n-Player Adversarial Search

COMP30024 Artificial Intelligence

Matt Farrugia

The University of Melbourne

Tuesday 2nd April 2019

Review and overview

Review of minimax

Minimax algorithm gives "perfect play" for *2-player*, *zero-sum*, *perfect information* games.

- True minimax requires search all the way to *terminal states* (where the *utility function* has known value).
- Optimality assumes that opponent does the same?
- This is usually Ridiculously Infeasible.

What can we do? Some ideas:

- Approximate minimax value: Switch to using a *cut-off test* and *evaluation function*. Note: Play no longer "perfect".
- Avoid unnecessary computation: *Alpha-beta pruning* skips branches that definitely cannot influence result.

Minimax was not designed to work with more than 2 players.

Plan

Two methods of approaching games with n players ($n > 2$):

- Approach 1: Reduce our n-player games to 2-player games.
- Approach 2: Extend our algorithms to handle more players.

Reducing games

Reducing games

What if we could somehow model an n-player game as a 2-player game?

Since the result is a 2-player game, we could then use the same techniques we already know (heuristic minimax, alpha-beta pruning, learned evaluation function) without much modification.

Two example models:

- *Paranoid* reduction: "It's just me against everyone else".
- *Alliance* reduction: More generally, partition players into any 2 "teams" of cooperating players.

Paranoid reduction

Idea: Consider all opponent players to be one single player aiming to minimise your objective (let's call this player "the enemy"). The enemy's possible actions are the possible *sequences of actions* that could be taken by each opponent between two of your turns.

Challenges:

- The enemy has a larger range of possible actions, might this increase our search's branching factor significantly?
- Is it possible to be *too* paranoid in some situations, and worry about actions other opponents (perhaps similarly paranoid) will likely avoid?

Alliance reduction

Intuitively justified: Playing *n*-player games often involves forging *temporary* and *informal* alliances to achieve shared goals (e.g. working together to prevent another player from winning).

Idea: Partition players into any 2 "teams". Modify algorithm as we did for paranoid reduction: Group together turns by players from the same team and consider all possible action sequences.

Challenges:

- How should we decide what partition to use? Since alliances are dynamic, how will we know when to change the partition?
- How can we prevent ourselves from unexpected "betrayal" from players we must trust to act a certain way?

Extending search

Extending search

Instead of making modelling assumptions about our *n*-player games, is it possible to extend the minimax algorithm and associated techniques to games with more than 2 players?

The answer is yes! Let's meet the *max^n algorithm*.

max^n is an extension of minimax that provably provides the same "perfect play" guarantees as minimax to *n*-player games.

That is... *if* you search the whole tree. As with a full minimax search, this is usually practically impossible, so we need to limit search depth and approximate state utility.

