

For Wednesday

- No reading
- Program 5 due

Research Paper

- Any questions?

Program 5

- Any questions?

Problem Solving as Search

- Many problems have a solution space that can easily be thought of as a directed graph or a tree
- We can solve problems of this type by searching for the optimal solution in the space of possible solutions (the **solution space**)

Implicit Trees/Graphs

- Note that we do NOT have to construct the graph of the entire solution space
- We only need a procedure for finding the next set of nodes

Backtracking

- Essentially, depth-first search in a solution space which can be represented as a directed graph
- When we discover that the current node does not produce the solution we want, we backtrack to a node where we can make an alternate decision and proceed from there

Backtracking Method Steps

- Define the solution space
- Organize the space appropriately to search in
- Search depth-first using bounding functions to avoid searching uninteresting parts of the space

Backtracking Approach

- We're going to load the first ship with containers that match its capacity as closely as possible
- If the second ship has capacity greater than the remaining boxes, we have a feasible solution

Bounding Functions

- We need to recognize infeasible solutions
- We need to recognize bad solutions

N-Queens

- Placing a set of N queens on an $N \times N$ board such that no two queens are attacking each other.

Game Playing Problem

- Instance of **general search problem**
- States where game has ended are **terminal states**
- A **utility function** (or **payoff function**) determines the value of the terminal states
- In 2 player games, MAX tries to maximize the payoff and MIN is tries to minimize the payoff
- In the search tree, the first layer is a move by MAX and the next a move by MIN, etc.
- Each layer is called a **ply**

Minimax Algorithm

- Method for determining the optimal move
- Generate the entire search tree
- Compute the utility of each node moving upward in the tree as follows:
 - At each MAX node, pick the move with maximum utility
 - At each MIN node, pick the move with minimum utility (assume opponent plays optimally)
 - At the root, the optimal move is determined

Recursive Minimax Algorithm

```
function Minimax-Decision(game) returns an operator
  for each op in Operators[game] do
    Value[op] <- Minimax-Value(Apply(op,
    game),game)
  end
  return the op with the highest Value[op]
```

```
function Minimax-Value(state,game) returns a utility value
  if Terminal-Test[game](state) then
    return Utility[game](state)
  else if MAX is to move in state then
    return highest Minimax-Value of Successors(state)
  else
    return lowest Minimax-Value of Successors(state)
```

Making Imperfect Decisions

- Generating the complete game tree is intractable for most games
- Alternative:
 - Cut off search
 - Apply some heuristic evaluation function to determine the quality of the nodes at the cutoff

Evaluation Functions

- Evaluation function needs to
 - Agree with the utility function on terminal states
 - Be quick to evaluate
 - Accurately reflect chances of winning
- Example: **material value** of chess pieces
- Evaluation functions are usually **weighted linear functions**

Alpha-Beta Pruning

- Concept: Avoid looking at subtrees that won't affect the outcome
- Once a subtree is known to be worse than the current best option, don't consider it further

General Principle

- If a node has value n , but the player considering moving to that node has a better choice either at the node's parent or at some higher node in the tree, that node will never be chosen.
- Keep track of MAX's best choice (α) and MIN's best choice (β) and prune any subtree as soon as it is known to be worse than the current α or β value

```
function Max-Value (state, game,  $\alpha$ ,  $\beta$ ) returns the minimax value
  of state
  if Cutoff-Test(state) then return Eval(state)
  for each s in Successors(state) do
     $\alpha$   $\leftarrow$  Max( $\alpha$ , Min-Value(s , game,  $\alpha$ ,  $\beta$ ))
    if  $\alpha$   $\geq$   $\beta$  then return  $\beta$ 
  end
  return  $\alpha$ 
```

```
function Min-Value(state, game,  $\alpha$ ,  $\beta$ ) returns the minimax value of
  state
  if Cutoff-Test(state) then return Eval(state)
  for each s in Successors(state) do
     $\beta$   $\leftarrow$  Min( $\beta$ ,Max-Value(s , game,  $\alpha$ ,  $\beta$ ))
    if  $\beta$   $\leq$   $\alpha$  then return  $\alpha$ 
  end
  return  $\beta$ 
```

Red Black Trees