Expectimax search

Some figures are derived from the slides created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley, available at http://ai.berkeley.edu.

Uncertain outcomes

- Single agent search tree assumed that you are the only agent taking actions and their result is predicted by the transition function $T(s, a) \to s'$
- Minmax assumed that there is also an opponent taking actions
    - But, in some sense, the zero sum opponent is also predictable
- What if we don't know the result of the action?
    - Inherent randomness in the environment: rolling a dice
    - Unpredictable opponents or bystanders with random behavior
    - Actions can fail or succeed partially (slipping wheels etc)

Probabilistic transition function

- A way to think about this is that the transition function is now probabilistic: $T(s, a, s') \in [0, 1]$
- We will talk more about this when we discuss Markov Decision Processes and reinforcement learning

Expectimax search

- We are still trying to compute the V value
- **max** nodes: return the max of successors
- **min** nodes: return the min of the successors
- **expectation** nodes: return the probability weighted average (*expectation*) of children

Reminder: expectation of a random variable

$$\mathbb{E}(f(x)) = \sum_i p(x_i) f(x_i)$$

or, in a continuous case:

$$\mathbb{E}(f(x)) = \int p(x) f(x) dx$$

Expectation of time to get to the airport:

- Drive time: 20 clear weather, 40 min in rain, 60 min in snow
- Likelihood of clear weather 80%, rain 15%, snow 5%
- Expectation: 20 * 0.8 + 40 * 0.15 + 60 * 0.05 = 25

Backgammon (or other dice-based zero-sum games)

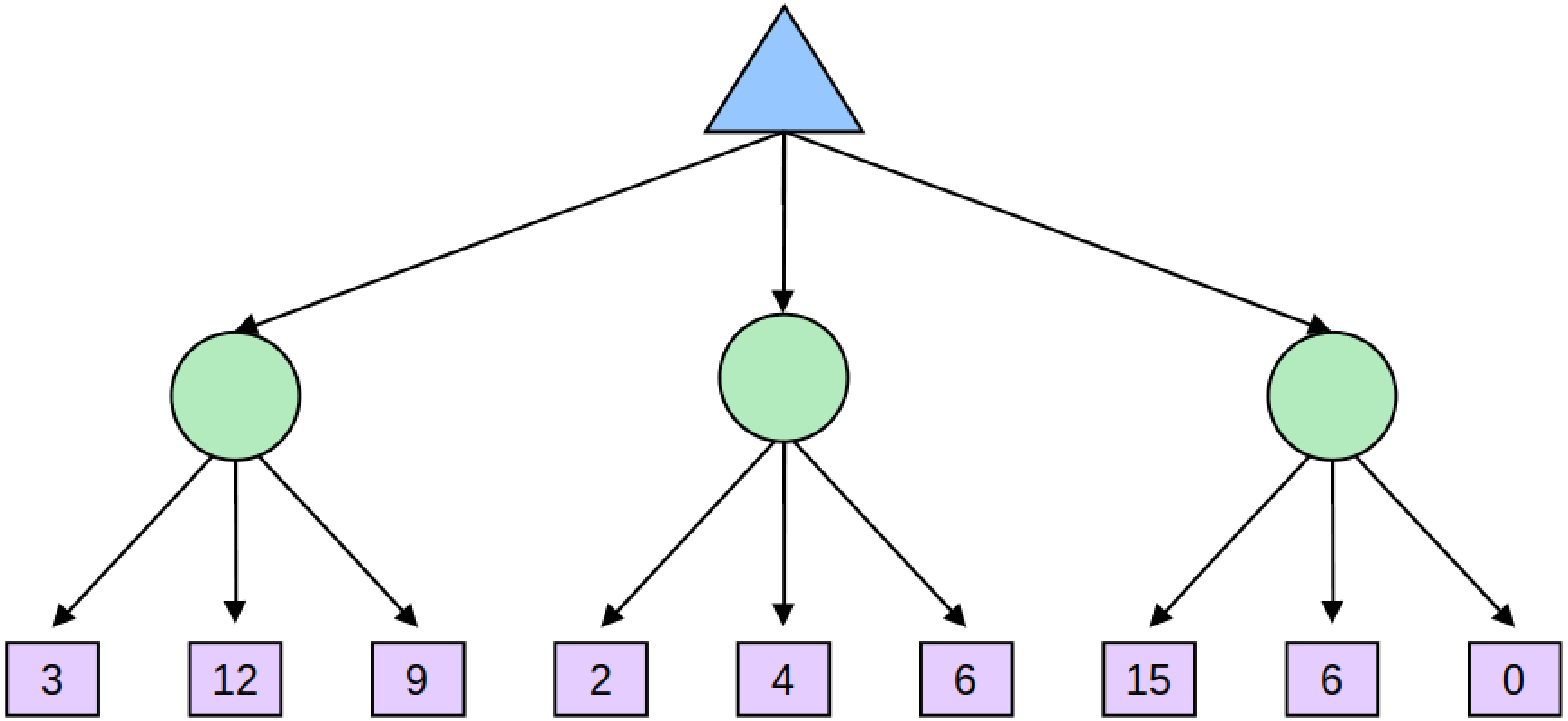
- Max node (move by ego, after knowing dice)
- Expect node (dice by opponent)
- Min node (move by opponent, after knowing dice)
- Expect node (dice by ego)
- Max node (move by ego, after knowing dice)... and so on

