An Update on Game Tree Research

Akihiro Kishimoto and Martin Mueller

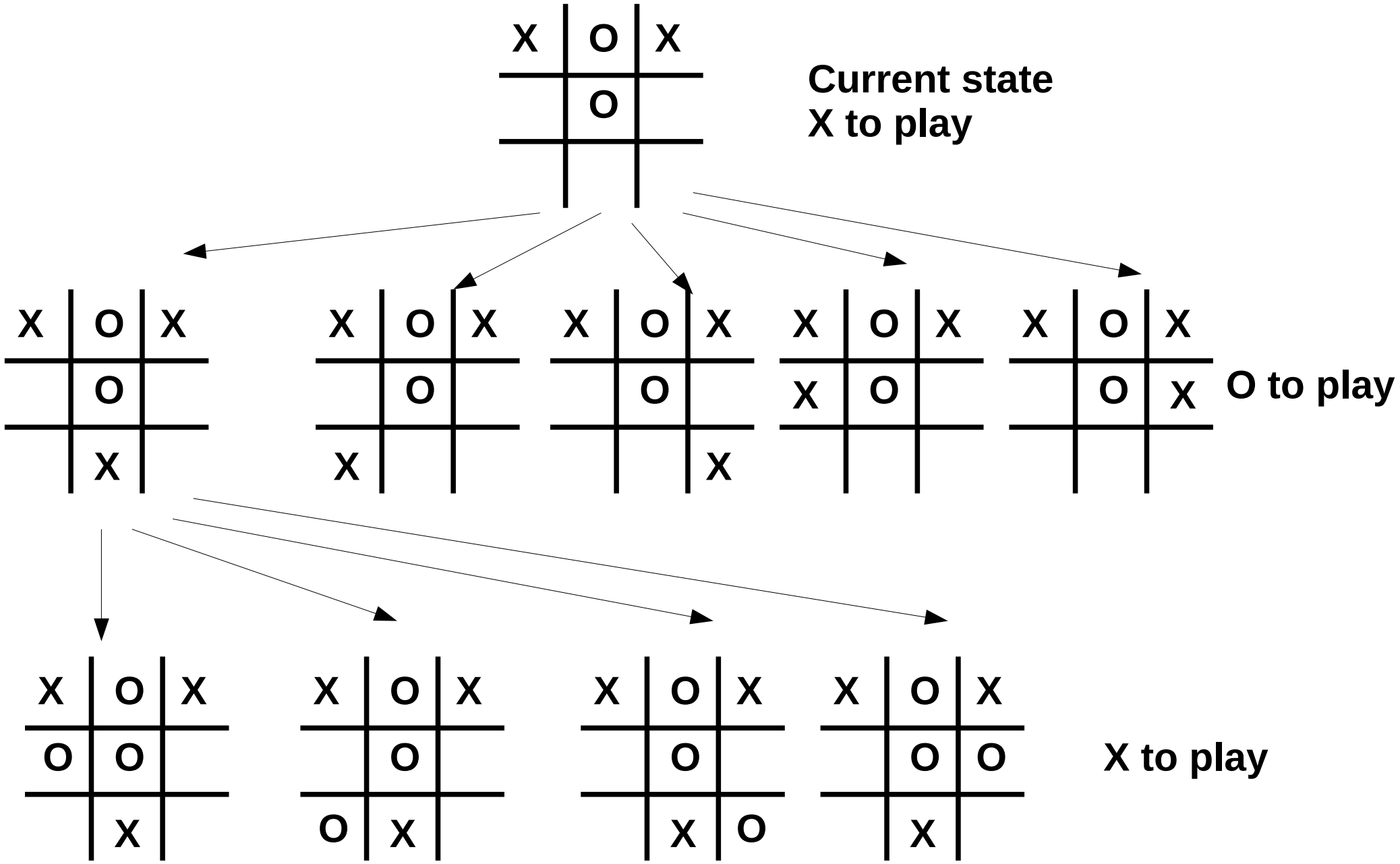
Tutorial 2: Solving and Playing Games

Presenter:
Akihiro Kishimoto, IBM Research - Ireland

Outline of this Talk

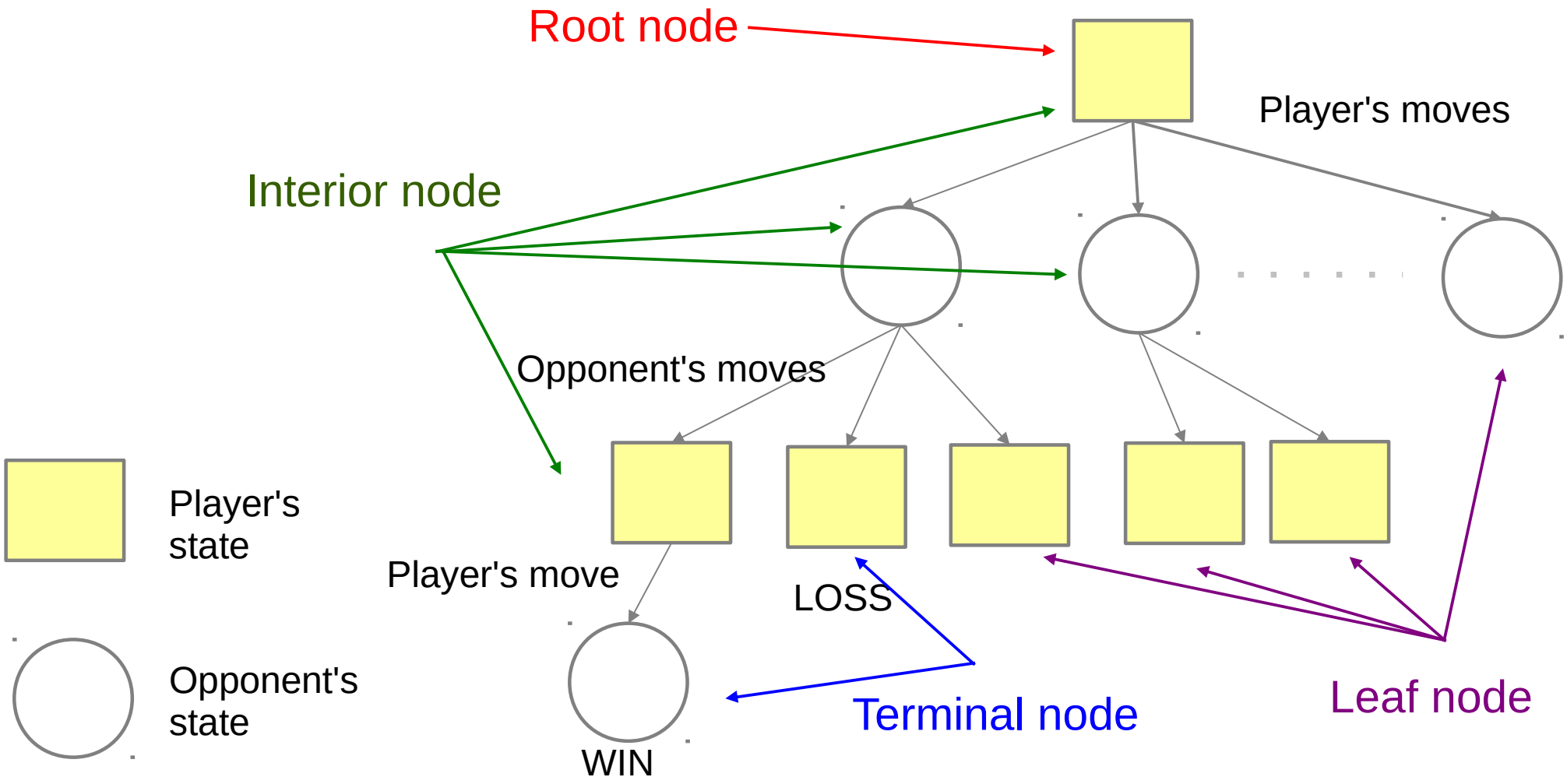
- Defines several notions required to understand detailed game research technologies
    - Minimax search (binary case)
    - AND/OR tree search
    - Minimax/Negamax search (general case)
    - Game-playing in practice

Game Tree Representation (1 / 2)

