Variations on Search

CS 480
Intro to Artificial Intelligence

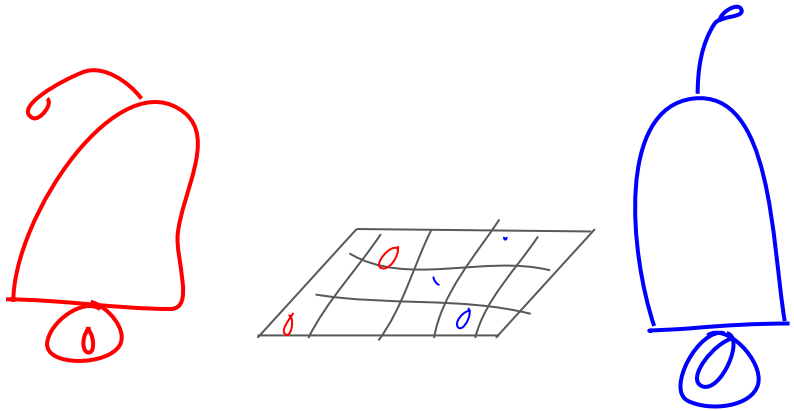
Adding in other agents

Recall, our environments were **single-agent**.

How can you plan a sequence of actions if you have no control over what other agents will do?

Simple **games** are an environment with multiple agents that keeps most of the other restrictions in place

- Checkers, Chess, Go
- Poker, Rock-paper-scissors, Blackjack
- Roulette, Backgammon, Monopoly



Two player, perfect information, zero-sum games

Two player
In general, we could have many players, but many classic games only have two.

Perfect information
All players know the exact state of the game.

Zero-sum
The result (**utility**) for each player at the end of the game is exactly the opposite of the opponent.

These assumptions let us **reason** about the goals of the other agent.

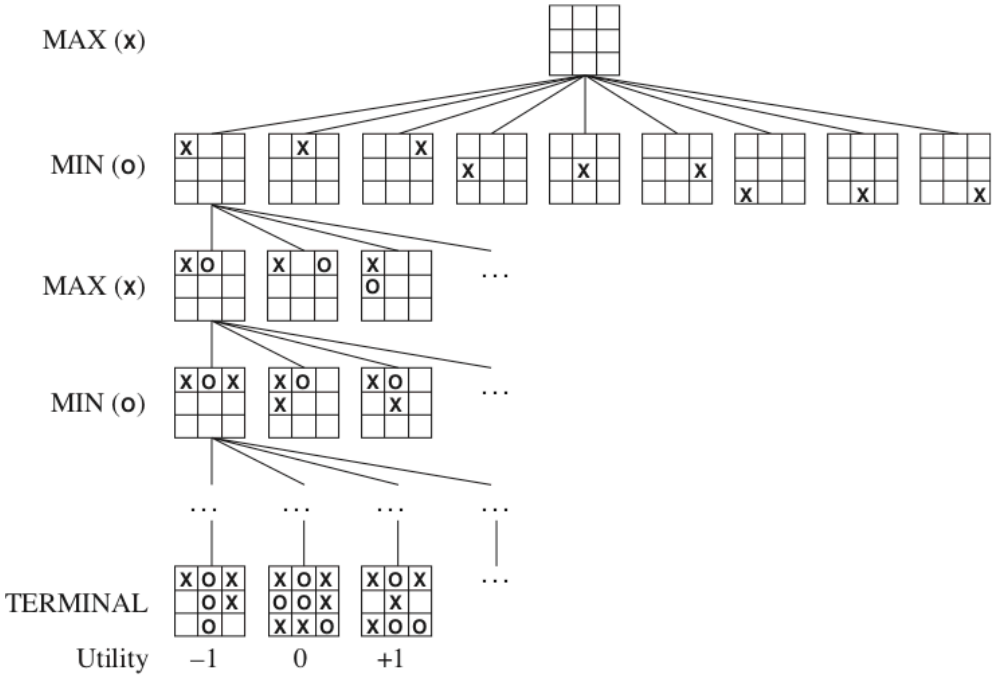
Search trees for 2-player zero-sum games (1)

We can still construct the state space as before, but the other player (**opponent**) gets to make every other move.

