Lecture 14

Midterm Review

CS 520: Intro AI

Jingjin Yu | Rutgers
Uninformed and Informed Search
Components of a Search Problem

- **State space** $S$: in this case, an edge-weighted graph

- **Initial** (start) and **goal** (final) states: $x_I$ and $x_G$
  - There can be more than one start/goal state: solve one side of a Rubik’s cube

- **Action**: in this case, moving from one state to a nearby state

- **Transition model**: tuples $(s_1, a, s_2)$ that are valid
  - Sometimes written as $T(s_1, a) = s_2$
  - There are usually costs/rewards associated with a transition, $R(s_1, a)$

- **Solution**: valid transitions connecting $x_I$ and $x_G$
  - Optimal solution: solution with lowest cost (e.g., length of the path)
Example: 8-puzzle

⇒ State space: arrangements of the 8 pieces
  ⇒ State space size: $9! = 362880$

⇒ Action/transition: shifting a piece to the empty cell

⇒ Initial state: an arbitrary state

⇒ Goal state: an arbitrary fixed state

⇒ Cost: number of actions (moves)
Tree Search: Structure

input: $S, x_i, G(\cdot)$

AddToQueue($x_i, \text{Queue}$); // Add $x_i$ to a queue of nodes to be expanded

while(!IsEmpty($\text{Queue}$))
    $x \leftarrow \text{Front}(\text{Queue})$; // Retrieve the front of the queue
    if($G(x)$) return solution; // Return if goal is reached
    for each successor $n_i$ of $x$ // Add all neighbors to the queue
        AddToQueue($n_i, \text{Queue}$)

return failure; Expanding a search node

⇒ Different search algorithms use different AddToQueue
    ⇒ BFS: FIFO queue – first in first out
    ⇒ DFS: LIFO queue – last in first out
    ⇒ Uniform-cost: Priority queue – node with smallest cost in the front

⇒ Tree search CANNOT tell whether a node has been seen before
Graph Search

**input:** $S, x_I, G(\cdot)$

AddToQueue $(x_I, Queue)$; // Add $x_I$ to a queue of nodes to be expanded

while (!IsEmpty (Queue))

    $x \leftarrow$ Front (Queue); // Retrieve the front of the queue

    if ($x.expanded == true$) continue; // Do not expand a node twice

    $x.expanded = true$; // Mark $x$ as expanded

    if ($G(x)$) return solution; // Return if goal is reached

    for each successor $n_i$ of $x$ // Add all neighbors of $x$ to the queue

        if ($n_i.expanded == false$) AddToQueue $(n_i, Queue)$

return failure;

- Keeps track of expanded nodes – each node is expanded once
- AddToQueue can be more efficient
  - Recall that tree search cannot tell nodes apart but graph search can
  - For priority queue, only a single copy of $n_i$ with smallest cost needs to be in the queue
BFS Tree Search and Graph Search Example

Problem graph

Running BFS tree and graph search

Tree search

Graph search

More compact search tree!
# Summary on Uninformed (Tree) Search

<table>
<thead>
<tr>
<th>Method</th>
<th>Complete?</th>
<th>Optimal?</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BFS</strong></td>
<td>Yes, for ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^d) )</td>
<td>( O(b^d) )</td>
</tr>
<tr>
<td>Uniform cost</td>
<td>Yes, for ( b &lt; \infty ) and ( \epsilon &gt; \text{fixed} , \delta &gt; 0 )</td>
<td>Yes</td>
<td>( O(b^{1+\frac{c^*}{\epsilon}}) )</td>
<td>( O(b^{1+\frac{c^*}{\epsilon}}) )</td>
</tr>
<tr>
<td><strong>DFS</strong></td>
<td>No</td>
<td>No</td>
<td>( O(b^m) )</td>
<td>( O(bm) )</td>
</tr>
<tr>
<td>Depth limited DFS</td>
<td>No</td>
<td>No</td>
<td>( O(b^\ell) )</td>
<td>( O(b\ell) )</td>
</tr>
<tr>
<td>Iterative deepening DFS</td>
<td>Yes, for ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^d) )</td>
<td>( O(bd) )</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>Yes, for BFS with ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^{\frac{d}{2}}) )</td>
<td>( O(b^{\frac{d}{2}}) )</td>
</tr>
</tbody>
</table>