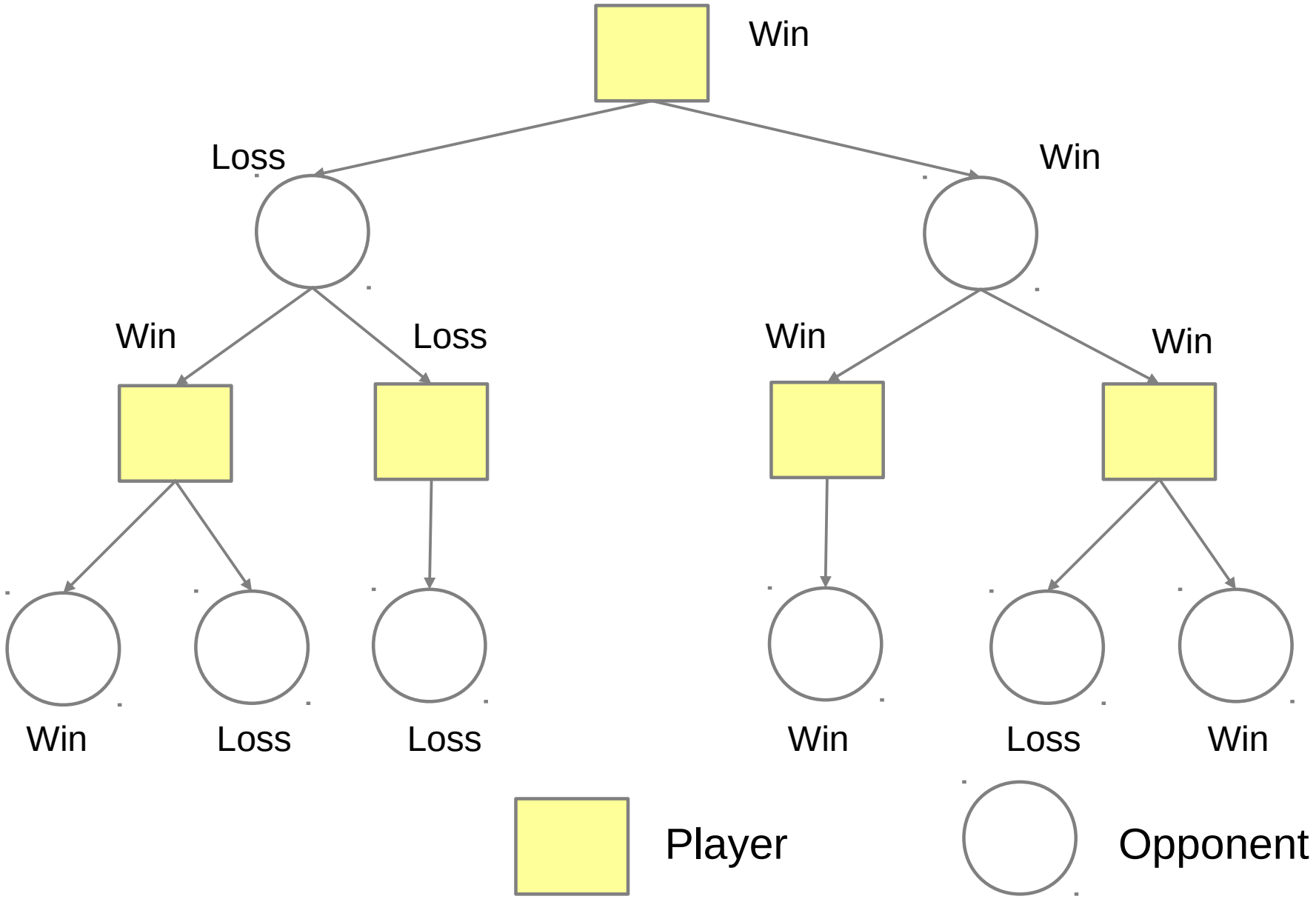


Game Tree Representation (2 / 2)

Perform look-ahead search by building game tree



Simplest Case of Minimax Search Binary Evaluation (1 / 3)



Simplest Case of Minimax Search Binary Evaluation (2 / 3)

- Each player tries to win. Zero-sum – opponent's win is my loss
- OR node (aka MAX node): If I have at least one winning move, I can win (by playing that move)
- If all my moves are losses, I lose.

```
//Basic minimax with Boolean outcome
bool MinimaxBooleanOR(GameState state) {
  if (state.IsTerminal())
    return state.StaticallyEvaluate();
  foreach legal move m of state
    state.Execute(m)
    bool isWin = MinimaxBooleanAND(state);
    state.Undo();
    if (isWin)
     return true;
  return false;
}
```

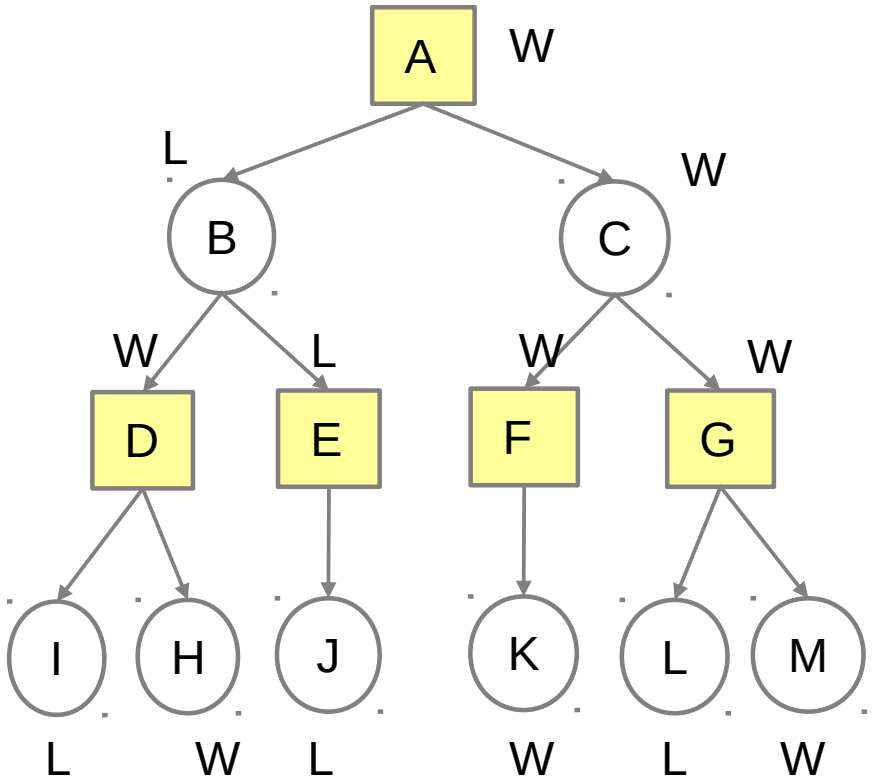
Simplest Case of Minimax Search Binary Evaluation (3 / 3)

- Each player tries to win. Zero-sum – opponent's win is my loss
- AND node (aka MIN node): All moves need to be winning
- If any of my moves are losses, I lose.

