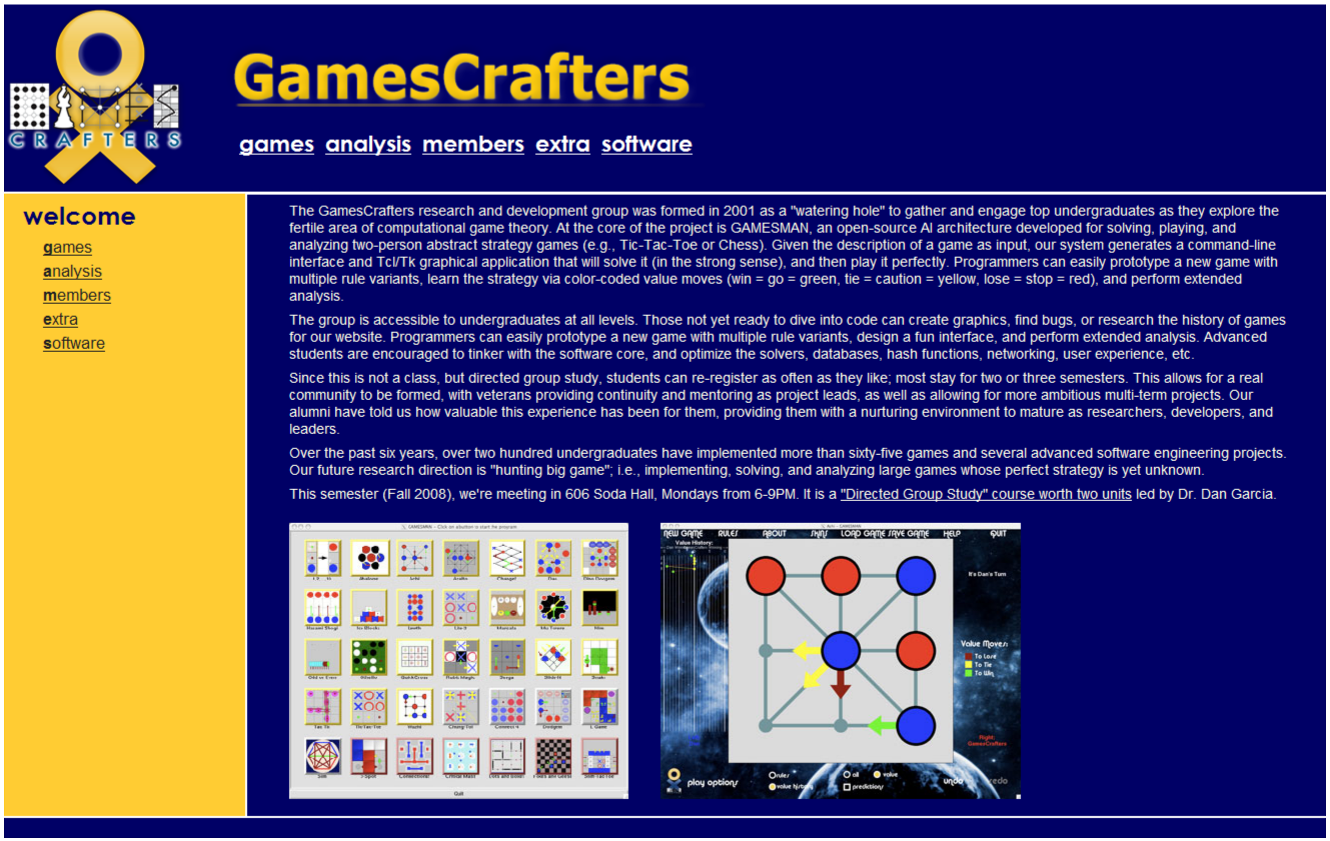
Game Playing State-of-the-Art

- **Checkers:** Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions. Checkers is now solved!
- **Chess:** Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Othello:** Human champions refuse to compete against computers, which are too good.
- **Go:** Human champions are just beginning to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances using Monte Carlo (randomized) expansion methods.
- **Pacman:** unknown

