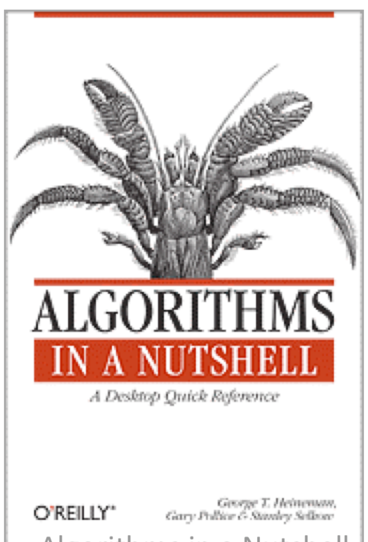
Algorithms in a Nutshell



Session 7

Path Finding in AI

1:50 – 2:40

Outline

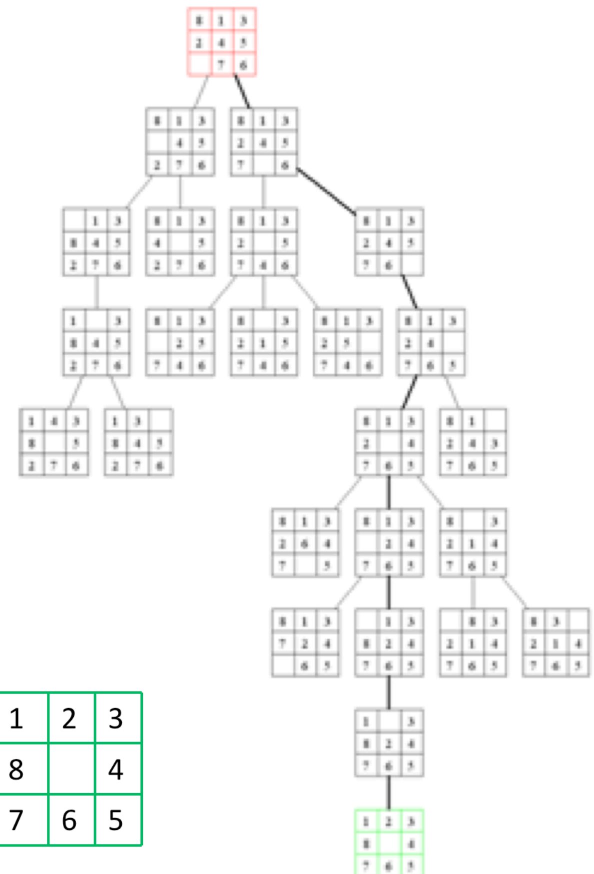
- Overview
- Search Trees
  - BREADTHFIRSTSEARCH, DEPTHFIRSTSEARCH
  - ASTAR (A^*)
- Game Trees
  - MINIMAX, ALPHABETA
- Themes
  - Blind search vs. Heuristics

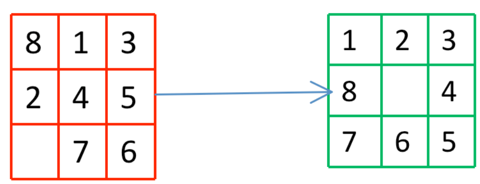
Path Finding in AI

- How to solve a problem when no clear solution exists?
  - Artificial Intelligence (AI)
  - Convert problem into a search
- Two common problem classes
  - One player game making moves
  - Two player games with alternating moves

Search Strategy

- Single player making moves
  - Start from *initial* board state
  - Target represents *goal state*
  - *Moves* transform board state
- Goal
  - Find sequence of moves from *initial* state to *goal* state
- Example 8-puzzle

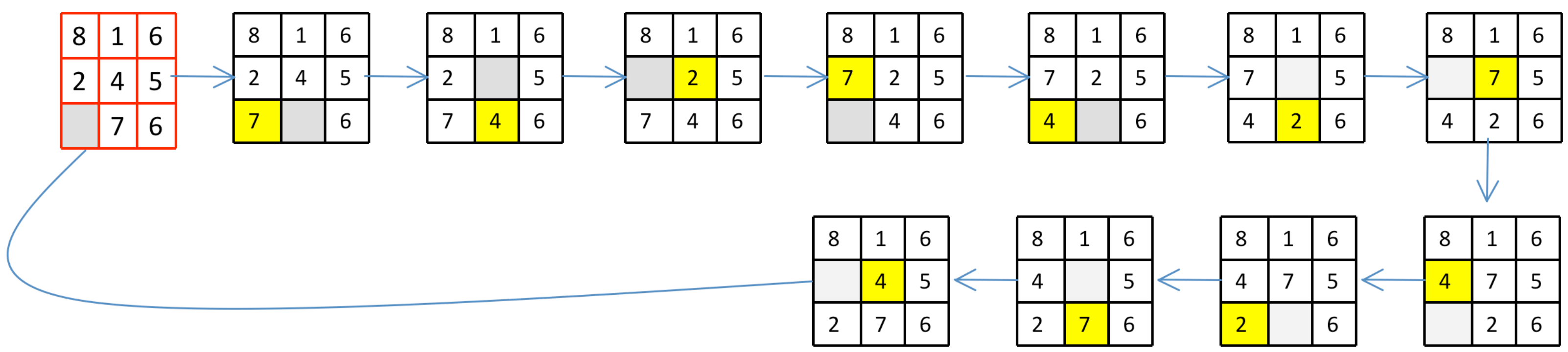




initial *goal*

Search Space

- Graph-like search space is possible
  - Must prevent infinite cycles as search proceeds
- Size of search space can be **Very Large**
  - 8-puzzle only has 362,880 unique states
  - Rubik's cube has 43,252,003,274,489,856,000