In tree form, we choose actions on **alternating layers**.

We are choosing actions that **maximize** our utility, opponent is choosing actions that **minimize** our utility

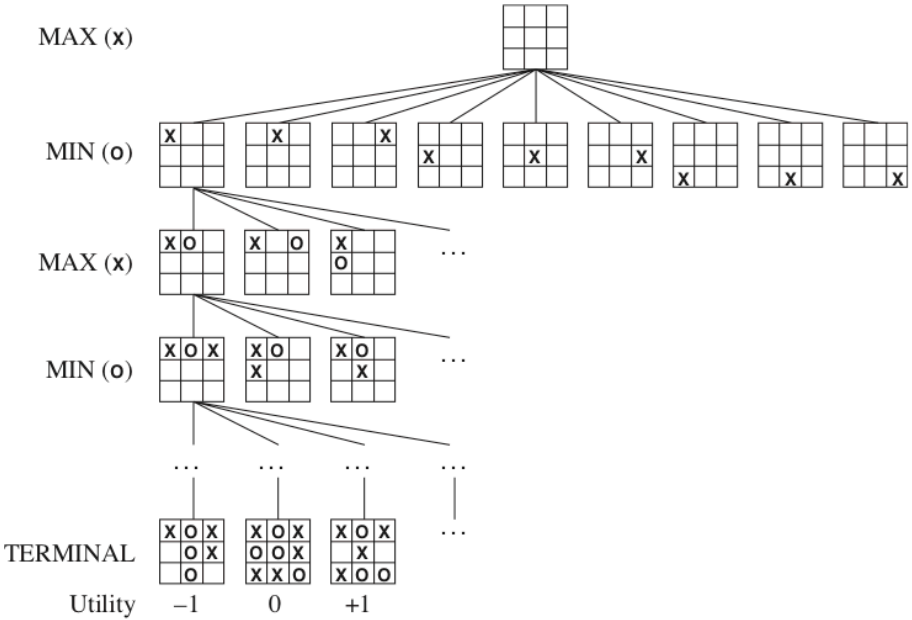
Idea: keep expanding nodes until we hit a **terminal** state, propagate the final utility back to initial action choice, pick the action with the best utility.



Search trees for 2-player zero-sum games (2)

Notes

- Result of searching this tree is a **single move** (have to wait for opponent's move)
- To pick the **optimal** move, we might need to expand the **entire search tree**
- We are **assuming** the opponent is **rational** (picks the best moves), but it doesn't matter if they're not!



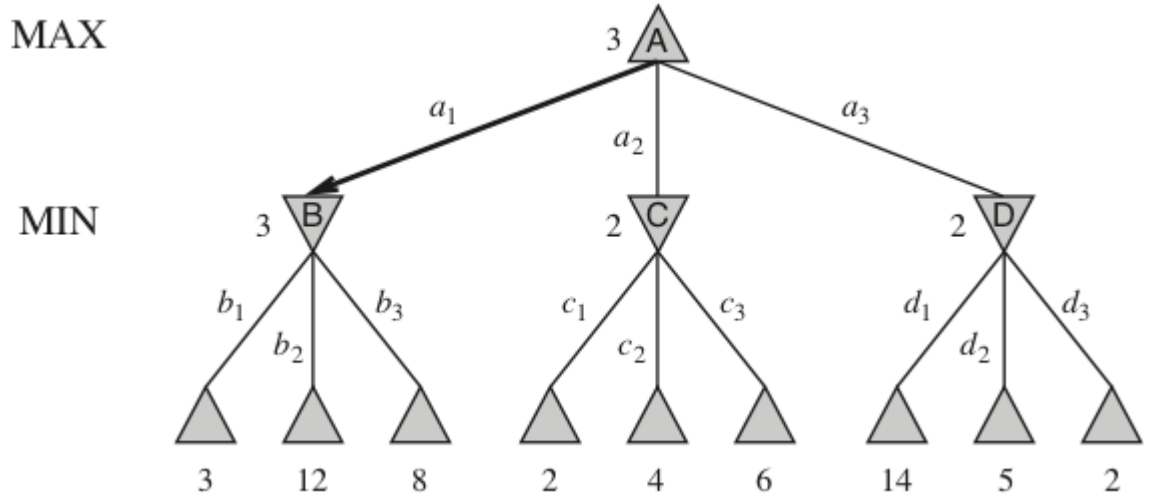
A recursive definition of the `Minimax` value of a **state**

