COMP30030: Introduction to Artificial Intelligence

Neil Hurley

School of Computer Science
University College Dublin
neil.hurley@ucd.ie

October 4, 2018

Alpha beta Effectiveness I

- Effectiveness depends on **node ordering**.
- Good move ordering improves effectiveness of pruning
- The best case occurs when each player's best move is the leftmost alternative (i.e., the first child generated). So, at MAX nodes the child with the largest value is generated first, and at MIN nodes the child with the smallest value is generated first.
- Worst Case ...
  - No advantage due to useless node ordering. That is, complexity $= O(b^d)$.
- Best Case ...
  - We could try and consider children nodes on a best-first basis because this is an effective ordering.
  - If this is possible then the $O(b^d)$ complexity of MiniMax becomes $O(b^{d/2})$.

Alpha beta Effectiveness II

- Since $b^{d/2} = \sqrt{b}^{d}$, this is the same as having a branching factor of $\sqrt{b}$ instead of b, thereby doubling the lookahead.

For example: Chess

- Average branching factor goes from 35 to 6.
- Allows for a much deeper search given the same amount of time.
- Allows computer chess to be competitive against humans.

Expected Case ...

- Empirical studies indicate an expected complexity of $O(b^{3d/4})$.

Multi-player Games

- MINMAX can be extended to multiplayer games by taking a vector of utility values.
- However, multiplayer games can be much more complicated.
  - alliances between players, that disrupt the normal purely selfish maximum utility behaviour.

Transposition Tables

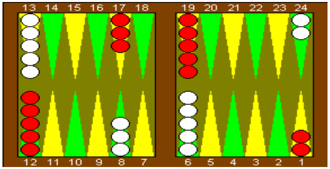
- Games often end up more than once in a state.
- We can cut down the time for searching if we store the states we have already computed the utility of in a place that we can access quickly.
- Typically this would be a hash,and the result is usually called a **transposition table**.
- This may result in dramatic savings.
- Usual problems with storing billions of states apply..
  - Again,there will need to be some strategy to store only states that are likely to be seen again.

Games and chance

- While heuristics introduce chance, there are some games in which chance is an inherent factor.
- E.g. when you play cards (more or less any game), you don't know beforehand which cards you'll be dealt.
- E.g. in backgammon you toss dice at every move. This typically widens the branching factor enormously..

Backgammon

- A two-player game, played on a one-dimensional track (although represented on a 2D board).
- Players take turns, roll 2 dice, move their checkers along the track based on the dice outcome.
- Win: moving all the checkers all the way to the end, and off the board.
- Gammon (double win): a player wins when the other still hasn't taken any checkers off the board.



Backgammon

- **Hitting a checker: landing on it, when it’s alone; it is sent all the way back.**
- **Blocking: it is possible to build structures that make it difficult to move forward for the opponent.**

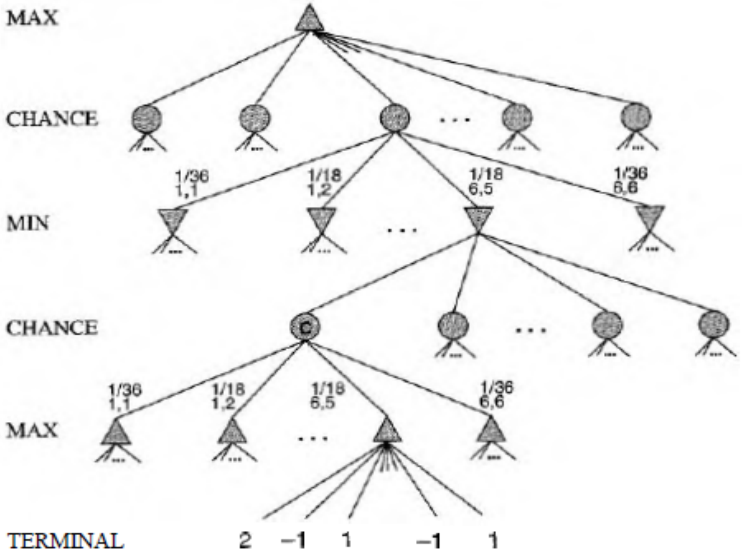
Horrid complexity

- Once you have rolled your dice, you know what legal moves you have.
- However, you do not know what the other player will roll.
- And of course you don't know what you will roll at the next move, etc.
- Theoretically you could still construct a tree and do MINMAX, but things get hairy almost instantly.