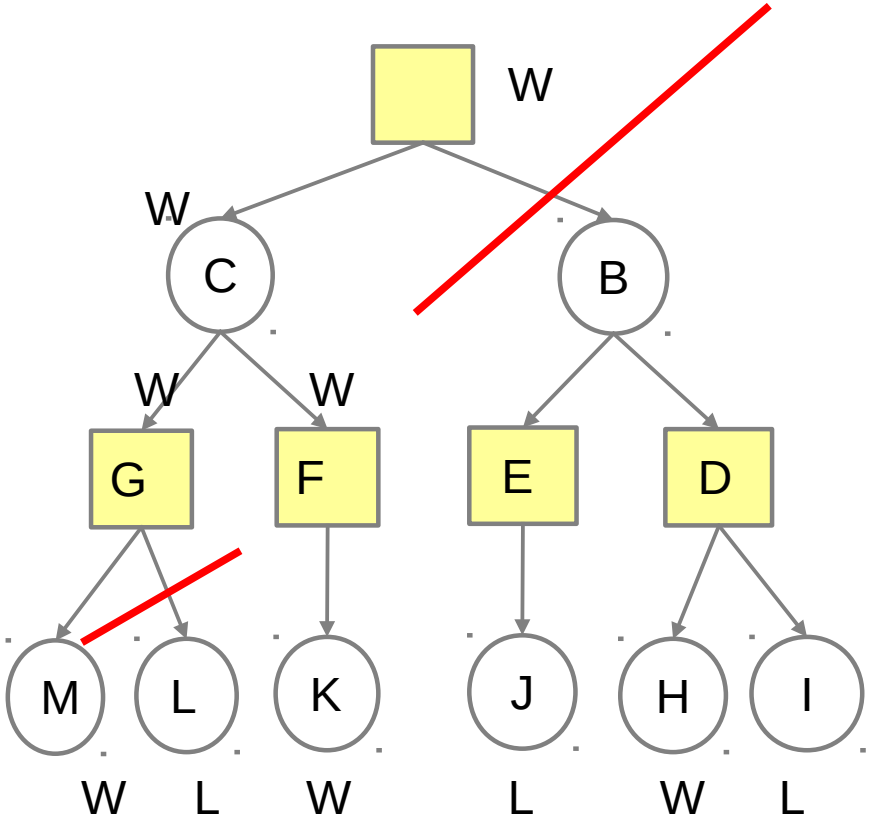
```
//Basic minimax with Boolean outcome
bool MinimaxBooleanAND(GameState state) {
  if (state.IsTerminal())
    return state.StaticallyEvaluate();
  foreach legal move m of state
    state.Execute(m)
    bool isWin = MinimaxBooleanOR(state);
    state.Undo();
    if (not isWin)
     return false;
  return true;
}
```

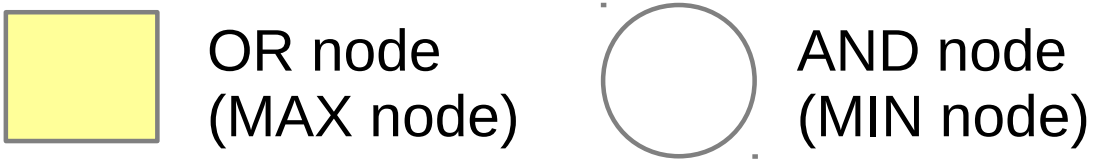
Example
Best and Worst Cases

Worst case



Best case





Boolean Minimax - Efficiency

- Time complexity (number of leaf nodes evaluated)
- Best case: about $b^{d/2}$, first move causes cutoff
- Worst case: about b^{d}, no move causes cutoff
- Space complexity O(*bd*) – depth-first exploration