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{Max} \\ \min_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{Min} \end{cases}$$

A simple example

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{Max} \\ \min_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{Min} \end{cases}$$

`Minimax`(B) = 3 (b_1)
`Minimax`(C) = 2 (c_1)
`Minimax`(D) = 2 (d_3)
`Minimax`(A) = 3 (a_1)



The Minimax algorithm

```
def Minimax_Decision(state):
  best_action,best_val = None,-math.inf
  for a in actions(state):
    s_prime = result(a,state)
    a_val = Min_Value(s_prime)
    if a_val>best_val:
      best_val = a_val
      best_action = a
  return best_action
```

```
def Max_Value(state):
  if isTerminal(state):
    return utility(state)
  v = -math.inf
  for a,s in successors(state):
    v = max(v,Min_Value(s))
  return v
```

```
def Min_Value(state):
  if isTerminal(state):
    return utility(state)
  v = math.inf
  for a,s in successors(state):
    v = min(v,Max_Value(s))
  return v
```

The Minimax algorithm - notes

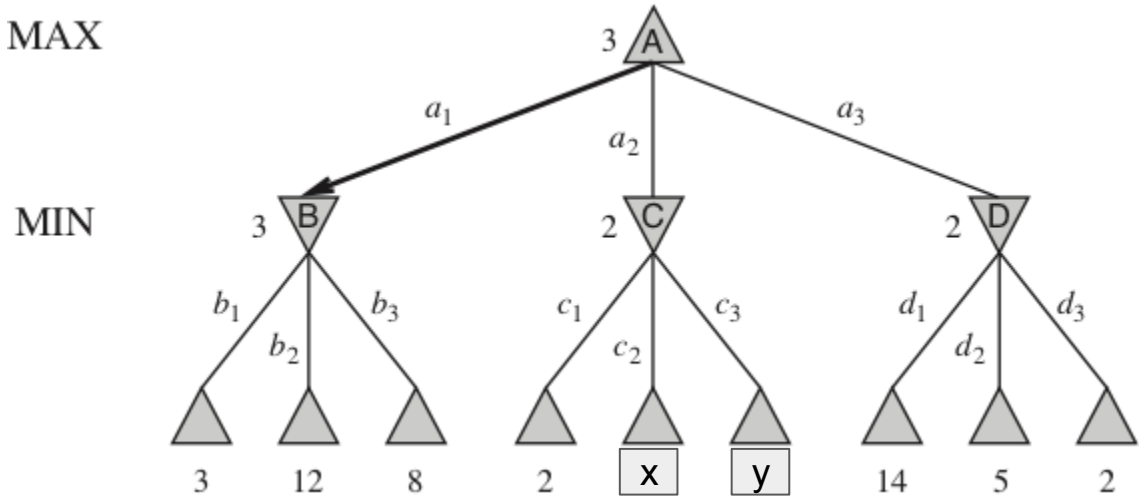
Basically, DFS!

- Complete? **Yes** (if tree is finite)
- Optimal? **Yes**
- Time complexity? **$O(b^m)$** (b: branching factor, m: depth of tree)
- Space complexity? **O(m*b)**

For chess, $b \cong 35$, $m \cong 100$

Expanding the full tree isn't going to work!