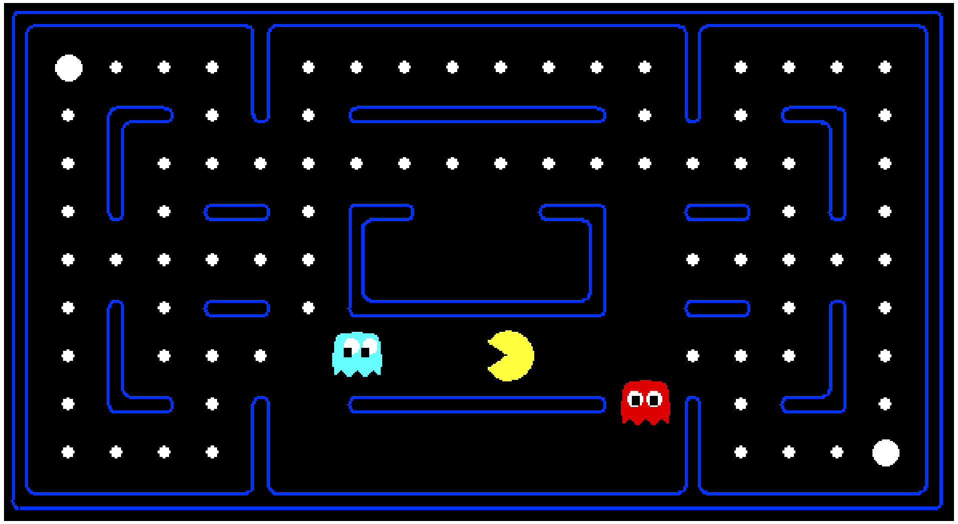
This slide deck courtesy of Dan Klein at UC Berkeley

GamesCrafters



http://gamescrafters.berkeley.edu/

Adversarial Search



[DEMO: mystery pacman]

Game Playing

- Many different kinds of games!

- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?

- Want algorithms for calculating a strategy (policy) which recommends a move in each state

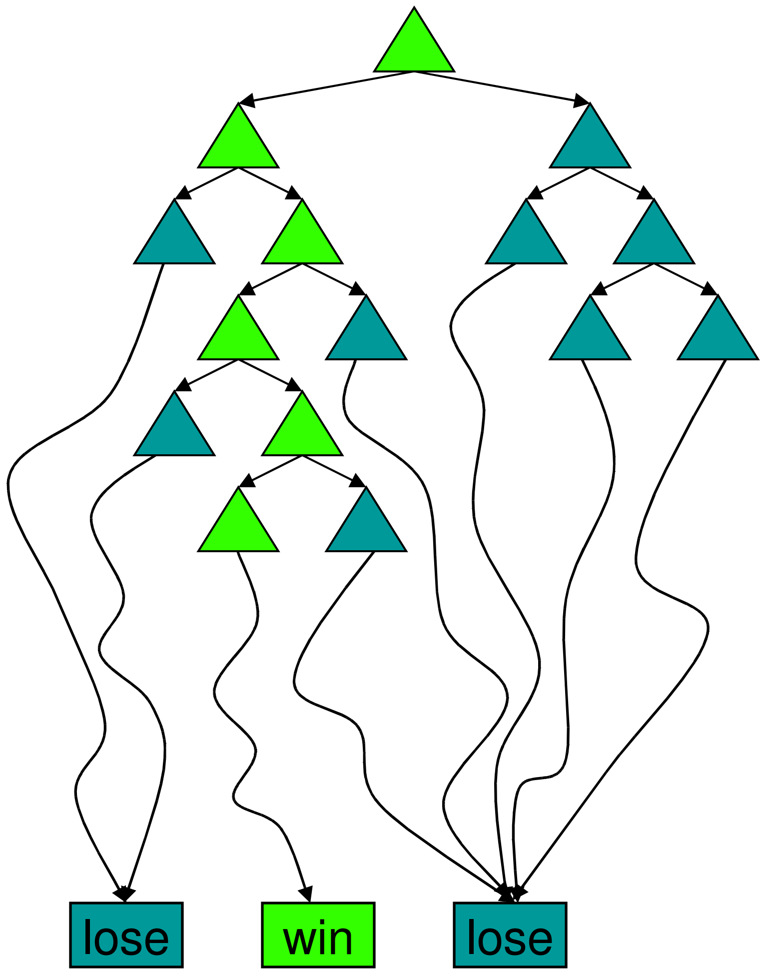
Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: $SxA \rightarrow S$
  - Terminal Test: $S \rightarrow \{t,f\}$
  - Terminal Utilities: $SxP \rightarrow R$

- Solution for a player is a policy: $S \rightarrow A$

Deterministic Single-Player?