Dice

- There are 36 ways to roll two dice.
- However, N-M is the same as M-N, so there are only 21 distinct ways of rolling.
- Doubles have a 1/36 chance, while any other combination has a 1/18 chance.
- You have a branching factor of 21 at the chance nodes, followed by a variable (but often large) branching factor thereafter.

Chance nodes



Expected MINMAX

- Instead of computing definite, deterministic MINMAX values, you have to compute *expected* MINMAX, which are weighted averages of the outcomes at the chance nodes.
- Essentially you sum the MINMAX values from the descendants with a weight which corresponds to the probability of the dice outcome leading there.

vs. MINMAX

- **Notice that in theory you could still do plain MINMAX, taking the worst dice outcome from the chance nodes.**
- **But that would hardly make any sense.**
- **There is an intrinsic difference between assuming that the adversary will always play the best move for them (which is reasonable, though perhaps slightly pessimistic), and assuming you'll always get the worst dice roll possible (which quickly borders impossibility).**
- **Besides, if you always get the worst roll you can just pack up and go home: you have no chance of winning.**

Combining it all

- EXPD_MINMAX(n) = {
- UTILITY(n) // if n is terminal
- MAX $_{\text{s in successors}}$ EXPD_MINMAX(s)
- // if n is MAX
- MIN $_{\text{s in successors}}$ EXPD_MINMAX(s)
- // if n is MIN
- SUM $_{\text{s in successors}}$ EXPD_MINMAX(s) x P(s)
- //if n is CHANCE

Hard to look ahead

- Although you average over chance nodes, you still have to look at them and at their successors.
- You do the maths differently with them, but they still contribute to the total branching factor of the tree the same as MAX or MIN nodes.
- With an effective branching factor of 400, you can look ahead 2 moves (4 ply – 400^4) at best..

What to do with Backgammon?

- **Brute force approach is hardly feasible.**
- **In general we need to develop positional judgement, rather than trying to look ahead explicitly: a position is good or bad *per se*.**
- **Put it another way: we need *very* good heuristics, because we don't stand a chance of getting close to the terminal states, unless we are in the very endgame.**

TD-Gammon

- On the other hand, rather than *designing* good heuristics by hand, so could instead try to *learn* them through game playing,.
- TD-Gammon. Heuristic evalutations are carried out by a **Neural Network**[1], which is trained through game playing.

[1]More about this later

Learning how to play

- Is Minimax really intelligent?
- By assuming the opponent is playing optimally, we might miss an opportunity to achieve a bigger win.
- In particular we might miss an opponent's weaknesses
- Shouldn't we learn from the experience of repeatedly playing against an imperfect opponent?

Formulating Reinforcement Learning

- The World described by a discrete, finite set of states and actions
- At every time step t, we are in a state s_t, and we:
  - Take an action a_t (possibly null action)
  - Receive some reward r_{t+1}
  - Move into a new state s_{t+1}
- Decisions can be described by a policy
  - a selection of which action to take, based on the current state
- Aim is to maximize the total reward we receive over time
- Sometimes a future reward is discounted by γ_{k-1}, where k is the number of time-steps in the future when it is received.