\( b \): branching factor (average)
\( d \): max. solution depth (i.e., depth of \( x_G \))
\( m \): max. depth of any node from \( x_I \)
\( \ell \): depth limit
\( C^* \): optimal solution cost
\( \epsilon \): smallest step (edge) cost, must be finite

⇒ For graph search on finite state space, DFS is complete and depth limited DFS is complete when \( \ell \geq d \)
Uninformed Search v.s. Informed Search

Uninformed search only uses information from problem definition

Example of uninformed and informed search: maze

Searching for a hole with unknown location in a maze

Searching for an exit at south
Admissible and Consistent Heuristic

⇒ Assume the cheapest path from \( x \) to a goal is \( C_x^* \), an **admissible heuristic** satisfies

\[
h(x) \leq C_x^*
\]

⇒ A **consistent** heuristic is defined as (a form of triangle inequality)

\[
h(n) \leq c(n, a, n') + h(n') \text{ and } h(x_G) = 0
\]

⇒ A consistent heuristic is always admissible

⇒ The reverse is not always true

⇒ Example of heuristic functions

⇒ Manhattan distance

⇒ Straight-line distance
Greedy Best-First Search

- Greedy best-first search uses $f(x) = h(x)$
- Example: path from S to G, tree/graph search (same for this example)

<table>
<thead>
<tr>
<th>State</th>
<th>$h(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>

Source: Berkeley CS188
A* Best-First Search

\[ A^* \text{ uses } f(x) = g(x) + h(x) \text{ (cost-to-come + estimated cost-to-go)} \]

\[ \text{Example: path from S to G, tree search} \]

Source: Berkeley CS188
**A* Graph Search with Admissible Heuristic**

⇒ Same example: path from S to G, *graph search*

### State vs. Heuristic Value

<table>
<thead>
<tr>
<th>State</th>
<th>(h(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>

\[h(x)\] is inconsistent, e.g., \(h(S) > c(S, B) + h(B),\) \(h(A) > c(A, B) + h(B)\)

Source: Berkeley CS188
A* Graph Search with Consistent Heuristic

→ Updated example: path from S to G, graph search

<table>
<thead>
<tr>
<th>State</th>
<th>$h(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>

Source: Berkeley CS188
Local Search
The Hill-Climbing Algorithm

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   current ← MAKE-NODE(problem.INITIAL-STATE)
   loop do
      neighbor ← a highest-valued successor of current
      if neighbor.VALUE ≤ current.VALUE then return current.STATE
      current ← neighbor
   end loop
```

Hill-Climbing Example: Eight-Queens

- One way to construct a cost function (so we are looking for minimum) is to **count the attacking pairs**

For the given setup, \( C = 3 + 4 + 2 + 3 + 2 + 2 + 1 = 17 \)

- The rest of the cost are computed **if a queen is moved in her own column**

- Best local move has cost of 12
Hill-Climbing Example: Eight-Queens

⇒ After 5 hill-climbing moves, \( C = 1 \)

⇒ Now all moves increase \( C \)
  ⇒ We are stuck in a (not too bad) local minimum
Simulated Annealing (SA)

- Similar to hill-climbing but allow “bad moves” with small probability
- Annealing: cooling hot metal slowly to achieve desired property

```plaintext
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
         schedule, a mapping from time to “temperature”

current ← MAKE-NODE(problem.INITIAL-STATE)
for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.VALUE - current.VALUE
    if ΔE > 0 then current ← next
    else current ← next only with probability e^{ΔE/T}