$\max^n$ algorithm

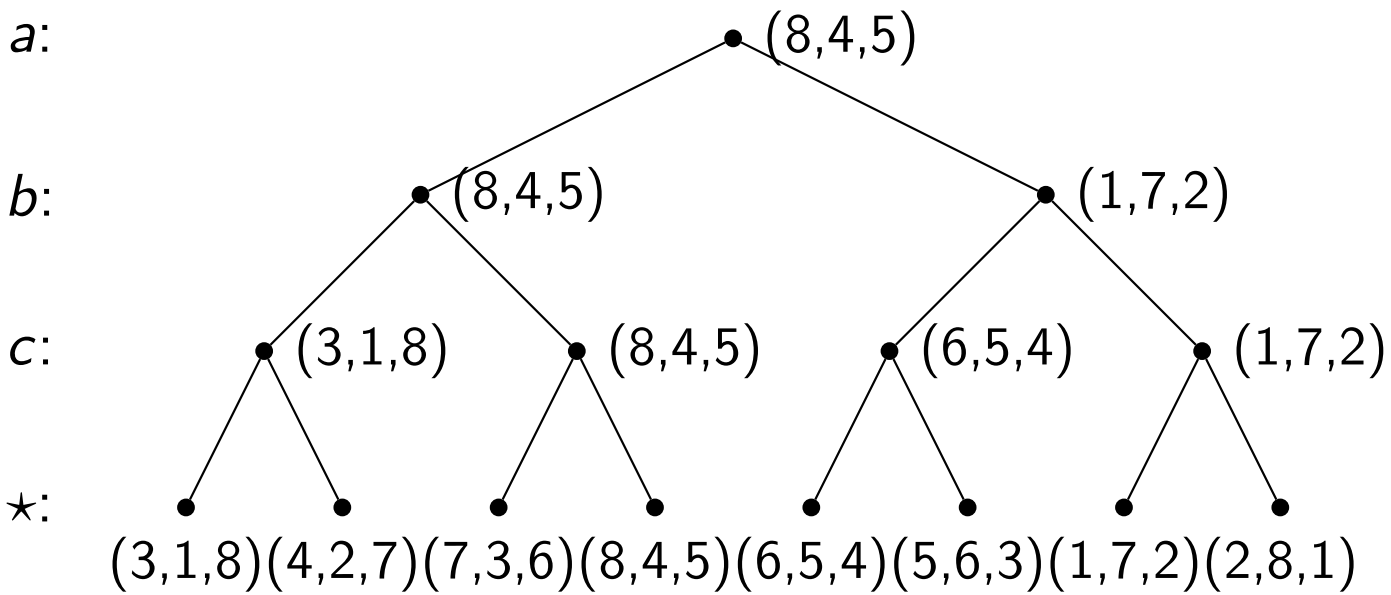
Idea: It's like minimax, but:

- Layers of the tree alternate between all n players.
- For each terminal state (or cut-off state), compute utility values (evaluation values) for all players and store in an n-dimensional *utility vector*.
- Backup values to non-terminal states by choosing the vector with the highest value in the next player's dimension.

Minimax can be viewed as a special case of $\max^n$ where the second dimension of the utility vector is implicit and is the negative of the first (thus *max* becomes *min* for the opponent).

$\max^n$ algorithm
An example

A three-ply lookahead $\max^n$ tree for a turn-based game with three players (a, b, and c):



Pseudo-code for $\max^n$

Pseudo-code for (heuristic) $\max^n$:

```
# Inputs: State s, Player ID p
# Output: (utility vector, best action)
function MAXN(s, p):
    if CUTOFF-TEST(s):
        return (EVALUATE(s), no-action)
    v-max  <- (-inf, -inf, ..., -inf) # n dimensions
    best-a <- no-action
    for each Action a in ACTIONS(s):
        (v, *) <- MAXN(RESULT(s, a), NEXT(p))
        if v[p] > v-max[p]:
            v-max  <- v
            best-a <- a
    return (v-max, best-a)
```

$\max^n$ tree pruning

In the general case, $\max^n$ trees are less amenable to pruning than minimax trees. Assuming utility vectors are bounded, some pruning is still possible:

- "Immediate pruning": If a child has the highest possible utility value for the maximising player, we need not consider other children.
- "Shallow pruning": If a child's utility vector guarantees that some other sibling of the parent will be chosen by the grandparent, we need not consider other children.

Unfortunately, "deep pruning" is no longer possible: Even if we can guarantee a utility vector from some deeper descendant will not make it back to the grandparent, in n-player games with $n > 2$, the values in such vectors can still influence which vector does make it back. We can't safely prune these branches.

