CS 188: Artificial Intelligence

Midterm Review

Instructors: Saagar Sanghavi – UC Berkeley

(Slides Credit: Dan Klein, Pieter Abbeel, Anca Dragan, Stuart Russell, Satish Rao, Ketrina Yim, and many others)
Midterm: Topics in Scope

- Utilities and Rationality, MEU Principle
- Search and Planning
- Constraint Satisfaction Programming
- Game Trees, Minimax, Pruning, Expectimax
- Probabilistic Inference, Bayesian Networks, Variable Elimination, D-Separation, Sampling
- Markov Models, HMMs, Viterbi Algorithm, Particle Filtering, Dynamic Bayes Nets
An agent *perceives* its environment through *sensors* and *acts* upon it through *actuators* (or *effectors*, depending on whom you ask).

The *agent function* maps percept sequences to actions.

It is generated by an *agent program* running on a *machine*. 
The task environment - PEAS

○ Performance measure
  ○ -1 per step; + 10 food; +500 win; -500 die; +200 hit scared ghost

○ Environment
  ○ Pacman dynamics (incl ghost behavior)

○ Actuators
  ○ Left Right Up Down or NSEW

○ Sensors
  ○ Entire state is visible (except power pellet duration)
Agent design

○ The environment type largely determines the agent design
  ○ *Partially observable* => agent requires *memory* (internal state)
  ○ *Stochastic* => agent may have to prepare for *contingencies*
  ○ *Multi-agent* => agent may need to behave *randomly*
  ○ *Static* => agent has time to compute a rational decision
  ○ *Continuous time* => continuously operating *controller*
  ○ *Unknown physics* => need for *exploration*
  ○ *Unknown perf. measure* => observe/interact with *human principal*
Utilities and Rationality

- **Utility**: map state of world to real value
- **Rational Preferences**

  - **Orderability**: \((A > B) \lor (B > A) \lor (A \sim B)\)
  - **Transitivity**: \((A > B) \land (B > C) \Rightarrow (A > C)\)
  - **Continuity**: \((A > B > C) \Rightarrow \exists p \ [p, A; 1-p, C] \sim B\)
  - **Substitutability**: \((A \sim B) \Rightarrow [p, A; 1-p, C] \sim [p, B; 1-p, C]\)
  - **Monotonicity**: \((A > B) \Rightarrow (p \geq q) \iff [p, A; 1-p, B] \geq [q, A; 1-q, B]\)

Given Rational Preferences, Exists \(U(X)\) s.t.

\[
U(A) \geq U(B) \iff A \geq B
\]

\[
U([p_1, S_1; \ldots; p_n, S_n]) = p_1 U(S_1) + \ldots + p_n U(S_n)
\]
Maximize Your Expected Utility
Search Problems
Search Problems

- **A search problem** consists of:
  - A state space
  - A successor function (with actions, costs)
  - A start state and a goal test

- **A solution** is a sequence of actions (a plan) which transforms the start state to a goal state

```
"N", 1.0

"E", 1.0
```
State Space Graphs vs. Search Trees

Each NODE in the search tree is an entire PATH in the state space graph.

We construct only what we need on demand.
General Tree Search

function Tree-Search( problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree
end

○ Important ideas:
  ○ Fringe
  ○ Expansion
  ○ Exploration strategy

○ Main question: which fringe nodes to explore?
Depth-First Search
Depth-First Search

Strategy: expand a deepest node first

Implementation:
Fringe is a LIFO stack
Breadth-First Search
Breadth-First Search

**Strategy:** expand a shallowest node first

**Implementation:**
Fringe is a FIFO queue
Cost-Sensitive Search

BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.
Uniform Cost Search
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)
Search Heuristics

- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Pathing?
  - Examples: Manhattan distance, Euclidean distance
Greedy Search
Greedy Search

- Expand the node that seems closest...
  - Move to smallest heuristic value

- Is it optimal?
  - No. Resulting path to Bucharest is not the shortest!
A* Search
Combining UCS and Greedy

- **Uniform-cost** orders by path cost, or *backward cost*  $g(n)$
- **Greedy** orders by goal proximity, or *forward cost*  $h(n)$

- **A* Search** orders by the sum: $f(n) = g(n) + h(n)$

Example: Teg Grenager
When should A* terminate?