Gradually lowering “temperature”
```

Small probability of picking a “bad move”
Genetic Algorithms (GA)

- Similar to local beam search
- As the name suggests, inspired by evolution through selection
- Key components: population of individuals, fitness function, crossover, mutation

- Example: 8-queens
  - Individual = state, with $8 \times \log_2{8} = 24$ bits
  - Fitness: number of pairs of queens not attacking, the higher the better
  - Crossover: combine portions of two states to form a valid state
  - Mutation: change bits in the individual with small probability
Genetic Algorithm Example: Eight-Queens

(a) Initial Population
24748552
32752411
24415124
32543213

(b) Fitness Function
24 31%
23 29%
20 26%
11 14%

(c) Selection
32752411

(d) Crossover
32748552
24752411
32752124
24415411

(e) Mutation
32748152
24752411
32252124
24415417

Chessboards:
Initial Board: (Left) 8 Queens placed
Crossed Board: (Middle) 4 Queens moved
Final Board: (Right) 8 Queens in new positions
Constraint Satisfaction Problems (CSPs)
Constraint Satisfaction Problems

† Definition: given a set of variables \(X = \{x_1, \ldots, x_n\}\), with domains \(D = \{D_1, \ldots, D_n\}\), \(x_i \in D_i\), and constraints \(C = \{c_1, \ldots, c_m\}\) that restrict the choice of the variables (e.g., \(x_1 + x_2 < 10\)), find an assignment \(x_1 = v_1, \ldots, x_n = v_n\) that satisfies all the constraints

† Examples: cryptarithmetic, eight-queens, map coloring

† Variables: \(X = \{x_{ij}\}, 1 \leq i, j \leq 8\)

† Domains: \(D = \{D_{ij}\}, D_{ij} = \{0,1\}\)

† Constraints
  † Total number of queens: \(\sum x_{ij} = 8\)
  † No two queens should attack
    † Rows and columns: \(\sum_i x_{ij} \leq 1, \sum_j x_{ij} \leq 1\)
    † Diagonals: \(\sum_{k=1}^8 x_{k,k} \leq 1, \sum_{k=2}^8 x_{k+1,k} \leq 1, \ldots\)
Backtracking Search

- Search with backtracking: essentially DFS with single variable assignment at each node expansion
Basic (Recursive) Backtracking

function \textsc{backtracking-search}(csp) \textbf{returns} a solution, or failure
\textbf{return} \textsc{recursive-backtracking}(\emptyset, csp)

function \textsc{recursive-backtracking}(assignment, csp) \textbf{returns} a solution, or failure
\textbf{if} assignment is complete \textbf{then return} assignment
\textbf{var} \leftarrow \textsc{select-unassigned-variable}(\textsc{variables}[csp], assignment, csp)
\textbf{for each} \textbf{value in} \textsc{order-domain-values}(\textbf{var}, assignment, csp) \textbf{do}
\hspace{1em} \textbf{if} value is consistent with assignment according to \textsc{constraints}[csp] \textbf{then}
\hspace{2em} add \{\textbf{var} = \textbf{value}\} to assignment
\hspace{2em} result \leftarrow \textsc{recursive-backtracking}(assignment, csp)
\hspace{2em} \textbf{if} result \neq \textbf{failure} \textbf{then return} result
\hspace{2em} remove \{\textbf{var} = \textbf{value}\} from assignment
\textbf{return} failure

\Rightarrow We can obtain additional efficiency through
\Rightarrow Carefully choosing the order of variables to assign values
\Rightarrow Carefully choosing the order of values to be assigned to a variable
\Rightarrow If we must fail, then fail early
Adversarial Search
Alternating 2-Player Games

Such games are sequential with the players taking turns.

The game ends with a terminal state with utilities for both players.

Zero-sum games: one player’s utility is the negation of the other player’s utility – hence summed utility is zero.

The process of playing a alternating 2-player game:

- Assuming Max and Min are playing a zero-sum game.
- At a step $k$ of the sequential game, Max wants to maximize her utility.
- At step $k + 1$, Min wants to maximize his utility, which means minimizing Max’s utility.