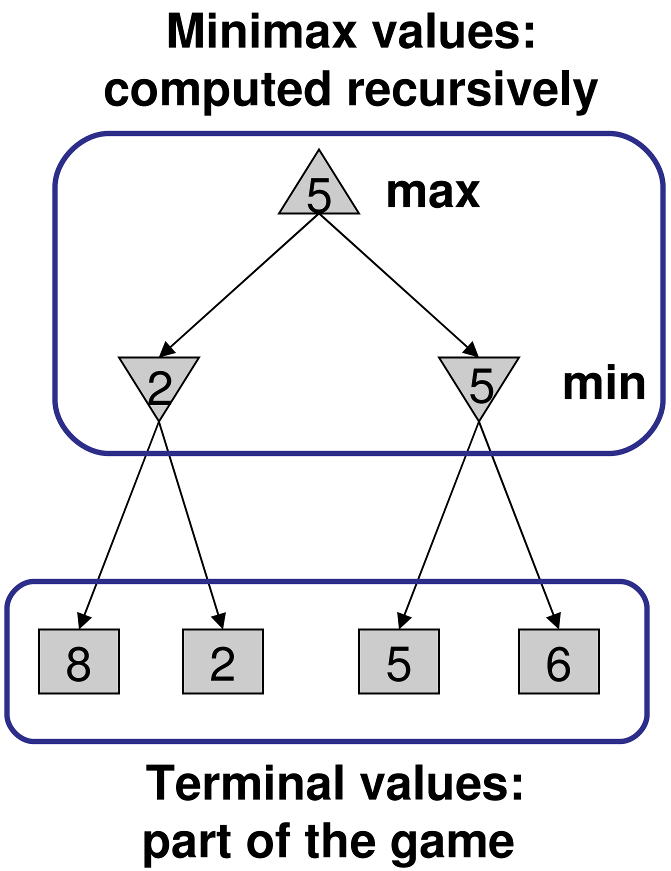
- Deterministic, single player, perfect information:
  - Know the rules
  - Know what actions do
  - Know when you win
  - E.g. Freecell, 8-Puzzle, Rubik's cube
- … it's just search!
- Slight reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children (the max value)
  - Note that we don't have path sums as before (utilities at end)
- After search, can pick move that leads to best node



Adversarial Games

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Each node has a minimax value: best achievable utility against a rational adversary



Computing Minimax Values

- Two recursive functions:
  - max-value maxes the values of successors
  - min-value mins the values of successors

---

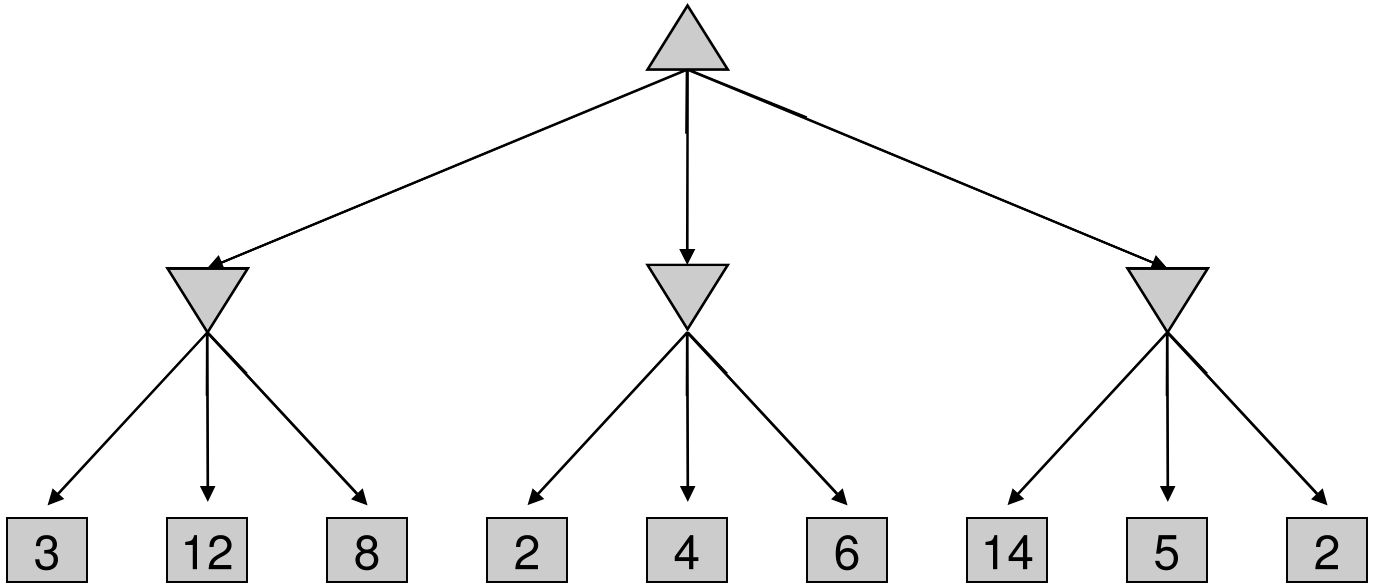
def value(state):

