, folded cards in poker, fogs in star craft.
- Different strategies may be applied and may all considered optimum against different opponents.
- If equilibrium exists, it's usually considered as optimum strategy.

Nash Equilibrium

- By John Nash — check out "A Beautiful Mind (2001)" if you want an informal introduction to him.

Information Definition

A set of strategies is a Nash equilibrium if no player can do better by unilaterally changing his or her strategy.

- Prisoners' dilemma

	B stays silent	B confesses
A stays silent	Each serves 1 yr	A: 5 yrs; B: free
A confesses	A: free; B: 5 yrs	Each serves 3 yrs

- Read Chapter 17 if you want to know more.

Summary

- MINIMAX search for zero-sum two-player games.
- Pruning techniques enable to search deeper.
- Due to time limit, heuristics are used to evaluate the “goodness” for a player.
- For stochastic games, we need to introduce chance nodes and search for expected maximum/minimum values.
- For stochastic games, $\alpha - \beta$ pruning is much less efficient, Monte Carlo simulations are often adopted to speed up the search.
- With limited observation, optimality is usually not well defined. If equilibrium exists, strategies in equilibrium are often considered optimum.