COMP9414: Artificial Intelligence

Adversarial Search

Wayne Wobcke

Room J17-433
wobcke@cse.unsw.edu.au
Based on slides by Maurice Pagnucco

Adversarial Search

- Shall investigate two uses of search where we can apply other strategies to search the state space
- In particular we shall investigate adversarial search in which we search through a space where not all operators (choices) are under our control
- We shall also briefly discuss constraint satisfaction problems
- Reference:
  - Ivan Bratko, Prolog Programming for Artificial Intelligence, Addison-Wesley, 2001. (Chapter 22)
  - Stuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, Second Edition, Pearson Education, 2003. (Chapter 6)

Adversarial Search

- In many problems — especially game playing — you're are pitted against an opponent
- This means that certain operators are beyond your control
- That is, you cannot control your opponent's moves
- You cannot search the entire space from the outset looking for a solution since your opponent may make a move which makes any path you find obsolete
- What you need is a strategy that leads to a winning position regardless of how your opponent plays

Overview

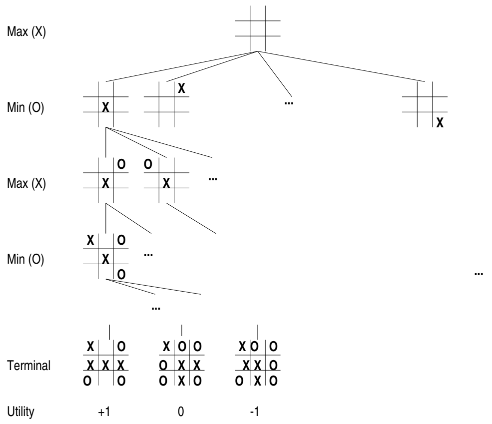
- Minimax
- Alpha-Beta Pruning
- Constraint Satisfaction as Search
- Conclusion

Games as Search Problems

Require the following components:

- initial state — board position plus which player has first move
- operators — legal moves
- terminal test — determines if game is completed
- utility function — numeric value for outcome of game

Example — Tic-Tac-Toe



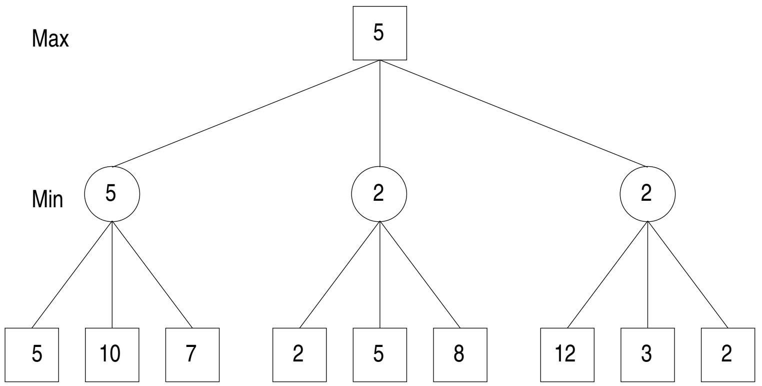
Minimax Criterion

Assume game tree of uniform depth (to simplify matters)

- Generate entire game tree
- Apply utility function to each terminal state
- To determine utility of nodes at any level, if Min's turn to play it will choose child with minimum utility, otherwise Max will choose child with maximum utility
- Continue backing up values from leaf to root, one level at a time

Maximizes utility under assumption that opponent will play perfectly to minimize it (assuming also opponent has same evaluation function)

Minimax Example



Minimax Algorithm

function MinimaxValue(state, game) **returns** utility value

  if TerminalTest[game](state)
    then return Utility[game](state)
  else if Max is to move in state
    then return highest MinimaxValue of successors(state)
  else return lowest MinimaxValue of successors(state)