    If the state is a terminal state: return the state's utility
    If the next agent is MAX: return max-value(state)
    If the next agent is MIN: return min-value(state)

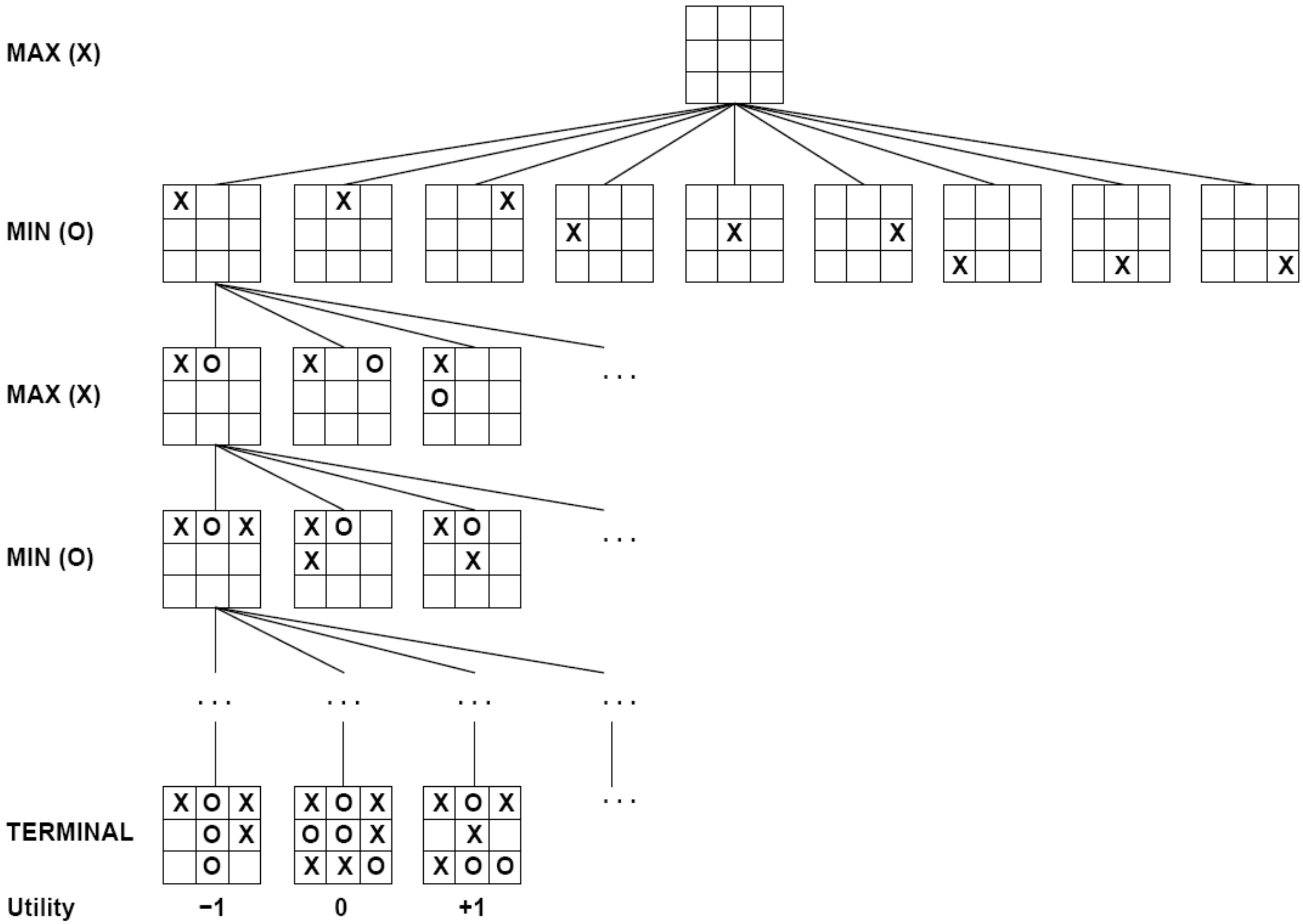
def max-value(state):