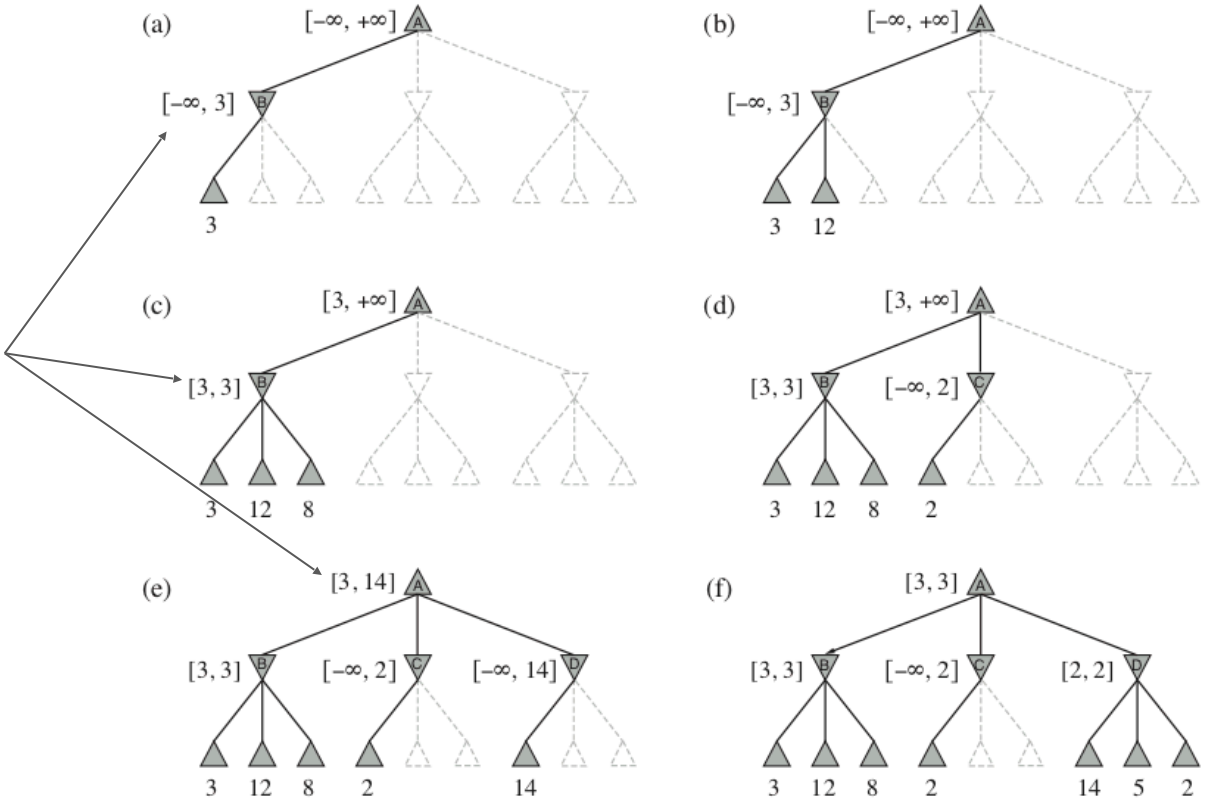
Do we always need to expand every node? (1)



$$\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3
\end{aligned}$$

Do we always need to expand every node? (2)

Range of possible Minimax values



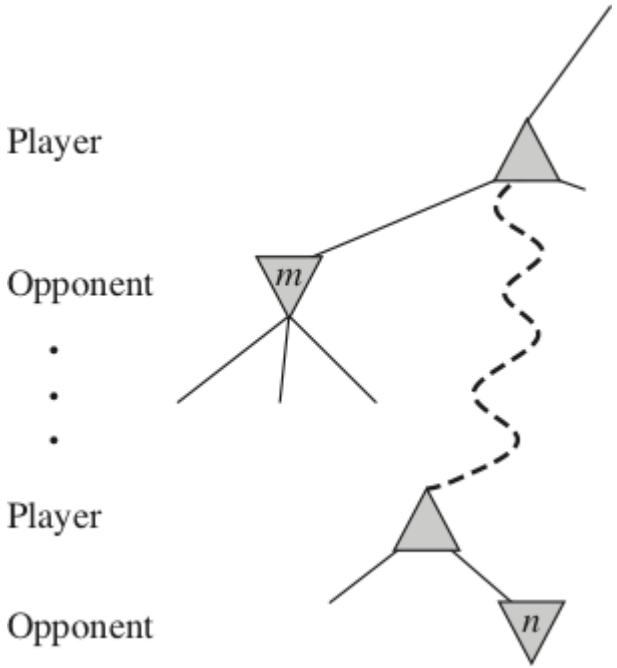
Unreachable nodes

If we can move to *m*, which has a strictly better `Minimax` value than *n*, we don't need to explore the path containing *n*.