- The process continues until the game reaches a terminal state.
- This gives us a game (search) tree for the game.
A Two-Ply Game Tree Example

⇒ Suppose Max and Min play according to the following game tree

⇒ The last layer shows the **terminal states** with utilities for Max
⇒ From here, Min **minimizes** the utility
⇒ Max then **maximizes** among Min’s choices
⇒ This is the gist of the **MinMax** algorithm
Alpha-Beta Pruning (Exact) Heuristic

- Visit all terminal states in MinMax is unnecessary
- Keep track of
  - $\alpha$ – the value of the best choice (highest value) so far for Max
  - $\beta$ – the value of the best choice (lowest value) so far for Min
Evaluation function

⇒ Replaces terminal states with evaluation functions
⇒ These functions must be computed quickly
⇒ A common evaluation function

\[ Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) \]

⇒ E.g., for chess, a naïve evaluation function may have \( w_i f_i(s) \) for a game piece \( i \).
  ⇒ \( w_i \) may represent the “power” of the piece, e.g., \( w_{\text{pawn}} = 1, w_{\text{queen}} = 10 \)
  ⇒ \( f_i \) may represent the current advantage of the piece. E.g., a pawn near opponent’s base may have much larger \( f \) than a starting pawn
Game Theory Basics
Simultaneous Move Game in Normal Form

- Set of players \( N = \{1, \ldots, n\} \),

- for two players, \( n = 2 \)

- Player \( 1 \leq i \leq n \) has a set of actions \( A^i \) available

  - \( A^i \) may potentially be infinite

  - The set of all choices are \( A = A^1 \times \cdots \times A^n \)

- For a set of choices from all players, \( \tilde{a} \in A \), there is a payoff vector \( \tilde{u} \)

  - More generally, \( R: A \rightarrow \mathbb{R}^n \)

- This gives us a payoff matrix (in the finite case)

- E.g.

  \[
  \begin{pmatrix}
  \text{Rock} & \text{Scissors} & \text{Paper} \\
  \text{Rock} & 0,0 & -1,1 & 1,-1 \\
  \text{Scissors} & 1,-1 & 0,0 & -1,1 \\
  \text{Paper} & -1,1 & 1,-1 & 0,0 
  \end{pmatrix}
  \]
Dominant Strategy

- **Dominant strategy**: a strategy that always yields better outcome for a player regardless of the fixed choice made by other players.

In prisoners’ dilemma, confess is a dominant strategy for P1.

Such strategies avoid “surprise” or regret.

Not necessarily the best overall outcome.

Clearly, (silent, silent) is the best overall choice here.

A **strictly dominant strategy** is a dominant strategy that does strictly better than any other strategy for a player.

<table>
<thead>
<tr>
<th></th>
<th>P1: Prisoner 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confess</td>
<td>Remain silent</td>
</tr>
<tr>
<td>Confess</td>
<td>-5,-5</td>
<td>-20,0</td>
</tr>
<tr>
<td>Remain silent</td>
<td>0,-20</td>
<td>-1,-1</td>
</tr>
</tbody>
</table>

P2: Prisoner 2
**Nash Equilibrium**

- **Nash equilibrium**: a strategy with which no player has anything to gain by changing his/her own action alone

<table>
<thead>
<tr>
<th>P1: Prisoner 1</th>
<th>Confess</th>
<th>Remain silent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Confess</strong></td>
<td>-5, -5</td>
<td>-20, 0</td>
</tr>
<tr>
<td><strong>Remain silent</strong></td>
<td>0, -20</td>
<td>-1, -1</td>
</tr>
</tbody>
</table>

- What are the Nash equilibriums for the players?
  - (-5, -5) is an Nash equilibrium
    - If P1 switches to being silent while P2 confesses, P2 gets 20 years
    - Same for P2 since the payoff matrix is symmetric
    - In fact, dominant strategies in a symmetric payoff matrix is always an equilibrium point

- Do we have another Nash equilibrium?
  - Nash equilibrium is the **best decision** a player can make taking into account of other players actions
Propositional Logic
Components of a Logic System