    Initialize max = -∞
    For each successor of state:
        Compute value(successor)
        Update max accordingly
    Return max

Minimax Example



Tic-tac-toe Game Tree



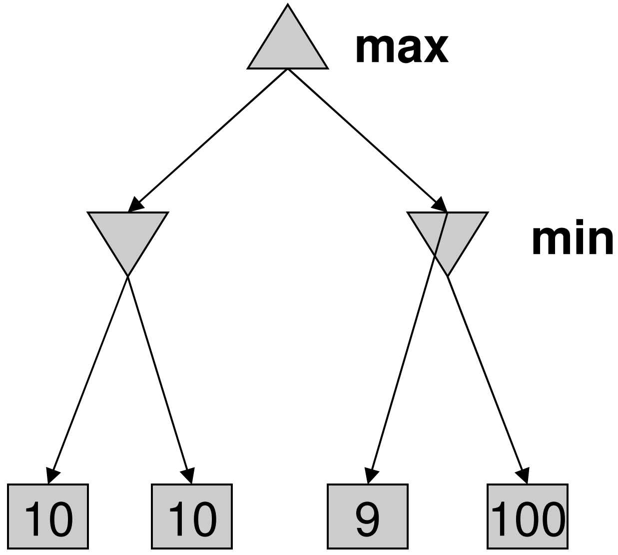
Minimax Properties

- Optimal against a perfect player. Otherwise?

- Time complexity?
  - $O(b^m)$

- Space complexity?
  - $O(bm)$

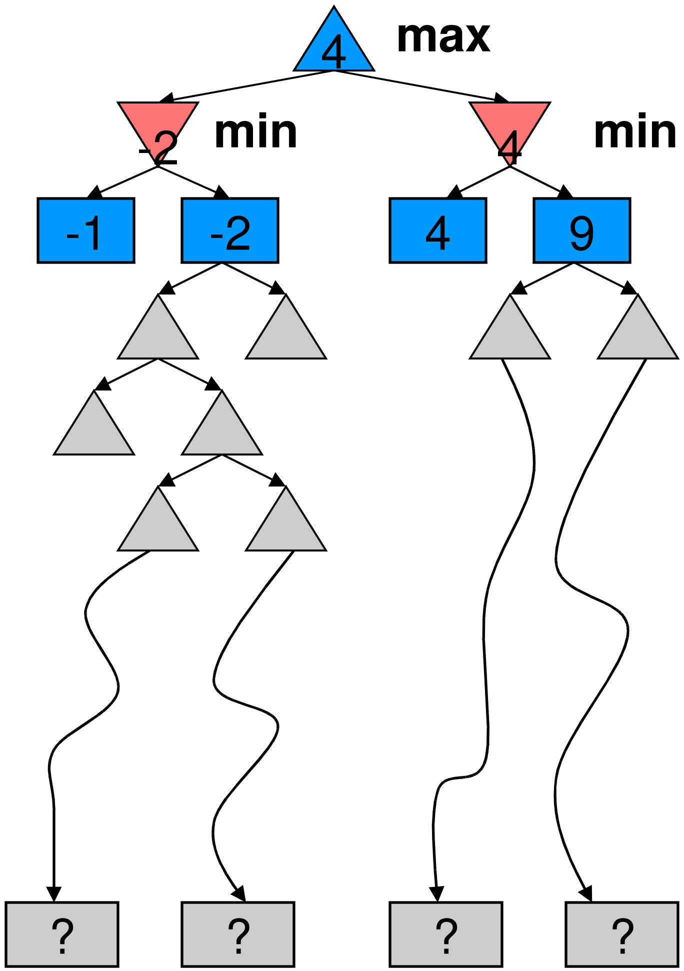
- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