- Should we stop when we enqueue a goal?

  - No: only stop when we dequeue a goal

![Diagram of A* algorithm](image)
Admissible Heuristics

- A heuristic $h$ is *admissible* (optimistic) iff:

  $$0 \leq h(n) \leq h^*(n)$$

  where $h^*(n)$ is the true cost to a nearest goal

- **Examples:**

  ![Diagrams showing examples of admissible heuristics]

- Coming up with admissible heuristics is most of what’s involved in using A* in practice.
Creating Admissible Heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics.

- Often, admissible heuristics are solutions to relaxed problems, where new actions are available.

- Inadmissible heuristics are often useful too.
Graph Search
function Graph-Search(problem, fringe) return a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      for child-node in EXPAND(STATE[node], problem) do
        fringe ← INSERT(child-node, fringe)
      end
  end
Consistency of Heuristics

- Main idea: estimated heuristic costs $\leq$ actual costs
  - Admissibility: heuristic cost $\leq$ actual cost to goal
    \[ h(v) \leq h^*(v) \text{ for all } v \in V \]
    Underestimate the true cost to the goal!
  - Consistency: heuristic “arc” cost $\leq$ actual cost for each arc
    \[ h(u) - h(v) \leq d(u, v) \text{ for all } (u, v) \in E \]
    Underestimate the weight of every edge!

- Consequences of consistency:
  - The $f$ value along a path never decreases
    \[ h(A) \leq \text{cost}(A \text{ to } C) + h(C) \]
  - A* graph search is optimal
Optimality of A* Search

- With a admissible heuristic, Tree A* is optimal.
- With a consistent heuristic, Graph A* is optimal.
  - With h=0, the same proof shows that UCS is optimal.
Constraint Satisfaction Problems
Constraint Satisfaction Problems

- $N$ variables
- domain $D$
- constraints

- states
  - partial assignment

- goal test
  - complete; satisfies constraints

- successor function
  - assign an unassigned variable
Standard Search Formulation

- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {} 
  - Successor function: assign a value to an unassigned variable 
  - Goal test: the current assignment is complete and satisfies all constraints 

- We’ll start with the straightforward, naïve approach, then improve it
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - “Incremental goal test”
- Depth-first search with these two improvements is called backtracking search (not the best name)
- Can solve n-queens for $n \approx 25$
Backtracking Example
Backtracking Search

Backtracking = DFS + variable-ordering + fail-on-violation

What are the choice points?

function BACKTRACKING-SEARCH(csp) returns solution/failure
    return Recursive-Backtracking({}, csp)

function Recursive-Backtracking(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(VARIABLES[csp], assignment, csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← Recursive-Backtracking(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
        return failure
Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options.
- Forward checking: Cross off values that violate a constraint when added to the existing assignment.

[Demo: coloring -- forward checking]
Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation*: reason from constraint to constraint
Consistency of A Single Arc

An arc $X \rightarrow Y$ is **consistent** iff for *every* $x$ in the tail there is *some* $y$ in the head which could be assigned without violating a constraint.