Define the following helper variables

α = the value of the **highest value** choice we have found so far at any choice point along the path for `Max`

β = the value of the **lowest value** choice we have found so far at any choice point along the path for `Min`



The Alpha-Beta search algorithm

```
def Alpha_Beta_Search(state):
  best_action,best_val = None,-math.inf
  for a in actions(state):
    s_prime = result(a,state)
    a_val = Min_Value(s_prime,-math.inf,math.inf)
    if a_val>best_val:
      best_val = a_val
      best_action = a
  return best_action
```

```
def Max_Value(state,alpha,beta):
  if isTerminal(state):
    return utility(state)
  v = -math.inf
  for a,s in successors(state):
    v = max(v,Min_Value(s,alpha,beta))
    if v >= beta:
      return v
    alpha = max(alpha,v)
  return v
```

```
def Min_Value(state,alpha,beta):
  if isTerminal(state):
    return utility(state)
  v = math.inf
  for a,s in successors(state):
    v = min(v,Max_Value(s,alpha,beta))
    if v<= alpha:
      return v
    beta = min(beta,v)
  return v
```

Alpha-Beta notes

Pruning does not affect the optimality of the final result

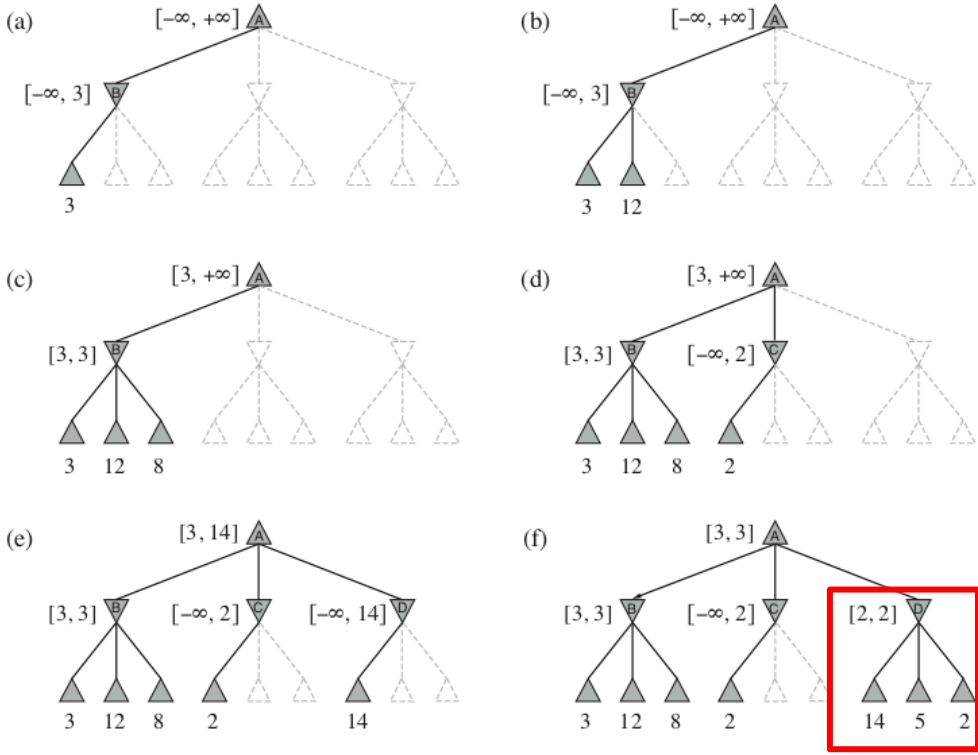
Which paths we can prune highly depends on the expansion order (just like DFS)

Worst case, same complexity as `Minimax`

With the best ordering, time complexity: **$O(b^{m/2})$** (effective branching factor $\sqrt{b}$)