[DEMO: minVsExp n]

Resource Limits

- Cannot search to leaves
- Depth-limited search
  - Instead, search a limited depth of tree
  - Replace terminal utilities with an eval function for non-terminal positions
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α -β reaches about depth 8 – decent chess program

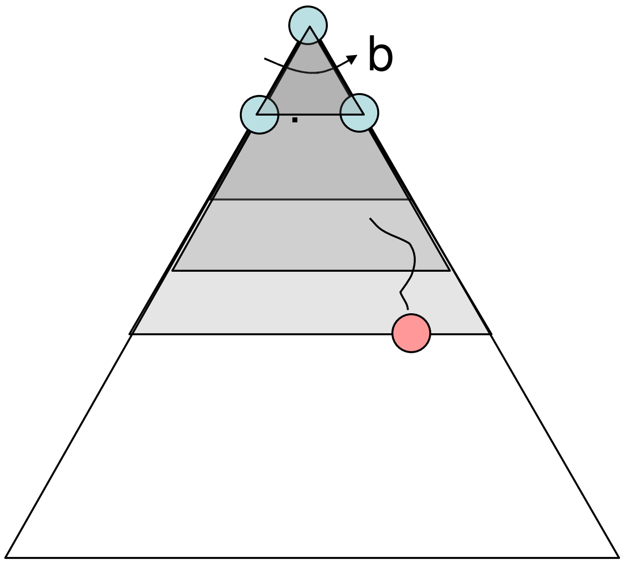


Iterative Deepening

Iterative deepening uses DFS as a subroutine:

3. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
4. If “1” failed, do a DFS which only searches paths of length 2 or less.
5. If “2” failed, do a DFS which only searches paths of length 3 or less.

….and so on.

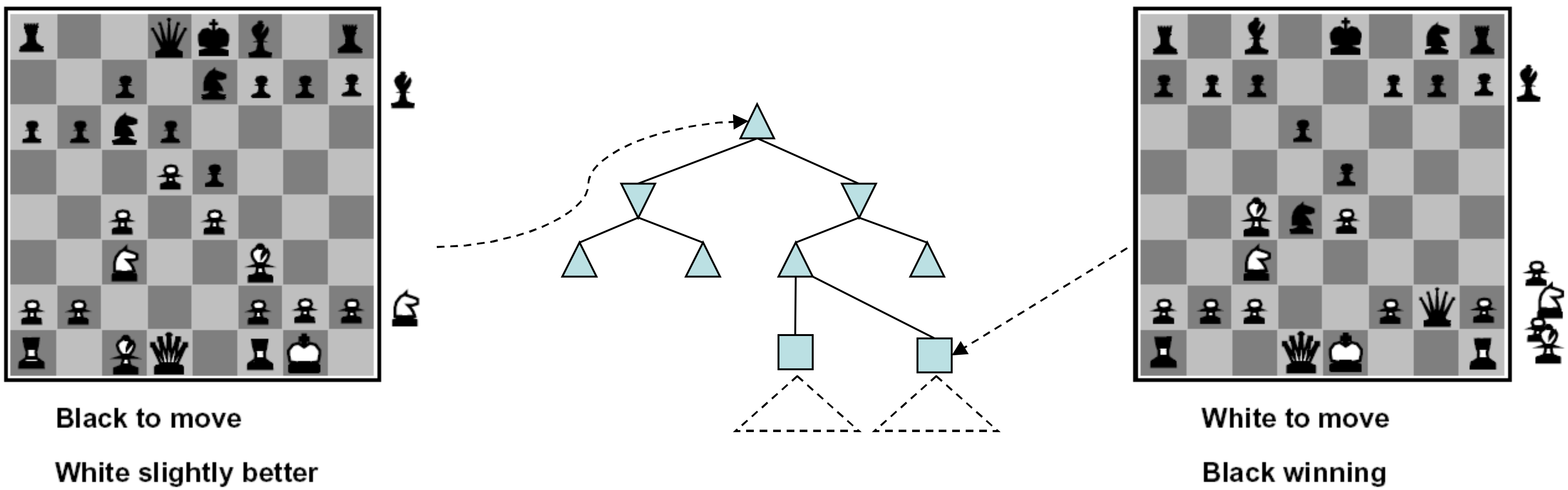


Why do we want to do this for multiplayer games?