Search Tree Approach

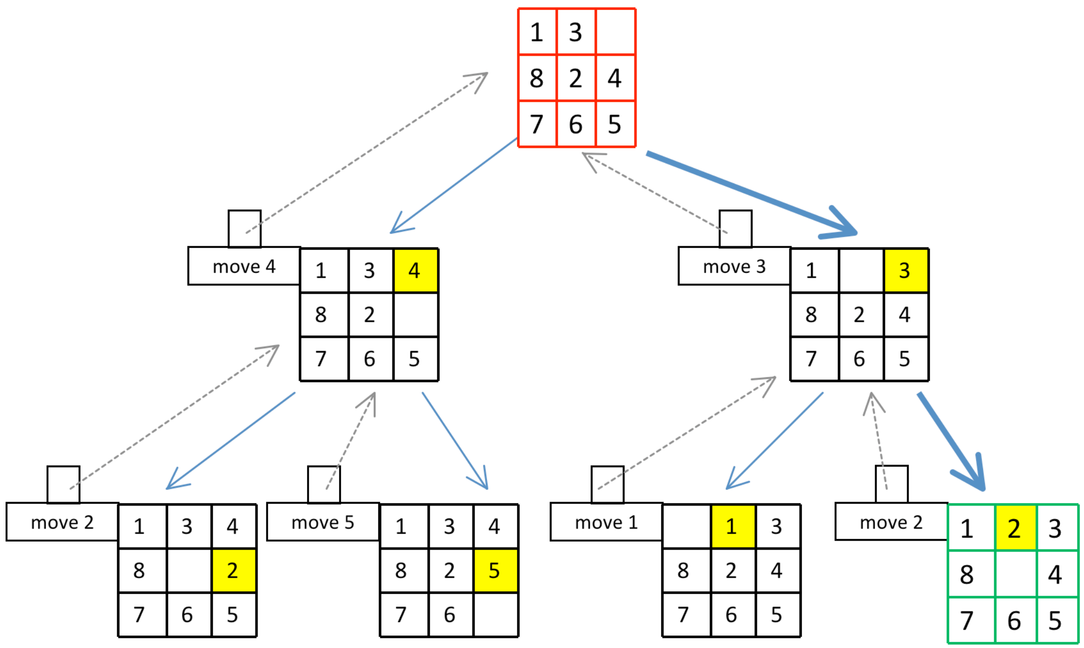
- Determine board state representation
- Identify valid moves given a board state
- Construct graph for search space?
  – Can't. In general, it's just too big
- Construct Search Tree
  – Nodes are valid board states
  – Edge between nodes represents the application of a move

Key decisions

- Must avoid revisiting prior board states
  - Maintain *closed* set of visited board states
  - Discard moves that result in a state that has already been visited
- Must be able to reproduce sequence of moves from initial board state
  - Each board state stores link to its prior state

8-Puzzle Search Tree Example

- Start at initial board state
- Levels represent number of moves from initial state
- Light blue arrows represent valid moves that create new board state
  - Solid blue represent winning solution
- Dashed grey lines represent *back links* for history

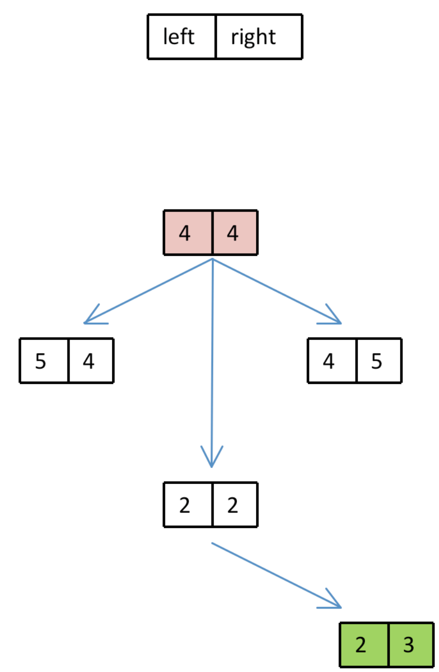


Strategies for searching Search Tree

- Recall DEPTHFIRSTSEARCH and BREADTHFIRSTSEARCH
  - Both still applicable **even though** graph never fully constructed
  - We explore graph and expand nodes as necessary
- Seek shortest sequence of moves to solution?
  - BREADTHFIRSTSEARCH
- Willing to try moves randomly until solution?
  - DEPTHFIRSTSEARCH
  - But must fix maximum depth bound to stop search because graph can be <u>very large</u>

Context: Small Puzzle

- Board state consist of two integers
  - Start from an initial board
- Three possible moves
  - Add one to left number
  - Add one to right number
  - If BOTH even, divide both in half
- Goal state is predefined (*m*,*n*)
  - *m* and *n* are ≥ 1