Forward checking?
Enforcing consistency of arcs pointing to each new assignment

Delete from the tail!
Enforcing Arc Consistency in a CSP

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
        for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
removed ← false
for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraint X_i ← X_j
        then delete x from DOMAIN[X_i]; removed ← true
return removed
```

○ Runtime: O(n^2d^3), can be reduced to O(n^2d^2)
○ … but detecting all possible future problems is NP-hard – why?
K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node’s domain has a value which meets that node’s unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k\textsuperscript{th} node.

- Higher k more expensive to compute

- (You need to know the k=2 case: arc consistency)
Strong K-Consistency

- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …
- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)
Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering
Value Ordering: Least Constraining Value

- Given a choice of variable, choose the least constraining value
- I.e., the one that rules out the fewest values in the remaining variables
- Note that it may take some computation to determine this! (E.g., rerunning filtering)

Why least rather than most?

Combining these ordering ideas makes 1000 queens feasible
Iterative Algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned.

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.

- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with $h(x) = \text{total number of violated constraints}$
Hill Climbing

- Objective function
- Global maximum
- Shoulder
- Local maximum
- "Flat" local maximum
- Current state
- State space
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children
  - Remove backward: For $i = n : 2$, apply $\text{RemoveInconsistent}(\text{Parent}(X_i), X_i)$
  - Assign forward: For $i = 1 : n$, assign $X_i$ consistently with $\text{Parent}(X_i)$

- Runtime: $O(n d^2)$ (why?)
Game Playing: Search with other agents
Adversarial Search
Adversarial Game Trees

Value of a state: The best outcome from that state
Minimax Values

States Under Agent’s Control:

\[ V(s) = \max_{s' \in \text{successors}(s)} V(s') \]

States Under Opponent’s Control:

\[ V(s') = \min_{s \in \text{successors}(s')} V(s) \]

Terminal States:

\[ V(s) = \text{known} \]
Minimax Implementation (Dispatch)

def value(state):
    if the state is terminal: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
Game Tree Pruning
Alpha-Beta Implementation

\( \alpha \): MAX’s best option on path to root
\( \beta \): MIN’s best option on path to root

**def max-value(state, \( \alpha \), \( \beta \)):**
- initialize \( v = -\infty \)
- for each successor of state:
  - \( v = \max(v, \text{value}\text{(successor, } \alpha \text{, } \beta)) \)
  - if \( v \geq \beta \) return \( v \)
  - \( \alpha = \max(\alpha, v) \)
- return \( v \)

**def min-value(state, \( \alpha \), \( \beta \)):**
- initialize \( v = +\infty \)
- for each successor of state:
  - \( v = \min(v, \text{value}\text{(successor, } \alpha \text{, } \beta)) \)
  - if \( v \leq \alpha \) return \( v \)
  - \( \beta = \min(\beta, v) \)
- return \( v \)
Alpha-Beta Example
Multi-Agent Utilities

- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…

1,6,6

1,6,6 7,1,2 6,1,2 7,2,1 5,1,7 1,5,2 7,7,1 5,2,5
Chance Nodes

- We don’t know what the result of an action will be:
  - Explicit randomness: rolling dice
  - Unpredictable opponents
  - Actions can fail

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes: calculate expected utilities
def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
Bayesian Networks
Conditional Probabilities

- Bayes Rule

\[ P(a|b) = \frac{P(a,b)}{P(b)} \]

\[
\begin{array}{|c|c|c|}
\hline
T & W & P \\
\hline
\text{hot} & \text{sun} & 0.4 \\
\text{hot} & \text{rain} & 0.1 \\
\text{cold} & \text{sun} & 0.2 \\
\text{cold} & \text{rain} & 0.3 \\
\hline
\end{array}
\]

\[
P(W = s | T = c) = \frac{P(W = s, T = c)}{P(T = c)} = \frac{0.2}{0.5} = 0.4
\]

\[
= P(W = s, T = c) + P(W = r, T = c) = 0.2 + 0.3 = 0.5
\]
Conditional Independence

- $X$ and $Y$ are **independent** iff
  \[ \forall x, y \ P(x, y) = P(x)P(y) \quad \Rightarrow \quad X \perp Y \]

- Given $Z$, we say $X$ and $Y$ are **conditionally independent** iff
  \[ \forall x, y, z \ P(x, y|z) = P(x|z)P(y|z) \quad \Rightarrow \quad X \perp Y|Z \]

- (Conditional) independence is a property of a distribution
Bayesian Networks

- A directed acyclic graph (DAG), one node per random variable
