Adversarial Search

This slide deck courtesy of Dan Klein at UC Berkeley
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
Minimax Example
Pruning in Minimax Search
Alpha-Beta Pruning

- **General configuration**
  - We’re computing the MIN-VALUE at $n$
  - We’re looping over $n$’s children
  - $n$’s value estimate is dropping
  - $a$ is the best value that MAX can get at any choice point along the current path
  - If $n$ becomes worse than $a$, MAX will avoid it, so can stop considering $n$’s other children
  - Define $b$ similarly for MIN
Alpha-Beta Pruning Example

Start at the root with a value of 3.

- For MAX, its best alternative is 3 or above.
- For MIN, its best alternative is 1 or below.

The diagram shows the tree structure with values at each node:
- At the root (3)
  - MAX's best alternative is 3 or above (value 3)
  - MIN's best alternative is 1 or below (value 1)
- At the level below:
  - MAX's best alternative is 12 or above (value 12)
  - MIN's best alternative is 2 or below (value 2)
- At the next level:
  - MAX's best alternative is 8 or above (value 8)
  - MIN's best alternative is value 5 or below

The pruning occurs when a value is determined to be worse than the current best value for MAX or better than the current best value for MIN, thus eliminating branches from further consideration.
Starting a/b

Raising a

Lowering b

Raising a

a is MAX’s best alternative here or above
b is MIN’s best alternative here or above
function Max-Value(state) returns a utility value
  if Terminal-Test(state) then return Utility(state)
  \[ v \leftarrow -\infty \]
  for \( a, s \) in Successors(state) do \[ v \leftarrow \max(v, \text{Min-Value}(s)) \]
  return \( v \)

function Max-Value(state, \( \alpha, \beta \)) returns a utility value
  inputs: state, current state in game
          \( \alpha \), the value of the best alternative for \( \max \) along the path to state
          \( \beta \), the value of the best alternative for \( \min \) along the path to state
  if Terminal-Test(state) then return Utility(state)
  \[ v \leftarrow -\infty \]
  for \( a, s \) in Successors(state) do
    \[ v \leftarrow \max(v, \text{Min-Value}(s, \alpha, \beta)) \]
    if \( v \geq \beta \) then return \( v \)
    \[ \alpha \leftarrow \max(\alpha, v) \]
  return \( v \)
Alpha-Beta Pruning Properties

- This pruning has no effect on final result at the root.

- Values of intermediate nodes might be wrong!
  - Important: children of the root may have the wrong value.

- Good child ordering improves effectiveness of pruning.

- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$.
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…

- This is a simple example of metareasoning (computing about what to compute).
Pruning Example
Multi-Agent Utilities

- Similar to minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own utility
  - Can give rise to cooperation and competition dynamically…
Expectimax Search Trees

- What if we don’t know what the result of an action will be? E.g.,
  - In solitaire, next card is unknown
  - In minesweeper, mine locations
  - In pacman, the ghosts act randomly

- Can do **expectimax search** to maximize average score
  - Max nodes as in minimax search
  - Chance nodes, like min nodes, except the outcome is uncertain
  - Calculate **expected utilities**
  - I.e. take weighted average (expectation) of values of children

- Later, we’ll learn how to formalize these underlying problems as **Markov Decision Processes**
Reminder: Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

Example: traffic on freeway?
- Random variable: $T =$ whether there’s traffic
- Outcomes: $T$ in {none, light, heavy}
- Distribution: $P(T=\text{none}) = 0.25$, $P(T=\text{light}) = 0.55$, $P(T=\text{heavy}) = 0.20$

Some laws of probability (more later):
- Probabilities are always non-negative
- Probabilities over all possible outcomes sum to one

As we get more evidence, probabilities may change:
- $P(T=\text{heavy}) = 0.20$, $P(T=\text{heavy} \mid \text{Hour}=8\text{am}) = 0.60$
- We’ll talk about methods for reasoning and updating probabilities later
Reminder: Expectations

- We can define function f(X) of a random variable X

- The expected value of a function is its average value, weighted by the probability distribution over inputs

- Example: How long to get to the airport?
  - Length of driving time as a function of traffic:
    - L(none) = 20, L(light) = 30, L(heavy) = 60
  - What is my expected driving time?
    - Notation: E[ L(T) ]
    - Remember, P(T) = {none: 0.25, light: 0.5, heavy: 0.25}

- \[ E[ L(T) ] = L(\text{none}) \times P(\text{none}) + L(\text{light}) \times P(\text{light}) + L(\text{heavy}) \times P(\text{heavy}) \]

- \[ E[ L(T) ] = (20 \times 0.25) + (30 \times 0.5) + (60 \times 0.25) = 35 \]
Expectimax Example
def value(s):
    if s is a max node return maxValue(s)
    if s is an exp node return expValue(s)
    if s is a terminal node return evaluation(s)

def maxValue(s):
    values = [value(s’) for s’ in successors(s)]
    return max(values)

def expValue(s):
    values = [value(s’) for s’ in successors(s)]
    weights = [probability(s, s’) for s’ in successors(s)]
    return expectation(values, weights)
Expectimax Pruning?
Depth-Limited Expectimax

Estimate of true expectimax value (which would require a lot of work to compute)
What Utilities to Use?