$\max^n$ tree pruning

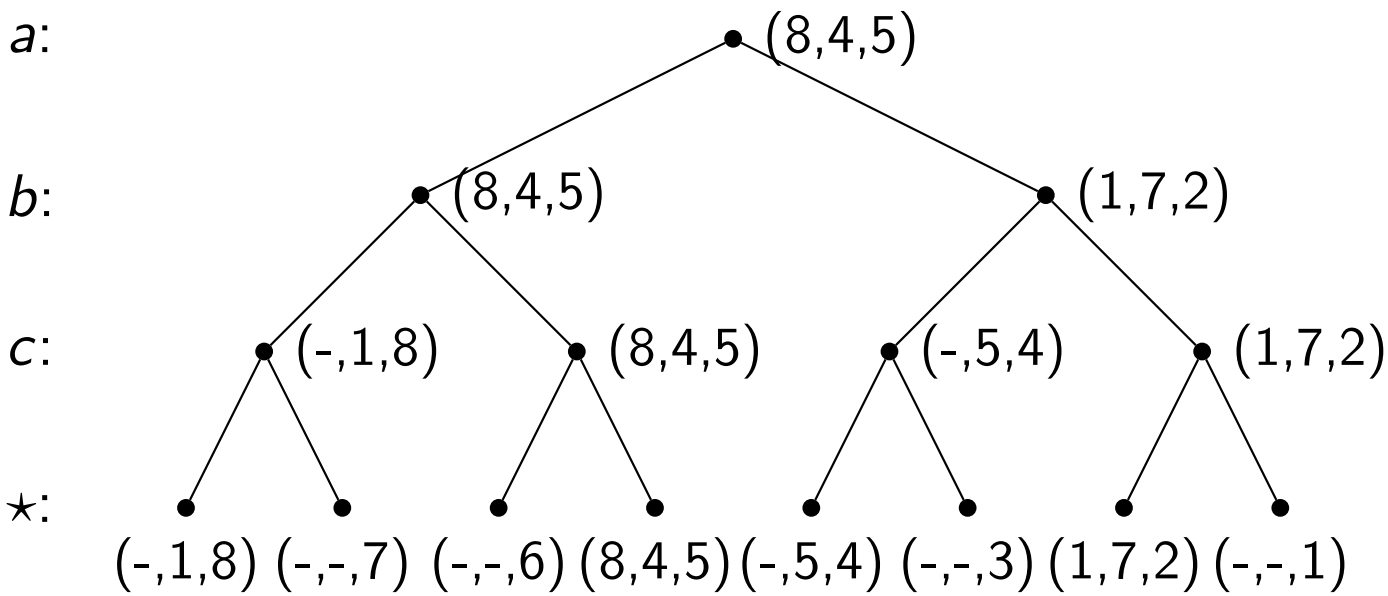
As a result, pruning may not provide the same average-case complexity improvements as for 2-player games:

- In the best case (that is, with a perfect move ordering), shallow $\max^n$ pruning approaches the performance of alpha-beta pruning, resulting in an effective branching factor of $\frac{1}{2}(1 + \sqrt{4b - 3})$ (compared to $\sqrt{b}$ for best-case alpha-beta pruning).
- In the average case (depending on average case model assumptions), shallow pruning does not reduce effective branching factor (unlike alpha-beta pruning).

Details and examples of these pruning techniques, and a shallow-pruning algorithm, are in (Korf, 1991).

Lazy evaluation

Another kind of pruning is now possible using a "lazy" evaluation function: When evaluating a state, we need only compute the components of the vector that will be compared to others.



This technique can reduce the number of components calculated by a constant factor.

Conclusion

Summary

Games with more than 2 players introduce new algorithmic and strategic challenges:

- We may reduce such games to 2-player games in an attempt to avoid these challenges. Or, we may extend our algorithms and embrace these challenges.
- In the latter case, we see that perfection is still unattainable, so we must approximate anyway.

Examples of skills expected (to be tested only through your projects, not in the final exam):

- Implement *n*-player game search algorithms or reductions.
- Discuss challenges associated with handling *n*-player games.
- Consider novel approaches for solving these challenges.

References and optional reading

- Russell, S. J., & Norvig, P. (2016). Artificial intelligence: a modern approach.
  - Section 5.2.2 touches on $\max^n$ and alliances.
- Luckhart, C., & Irani, K. B. (1986). An Algorithmic Solution of N-Person Games.
  - Introduces $\max^n$ and "lazy" evaluation idea (confusingly, calling it shallow pruning).
- Sturtevant, N. R., & Korf, R. E. (2000). On pruning techniques for multi-player games.
  - Analyses $\max^n$ alpha-beta pruning, introduces "paranoid" reduction, includes some experiments (on card games).
- Korf, R. E. (1991). Multi-player alpha-beta pruning.
  - Analyses $\max^n$ alpha-beta pruning in greater detail.

The 3 papers are *short* (5, 7, and 13 pages) and *available online*.