For chess, $b \cong 6$ instead of 35

6^{100} is still pretty big



Alternatives to finding the optimal solution

- Use `Cutoff-Test` and `Eval` instead of `Terminal` and `Utility`
  - `Cutoff-Test` checks to see if terminal, or if we've already expanded past some depth limit
  - `Eval` provides an **estimate** of the true utility of this node without searching all the way down to the leaves. Similar to the **heuristic** from previous lectures
- Use **Iterative Deepening Search** and return the move selected by the deepest completed search (alternatively, use IDS to help with move ordering)
- Use a lookup table of previously visited states (**transposition table**, **memoization**)
  - Kind of like the closed list from previous lectures
  - Need to have a cache-like structure to keep memory usage bounded

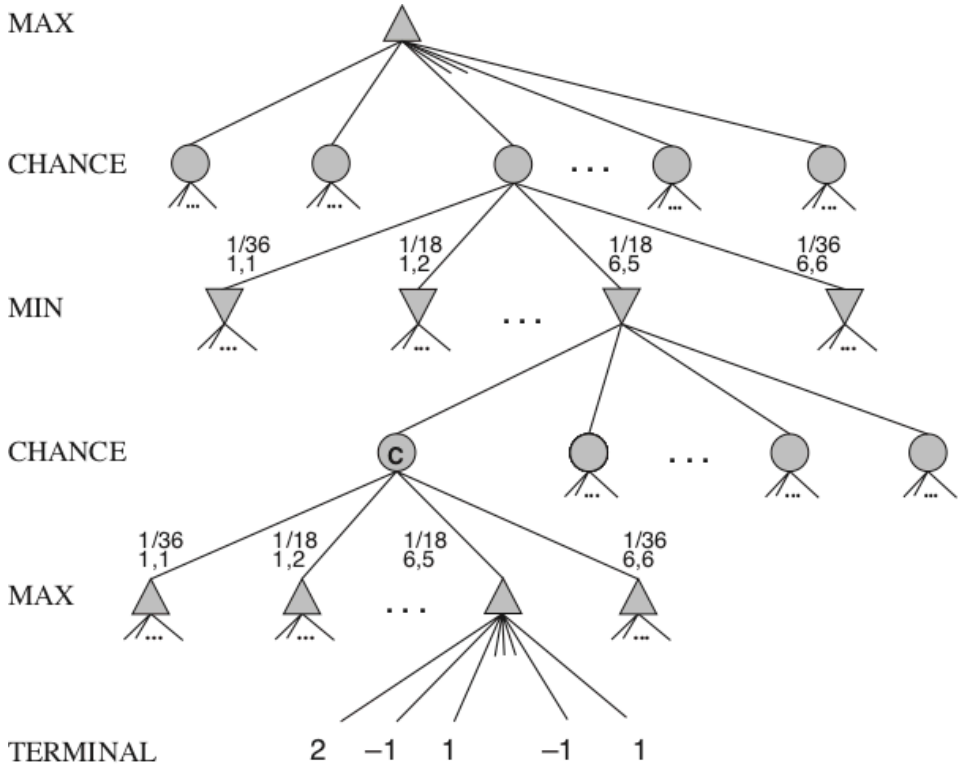
Games of Chance

Plenty of games include elements of randomness

- Poker, Blackjack, Solitaire (shuffling)
- Backgammon, Monopoly (dice)
- Roulette, Pachinko, Slots (mechanical)

We can still use search, if we modify our search tree to include probabilities with edges

For adversarial games, we can introduce another player, “Chance”, and use a slightly modified version of `Minimax`



Expectiminimax

The "`Expectiminimax`" value is the same as `Minimax` except that for chance nodes we take the **expected value** of all the children

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_a P(a)\textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

We end up with a version of adversarial search that uses this instead of `Minimax-Value`.