- A conditional probability table (CPT) for each node
  - Probability of $X$, given a combination of values for parents.
    \[ P(X|a_1 \ldots a_n) \]
- Bayes nets implicitly encode joint distributions as a product of local conditional distributions
  - To see what probability a BN gives to a full assignment, multiply all the relevant conditionals together:
    \[ P(x_1, x_2, \ldots x_n) = \prod_{i=1}^{n} P(x_i|\text{parents}(X_i)) \]
Independence Assumptions

- Definition: Each node, given its parents, is conditionally independent of all its non-descendants in the graph.

  MarkovBlanket refers to the parents, children, and children's other parents.

- Each node, given its MarkovBlanket, is conditionally independent of all other nodes in the graph.

  MarkovBlanket refers to the parents, children, and children's other parents.
Inference by Enumeration

- General case:
  - Evidence variables: \( E_1 \ldots E_k = e_1 \ldots e_k \)
  - Query variable: \( Q \)
  - Hidden variables: \( H_1 \ldots H_r \)

- We want:
  \[
P(Q|e_1 \ldots e_k)
  \]

- Step 1: Select the entries consistent with the evidence

- Step 2: Sum out \( H \) to get joint of Query and evidence

- Step 3: Normalize

\[
P(Q|e_1 \ldots e_k) = \frac{1}{Z} \sum_{h_1 \ldots h_r} P(Q, h_1 \ldots h_r, e_1 \ldots e_k | X_1, X_2, \ldots X_n)
\]

\[
Z = \sum_q P(Q, e_1 \ldots e_k)
\]
Inference on Bayes Nets

\[ P(L) = ? \]

- **Inference by Enumeration**
  \[ = \sum_t \sum_r P(L|t)P(r)P(t|r) \]

- **Variable Elimination**
  \[ = \sum_t P(L|t) \sum_r P(r)P(t|r) \]
Marginalizing Early (Variable Elimination)
Variable Elimination

\[ P(R) \]
<table>
<thead>
<tr>
<th>T</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>+r</td>
<td>0.1</td>
</tr>
<tr>
<td>-r</td>
<td>0.9</td>
</tr>
</tbody>
</table>

\[ P(T|R) \]
| +r | +t | 0.8 |
| +r | -t | 0.2 |
| -r | +t | 0.1 |
| -r | -t | 0.9 |

Join R

\[ P(R,T) \]
| +r | +t | 0.08 |
| +r | -t | 0.02 |
| -r | +t | 0.09 |
| -r | -t | 0.81 |

Sum out R

\[ P(L|T) \]
| +t | +l | 0.3 |
| +t | -l | 0.7 |
| -t | +l | 0.1 |
| -t | -l | 0.9 |

Join T

\[ P(T,L) \]
| +t | +l | 0.051 |
| +t | -l | 0.119 |
| -t | +l | 0.083 |
| -t | -l | 0.747 |

Sum out T
General Variable Elimination

- Query: \[ P(Q|E_1 = e_1, \ldots, E_k = e_k) \]

- Start with initial factors:
  - Local CPTs (but instantiated by evidence)

- While there are still hidden variables (not Q or evidence):
  - Pick a hidden variable H
  - Join all factors mentioning H
  - Eliminate (sum out) H

- Join all remaining factors and normalize
Independence Assumptions in a Bayes Net

Assumptions we are required to make to define the Bayes net when given the graph:

\[ P(x_i|x_1 \cdots x_{i-1}) = P(x_i|\text{parents}(X_i)) \]

Important for modeling: understand assumptions made when choosing a Bayes net graph
Active / Inactive Paths

- **Question:** Are X and Y conditionally independent given evidence variables \( \{Z\} \)?
  - Yes, if X and Y “d-separated” by Z
  - Consider all (undirected) paths from X to Y
  - No active paths = independence!

- **A path is active if each triple is active:**
  - Causal chain \( A \rightarrow B \rightarrow C \) where B is unobserved (either direction)
  - Common cause \( A \leftarrow B \rightarrow C \) where B is unobserved
  - Common effect (aka v-structure) \( A \rightarrow B \leftarrow C \) where B or one of its descendants is observed

- **All it takes to block a path is a single inactive segment**
D-Separation

- Query: \( X_i \perp\!\!\!\!\!\!\!\!\!\perp X_j \mid \{X_{k_1}, \ldots, X_{k_n}\} \) ?

- Check all (undirected!) paths between \( X_i \) and \( X_j \)
  - If one or more active paths, then independence not guaranteed
    \( X_i \nabla X_j \mid \{X_{k_1}, \ldots, X_{k_n}\} \)
  - Otherwise (i.e. if all paths are inactive),
    then independence is guaranteed
    \( X_i \perp\!\!\!\!\!\!\!\!\!\perp X_j \mid \{X_{k_1}, \ldots, X_{k_n}\} \)
Another Perspective: Bayes Ball

An undirected path is active if a Bayes ball travelling along it never encounters the “stop” symbol: \[ \rightarrow \leftarrow \]

If there are no active paths from \( X \) to \( Y \) when \( \{Z_1, \ldots, Z_k\} \) are shaded, then \( X \perp Y | \{Z_1, \ldots, Z_k\} \).
Topology Limits Distributions

- Given some graph topology $G$, only certain joint distributions can be encoded.
- The graph structure guarantees certain (conditional) independences.
- (There might be more independence)
- Adding arcs increases the set of distributions, but has several costs.
- Full conditioning can encode any distribution.