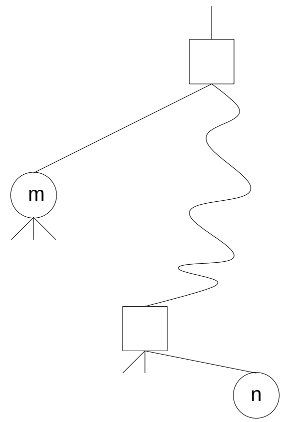
Alpha-Beta Pruning

- In most games it will be impossible to try and calculate minimax as described — the game tree will be just too big
- There is however a way of pruning the amount of work to be done and still make the correct minimax decision
- Pruning — elimination of branches from the search without examination
- Alpha-beta pruning returns a pruned minimax tree

Alpha-Beta Pruning

- **Idea:** Consider node *n* in search tree such that certain player has a choice of moving to that node
- If the player has a better choice *m* either at the parent node of *n*, or at any choice point further up, then *n* will never be reached in actual play
- Once we have ascertained enough information about *n* by looking at some of its successors to reach this decision, we can prune it

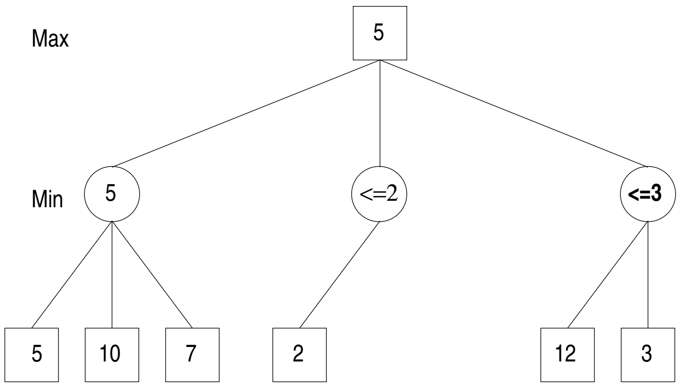
Alpha-Beta Pruning



Alpha-Beta Pruning

- Minimax is depth-first
- At any point we only have to consider the nodes on a single path in the search tree
- Suppose α is the value of the best choice for Max on the path and β the value of the best choice for Min on the path
- Alpha-beta updates the values of α and β and prunes any subtree as soon as it can determine whether it is worse than the current α or β

Alpha-Beta Pruning Example



Alpha-Beta Pruning Algorithm

function MaxValue(state, game, α, β) **returns** minimax value of state

  if CutoffTest(state) **then return** Eval(state)
  for each *s* in Successors(state) **do**
    $\alpha \leftarrow Max(\alpha, MinValue(s,\ game,\ \alpha, \beta))$
    if $\alpha \geq \beta$ **then return** β
  return α

function MinValue(state, game, α, β) **returns** minimax value of state

  if CutoffTest(state) **then return** Eval(state)
  for each *s* in Successors(state) **do**
    $\beta \leftarrow Min(\beta, MaxValue(s,\ game,\ \alpha, \beta))$
    if $\beta \leq \alpha$ **then return** α
  return β

Games of Chance

- Many problems and many games include an element of chance
- For example, the roll of dice (backgammon)
- The game tree must now include chance nodes representing the element of chance and labelled with the likelihood that the given chance event will occur
- We must now work with expected values
  $expectimax(C) = \sum_i P(d_i) max_{s \in S(C,\ d_i)}(utility(s))$

Constraint Satisfaction Problems

- Constraint Satisfaction Problems (CSPs) are problems in which states are defined by the values taken by a set of variables and the goal test specifies a set of constraints the values must satisfy
- Problems that can be expressed as CSPs: N-queens, VLSI layout, scheduling, cryptarithmetic
- Can use search to look for an assignment of values to variables such that the constraints are satisfied
- CSP has become a powerful and commonly used technique in AI with its own algorithms for determining variable assignments (e.g. arc consistency, hill climbing, simulated annealing, etc.)

Constraint Satisfaction Problems

	S	E	N	D	
					+
	M	O	R	E	
M	O	N	E	Y	

Conclusion

- Search is a common technique in problem solving especially when our knowledge of the problem or domain is limited
- It is important to spend some time thinking about the problem in order to decide how the problem states will be represented and which search strategy to apply
- We have only investigated a small number of search techniques
- We have examined some uninformed (blind) and informed (heuristic) strategies plus some techniques for adversarial search and constraint satisfaction problems