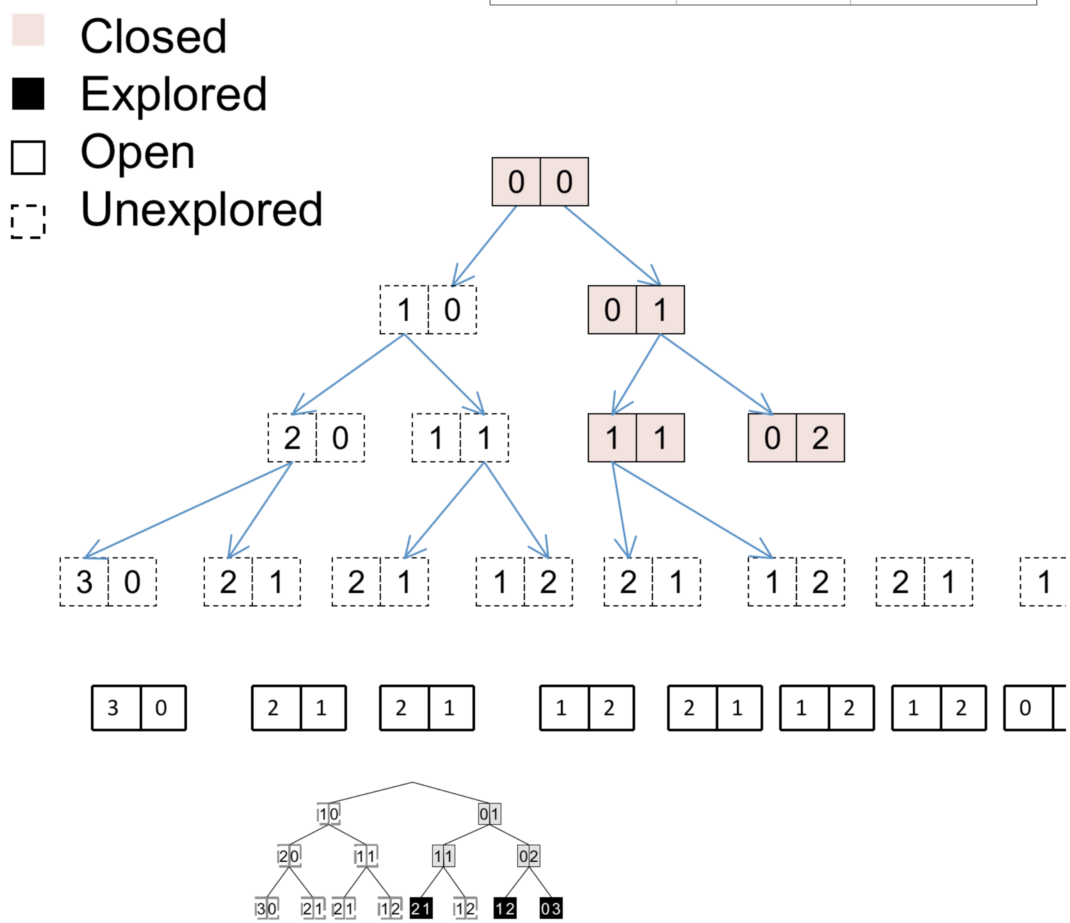


182

DEPTHFIRSTSEARCH

```
search (initial, goal)
if (initial = goal) then return "Solution"
initial.depth = 0
open = new Stack
closed = new Set
insert (open, copy (initial))
while (open is not empty)
  n = pop (open)
  insert (closed, n)
  foreach valid move m at n do
    next = state when playing m at n
    if (closed doesn't contain next) then
      next.depth++
      if (next = goal) then return "Solution"
      if (next.depth < maxDepth) then
        insert (open, next)
return "No Solution"
end
```

Best case	Average case	Worst case
O(b*d)	$O(b^d)$	$O(b^d)$



DEPTHFIRSTSEARCH

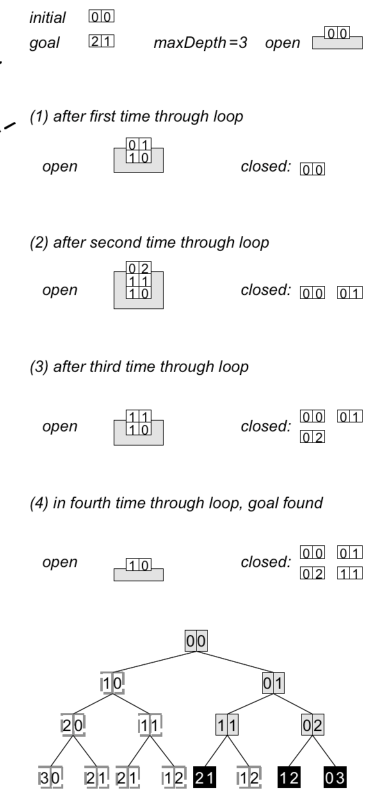
- *d* defines how deep to search
  - Stop searching past this depth
- *b* defines average number of available moves per board state
- *open* is stack representing state of search
  - Backtrack by "popping" board states off stack
- *closed* is set of states that have been visited
  - Must support lookup

DEPTHFIRSTSEARCH

Best case	Average case	Worst case
O(b*d)	$O(b^d)$	$O(b^d)$

Stack Backtracking Set

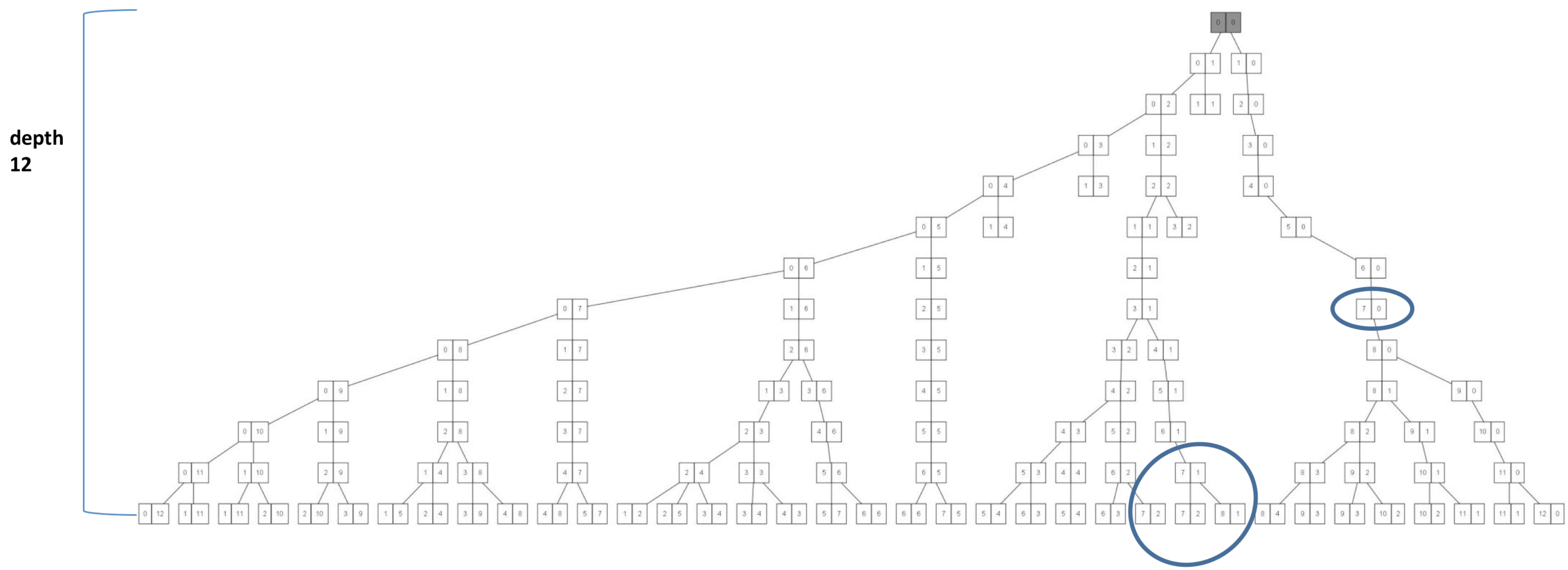
```
search (initial, goal)
  if (initial = goal) then return "Solution"
  initial.depth = 0
  open = new Stack
  closed = new Set
  insert (open, copy(initial))
  while (open is not empty)
    n = pop (open)
    insert (closed, n)
    foreach valid move m at n do
      next = state when playing m at n
      if (closed doesn't contain next) then
        next.depth = n.depth+1
        if (next = goal) then return "Solution"
        if (next.depth < maxDepth) then
          insert (open, next)
  return "No Solution"
end
```



188

Small DEPTHFIRSTSEARCH Example

- Goal node: [7|3]
  - – Why does DFS with depth-bound 12 fail?