- For minimax, terminal function scale doesn’t matter
  - We just want better states to have higher evaluations (get the ordering right)
  - We call this **insensitivity to monotonic transformations**

- For expectimax, we need *magnitudes* to be meaningful
What Probabilities to Use?

- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a node for every outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!
- For now, assume for any state we magically have a distribution to assign probabilities to opponent actions / environment outcomes

Having a probabilistic belief about an agent’s action does not mean that agent is flipping any coins!
Expectimax for Pacman

- Notice that we’ve gotten away from thinking that the ghosts are trying to minimize pacman’s score
- Instead, they are now a part of the environment
- Pacman has a belief (distribution) over how they will act
- Quiz: Can we see minimax as a special case of expectimax?
- Quiz: what would pacman’s computation look like if we assumed that the ghosts were doing 1-ply minimax and taking the result 80% of the time, otherwise moving randomly?
- If you take this further, you end up calculating belief distributions over your opponents’ belief distributions over your belief distributions, etc…
  - Can get unmanageable very quickly!
## World Asssumptions

### Results from playing 5 games

<table>
<thead>
<tr>
<th></th>
<th>Minimizing Ghost</th>
<th>Random Ghost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Minimax Pacman</strong></td>
<td>Won 5/5</td>
<td>Won 5/5</td>
</tr>
<tr>
<td></td>
<td>Avg. Score: 483</td>
<td>Avg. Score: 493</td>
</tr>
<tr>
<td><strong>Expectimax Pacman</strong></td>
<td>Won 1/5</td>
<td>Won 5/5</td>
</tr>
<tr>
<td></td>
<td>Avg. Score: -303</td>
<td>Avg. Score: 503</td>
</tr>
</tbody>
</table>

Pacman used depth 4 search with an eval function that avoids trouble. Ghost used depth 2 search with an eval function that seeks Pacman.
Mixed Layer Types

- E.g. Backgammon
- Expectiminimax
  - Environment is an extra player that moves after each agent
  - Chance nodes take expectations, otherwise like minimax

```
ExpectiMinimax-Value(state):
    if state is a MAX node then
        return the highest ExpectiMinimax-Value of Successors(state)
    if state is a MIN node then
        return the lowest ExpectiMinimax-Value of Successors(state)
    if state is a chance node then
        return average of ExpectiMinimax-Value of Successors(state)
```
Stochastic Two-Player

- Dice rolls increase $b$: 21 possible rolls with 2 dice
  - Backgammon $\approx 20$ legal moves
  - Depth $2 = 20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier…
- TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play
- 1st AI world champion in any game!
Maximum Expected Utility

- Why should we average utilities? Why not minimax?

- Principle of maximum expected utility:
  - A rational agent should choose the action which maximizes its expected utility, given its knowledge

- Questions:
  - Where do utilities come from?
  - How do we know such utilities even exist?
  - Why are we taking expectations of utilities (not, e.g. minimax)?
  - What if our behavior can’t be described by utilities?
What’s Next?

- Make sure you know what:
  - Probabilities are
  - Expectations are

- Next topics:
  - Dealing with uncertainty
  - How to learn evaluation functions
  - Markov Decision Processes
Deterministic Two-Player

- E.g. tic-tac-toe, chess, checkers
- Zero-sum games
  - One player maximizes result
  - The other minimizes result
- Minimax search
  - A state-space search tree
  - Players alternate
  - Each layer, or ply, consists of a round of moves*
  - Choose move to position with highest minimax value = best achievable utility against best play

* Slightly different from the book definition
Minimax Example
Minimax Search

**function** Max-Value(*state*) **returns** a utility value

if Terminal-Test(*state*) then return Utility(*state*)

\[ v \leftarrow -\infty \]

for \( a, s \) in Successors(*state*) do \[ v \leftarrow \max(v, \text{Min-Value}(s)) \]
return \( v \)

---

**function** Min-Value(*state*) **returns** a utility value

if Terminal-Test(*state*) then return Utility(*state*)

\[ v \leftarrow \infty \]

for \( a, s \) in Successors(*state*) do \[ v \leftarrow \min(v, \text{Max-Value}(s)) \]
return \( v \)
What is Search For?