⇒ **Syntax** defines what are valid sentences
  ⇒ E.g., $x4y + =$ is not a valid equation

⇒ **Semantics** give valid sentences meanings
  ⇒ E.g., certain political candidates this year are orgulous
  ⇒ If you do not know what “orgulous” means, the sentence is useless
  ⇒ In logic, this is done by assigning sentences to be *true* or *false*

⇒ **Model** and **possible worlds** – the truth value of a sentence is not absolute. A particular set of assignments of truth values to sentences form a possible world or a model

⇒ **Entailment** (⇒): $\alpha \models \beta$ says that $\beta$ follows logically from $\alpha$

⇒ **Soundness**: an inference algorithm is sound if it only derives entailed sentences

⇒ **Completeness**: an inference algorithm is complete if all entailed sentences can be derived using the algorithm
### Propositional Logic Syntax

- **Syntax of propositional logic**

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Symbol(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Sentence</code></td>
<td>( \rightarrow ) AtomicSentence ( \mid ) ComplexSentence</td>
</tr>
<tr>
<td>AtomicSentence</td>
<td>( \rightarrow ) True ( \mid ) False ( \mid ) ( P \mid Q \mid R \mid \ldots )</td>
</tr>
<tr>
<td>ComplexSentence</td>
<td>( \rightarrow ) ( ( \text{Sentence} ) \mid [ \text{Sentence} ] \mid \neg \text{Sentence} \mid \text{Sentence} \land \text{Sentence} \mid \text{Sentence} \lor \text{Sentence} \mid \text{Sentence} \implies \text{Sentence} \mid \text{Sentence} \iff \text{Sentence} )</td>
</tr>
</tbody>
</table>

- **A sentence can be an atomic sentence or a complex sentence**

- **E.g.** \((P \lor Q) \implies (R \lor S)\)
  - **Propositions** \(P, Q, R, S\) are atomic sentences
  - **\((P \lor Q)\)** is a complex sentence
  - **So is** \((R \lor S)\) and **\((P \lor Q) \implies (R \lor S)\)**
Propositional Logic Semantics

⇒ Semantics tells us how to interpret a sentence
⇒ For propositional logic (a world or model is always assumed)
  ⇒ *true* is always true and *false* is always false
  ⇒ Other propositions, e.g., *P*, *Q*, *R*, must be given true or false values
  ⇒ For complex sentences
    ⇒ \( \neg P \) is true iff (reads if and only if) \( P \) is false
    ⇒ \( P \land Q \) is true iff both \( P \) and \( Q \) are true
    ⇒ \( P \lor Q \) is true iff either \( P \) or \( Q \) is true
    ⇒ \( P \Rightarrow Q \) is true unless \( P \) is true and \( Q \) is false
    ⇒ \( P \iff Q \) is true iff \( P \) and \( Q \) are both true or both false
  ⇒ In truth table form

<table>
<thead>
<tr>
<th>( P )</th>
<th>( Q )</th>
<th>( \neg P )</th>
<th>( P \land Q )</th>
<th>( P \lor Q )</th>
<th>( P \Rightarrow Q )</th>
<th>( P \iff Q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
Inference in Propositional Logic

⇒ Inference rules

⇒ **Modus Ponens**: if \( \alpha \) is true and \( \alpha \Rightarrow \beta \), then \( \beta \) is true

\[
\frac{\alpha \Rightarrow \beta, \alpha}{\beta}
\]

⇒ **And-Elimination**: if \( \alpha \land \beta \) is true, then \( \alpha \) (as well as \( \beta \)) must be true

\[
\frac{\alpha \land \beta}{\alpha}
\]

⇒ More complex rules can also be composed, e.g., **bidirectional elimination**:

\[
\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)}
\]

\[
\frac{(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}
\]

⇒ **Unit resolution**

\[
\frac{\ell_1 \lor \cdots \lor \ell_n \lor \neg \ell_m}{\ell_1 \lor \cdots \ell_{m-1} \lor \ell_{m+1} \lor \cdots \lor \ell_n}
\]

⇒ Example: derive \( C \) from \( A \Leftrightarrow B, \neg B \lor C, A \)
Probability Basics
We study **probability** because it provides a principled way for **summarizing uncertainty**

- Uncertainty arises from (inherent) laziness or ignorance