Algorithm Detail

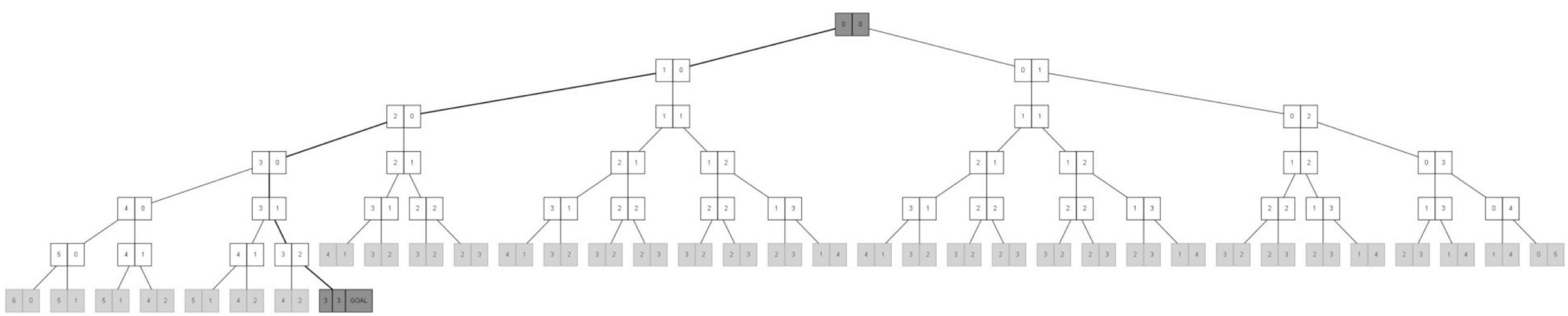
- Why are duplicate states visited?
  - *open* is a stack and the same state may be inserted

```
while (!open.isEmpty()) {
  INode n = open.remove();
  closed.insert(n);

  // All successor moves translate into appended OPEN states.
  DoubleLinkedList<IMove> moves = n.validMoves();
  for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
    IMove move = it.next();

    // Execute move on a copy since we maintain sets of board states
    INode successor = n.copy();
    move.execute(successor);

    // If already visited, try another state
    if (closed.contains(successor) != null) { continue;  }
    if (successor.equals(goal)) { return new Solution (initial, successor); }

    if (depth < depthBound) { open.insert (successor); }
  }
}

// No solution.
return new Solution (initial, goal, false);
```

Some code deleted to make this easier to read.

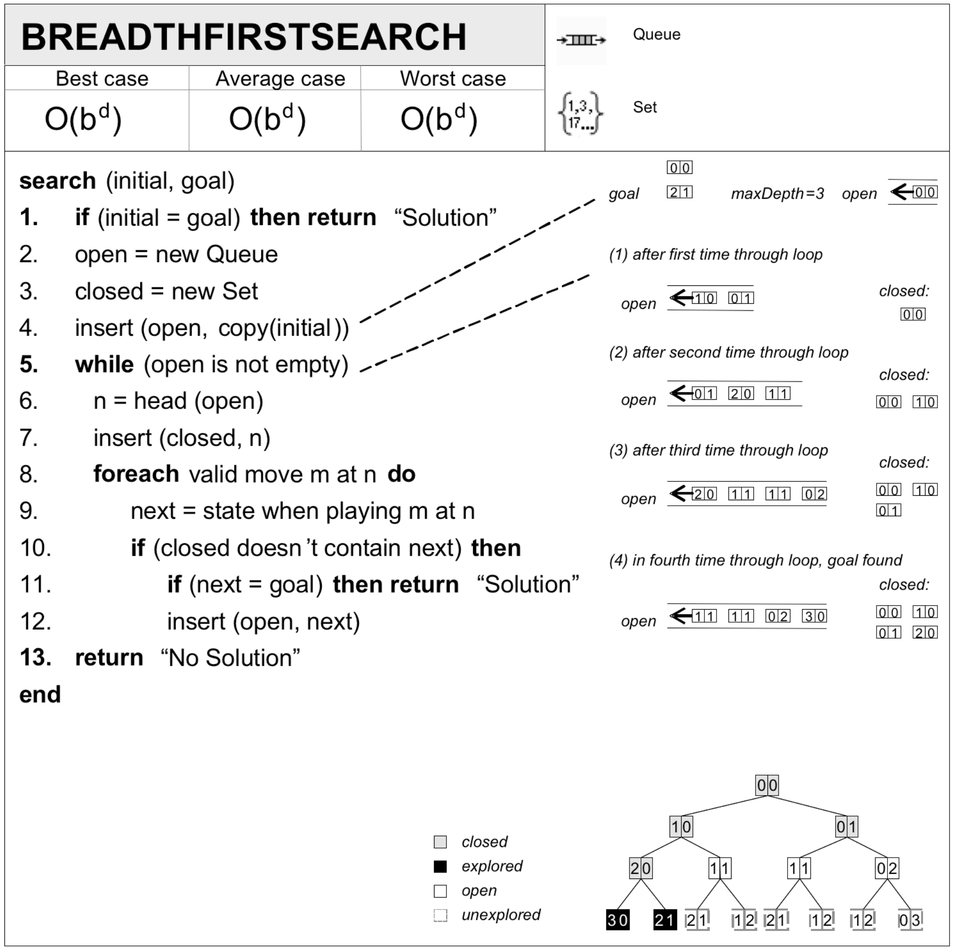
Small BREADTHFIRSTSEARCH Example

- Systematic exploration of search tree
  - All possible states *k* moves away are visited before states *k+1* moves away
  - Note distinct structure of search tree



BREADTHFIRSTSEARCH

- *d* defines depth of search
  - However, search continues until solution found or run out of memory
- *b* defines average number of available moves per board state
- *open* is queue representing state of search
  - Ensures all board states *k* moves away are visited before those *k+1* away
- *closed* is set of states that have been visited
  - Must support lookup



