Contents

1 Foundations of Intelligent Systems

1.1 Probability Theory

- **Outcome Space (Sample Space)**: The set of all possible outcomes of a random experiment. Denoted by Ω.
  Example: Rolling a die, $\Omega = \{1, 2, 3, 4, 5, 6\}$.

- **Probability Distribution**: A function that assigns a probability to each outcome in the outcome space. Must satisfy:

  - $0 \leq P(\omega) \leq 1$ for each outcome $\omega \in \Omega$
  - $\sum P(\omega) = 1$ for all $\omega \in \Omega$

  Example: For a fair die, $P(1) = P(2) = ... = P(6) = \frac{1}{6}$.

- **Random Variable**: A variable whose value is determined by the outcome of a random experiment.

  - **Discrete Random Variable**: Takes on a countable number of values.
    Example: Number of heads in three coin tosses.
  - **Continuous Random Variable**: Takes on values within a continuous range.
    Example: Temperature readings.

- **Event**: A subset of the outcome space, representing a collection of possible outcomes.
  Example: Getting an even number when rolling a die, Event = {*2, 4, 6*}.

- **Probability Mass Function (PMF)**: For a discrete random variable X,
  $$PMF(x) = P(X = x)$$
  gives the probability that X takes on the value x.

- **Probability Density Function (PDF)**: For a continuous random variable X, PDF(x) represents the relative likelihood of X taking on

the value x. The probability of X falling within a range [a, b] is given by the integral of the PDF over that range:

$$P(a \leq X \leq b) = \int PDF(x)dx$$

- **Cumulative Distribution Function (CDF)**: For a random variable X,
$$CDF(x) = P(X \leq x)$$
gives the probability that X takes on a value less than or equal to x.

- **Expectation**: The average or expected value of a random variable X.
  - For discrete X: $E_p[X] = \sum_x x * PMF(x))$
  - For continuous X: $E_p[X] = \int x * PDF(x)dx$

- **Variance**: Measures how much the values of X deviate from the expected value.
$$Var_p[X] = E_p[(X - E_p[X])^2]$$

- **Chain Rule**: $P(A \cap B) = P(A) * P(B \mid A) = P(B) * P(A \mid B)$, extends to multiple events:
  - $P(A_1 \cap A_2 \cap ... \cap A_n) = P(A_1 \mid A_2, ...A_n) * P(A_2 \mid A_3, ...A_n) * ... * P(A_n)$.

- **Bayes' Rule**: $P(A \mid B) = \dfrac{P(B \mid A) * P(A)}{P(B)}$, allows us to calculate the probability of a hypothesis (A) given some evidence (B). Variations:
  - $P(A \mid B) = \dfrac{P(A)P(B \mid A)}{P(B)} = \dfrac{P(A)P(B \mid A)}{\sum P(B \mid A_i)P(A_i)}$
  - $P(A \mid B \cap C) = \dfrac{P(B \mid A \cap C)P(A \mid C)}{P(B \mid C)}$

- **Conditional Probability**: $P(A \mid B) = \dfrac{P(A \cap B)}{P(B)}$, the probability of event A happening given that event B has already happened.

- **Independence**: Events A and B are independent if

$$P(A \mid B) = P(A) \text{ or } P(A \cap B) = P(A)P(B)$$

- **Conditional Independence**: Events A and B are conditionally independent given C if

$$P(A \cap B \mid C) = P(A \mid C)P(B \mid C)$$

- **Joint Distribution**: A probability distribution that gives the probability of every possible combination of values for a set of random variables.

- **Probabilistic Inference**: The process of using probability theory to reason about uncertain events and make predictions based on available evidence.

- **Examples of Probability Distributions in AI**: Bernoulli, Binomial, Gaussian distributions are commonly used to model various phenomena in AI applications.

- **Importance of Probability Theory in AI**: Probability theory plays a crucial role in AI by providing a principled framework for representing and reasoning with uncertainty, which is inherent in most real-world applications due to incomplete or noisy information. It enables intelligent agents to make informed decisions and predictions even when faced with ambiguity.

1.2 Graph Theory

Graphs are fundamental in AI for representing relationships between entities (e.g., in social networks or knowledge graphs), modeling problem spaces (e.g., state-space representations in search), and enabling efficient algorithms for search, planning, and reasoning.

Graph is a collection of vertices (nodes) connected by edges. Represented as G = (V, E) where,

- V is the set of vertices.

- E is the set of edges, where each edge is a pair of vertices (u, v) indicating a connection between nodes u and v.