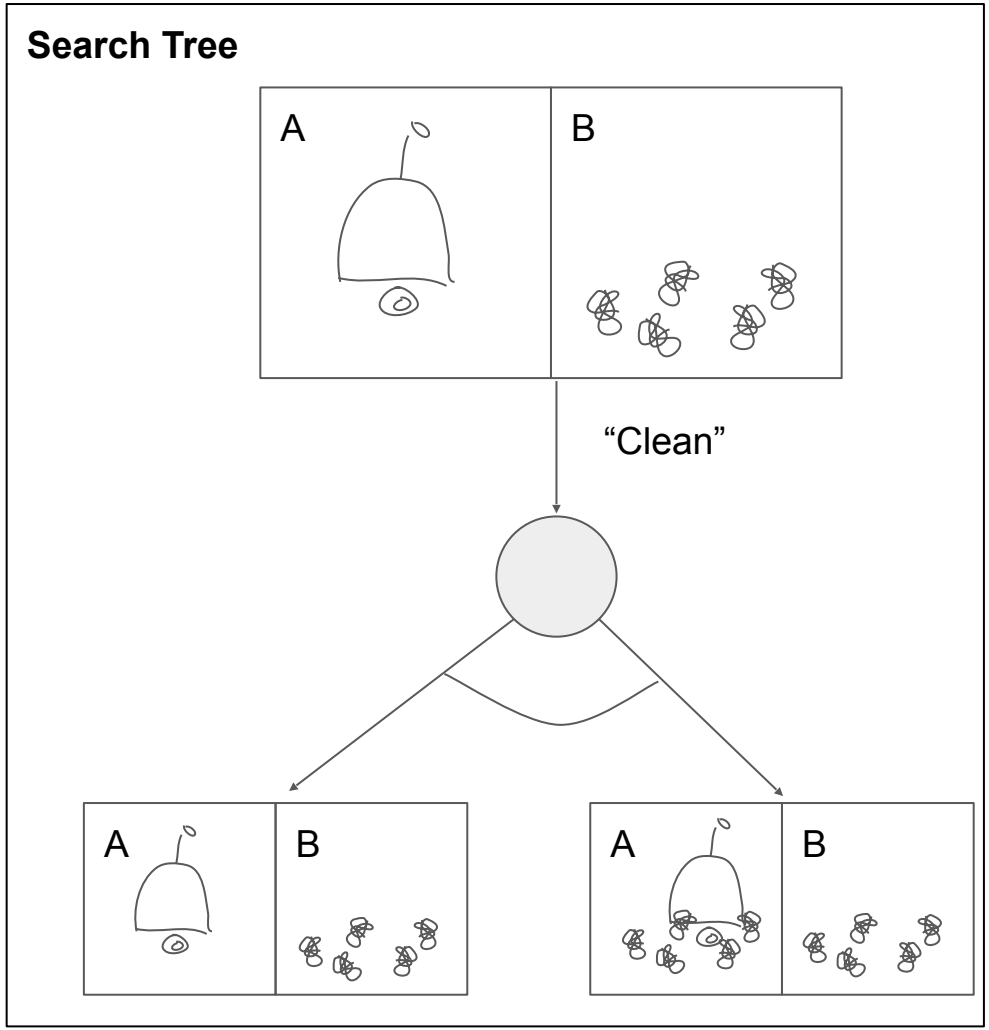
Stochastic Actions

In the single-agent case, instead of having a "Chance" player, we can model actions as having **non-deterministic** outcomes

Deterministic actions
`Result`(s,a) = s'

Stochastic actions
`Result`(s,a) = $\{s_1, s_2, ..., s_k\}$

This type of representation is called an "`And-Or` search tree"