- Models of the world: single agents, deterministic actions, fully observed state, discrete state space

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics to guide, fringe to keep backups

- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems
Constraint Satisfaction Problems

- Standard search problems:
  - State is a “black box”: arbitrary data structure
  - Goal test: any function over states
  - Successor function can be anything

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables $X_i$ with values from a domain $D$ (sometimes $D$ depends on $i$)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Simple example of a formal representation language

- Allows useful general-purpose algorithms with more power than standard search algorithms
Example: N-Queens

- **Formulation 1:**
  - **Variables:** $X_{ij}$
  - **Domains:** $\{0, 1\}$
  - **Constraints**
    \[
    \forall i, j, k \ (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}
    \]
    \[
    \forall i, j, k \ (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}
    \]
    \[
    \forall i, j, k \ (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}
    \]
    \[
    \forall i, j, k \ (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}
    \]
    \[
    \sum_{i,j} X_{ij} = N
    \]
Example: N-Queens

- **Formulation 2:**
  - **Variables:** $Q_k$
  - **Domains:** $\{1, 2, 3, \ldots N\}$

- **Constraints:**
  - Implicit: $\forall i, j$ non-threatening($Q_i$, $Q_j$)
  - or-
  - Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
    - \ldots
Example: Map-Coloring

- **Variables:** \( WA, NT, Q, NSW, V, SA, T \)
- **Domain:** \( D = \{ \text{red, green, blue} \} \)
- **Constraints:** adjacent regions must have different colors
  \[ WA \neq NT \]
  \[ (WA, NT) \in \{ (\text{red, green}), (\text{red, blue}), (\text{green, red}), \ldots \} \]

- **Solutions** are assignments satisfying all constraints, e.g.:
  \[ \{ WA = \text{red}, NT = \text{green}, Q = \text{red},\]
  \[ NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green} \} \]
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables

- Binary constraint graph: nodes are variables, arcs show constraints

- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!
Example: Cryptarithmetic

- Variables (circles):
  \[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]

- Domains:
  \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

- Constraints (boxes):
  \[ \text{alldiff}(F, T, U, W, R, O) \]
  \[ O + O = R + 10 \cdot X_1 \]
  \[ \ldots \]
Example: Sudoku

- Variables:
  - Each (open) square

- Domains:
  - \{1,2,\ldots,9\}

- Constraints:
  - 9-way alldiff for each column
  - 9-way alldiff for each row
  - 9-way alldiff for each region
Example: Boolean Satisfiability

- Given a Boolean expression, is it satisfiable?
- Very basic problem in computer science

\[ p_1 \land (p_2 \rightarrow p_3) \land ((\neg p_1 \land \neg p_3) \rightarrow \neg p_2) \land (p_1 \lor p_3) \]

- Turns out you can always express in 3-CNF

\[ (p_1) \land (\neg p_2 \lor p_3) \land (p_1 \lor p_3 \lor \neg p_2) \land (p_1 \lor p_2 \lor p_3) \]

- 3-SAT: find a satisfying truth assignment
Example: 3-SAT

- **Variables:** \( p_1, p_2, \ldots, p_n \)
- **Domains:** \{true, false\}
- **Constraints:**
  \[
  p_i \lor p_j \lor p_k \\
  \neg p_i' \lor p_j' \lor p_k' \\
  \vdots \\
  p_i'' \lor \neg p_j'' \lor \neg p_k''
  \]
  Implicitly conjoined (all clauses must be satisfied)
Varieties of CSPs

- **Discrete Variables**
  - Finite domains
    - Size $d$ means $O(d^n)$ complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable

- **Continuous variables**
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by LP methods
Varieties of Constraints

- **Varieties of Constraints**
  - Unary constraints involve a single variable (equiv. to shrinking domains):
    \[ SA \neq \text{green} \]
  - Binary constraints involve pairs of variables:
    \[ SA \neq WA \]
  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints

- **Preferences (soft constraints):**
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these until we get to Bayes’ nets)
Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Floorplanning
- Fault diagnosis
- ... lots more!

- Many real-world problems involve real-valued variables...
Backtracking Example
Improving Backtracking

- General-purpose ideas can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
  - Can we take advantage of problem structure?
Summary

- CSPs are a special kind of search problem:
  - States defined by values of a fixed set of variables
  - Goal test defined by constraints on variable values

- Backtracking = depth-first search with incremental constraint checks
- Ordering: variable and value choice heuristics help significantly
- Filtering: forward checking, arc consistency prevent assignments that guarantee later failure
- Structure: Disconnected and tree-structured CSPs are efficient

- Iterative improvement: min-conflicts is usually effective in practice
Some Hard Questions…

- Who is liable if a robot driver has an accident?
- Will machines surpass human intelligence?
- What will we do with superintelligent machines?
- Would such machines have conscious existence? Rights?
- Can human minds exist indefinitely within machines (in principle)?