Directed Graph: A graph where edges have a direction, represented by arrows. An edge (u, v) indicates a connection from node u to node v.

Undirected Graph: A graph where edges have no direction, representing a symmetric relationship between nodes. An edge u, v (or equivalently v, u) indicates a connection between nodes u and v.

1.2.1 Graph Properties and Structures

- **Cyclic Graph**: A graph that contains at least one cycle.
  - **Cycle**: A path that starts and ends at the same node, visiting at least one other node in between.
- **Acyclic Graph**: A graph that does not contain any cycles.
- **Connected Graph**: An undirected graph where there is a path between every pair of nodes.
- **Directed Acyclic Graph (DAG)**: A directed graph without any cycles.
  - Often used to represent Bayes Networks.
- **Node Relationships in Directed Graphs**
  - **Parent Node**: A node that has an outgoing edge to another node (it's child).
  - **Child Node**: A node that has an incoming edge from another node (it's parent).
  - **Ancestor**: A node that can be reached by following a sequence of directed edges from another node.
  - **Descendant**: A node that can reach another node by following a sequence of directed edges.
- **Node Degrees**
  - **Degree**: The number of edges connected to a node.

  - **In-degree**: In a directed graph, the number of incoming edges to a node.
  - **Out-degree**: In a directed graph, the number of outgoing edges from a node.

- **Subgraphs**
  - **Subgraph**: A graph formed by a subset of vertices and edges from the original graph.
  - **Induced Subgraph**: A subgraph formed by selecting a subset of vertices and including all the edges between those vertices from the original graph.

- **Special Subgraphs**
  - **Clique**: A subgraph where every pair of vertices is connected by an edge (i.e., a complete subgraph).

- **Paths and Walks**
  - **Walk**: A sequence of alternating vertices and edges, where each edge connects the vertex preceding it to the vertex following it.
  - **Trail**: A walk in which all edges are distinct.
  - **Path**: A walk in which all vertices (and hence edges) are distinct.

1.3 Linear Programming

- **Definition**: A linear program is an optimization problem where the objective function and constraints are all linear. It involves finding the best values for a set of variables to maximize or minimize the objective function, subject to the given constraints.

- **Components**:
  - A set of real-valued variables.
  - A linear objective function (to be maximized or minimized).
  - A set of linear constraints (inequalities or equalities).

- **Solution**:

- A linear program may be
  - infeasible (no solution satisfies all constraints)
  - unbounded (the objective function can be made arbitrarily large or small)
  - have a finite optimum.
- The set of feasible solutions forms a convex polyhedron.
- The Simplex algorithm is a common method for solving linear programs by iteratively moving along the edges of the polyhedron

1.4 Graph Search Methods

1.4.1 Uninformed Search

These methods does not utilize any problem-specific knowledge (heuristics). Examples,

- **Breadth-First Search (BFS)**: Expands nodes in a level-by-level manner, guaranteeing to find the shortest (not necessarily the best) path if one exists (complete). Both time and space complexities are exponential $O(b^d)$. Uses a queue.
- **Depth-First Search (DFS)**: Explores a path until it reaches a dead end or the goal, then backtracks. May (meaning it's not complete) not find the shortest path. Uses a stack. Space complexity is $O(bd)$
- **BFS Optimal**: Optimal search method. Uses a priority queue. Keep the nodes in increasing order of cost of path so far. Once explored keep only the node with least cost. Doesn't explore previously expanded nodes.

1.4.2 Heuristic (Informed) Search

Heuristic search plays a vital role in AI by enabling intelligent agents to efficiently navigate complex problem spaces and make decisions, even when faced with uncertainty or incomplete information. It allows agents to leverage problem-specific knowledge (heuristics) to guide their search for solutions, leading to faster and more effective decision-making. Heuristics are very much domain specific. Examples include,

- **Best-First Search**: Greedy Algorithm. Expands nodes which has the least cost to reach the goal (using the heuristic function). Stops once reached to the goal. Time Complexity is $O(b^d)$. If heuristic is always 0, then space complexity same as breadth first search, otherwise space complexity looks like $O(bd)$

- **A* Search**: Combines the cost to reach a (the current) node (g(n)) with the estimated cost to reach the goal from that (current) node (h(n)) to prioritize expansion. It guarantees to find the optimal path if the heuristic is admissible. Won't stop the process until goal node has the least estimated cost.

Components of the Heuristic Search

- **State Space**: The set of all possible states that the problem can be in.