Approximate Inference: Sampling
Prior Sampling

- For $i = 1, 2, \ldots, n$ in topological order
  - Sample $x_i$ from $P(X_i | \text{Parents}(X_i))$
- Return $(x_1, x_2, \ldots, x_n)$
Rejection Sampling

- Input: evidence instantiation
- For $i = 1, 2, \ldots, n$ in topological order
  - Sample $x_i$ from $P(X_i | \text{Parents}(X_i))$
  - If $x_i$ not consistent with evidence
    - Reject: return – no sample is generated in this cycle
  - Return $(x_1, x_2, \ldots, x_n)$
Likelihood Weighting

- Input: evidence instantiation
- \( w = 1.0 \)
- for \( i = 1, 2, \ldots, n \) in topological order
  - if \( X_i \) is an evidence variable
    - \( X_i = \) observation \( x_i \) for \( X_i \)
    - Set \( w = w \times P(x_i | \text{Parents}(X_i)) \)
  - else
    - Sample \( x_i \) from \( P(X_i | \text{Parents}(X_i)) \)
- return \((x_1, x_2, \ldots, x_n), w\)
Gibbs Sampling

- Step 1: Fix evidence
  - $R = +r$
- Step 2: Initialize other variables
  - Randomly

- Steps 3: Repeat:
  - Choose a non-evidence variable $X$
  - Resample $X$ from $P(\ X | \ MarkovBlanket(X))$

Sample from $P(S| +c, -w, +r)$
Sample from $P(C| +s, -w, +r)$
Sample from $P(W| +s, +c, +r)$
Hidden Markov Models
Markov Chains (Review from EE 16A, CS 70)

- Value of $X$ at a given time is called the state

<table>
<thead>
<tr>
<th>$P(X_0)$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sun</td>
<td>rain</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

$P(X_1)$ $P(X_t | X_{t-1})$

| $X_{t-1}$ | $X_t$ | $P(X_t | X_{t-1})$ |
|-----------|-------|-------------------|
| sun       | sun   | 0.9               |
| sun       | rain  | 0.1               |
| rain      | sun   | 0.3               |
| rain      | rain  | 0.7               |

State Transition Diagram (Flow Graph)

State Trellis
Mini-Forward Algorithm

- Question: What’s P(X) on some day t?

\[ P(x_1) = \text{known} \]

\[ P(x_t) = \sum_{x_{t-1}} P(x_{t-1}, x_t) = \sum_{x_{t-1}} P(x_t \mid x_{t-1})P(x_{t-1}) \]

Forward simulation
Stationary Distribution

- **For most chains:**
  - Influence of the initial distribution gets less and less over time.
  - The distribution we end up in is independent of the initial distribution

- **Stationary distribution:**
  - The distribution we end up with is called the stationary distribution $P_\infty$ of the chain
  - It satisfies
    \[
    P_\infty(X) = P_{\infty+1}(X) = \sum_x P(X|x)P_\infty(x)
    \]
Hidden Markov Models

- Markov chains not so useful for most agents
  - Need observations to update your beliefs

- Hidden Markov models (HMMs)
  - Underlying Markov chain over states $X_i$
  - You observe outputs (effects) at each time step
Inference tasks

Prediction: $P(X_{t+k} | e_{1:t})$

Filtering: $P(X_t | e_{1:t})$

Smoothing: $P(X_k | e_{1:t}), k < t$

Explanation: $P(X_{1:t} | e_{1:t})$
Inference: Find State Given Evidence

- We are given evidence at each time and want to know $P(X_t|e_{1:t})$
- Idea: start with $P(X_1)$ and derive $P(X_t | e_{1:t})$ in terms of $P(X_{t-1} | e_{1:t-1})$
- Two steps: Passage of time + Incorporate Evidence

\[
P(X_{t+1} | e_{1:t})
\]

\[
P(X_{t+1} | e_{1:t+1})
\]
Forward Algorithm

- Every time step, we start with current $P(X \mid \text{evidence})$
- We update for time:

$$P(x_t|e_{1:t-1}) = \sum_{x_{t-1}} P(x_{t-1}|e_{1:t-1}) \cdot P(x_t|x_{t-1})$$

- We update for evidence:

$$P(x_t|e_{1:t}) \propto_x P(x_t|e_{1:t-1}) \cdot P(e_t|x_t)$$

- The forward algorithm does both at once
Most likely explanation = most probable path

- **State trellis**: graph of states and transitions over time

  ![State Trellis Diagram]

  \[ X_1 \quad X_2 \quad \ldots \quad X_T \]

- Each arc represents some transition \( X_{t-1} \rightarrow X_t \)
- Each arc has weight \( P(x_t \mid x_{t-1}) \cdot P(e_t \mid x_t) \) (arcs to initial states have weight \( P(x_0) \))
- The **product** of weights on a path is proportional to that state seq’s probability
- Forward algorithm: sums of paths
- **Viterbi algorithm**: best paths
  - Dynamic Programming: solve subproblems, combine them as you go along
Forward / Viterbi Algorithms

Forward Algorithm (Sum)
For each state at time $t$, keep track of the total probability of all paths to it

$$f_t[x_t] = P(x_t, e_{1:t})$$
$$= P(e_t|x_t) \sum_{x_{t-1}} P(x_t|x_{t-1}) f_{t-1}[x_{t-1}]$$

Viterbi Algorithm (Max)
For each state at time $t$, keep track of the maximum probability of any path to it

$$m_t[x_t] = \max_{x_{1:t-1}} P(x_{1:t-1}, x_t, e_{1:t})$$
$$= P(e_t|x_t) \max_{x_{t-1}} P(x_t|x_{t-1}) m_{t-1}[x_{t-1}]$$
Viterbi Algorithm Pseudocode