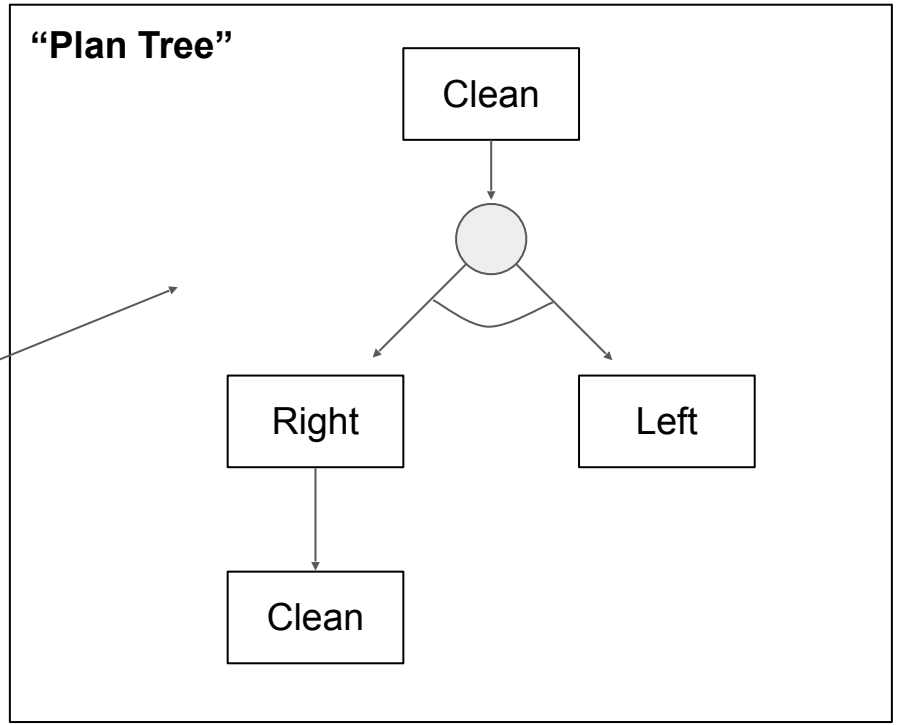
Solutions for `And-Or` search trees (no cycles)

As long as there are no cycles, we can still do search!

The result of searching an `And-Or` tree is no longer a sequence of actions, but a **contingency plan**:

[Clean, **if** S=5 **then** [Right, Clean] **else** [Left]]

An acyclic contingency plan has a nested set of sub-plans for each possible outcome, and can **also** be represented as a tree



One version of `And-Or` Search

```
def and_or_search(prob):
  or_search(prob.initstate,prob,[])
```

```
def or_search(state, prob, path):
  if prob.goal_test(state):
    return empty plan
  if state in path:
    return failure   #cycle
  for action in prob.acts(state):
    S = prob.result(state,action)
    np = [state,]+path
    plan = and_search(S, prob, np)
    if plan != failure:
      return [action,]+plan
  return failure
```

```
def and_search(states, prob, path):
  plans = list()
  for s in states:
    subplan=or_search(s,prob,path)
    if subplan == failure:
      return failure
    plans.append(subplan)
  return plans
```

Like DFS, but with a base-case for cycles, and alternating AND/OR layers. Compare Fig 4.11

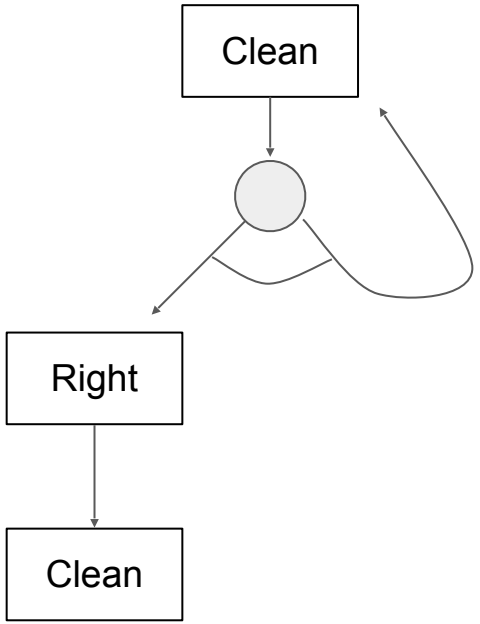
Solutions for `And-Or` search trees (with cycles)

Actually, even if there are cycles, we can *sometimes* find a solution.

If each outcome of a non-deterministic action occurs **eventually**, we can still find a “solution” that will… eventually… get to the goal.

To keep our solutions compact, we can introduce **labels** for repeated steps

[L_1: Clean, **if** S=5 **then** L_1 **else** [Right ...]]



Summary and preview

Wrapping up

- **Games** with multiple agents can be solved with similar search methods, when certain assumptions are made that let us reason about the objectives of the other agents
- `Minimax` uses an alternating, DFS-like search to pick the best **action** for the current state
- We can effectively increase the depth of the trees we can search in a fixed time by using Alpha-Beta **pruning**

Next time

- Introducing stochasticity