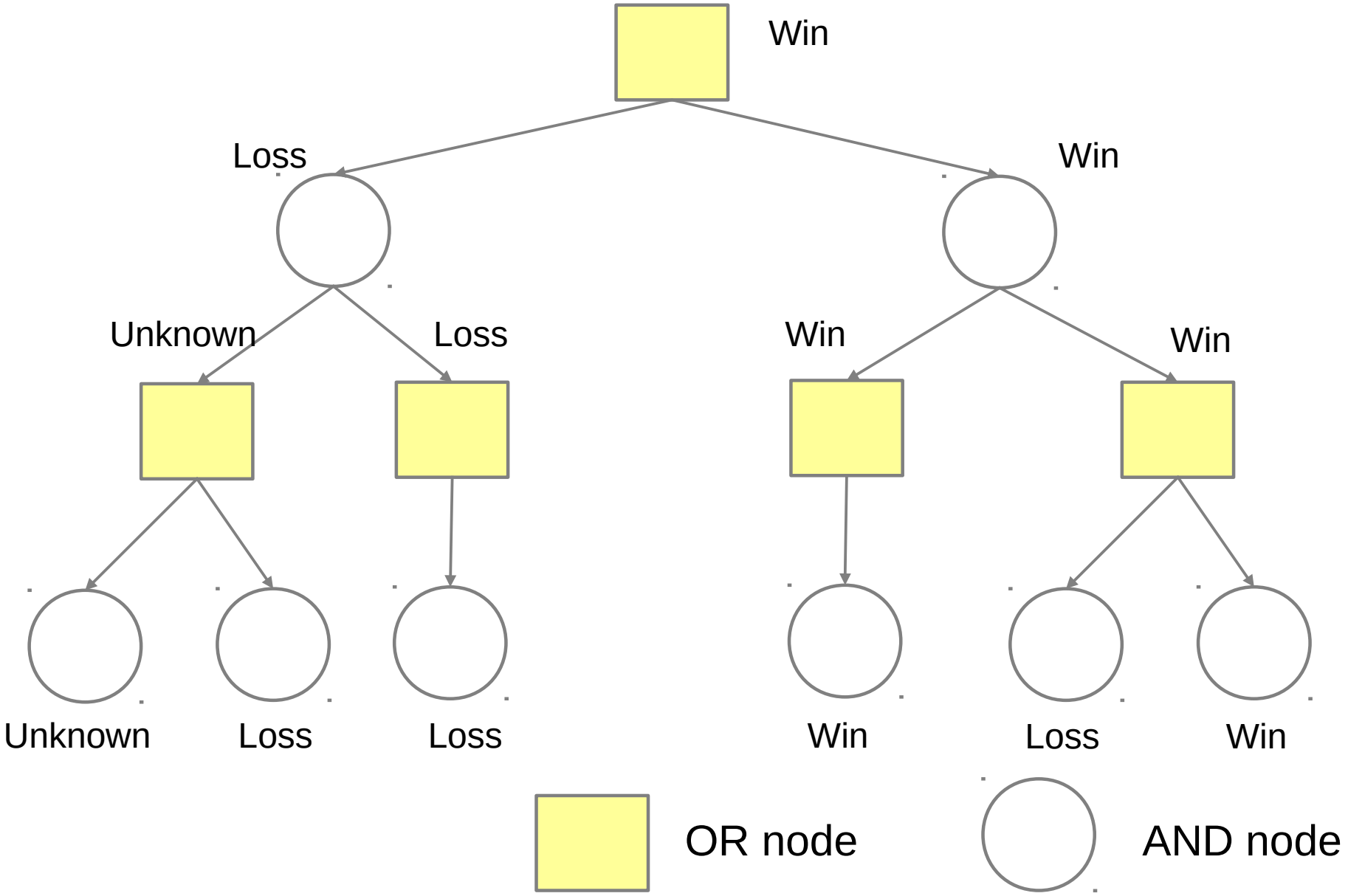
  b: number of available moves (branching factor)

  d: search depth

AND/OR Tree

- Formalizes concept of game tree with alternating players
- OR node: player's turn – can win if move 1 OR move 2 OR … wins
- AND node: opponent's turn – player wins only if opponent's move 1 AND move 2 AND … all win (for player)
- Many applications when goal can be expressed recursively as conjunction/disjunction of subgoals
- Normal form: alternating layers of AND, OR nodes
- Generalization AND/OR DAG or DCG
- Introduce three values, *win, loss, and unknown* in this lecture