BREADTHFIRSTSEARCH

Best case	Average case	Worst case
$O(b^d)$	$O(b^d)$	$O(b^d)$

```
search (initial, goal)
1.  if (initial = goal) then return "Solution"
2.  open = new Queue
3.  closed = new Set
4.  insert (open, copy(initial))
5.  while (open is not empty)
6.    n = head (open)
7.    insert (closed, n)
8.    foreach valid move m at n do
9.      next = state when playing m at n
10.     if (closed doesn't contain next) then
11.       if (next = goal) then return "Solution"
12.       insert (open, next)
13. return "No Solution"
end
```

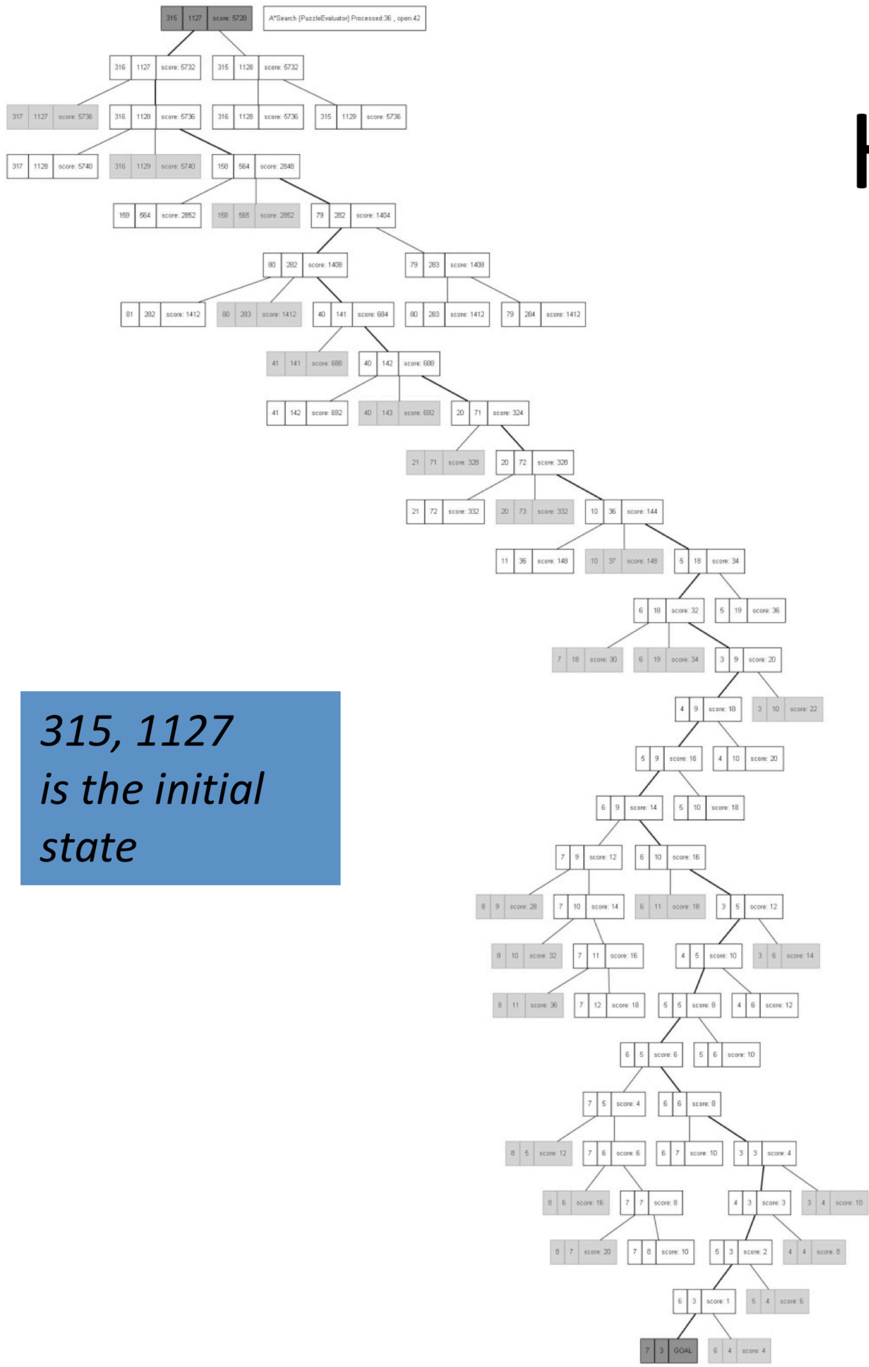
Heuristics to the rescue

- BREADTHFIRSTSEARCH will find solution should it exist
  - But it may require an incredible amount of resources
- DEPTHFIRSTSEARCH may find solution
  - But only if bound is properly set
- Both methods are “blind” searches
  - How do we add knowledge to the search?
  - Through heuristics
- Heuristics
  - Methods that helps solve a problem; rules of thumb; common sense ideas

Sample heuristic for Small Puzzle

- Produce value that estimates number of moves to reach solution

```
public int eval(INode state) {
  SmallPuzzle tp = (SmallPuzzle) state;

  // manhattan distance to target[]
  int diff = Math.abs(target[0] - tp.s[0]) + Math.abs(target[1] - tp.s[1]);

  // if we have gone too far, then DOUBLE the cost, since we have to cut in half
  // and then add back. Same for both
  if (tp.s[0] > target[0]) { diff *= 2; }
  if (tp.s[1] > target[1]) { diff *= 2; }

  return diff;
}
```