Expectimax (and other variants) pseudocode

```
def value(state):
  if state is TERMINAL: return value
  if state is MAX: return maxvalue(state)
  if state is MIN: return minvalue(state)
  if state is EXPECT: return expvalue(state)

def maxvalue(s)
  v = −∞
  for s' in succ(s)
    v = min (v, maxvalue(s'))
  return v

def expvalue(s)
  v = 0
  for s' in succ(s)
    v = v + probability(s') ∗ value(s')
  return v
```


3
12
9
2
4
6
15
6
0

Expectimax pruning

- Can we prune expectimax?
- Problem: expectation can go both up and down with new nodes!
  - You need all the subnodes to calculate the expectation
- Heuristics:
  - prune branches if their contribution to the expectation is small enough to be negligible (e.g. they are unlikely)
  - prune branches if you predict their values as being below a threshold

Depth limited expectimax

- Expectimax nodes can really blow up the computation time, because you need to evaluate everything below
- It is useless to make long plans when they depend on repeated dice throws to come out just so:
  - I will throw an 8 and move like this, then my opponent will throw a 4 and move like that, then I will throw an 11…
- Game programs for games with significant random component:
  - Think ahead only 1..4 plies
  - Use a **very good** evaluation function

Where do we get the probabilities from?

- In expectimax search, we need to know the probabilities of outcomes
    - Sometimes it is some uniform or near randomness (eg. dice)
    - Sometimes it is a small uncertainty on a positive or negative action.
- Where do we get the probabilities? - The **model**
    - Sometimes it is simple - eg. dice roll
    - Sometimes it is very complex
- We will revisit this later

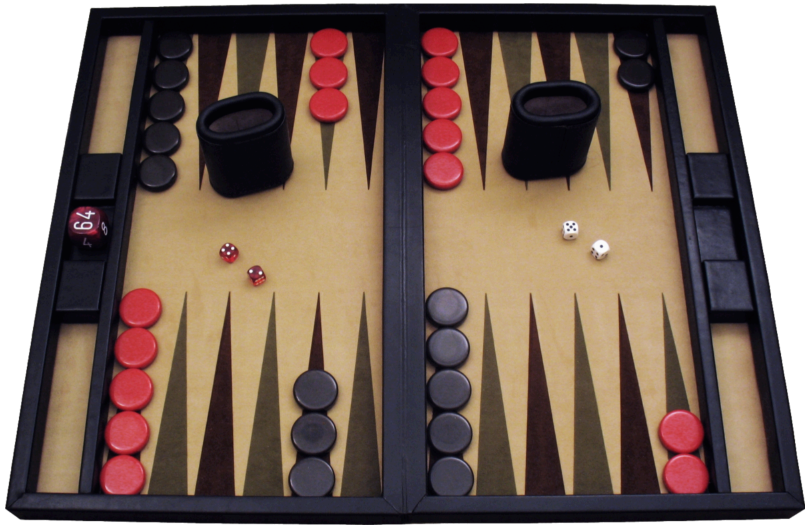
Informed probabilities

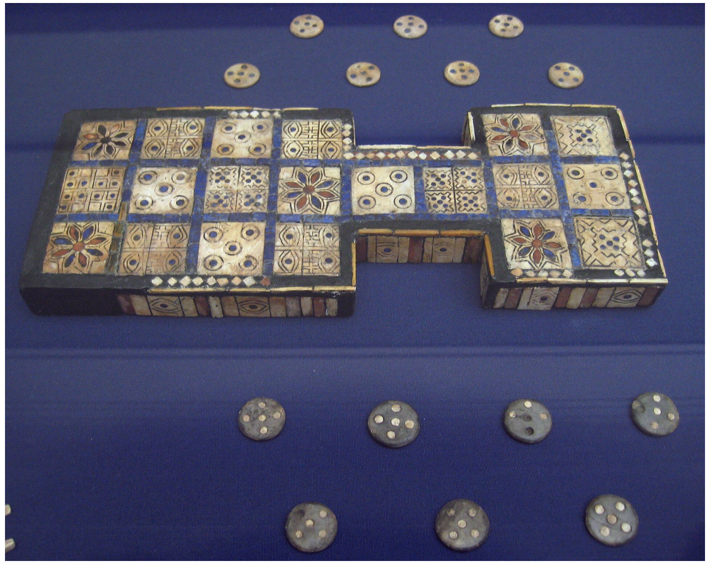
- Expectimax can also handle situations where you try to model an imperfect opponent:
- Let us say that the opponent is doing the perfect minmax move 90% of the time, but moves randomly 10% of the time
  - This is an expectation node. But you don't know the probability of the moves ahead of time, you need to calculate it!
- You need to run a **simulation of your opponent**, with the opponent **simulating you**
  - This is very expensive for expectimax
  - It is much cheaper for minmax, because the two simulations are folded into the same tree.

Mixed layers

- Different layers (max/min/expectations) can be mixed randomly.
- Often, we consider the environment an additional "random" player.
- Each node computes the appropriate combination of its children.

Example: Backgammon


64



Example: Backgammon

- Dice rolls increase b: 21 possible rolls with 2 dice
  - About 20 legal moves
  - With depth 2 number of nodes is $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node is getting smaller
  - Searching for the best outcome is getting less and less useful
  - There is no point searching ten moves ahead: it is very unlikely that the dice will play out just so
- **History**
  - TDGammon (Gerald Tesauro, 1990s): 1st AI world champion in any significant game. Uses depth-2 search + very good evaluation function + reinforcement learning

Multi-agent games

- What if the game is not zero sum, or has multiple players?
- Node values are not tuples of utility
- Each player maximizes its own utility
- Emergent cooperation and competitition