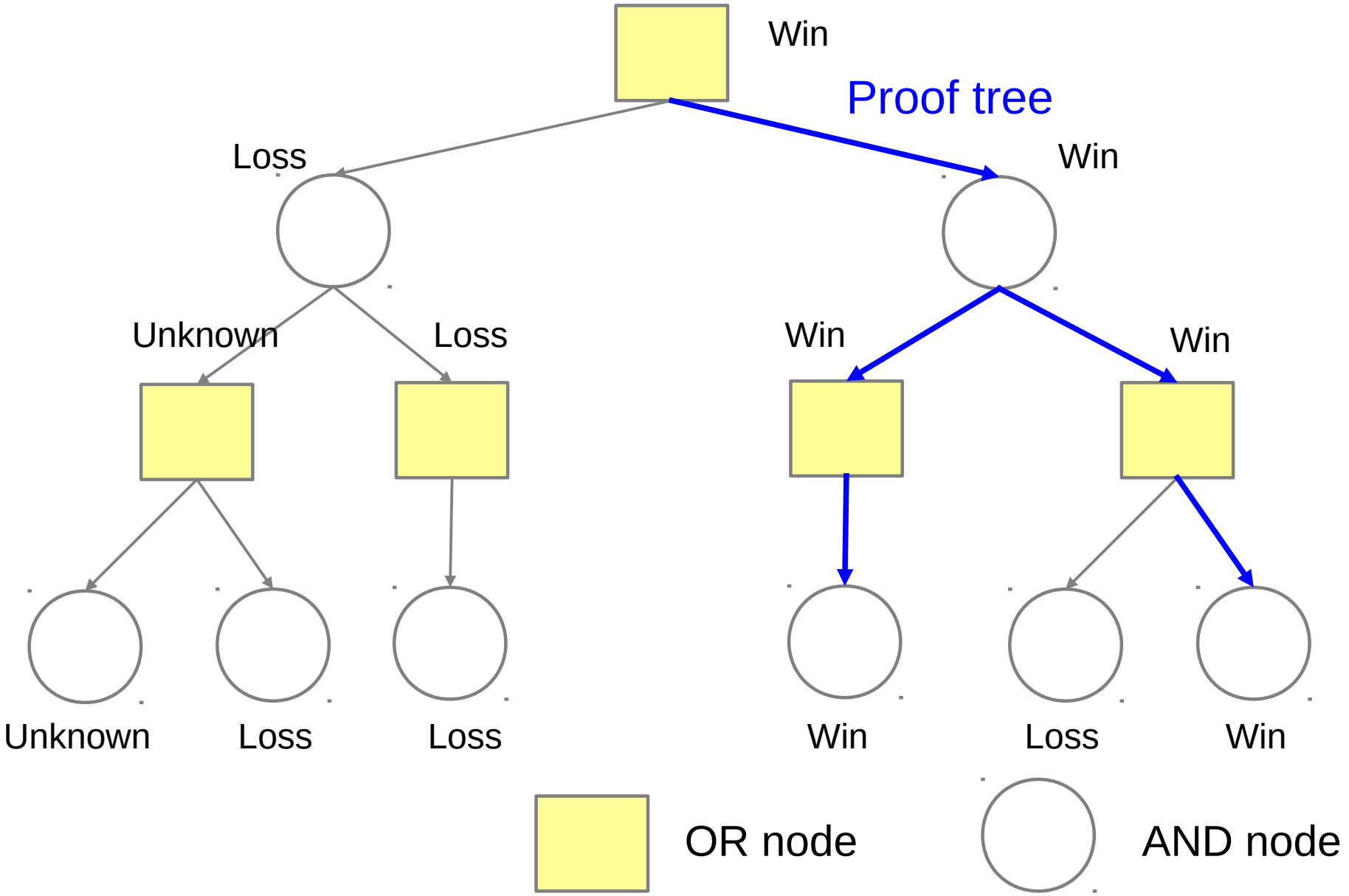
Example of AND/OR Tree



Proof Tree

- A winning strategy for player
- Dual concept: disproof tree – proves we cannot win
- Subset of game tree, covers, all possible opponent replies
- Subtree *P* of game tree *G* is proof tree iff:
    - *P* contains root of *G*
    - All terminal nodes of *P* are wins
    - If interior AND node is in *P*, *all its children* are in *P*
    - If interior OR node is in *P*, *at least one child* is in *P*

Example of Proof Tree



Comments on Proof Tree

- Some definition work on DAG, even arbitrary graph
- There may be more than one proof tree
- Efficiency: want to find minimal or at least small proof tree
- In uniform (*b*,*d*) tree, with OR node at root, number of leaf nodes in best case is 1, 1, *b*, *b*, b^2, b^2,...

  b: branching factor, *d*: depth
- Search is most efficient if it examines only at the proof tree
- In practice, that's impossible. But good move ordering is crucial