Back to TicTacToe

Consider the game tic-tac-toe:

- reward: win/lose/draw the game (+1/-1/0) (only at final move in given game)
- state: positions of X's and O's on the board
- policy: mapping from states to action
  - based on rules of game
- value function: prediction of reward in future, based on current state
  - In tic-tac-toe, since state space is tractable, we can use a table to represent the value function

Learning as we play I

- Consider a table, where we keep, *for each state of the game*, the probability of winning from that state.
- For terminal states, we can set this probability exactly – 1.0 for a winning terminal state and 0.0 for a losing terminal state.
- For all other states, we initially set the probability to 0.5. Let's call that probability the *estimated* value of the state, $V(s)$.

- During play we use a greedy approach:
  - At the current position, we generate all the child states that can be reached (all the possible moves)
  - We select from among these moves, the child state with the greatest value, according to our table.

Learning as we play II

- But we learn as we go:
  - We attempt to improve our estimate of the value of parent state, through a learning rule such as

$$V(s) \leftarrow V(s) + \alpha[V(s') - V(s)]$$

  - $\alpha \in [0, 1]$ is a small positive number, called the *step-size* parameter, and affects the *learning rate*
  - s' is the child state and s the parent state.
  - We are using our estimate of the value of the child state to improve our estimate of the parent state.

Learning as we play III

- We play the opponent not once, but many times, improving the value function with each play.
  - Imagine this improvement filtering back from the terminal states along the paths used to reach them.
- This simple update is an example of *temporal difference* learning.
- If α is reduced towards 0 properly over time, this method converges for any fixed opponent.

Exploration vs Exploitation

- By adopting a greedy approach to playing we are fully exploiting our current estimate of the value function.
- However, with such an approach, we are inclined to follow paths that were found to be good in the past.
- The danger is that we might never explore other paths, that are also good (or better), because they are never chosen and the estimate of their value is poor.
- We should therefore sometimes ignore the value function and just pick a random move — exploration.
- This is not done in order to improve our chances of winning the current game, but rather to learn more about the search space to improve our playing of future games.

- ϵ-greedy approach: with probability ϵ, choose any child state at random, otherwise, with probability $1 - \epsilon$, use the greedy strategy to choose the child state. ϵ controls the amount of exploration.

Markov Decision Processes

- The world and the actor may not be deterministic or our model of the world may be imperfect.
- We assume the Markov property: the future depends on the past only through the current state.
- We describe the environment by a distribution over rewards and state transitions:

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

- A policy is then also described as a probability distribution over the actions, given the state:

$$\pi(s, a) = P(a_t = a | s_t = s)$$

The policy is not a fixed sequence of actions, but rather a conditional plan.

Markov Decision Problem

- Markov Decision Problem: consists of a tuple (S, A, P, R, γ) where
  - S is the state space
  - A is the set of possible actions to move through the state space
  - R is the reward function, $(R(s, s', a)$ is the immediate reward obtained on moving from s to s' through action a
  - P is the transition matrix, that governs how we move through the state space

$$P(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a)$$

  - γ is the discount value that weights future

Standard MDP problems include:

1. Planning: given a complete Markov decision problem as input, compute a policy with optimal expected return.
2. Learning: If only have access to past experience, learn a near-optimal strategy.

MDP Formulation

- Goal: find a policy π that maximises expected accumulated future rewards, obtained by following π from state s_t.
- The *value* can be written as the sum of all rewards obtained from time t onwards (discounted by γ)
- The value depends on the policy that selects which action to take at each decision point:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

Approach

- We might try to learn the function V^* given by

$$V^*(s) = \max_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

  where $\delta()$ is the state reached by applying action a at state s
- We could then do a lookahead search to choose the best action from any state

$$\pi^*(s) = \arg\max_a [r(s,a) + \gamma V^*(\delta(s,a)]$$

- But this only works if we know δ and r.