- **Initial State**: The starting state of the problem.

- **Goal State**: The desired state or states that represent solutions to the problem.

- **Actions**: The set of possible actions that can be taken in each state to transition to other states.

- **Transition Model**: Defines how actions change the state of the problem.

- **Cost Function**: Assigns a cost to each action or path, representing the effort or resources required.

- **Heuristic Function**: An estimate of the cost to reach the goal from a given state, used to guide the search process.

Heuristic Functions:

- **Admissible Heuristic**: A heuristic function that never overestimates the cost to reach the goal. This property is crucial for A* search to guarantee optimality.

- **Consistent Heuristic**: A heuristic function that satisfies the triangle inequality:
$$h(n) \leq c(n, n') + h(n')$$
where $h(n)$ is the heuristic estimate for node n, $c(n, n')$ is the actual cost to go from node n to node n', and $h(n')$ is the heuristic estimate for node n'. Consistent heuristics are also admissible.

- **Examples**:
  - **8-puzzle**: Manhattan distance (sum of the distances of each tile from its goal position) is an admissible and consistent heuristic.
  - **Path-finding**: Straight-line distance (Euclidean distance) to the goal is an admissible heuristic.

Trade-off between Informed-ness and Computational Cost: While more informed search strategies (using better heuristics) can lead to faster solutions by focusing the search on promising areas, they often incur a higher computational cost due to the additional computations required to evaluate the heuristic function. The choice of heuristic involves balancing the trade-off between improved search efficiency and increased computational complexity.

1.5 Graphical Models

1.5.1 Independence

If we have $P(X \mid Y) = P(X)$,

- Then X and Y are independent of each other
- $P(X, Y) = P(X \mid Y).P(Y) = P(X).P(Y)$

This is written as $X \perp Y$

1.5.2 Conditional Independence

Two random variables X and Y are conditionally independent given Z if and only if,
$$P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$$
Represented as $X \perp Y \mid Z$
The above equation must hold for every value of X, Y, Z

1.5.3 Joint Probability Distributions

- Probability distributions over 2 or more variables.
- $P(X1) = \Sigma_{x2 \in val(X2)} P(X1, X2 = x2)$
- Space requirement is exponential to number of variables.

1.5.4 Types of Graphs

- **Directed Graphs:** Bayesian networks
- **Undirected Graphs:** Markov networks

These graphs provide a way to represent the joint distributions in a compact manner by exploiting conditional independence between random variables.

2 Graphical Models

2.1 Representation

2.1.1 Bayesian Networks (Directed Graphical Models)

With the Bayesian networks, joint probability distribution can be written as a product of local probability tables. This greatly simplifies the computation. Bayesian networks are used to model the causal relationship between variables. There are two main components.

- **Components**:
  1. **Structure**: A Bayesian network is represented by a directed acyclic graph (DAG), where:
     - Nodes: Represent random variables (discrete or continuous).
     - Directed Edges: Represent direct causal or influential relationships between variables. An edge from node A to node B signifies that A directly influences B.
  2. **Local Probability models (Conditional Probability Distributions)**:
     - Each node in the DAG is associated with a conditional probability distribution (CPD) that quantifies the probability of that variable taking on different values given the values of it's parent nodes.
     - For a discrete variable X with parents U, the CPD is represented as a table $P(X \mid U)$, specifying the probability of each value of X for every possible combination of values of U.
     - For continuous variables, CPDs can be represented using parametric distributions (e.g., Gaussian) or other suitable representations.
- **Joint Distribution Factorization**: The joint probability distribution over all variables in the Bayesian Network can be factorized based on the conditional independence relationships encoded in the DAG,

$$P(X_1, X_2, ..., X_n) = \prod_{i=1}^{n} P(X_i | Parents(X_i))$$

Where Parents(Xi) denotes the set of parent nodes of Xi in the DAG.

- **Conditional Independence**: Bayesian networks explicitly represent conditional independence relationships. A variable is conditionally independent of its non-descendants given its parents. This property allows for compact representation and efficient inference.
- **D-Separation**: Graph structure can be used for determining conditional independence in Bayesian networks. Two sets of variables A and B are d-separated by a third set C if all trails between A and B are "blocked" by C. Trail A is blocked by C if it contains a node Z such that either,
  - The arrows on the trail meet head-to-tail or tail-to-tail at Z, and Z is given (in evidence).
  - The arrows on the trail meet head-to-head at Z, and neither Z nor any of its descendants are in evidence.