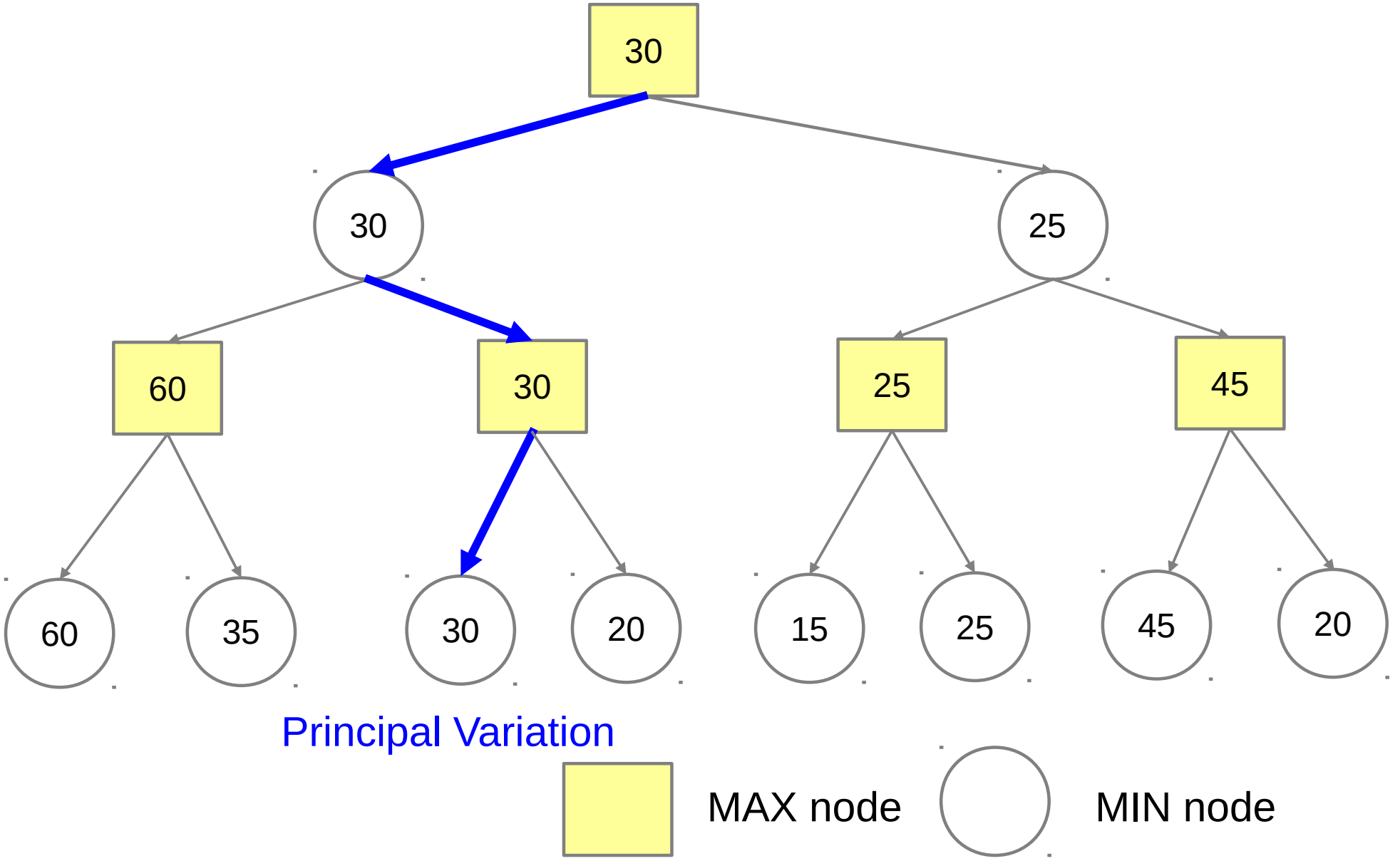
Minimax Search

- General case – score of position can be any finite number
- Frequent special case: small set of values, e.g., win-draw-loss
- We try to maximize the score, opponent tries to minimize it
- Zero-sum: each extra point we win, the opponent loses

Full Search versus Heuristic Search

- Code so far searches until the end of game
- For heuristic play, stop search earlier (e.g., after *N* moves)
- Depth-limited search can be good for move ordering – iterative deepening idea (next lecture)
- Minimax search code with depth-limit
- Can exactly solve positions (when search finds proof tree)
- Evaluate positions at leaf nodes by calling *evaluation function* that approximates a chance of winning
    - Scores are assumed to be integer in this lecture
- *Principal variation* – best-scoring path for both players

Example of Minimax Search



Minimax Search – OR Node (MAX Node)

```
int MinimaxOR(GameState state, int depth) {
  // Evaluate from root player's view
  if (state.IsTerminal() or depth=0)
    return state.StaticallyEvaluate()
  int best = -INF
  foreach legal move m from state
    state.Execute(m)
    int score = MinimaxAND(state, depth-1)
    best = max(best, score)
    state.Undo()
  return best
}
```

