

Adversarial Search

CS 121

Lecture 8

Chris Archibald

July 20, 2009

1 Big picture

1.1 Previously in search

Until now we have focused on search with a single-agent in an observable environment, where the results of actions are deterministic. We have looked at blind search, heuristic search, local search, constraint satisfaction and action planning. This is the material that will be on the midterm. Any questions about those topics?

For the rest of the class, we will begin to change some of the assumptions we made, and see how we can still design rational agents for these new environments and problem types.

2 Adversarial search

2.1 Today

Today we will look first at a multi-agent environment. This is typically called a *game*. If the goals of the agents are in opposition to each other (if I win you lose), then we have an *adversarial game*. The other assumptions that we made will still hold for the present, we can observe the current state, actions have deterministic results.

Let's see a quick example:

2.2 Tic-Tac-Toe

Draw out some of the game tree for tic-tac-toe, showing how this will work.

3 Games

3.1 why games?

- Long engaged and judged human intelligence, typically easy to represent . Chess playing programs have been around as long as programs have.
- Hard to solve
- Must make some decision, might not be able to make optimal decision
- Bad decisions are punished by losses
- Making use of time.

3.2 Types of games

Type of information (perfect / imperfect) and Is chance involved? (yes / no)

4 Today's plan

- Adversarial search, the basics
- minimax search
- Evaluating leaf nodes
- Modifications to minimax ($\alpha - \beta$ -pruning)
- Different environments (chance, imperfect information)

5 The basics

5.1 Definition

A game is

- State space = all the states of the game
- Initial state = board position and player whose move it is
- Successor function = actions available and what results
- Terminal test = is the game over?
- Utility function = who won in this terminal state?

5.2 Example game tree

Definitions: ply, MAX player, MIN player, triangles, squares, utility

5.3 What should each player do?

A *strategy* in a game is a complete contingent plan. We must be ready for any eventuality that arises. *Optimal strategies* in general depend on the opponent. But, what decisions can we make in the absence of knowledge about the opponent?

5.4 Minimax

What if we assume that our opponent is perfect? Let's see what that gives us.

5.4.1 Look at example tree

Work out how perfect player would play, and how we could respond perfectly

We can *back up* a value for each node by looking at its children. This is called the *minimax value* for the node

Define minimax-value

$\text{minimax-value}(n) = \text{utility}(n)$ if n is terminal, $\max_{s \in \text{successors}(n)} \text{minimax-value}(s)$ if n is max node, min otherwise

5.4.2 What if our opponent is not perfect?

In this case we can't do worse, but we might have been able to do better with a different strategy

5.5 Minimax-algorithm

Depth-first search of game tree, backing up values recursively.

5.6 Extension to three players

page 166 in book

6 Pruning

The search tree is far too big for most games to actually run the minimax algorithm in order to determine what action to take.

It would be nice if we could avoid searching some of the space, but we don't want to hurt the optimality of our algorithm

Let's revisit our example We don't even need the leaf values from 2 nodes. Can we utilize this fact?

6.1 $\alpha - \beta$ pruning

α = value of best choice we have found along path for MAX

β = value of best choice we have found along path for MIN

We can stop looking at the descendants of a node when its value is worse than the current α or β values for MAX or MIN.

6.1.1 Let's revisit the example, this time with $\alpha - \beta$ pruning

Any questions?

6.1.2 Let's look at a slide example

7 Evaluation functions

Even with the help that alpha-beta provides, we still cannot do anything close to expanding all of the nodes in the tree that we need to make a perfect decision.

What can we do?

7.1 Idea: evaluation function

What if we had a function that told us how good each state was? This would help us make our decision.

Restrictions? How to design?

- Must order the terminal states correctly
- Relatively fast (could compute exact value, but that would defeat the whole purpose)
- For non-terminal nodes, value should correspond to chances of winning

What do we mean by chances of winning? There is no chance. Well, we have uncertainty from computational limitations. Basically, evaluation functions will typically give the same value to a large set of different states, we can think of the chance of winning as being the fraction of the states from that set that lead to a win.

7.2 Weighted linear function

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s)$$

This assumes that features are independent. We can also use non-linear combinations. These weights and features are *not* part of the rules of chess. Typically hand created from experience

Notice that the values returned can be arbitrary, only order is important

8 Cutting off search

Issues: need something more sophisticated. Otherwise, if we are one move away from a loss, we won't realize it.

Issues:

- Quiescence - positions less likely to exhibit wild swings in value
- Horizon effect - Push bad things over the horizon
- singular extensions - search deeper with *clearly better* moves
- forward pruning- what if we cut the best moves, can't do from root, because might miss some obvious things

9 Current game playing

Back to slides, then show video

10 Welcome back

Now we will talk about some other types of games, and how to deal with them

11 Games with chance

Backgammon has rolls of the dice

Change search tree to include *chance nodes* (circles)

How to make a correct decision now? Must now use expected value, where expectation is taken over chance event

we now have an expectiminimax value for games with chance nodes

11.1 Evaluation values

Now we need to be careful about what they mean

example from R and N slides

11.2 Computational complexity

$O(b^m n^m)$. Extra cost makes looking ahead very hard.

What about alpha-beta in this case?

Well, alpha-beta ignores things that just won't happen, given perfect play. But with chance, we can't say that, and it is hard to prune anything.

What about something similar? We can look at the highest and lowest that each child should be. If we have bounds on the evaluation function, then we can do this.

Talk about poker