Q Learning

- Define a new function very similar to V^*

$$Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$$

- If we learn Q, we can choose the optimal action without knowing δ

$$\begin{aligned}\pi^*(s) &= \arg\max_a [r(s,a) + \gamma V^*(\delta(s,a))] \\ &= \arg\max_a Q(s,a)\end{aligned}$$

Q Learning

- Q and V^* are related by

$$V^*(s) = \max_a Q(s, a)$$

- Applying recursively, we get

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma V^*(\delta(s_t, a_t)) \\ &= r(s_t, s_t) + \gamma \max_{a'} Q(s_{t+1}, a') \end{aligned}$$

- Suppose the learner has a current approximation to Q, say $\hat{Q}$. Then we can update that approximation using the Q learning rule:

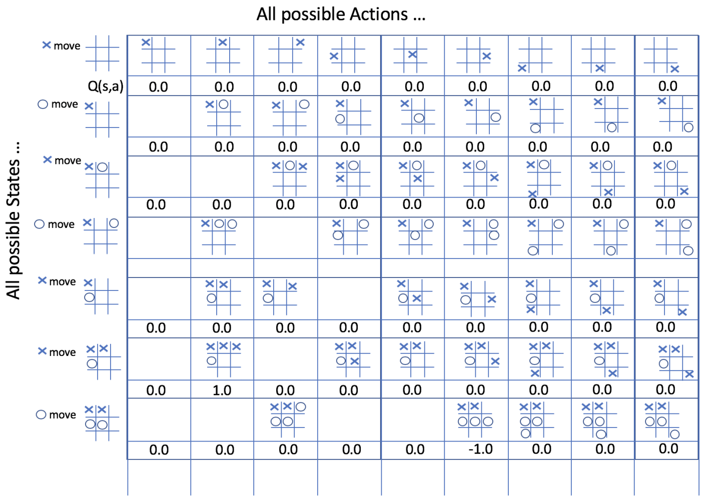
$$\hat{Q}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')$$

where s' is the state resulting from applying action a in state s.

Q Learning Algorithm

```
1 For each state $s$, initialise a table entry with $\hat{Q}(s,a) \leftarrow 0$
2 Start in some initial state $s$
3 Do forever
    3.1 Select an action $a$ and execute it.
    3.2 Receive immediate reward $r$
    3.3 Observe the new state $s'$
    3.4 Update the table entry for $\hat{Q}(s,a)$, using the Q learning rule.
    3.5 $s \leftarrow s'$
4 If we get stuck at a particular state, we can reset to a new
  initial state and rerun the loop.
```

Q table for TicTacToe



- Need to keep a track of Q value for all possible states and all possible actions from each state.

Q table for TicTacToe

- Table of $Q(s, a)$ is feasible for tic-tac-toe, but not for large problems.
- Imagine instead that we could learn some function, f, such that

$$f(\beta_1, \ldots \beta_m, s, a) \approx Q(s, a)$$

- Here $\beta_1, \ldots \beta_m$ are parameters, which are adjusted by the learning algorithm.
- Much fewer parameters than entries in the table.
- Learning functions such as this is an approach used widely in Machine Learning.

Where are we at with Games? I

- At checkers an Othello, computers can beat humans because branching factors are relatively low and terminal states can often be reached by a MINMAX algorithm. (In one instance, a program declared a checkers win after 5 moves).
- In chess, we have managed to match and beat world champions. As this has required an enormous amount of computer power, some have argued that is it not a particularly "intelligent" way to win – human heuristics would seem to still be much stronger than computer heuristics.
- Google have recently created a Go program—"Alpha Go" that has beaten the best human Go champions. It was not expected that this could be achieved as the branching factor in Go is as high as 361 at the beginning of the game.
- Alpha Go is a famous recent success of Deep Learning. Neural networks were used to learn good evaluation functions from a database of games played by expert human players.

Where are we at with Games? II

- Alpha Go uses reinforcement learning. Once the program became strong by learning from human players, it improved its performance even more by *playing against itself*. A newer version **AlphaZero** learned to outperform Alpha Go, by *playing against itself*, with no bootstrapping from human expert games.
- Similarly, at backgammon, programs can now beat world champions. And the best programs, such as TD-Gammon, again learn from self play.
- Now human game-players are learning new strategies by observing the strategies developed by game-playing programs.