Note: wrongness of eval functions matters less and less the deeper the search goes!

Evaluation Functions

- Function which scores non-terminals

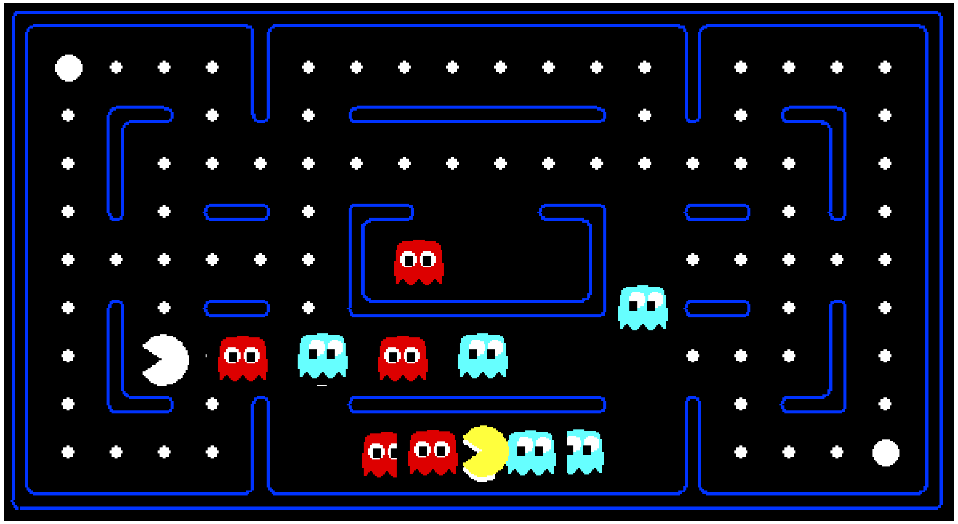


- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

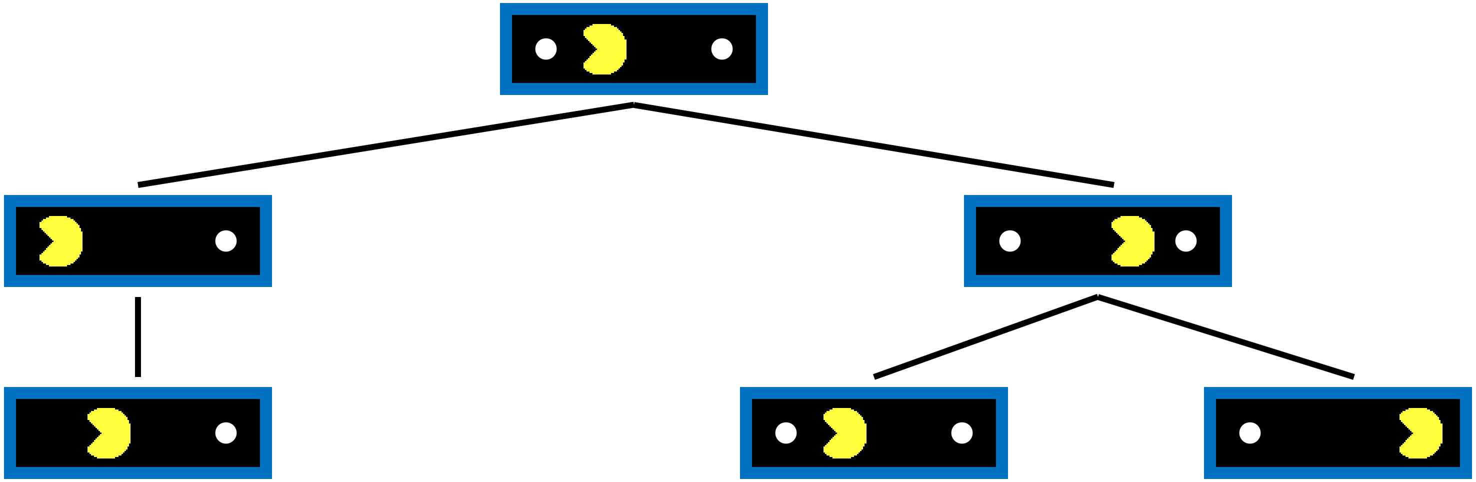
- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

Evaluation for Pacman



$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Why Pacman Starves



- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

Minimax Example