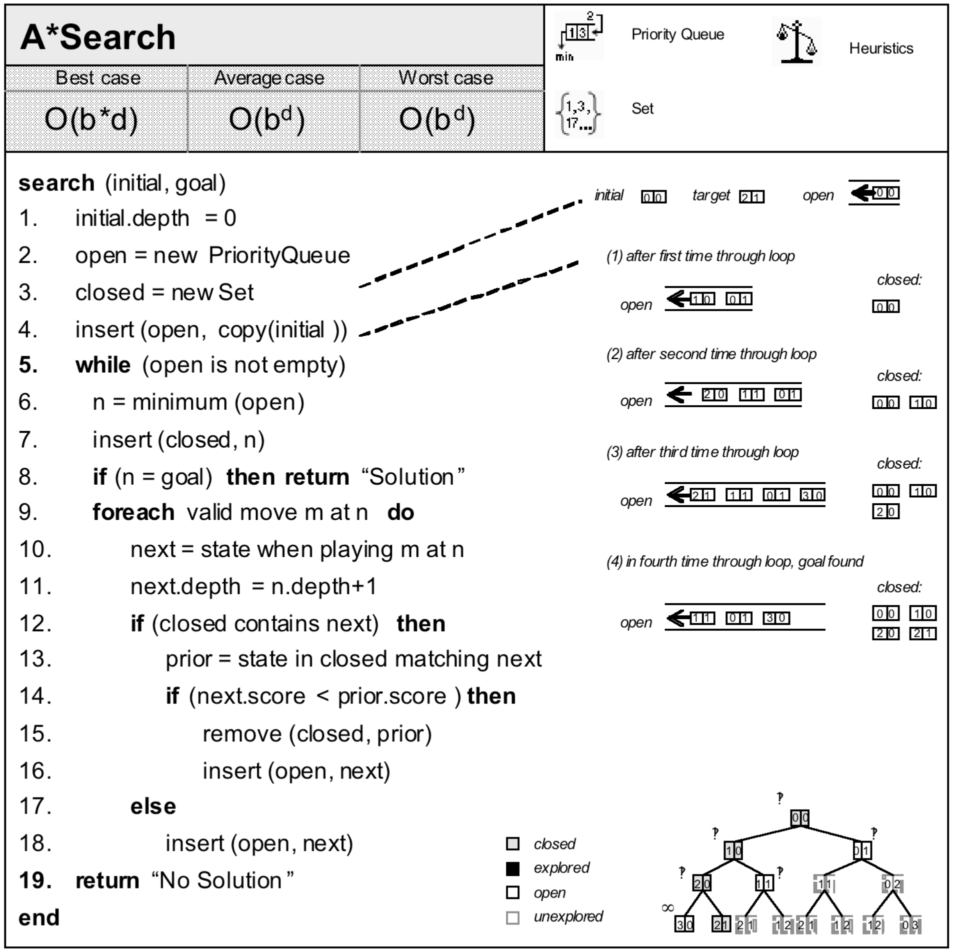
315, 1127
is the initial
state

Heuristic at work

- Handles dead ends
  - Pursues more profitable sequences
  - Judged by an evaluator that rates each board state
  - Gray board states are still “open” and available if no better ones are found
- Search power determined entirely by utility of heuristic

ASTAR (A*) Search

- *d* defines depth of search
- *b* defines average number of available moves per board state
- *open* is priority queue representing state of search
  - Extract board state with best evaluation at each iteration
- *closed* is set of states that have been visited
  - A* may remove states from *closed* if their evaluation score is better



ASTAR (A*) Detail

```
while (!open.isEmpty()) {
    // Remove node with smallest evaluation function and mark closed.
    INode n = open.remove();
    closed.insert(n);

    // Return if goal state reached.
    if (n.equals(goal)) { return new Solution (initial, n); }

    DoubleLinkedList<IMove> moves = n.validMoves();
    for (Iterator<IMove> it = moves.iterator(); it.hasNext(); ) {
      IMove move = it.next();

      // Make move and score the new board state.
      INode successor = n.copy();
      move.execute(successor);

      // Compute evaluation function to see if we have improved upon already-closed state
      scoringFunction.score(successor);

      // If already visited, see if we are revisiting with lower cost; if so, pull from closed
      INode past = closed.contains(successor);
      if (past != null) {
        if (successor.score() >= past.score()) { continue; }
        closed.remove(past);
      }

      open.insert (successor); // place into open.
    }
  }

  // No solution.
  return new Solution (initial, goal, false);
}
```

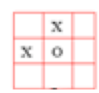
*Only check against solution when removing from **open***

*Only searching method we describe that removes elements from **closed***

Game Tree

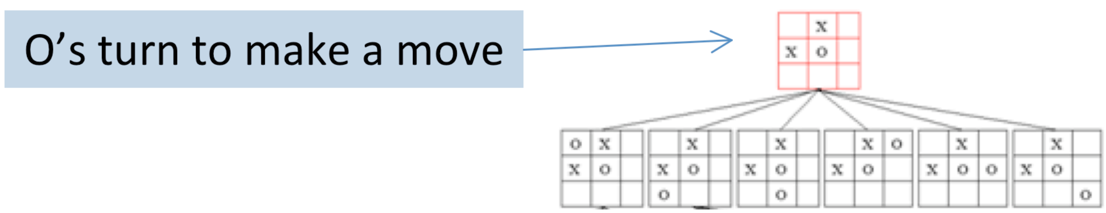
- Represents two player games
  - Players alternate turns
  - Start from initial game state
  - Many states in which either player can win
  - Draws are possible (neither player wins)

O’s turn to make a move



Game Tree

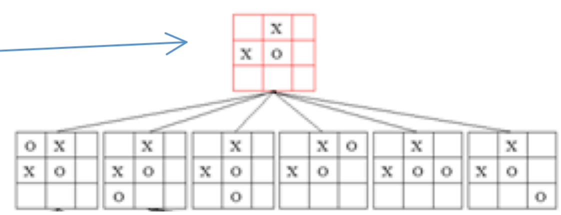
- Represents two player games
  - Players alternate turns
  - Start from initial game state
  - Many states in which either player can win
  - Draws are possible (neither player wins)



Game Tree

- Represents two player games
  - Players alternate turns
  - Start from initial game state
  - Many states in which either player can win
  - Draws are possible (neither player wins)

O's turn to make a move



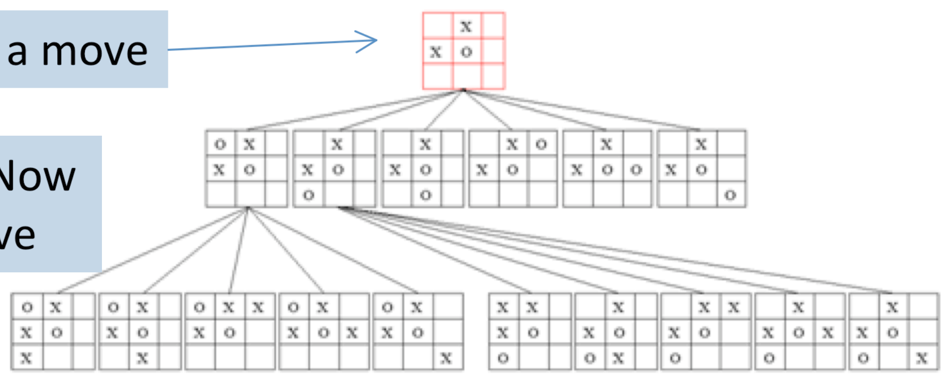
O has six possible moves. Now
It is X's turn to make a move