- Has objective (frequentism) and subjective interpretations

We may then reason based on such summarization

**Probability space** \( (\Omega, \mathcal{F}, P) \)

- **\( \Omega \)**: **sample space** – the set of all possible outcomes
  - Mutually exclusive, atomic events
- **\( \mathcal{F} \)**: **event space** – each event has zero or more outcomes
  - For discrete \( \Omega \), \( |\mathcal{F}| = 2^{|\Omega|} \), \( \emptyset \in \mathcal{F}, \Omega \in \mathcal{F} \)
- **\( P \)**: **probability measure**, which assigns probabilities to events
  - Note that \( P(\emptyset) = 0, P(\Omega) = 1 \)
  - For \( \phi \in \mathcal{F}, P(\phi) = \sum_{\omega \in \phi} P(\{\omega\}) \)

E.g., tossing two fair coins

- \( \Omega = \{HH, HT, TH, TT\} \)
- \( P(\{HH\}) = P(\{HT\}) = P(\{TH\}) = P(\{TT\}) = 0.25 \)
Venn Diagram & Inclusion-Exclusion

⇒ **Venn diagram**: a diagram showing the relationship among event sets

![Venn Diagram](image)

⇒ **The inclusion-exclusion principle**

⇒ For two events

\[ P(A \cup B) = P(A) + P(B) - P(A \cap B) \]

⇒ For \( n \) events

\[ P(A_1 \cup \cdots \cup A_n) = \sum_{1 \leq i \leq n} P(A_i) - \sum_{1 \leq i \leq j \leq n} P(A_i \cap A_j) + \sum_{1 \leq i \leq j \leq k \leq n} P(A_i \cap A_j \cap A_k) - \cdots \]
Review: Random Variable

A random variable (RV) $X: \Omega \rightarrow E$ is a function from the sample space to some other space.

- Often $E$ is $\mathbb{R}$ or $\mathbb{N} \cup \{0\}$

- E.g., coin toss, $\Omega = \{\text{head, tail}\}$, $X(\omega) = \begin{cases} 1, & \omega = \text{head} \\ 0, & \omega = \text{tail} \end{cases}$

**Expectation**: the expected value of an RV

$$E[X] = \sum_{1 \leq i \leq n} x_i P(X = x_i) = x_1 P(X = x_1) + \cdots + x_n P(X = x_n)$$

**Expectation commutes with addition**

$$E[\sum X_i] = \sum (E[X_i])$$

**Example**: expected number of trials to get all sides of a dice

- $E_{\text{tosses to get all sides}} = E_{\text{tosses to get any one side}} + E_{\text{tosses to a second side}} + \cdots + E_{\text{tosses to a sixth side}} = 1 + \frac{6}{5} + \frac{6}{4} + \frac{6}{3} + \frac{6}{2} + \frac{6}{1}$
Probability Basics

- A joint probability distribution assigns probabilities to all atomic events, e.g., provides $P(X_1, \ldots, X_n)$ for all possible value assignments $(x_1, \ldots, x_n)$ to $(X_1, \ldots, X_n)$.
  - It provides everything we need to know about $X_1, \ldots, X_n$.

- Marginal distributions can be computed via marginalization:

  $$P(X_1 = x_1, \ldots, X_m = x_m) = \sum_{\text{all choices of } x_{m+1}, \ldots, x_n} P(X_1 = x_1, \ldots, X_m = x_m, X_{m+1} = x_{m+1}, \ldots, X_n = x_n)$$

- Definition of conditional probability: $P(A|B) = \frac{P(A,B)}{P(B)}$.

- This gives the product rule: $P(A, B) = P(B)P(A|B) = P(A)P(B|A)$.

- Which generalizes to chain rule:

  $$P(X_1, \ldots, X_n) = P(X_1)P(X_2|X_1) \cdots P(X_n|X_1, \ldots, X_{n-1}) = \prod_{i=1}^{n} P(X_i|X_1, \ldots, X_{i-1})$$
Review: Probability Basics, Continued

⇒ Two RVs $X, Y$ are **independent** if $P(X, Y) = P(X)P(Y)$
⇒ Equivalent to $P(X) = P(X|Y)$ and $P(Y) = P(Y|X)$
⇒ $X, Y$ are **cond. independent** given $Z$ if $P(X, Y|Z) = P(X|Z)P(Y|Z)$
⇒ Equivalent to $P(X|Y, Z) = P(X|Z)$ and $P(Y|X, Z) = P(Y|Z)$
⇒ Independence and cond. Independence can greatly reduce the size of a joint distribution

⇒ **Bayes’ rule** is derived from the product rule

\[
P(H|D) = \frac{P(D|H)P(H)}{P(D)}
\]

Data collected given a hypothesis

Current belief

Hypothesis based on data
Bayesian Networks
Just Finished – Need to Know

› Making decision using naïve Bayes (the male/female example)
› Apply Bayes rule (e.g., the desert marriage and the drug examples)
› Exact inference
› Approximate inference