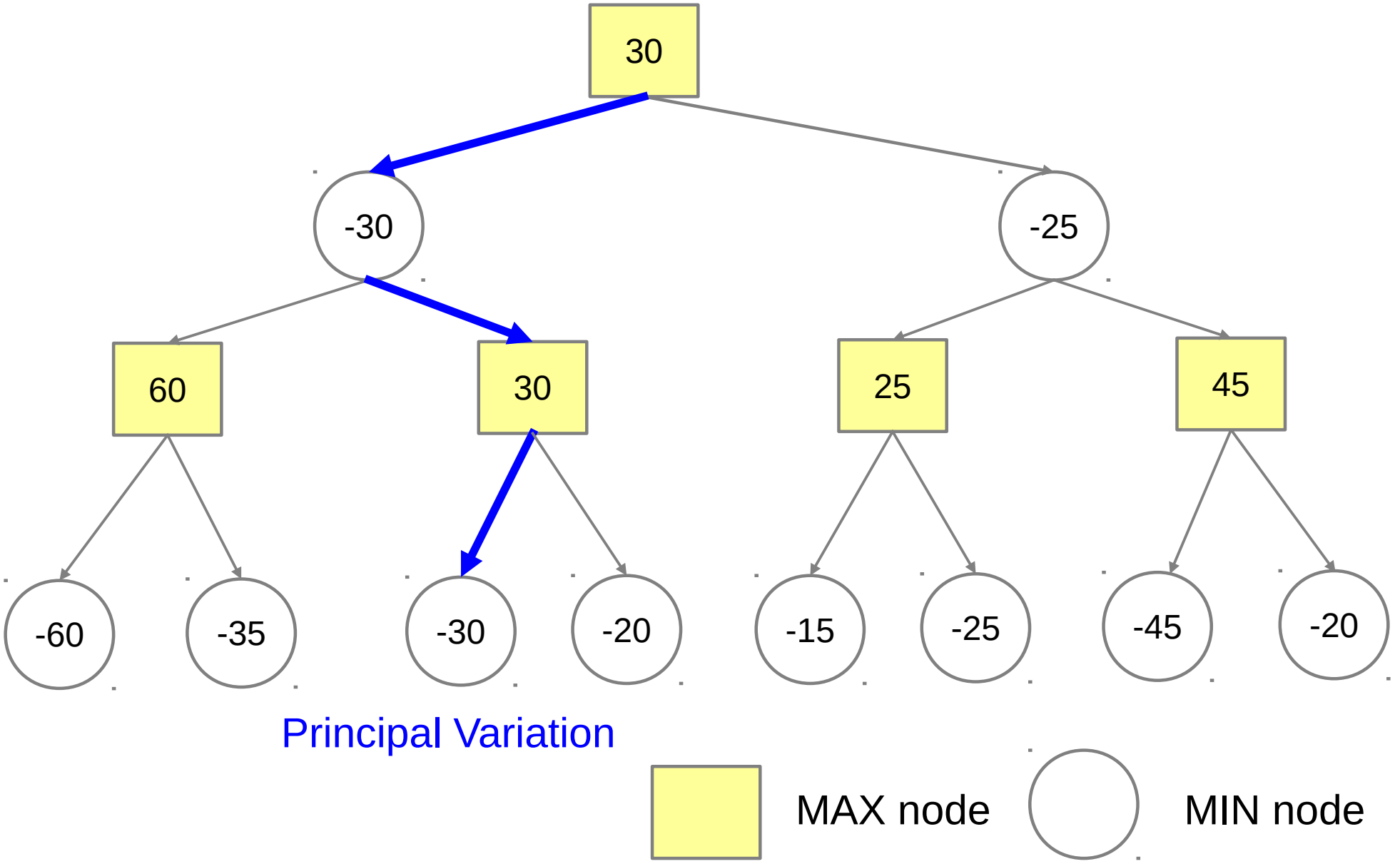
Minimax Search – AND Node (MIN Node)

```
int MinimaxAND(GameState state, int depth) {
  // Evaluate from root player's view
  if (state.IsTerminal() or depth=0)
    return state.StaticallyEvaluate()
  int best = INF
  foreach legal move m from state
    state.Execute(m)
    int score = MinmaxOR(state, depth-1)
    best = min(best, score)
    state.undo()
  return best
}
```

Negamax Search

- Minimax search uses max and min procedures
- Negamax always maximizes the score by negating returned scores from children
- Evaluate states at leaf nodes from *current player's viewpoint*

Example of Negamax Search



Negamax Search – Pseudo Code

```
int Negamax(GameState state, int depth) {
  // Evaluate from current player's view
  if (state.IsTerminal() or depth=0)
    return state.StaticallyEvaluate()
  int best = -INF
  foreach legal move m from state
    state.Execute(m)
    int score = - Negamax(state,depth-1)
    best = max(best, score)
    state.Undo()
  return best
}
```

Comments on Plain Minimax/Negamax

- Inefficient. No pruning as opposed to Boolean case above. In (b,d) tree, searches all b^d paths
- How can we add pruning? (next lecture)
- How to set a proper depth to search (next lecture)
- Simple idea: prune if max. value reached (usually does not help much)

Game-Playing Program in Practice

- Incorporates several approaches such as
  - Opening book
  - Search engine, e.g., alpha-beta (next lecture) and MCTS (afternoon)
  - Endgame database
  - Specialized search

Opening Book

- Databases that collect positions and moves particularly in the beginning of games
- Collected from human experts' game records if available e.g. chess, checkers, shogi, Go
- If position to query is stored in opening book, play stored move immediately at that position
- If more than one move is available, select one randomly
- Can provide a high-quality move and non-deterministic behavior of game-playing program, and save time
- Blunder moves must be filtered out when book is constructed

Endgame Database

- In some games, all positions in endgame can be enumerated by single/parallel computing resources
    - e.g., positions with <= 6 pieces in chess and with <= 10 pieces in checkers
    - Precompute win-draw-loss values of these positions and save them in database
- Perform *retrograde analysis* that backs up scores from terminal positions to build database
- Perfect evaluation can be achieved by fast database lookup
- Paging-based approach is used if database does not fit into memory

Specialized Search

- In some game-playing systems, specialized search is incorporated to efficiently check if sub-goal can be achieved
    - E.g., tactical search in Go and check-mate (tsume-shogi) search in shogi
- Main search invokes such specialized search by limited time/node expansions
- Specialized search has much higher overhead than endgame database lookups
- When specialized search is invoked must be carefully considered (see next lecture in case of shogi)

Summary

- Explained basic notions required to understand remaining material
    - AND/OR tree search and proof tree
    - Minimax/Negamax search
    - Game-playing program in practice e.g. opening book, endgame database, specialized search