2.1.2 Undirected Graphical Models (Markov Networks)

Captures the correlation, not causation. Useful because joint probability distribution can be written as product of cliqueue potential.

- **Structure**: A Markov network is represented by an undirected graph, where:
  - Nodes: Represent random variables.
  - Undirected Edges: Represent correlations (not causations) between variables.
- **Conditional Independence**: In Markov networks, conditional independence is determined by graph separation.
  - Two sets of variables A and B are conditionally independent given C (C is in evidence) and all paths between A and B are through C.
- **Cliques**: A clique in a Markov network is a subset of nodes where every pair of nodes is connected by an edge (i.e., a fully connected subgraph).

- **Clique Potentials**: Non-negative functions defined over cliques, capturing the compatibility or affinity between variable assignments within a clique. Potentials are not probabilities.
  - Denoted by $\phi_C(X_C)$, where ϕ_C is a clique and X_C represents the variables in the clique.
- **Joint Distribution Representation**: The joint probability distribution in a Markov network is represented as a product of clique potentials, normalized by a partition function:
  - $P(X_1, X_2, ..., X_n) = \frac{1}{Z} \prod_C \phi_C(X_C)$
  - Where Z is the partition function ensuring that the distribution sums to 1.

2.2 Inference

Inference in graphical models refers to the process of computing probabilities of interest given some observed evidence or query variables. This involves reasoning about the dependencies and uncertainties encoded in the graphical structure and associated probability distributions.

Posterior Probability Inference: Process of calculating the probability of certain events or states occurring after some other events or evidence are observed. This is in contrast to prior probability, which is the probability of an event before any evidence is observed.

2.2.1 Exact Inference

Exact inference algorithms compute the desired probabilities precisely, but the computational complexity can be exponential with respect to the tree-width of the graph, making them intractable for large or densely connected networks.

- **Naive Solution** The naive solution to inference in Bayesian Networks involves directly computing the full joint probability distribution over all variables and then using it to calculate the desired conditional probabilities. Steps as follows,

1. Calculate joint distribution
2. Introduce evidence
3. Marginalize (sum over)
4. Normalize (so it becomes a probability)

This approach, although conceptually simple, suffers from exponential space and time complexity. As the number of variables in the network increases, the size of the joint distribution grows exponentially, making it impractical to store and manipulate.

Complexity: The complexity of the naive solution is dominated by the initial construction of the joint distribution, which is exponential in the number of variables ($O(2^n)$ in this example).

Key Takeaway: While the naive solution is straightforward, its exponential complexity makes it impractical for larger Bayesian Networks. This motivates the need for more efficient inference algorithms like Variable Elimination, which exploit conditional independence to reduce computations.

- **Variable Elimination**: Variable elimination is a general algorithm for exact inference in graphical models. It works by systematically eliminating variables from the joint distribution, exploiting conditional independence relationships to simplify computations. The basic idea is to marginalize out (sum or integrate) variables one by one until only the query variables remain. The order of variable elimination can significantly impact the efficiency of the algorithm. Elimination should start with the least connected nodes.

  Complexity: The complexity of variable elimination is exponential with respect to the tree-width of the graph, which is a measure of how "tree-like" the graph is. For graphs with low tree-width, variable elimination can be efficient. However, for densely connected graphs with high tree-width, it can become computationally intractable.

- **Singly Connected Networks (Polytrees)**:
  - A special class of directed acyclic graphs (DAGs) where there is at most one path between any two nodes.
  - Polytrees have treewidth 1, enabling efficient exact inference.

2.2.2 MPE and MAP

- **Most Probable Explanation (MPE)**:

$$MPE(e) = \arg \max_{x_1,...,x_n} P(x_1, ..., x_n \mid e)$$

Using Bayes Rule,

$$\arg \max_{x_1,...,x_n} P(x_1, ..., x_n \mid e) = \frac{\arg \max_{x_1,...,x_n} P(x_1, ..., x_n, e)}{P(e)}$$

Since denominator is a constant it's sufficient to argmax the numerator.

$$\arg \max_{x_1,...,x_n} P(x_1, ..., x_n \mid e) = \arg \max_{x_1,...,x_n} P(x_1, ..., x_n, e)$$

The goal is to find the most likely joint assignment of values to all variables in the network given some observed evidence. This can be solved using a modified version of variable elimination, where instead of summing out variables, we take the maximum over their possible values.

There are 2 phases in the process,

1. **Forward pass**: Eliminate all the variables that's not part of the query using the joint probability values. Then find the value for the variable at the root.
2. **Backward pass**: Use the value at the root node variable to find the values for rest of the nodes.