Game Tree

- Represents two player games
  - Players alternate turns
  - Start from initial game state
  - Many states in which either player can win
  - Draws are possible (neither player wins)

O's turn to make a move

O has six possible moves. Now
It is X's turn to make a move



Game Tree Changes in Search

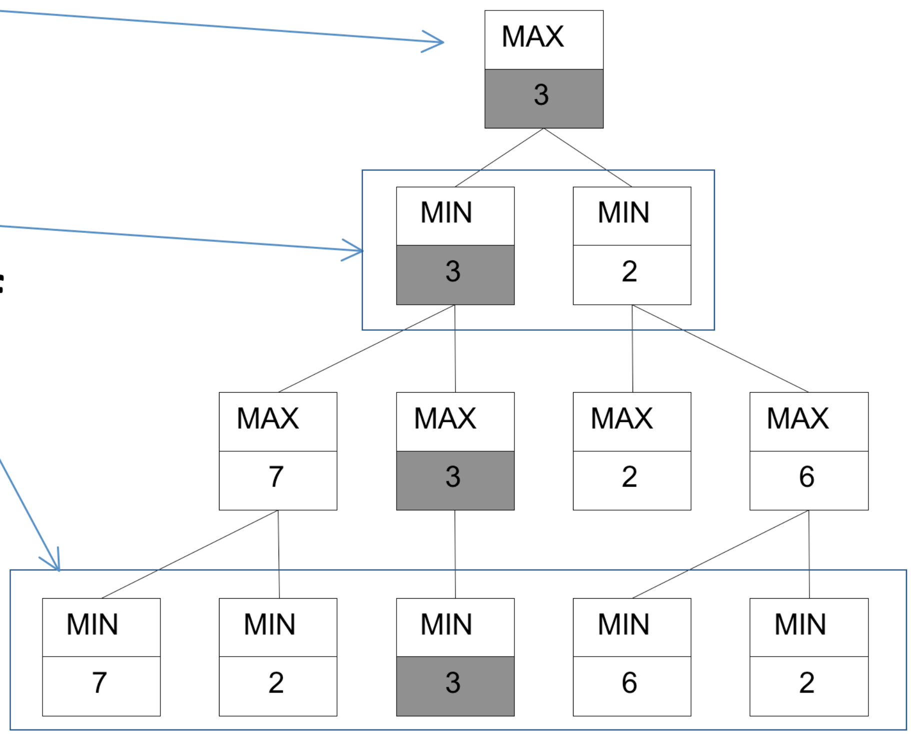
- Goal is no longer a search for a path to known target
  - Given a player's game state, select a move that has best probability of leading to a win, securing a draw, or avoiding a loss
  - Infeasible to expand game tree all the way to a solution
- Must evaluate game states
  - Take into account perspective of player
- No longer maintain sets of board states
  - Make moves (and undo them) while traversing game tree

MINIMAX

- The grandfather algorithm of all two player strategies
  - Book continues with two other enhancements, namely NEGMAX and ALPHABETA
  - We don't have enough time to cover these today
  - Also (honestly) you need to patiently work through these two algorithms yourself to understand fully **how** they work

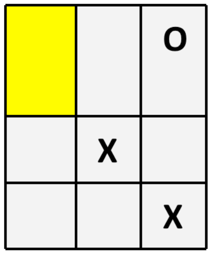
MINIMAX Search Strategy

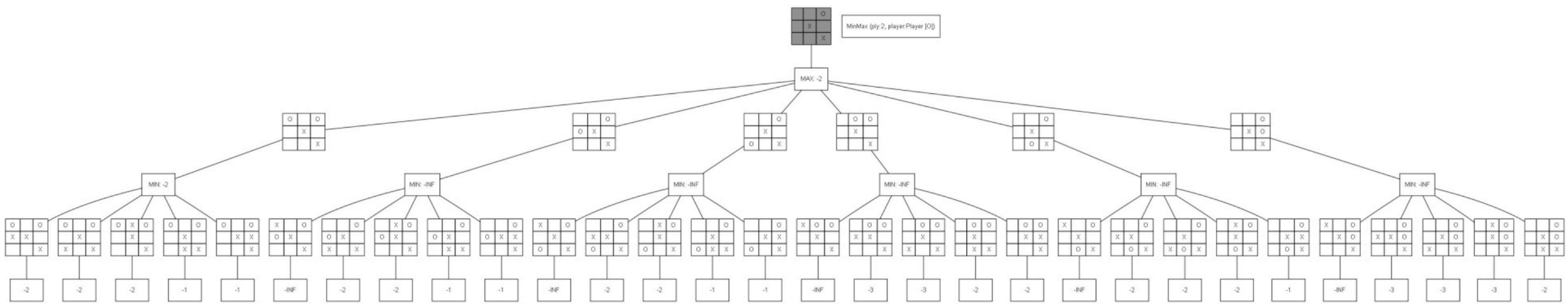
- Recursively explore **first state** to a specific depth, the *ply*
  - Return best move from available moves
- Scoring function evaluates **leaf game states** from the **perspective of the player making the first move**
  - That is the MAX player
  - The opponent is MIN
- Score of interior nodes
  - MAX is largest of its children
  - MIN is smallest of its children



Sample TicTacToe game state

- Given first state as shown
  - Only move that O can make to avoid immediate loss is upper left corner
  - This translates into MinMax search that discovers that “-2” is the best O can do from its children





MINIMAX

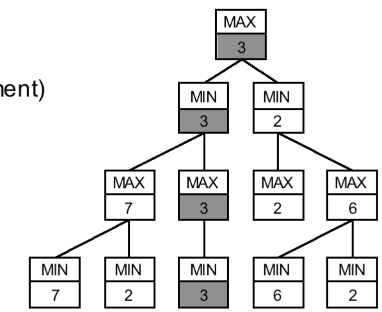
- Recursive execution
- Swap player and opponent
  - To choose proper moves in line 5
  - To choose proper scoring in lines 9-12
- Evaluate all leaf nodes from original player
- Associate scores with moves
  - Return best one

MiniMax

Best case	Average case	Worst case
$O(b^{ply})$	$O(b^{ply})$	$O(b^{ply})$

Recursion
Backtracking
Brute Force

```
bestMove (s, player, opponent)
1.   original = player
2.   [move,score ] = minimax (s, ply, player, opponent)
3.   return move
end

minimax (s, ply, player, opponent)
1.   best = [∅, ∅]
2.   if (ply = 0 or no valid moves) then
3.     score = evaluate s for original player
4.     return [∅, score]

5.   foreach valid move m for player in state s do
6.     execute move m on s
7.     [move, score] = minimax (s, ply –1, opponent, player)
8.     undo move m on s
9.     if (player is original) then
10.        if (score > best.score ) then best = [m, score]
11.    else
12.        if (score < best.score ) then best = [m, score]
13.  return best
end
```



Game tree is recursively explored, to a fixed ply depth.