```
function VITERBI(O, S, Π, Y, A, B) : X

    for each state i = 1, 2, ..., K do
        T₁[i, 1] ← πᵢ · Bᵢy₁
        T₂[i, 1] ← 0
    end for

    for each observation j = 2, 3, ..., T do
        for each state i = 1, 2, ..., K do
            T₁[i, j] ← max̄(T₁[k, j - 1] · Aᵦ · Bᵦyⱼ)
            T₂[i, j] ← arg max̄(T₁[k, j - 1] · Aᵦ · Bᵦyⱼ)
        end for
    end for

    zₜ ← arg max̄(T₁[k, T])
    xₜ ← sₜzₜ

    for j = T, T - 1, ..., 2 do
        zⱼ₋₁ ← T₂[zⱼ, j]
        xⱼ₋₁ ← sⱼ₋₁
    end for

    return X
end function
```

Observation Space
\( O = \{o_1, o_2, \ldots, o_N\} \)

State Space
\( S = \{s_1, s_2, \ldots, s_K\} \)

Initial probabilities
\( Π = (\pi_1, \pi_2, \ldots, \pi_K) \)

Observations
\( Y = (y_1, y_2, \ldots, y_T) \)

Transition Matrix
\( A \in \mathbb{R}^{K \times K} \)

Emission Matrix
\( B \in \mathbb{R}^{K \times N} \)

Matrix \( T₁[i, j] \) stores probabilities of most likely path so far with \( x_j = s_i \)

Matrix \( T₂[i, j] \) stores \( x_{j-1} \) of most likely path so far with \( x_i = s_i \)
Particle Filtering: Approximate Inference on HMMs

- Particles: track samples of states rather than an explicit distribution
Dynamic Bayes Nets (DBNs)

- We want to track multiple variables over time, using multiple sources of evidence
- Idea: Repeat a fixed Bayes net structure at each time
- Variables from time $t$ can condition on those from $t-1$

Dynamic Bayes nets are a generalization of HMMs