Example systems :High traffic is observed in junction A, what's the most probable explanation of traffic in all other junctions (leading to junction A).

- **Maximum A-posteriori Probability (MAP)** (Not covered in this module):
The task of finding the most likely joint assignment of values to a subset of query variables given some observed evidence. More challenging than MPE as it involves optimizing over a subset of variables while marginalizing out the rest. Can be solved using various techniques, including,

- Variable elimination with max-marginalization.
- Junction tree algorithm.
- Linear programming relaxation.

Example: High traffic is observed in junction A, what's the most probable explanation of traffic in one other junction (leading to junction A)

2.2.3 Approximate Inference

Approximate inference algorithms are used when exact inference is computationally intractable. They provide approximate solutions but can be much faster than exact methods.

- **Stochastic Simulation (Monte Carlo Methods)**: Generate samples from the distribution represented by the graphical model. Then estimate probabilities based on the frequencies of events in the samples.

  Advantages: Can handle complex models and arbitrary queries.

  Disadvantages: Can be slow to converge, especially for rare events or when evidence is strong. When there are lots of constraints most simulations will be thrown away.
- **Likelihood Weighting**:

$$Pr(X_k = q \mid E = e) = \frac{\textit{Weights of samples with } x_k = q}{\textit{Total weight of all samples}}$$

  * A variation of Monte Carlo sampling that focuses on generating samples consistent with the evidence.
  * Each sample is weighted by its likelihood given the evidence.
  * **Advantages**: More efficient than direct sampling when evidence is present. No samples get thrown away.
  * **Disadvantages**: Can still be slow to converge for rare events or strong evidence.

Event $E = e$, Variables $X_1, ...X_n$
$w \leftarrow 1$;
for $i = 1..n$ **do**
  if X_i *not belongs to* E **then**
    sample x_i from $P(X_i \mid Parents(X_i)$;
  else
    $x_i \leftarrow e(X_i)$;
    $w \leftarrow w * P(x_i \mid Parents(X_i)$
  end
end
return $P(X = x, E = e), w$

Algorithm 1: Likelihood Weighting

2.3 Learning

In the context of graphical models, learning involves estimating the parameters or/and structure of the model from data. This allows us to discover the underlying probabilistic relationships among variables and make predictions or inferences about new, unseen data.

2.3.1 Types of Learning

- **Parameter Learning**: Given the structure of the graphical model (i.e., the DAG or undirected graph), estimate the parameters of the local probability distributions (CPDs in Bayesian networks or clique potentials in Markov networks) from data.
- **Structure Learning**: Learn the graph structure itself (i.e., the connections between variables) from data. (This is not covered in this module)
  - This is generally a more challenging problem than parameter learning, as it involves searching over a potentially vast space of possible graph structures. Heuristics or constraints are often necessary to guide the search and make it computationally feasible.

2.3.2 Parameter Learning with Known Structure

- **Full Observability (Complete Data)**:

- All variables in the training data are observed.
- **Maximum Likelihood Estimation (MLE)**:
  - The most common approach for parameter learning with complete data.
  - Finds the parameter values that maximize the likelihood of the observed data given the model structure.
  - Involves computing the sufficient statistics (counts or frequencies) from the data and using them to estimate the parameters.

- **Partial Observability (Incomplete Data)**:
  - Some variables in the training data are missing or unobserved.
  - **Problems**:
    1. Estimating parameters
    2. Hypothesizing missing values.
  - **Expectation-Maximization (EM) Algorithm** (Not tested in Exam):
    - An iterative algorithm for parameter estimation with incomplete data.
    - Alternates between two steps:
      - **E-step** (Expectation step): Computes the expected values of the missing data given the current parameter estimates and the observed data.
      - **M-step** (Maximization step): Updates the parameter estimates to maximize the expected log-likelihood of the complete data (observed + missing), using the expected values from the E-step.
    - The algorithm converges to a local maximum of the likelihood function.

2.3.3 Connection to Intelligent Agents

The ability to learn from data is a crucial aspect of intelligent agents, as it allows them to adapt to their environment and improve their performance

over time. By learning the parameters and/or structure of graphical models, agents can acquire new knowledge, refine existing knowledge, and make more accurate predictions or inferences about the world, leading to better decision-making and overall performance. *Reference: Russell & Norvig (2010), Chapter 2*

2.3.4 Structure Learning

Not covered in this module.