MIN nodes select the smallest of their child states.

MAX nodes select the largest of their child states.

Leaf nodes evaluate from position of original player

Code Check

```
private MoveEvaluation minimax (int ply, IComparator comp, IPlayer player, IPlayer opponent) {
  // If no allowed moves or a leaf node, return game state score.
  Iterator<IMove> it = player.validMoves(state).iterator();
  if (ply == 0 || !it.hasNext()) {
    return new MoveEvaluation (original.eval(state));
  }

  // Try to improve on this lower-bound (based on selector).
  MoveEvaluation best = new MoveEvaluation (comp.initialValue());

  // Generate game states that result from all valid moves for this player.
  while (it.hasNext()) {
    IMove move = it.next();
    move.execute(state);

    // Recursively evaluate position. Compute Minimax and swap player and opponent
    MoveEvaluation me = minimax (ply-1, comp.opposite(), opponent, player);
    move.undo(state);

    // Select maximum (minimum) of children if we are MAX (MIN)
    if (comp.compare(best.score, me.score) < 0) {
       best = new MoveEvaluation (move, me.score);
    }
   }
  return best;
}
```

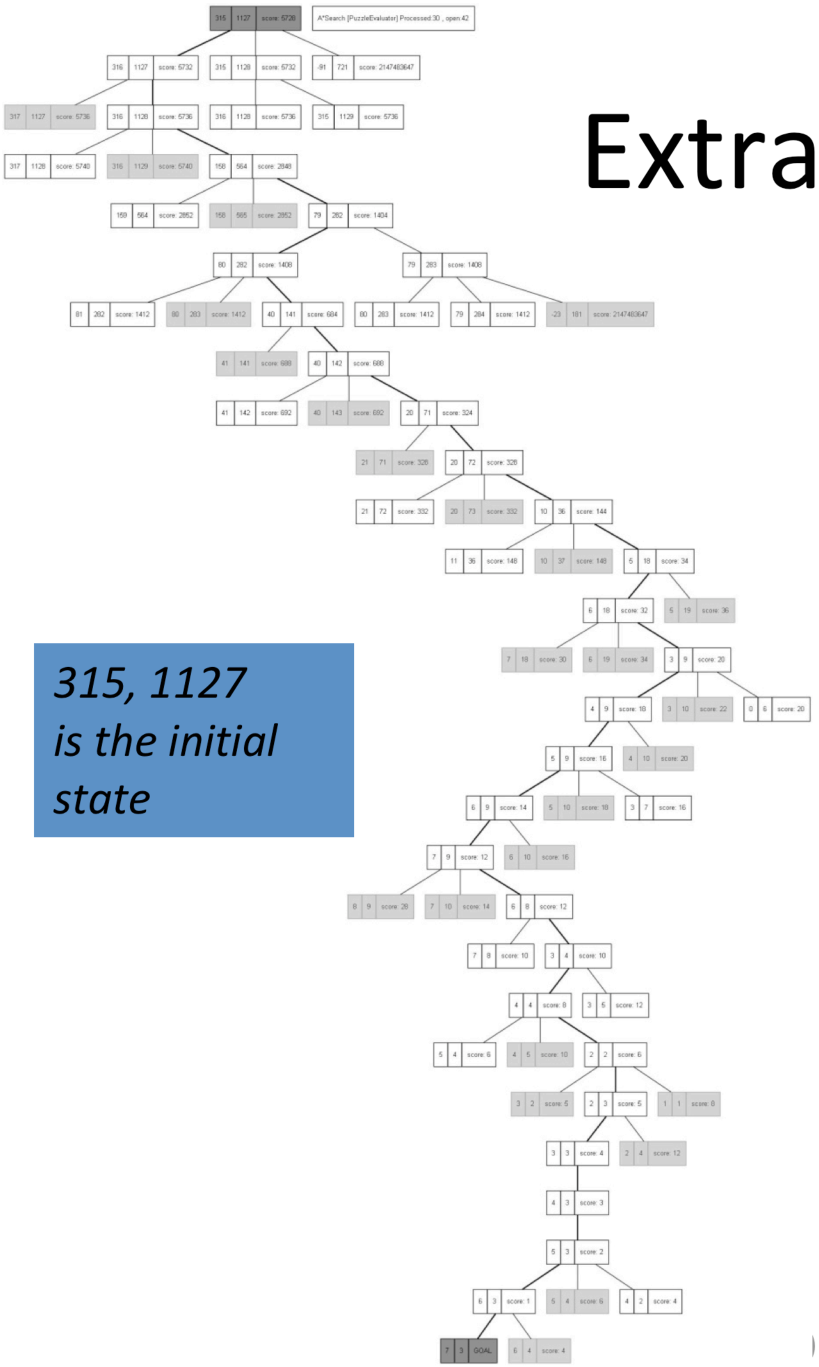
MiniMax

- Domain-independent search strategy
- Any two-player game can be used, if…
  – Can iterate over all player moves for a game state
  – Can design evaluation function that represents
- Success based upon several factors
  – Memory usage
  – Show execution of TicTacToe tournaments against random player

ASTAR Exercise

- Add new move type
  - If two numbers are both ODD, then compute difference *d*, and subtract *d*/2 from both numbers
  - Add `SubtractHalfMove` to the `smallpuzzle` package
  - Modify `validMoves` within SmallPuzzle
- How does heuristic perform with new move?
  - What if either (a,b) becomes negative?

Extra Slide ASTAR Search



315, 1127
is the initial
state

- Must take care to prune away nodes heading into negative territory
  - unproductive