3 Decision Making

3.1 Decision Theory

Decision theory provides a formal framework for making optimal decisions in the face of uncertainty. It helps agents choose actions that maximize their expected utility, taking into account both the probabilities of different outcomes and the agent's preferences over those outcomes. *Reference: Russell & Norvig (2010), Chapter 16*

3.1.1 Key Concepts

- **Decision**: A choice among a set of available actions.
- **Outcome**: The result of taking an action, which may be uncertain.
- **Utility**: A numerical value representing the desirability or preference for an outcome.
- **Expected Utility**: The average utility of an action, weighted by the probabilities of its possible outcomes.
- **Rational Agent**: An agent that chooses actions to maximize its expected utility.

3.1.2 Maximum Expected Utility (MEU) Principle

The MEU principle states that a rational agent should choose the action that maximizes its expected utility.

Expected utility

$$EU(A_j) = \sum_{s_j} Pr(s_j \mid A_j)U(s_j)$$

Action that maximize the EU

$$MEU = \max_{A_i} EU(A_j)$$

- A is an action
- s is a possible outcome or state

- $P(s \mid A)$ is the probability of outcome s given action A
- $U(s)$ is the utility of outcome s

3.1.3 Utility Functions

Utility functions assign numerical values to outcomes, reflecting the agent's preferences. They can capture different attitudes towards risk.

Different attitudes towards risk,

- **Risk-averse**: Prefers certain outcomes with lower expected value over risky outcomes with higher expected value.
- **Risk-neutral**: Indifferent between certain and risky outcomes with the same expected value
- **Risk-seeking**: Prefers risky outcomes with higher expected value over certain outcomes with lower expected value

Example:
Consider a decision problem where you can either:

- Action 1: Invest in a risky stock
  - 50% chance of gaining $1000
  - 50% chance of losing $500
- Action 2: Keep the money in a safe savings account
  - 100% chance of gaining $100

Assuming a risk-neutral utility function (U(x) = x), the expected utilities are:

- EU(Action 1) = 0.5 * 1000 + 0.5 * (-500) = 250
- EU(Action 2) = 1.0 * 100 = 100

According to the MEU principle, a rational agent would choose Action 1 (invest in the risky stock) as it has a higher expected utility.

3.1.4 Limitations of the MEU Principle in Real World

The MEU principle is a normative model for ideal decision-making. However, human decision-making often deviates from this ideal due to various cognitive biases, bounded rationality, and the difficulty of accurately assessing probabilities and utilities in complex real-world situations. *Reference: Russell & Norvig (2010), Chapter 16*

3.2 Markov Decision Processes (MDPs)

MDPs provide a mathematical framework for modeling sequential decision-making problems in environments with uncertainty. They are widely used in AI to represent problems where an agent interacts with an environment over time, taking actions and receiving rewards based on its decisions. *Reference: Russell & Norvig (2010), Chapter 17*

Examples of MDPs: Grid-world navigation, robot control, inventory management, and many other real-world problems can be formulated as MDPs. *Reference: Russell & Norvig (2010), Chapter 17*

3.2.1 Key Components of an MDP

An MDP is defined by a tuple (S, A, P, R, γ), where:

- **States (S)**: A finite set of possible states the agent can be in.
- **Actions (A)**: A finite set of actions the agent can take in each state.
- **Transition Probabilities (P)**: $P(s' \mid s, a)$ represents the probability of transitioning to state s' after taking action a in state s.
- **Rewards (R)**: $R(s, a, s')$ represents the immediate reward received after transitioning to state s' from state s by taking action a.
- **Discount Factor (γ)**: A value between 0 and 1 that discounts future rewards, reflecting the preference for immediate rewards over delayed rewards.

3.2.2 Applicability of MEU for MDPs

Need to consider multiple starting states, calculte future utility as well as the immediate utility. And find the best action for each state.

$$EU((1,1),N) = \Sigma_{s'} Pr(s' \mid (1,1),N)[R((1,1),N,s') + MEU(s')]$$

$$Pr(s' \mid (1,1),N) = \text{Transition probability}$$

$$R((1,1),N,s') = \text{Immediate reward}$$

$$MEU(s') = \text{Future reward}$$

3.2.3 MDP terminology

- **MEU analogy**
  - $EU(s,a) -> Q(s,a)$
  - $MEU(s) -> V(s)$
  - $MEU(s) = \underset{a}{max} EU(s,a) \rightarrow V(s) = \underset{a}{max} Q(s,a)$
  - $EU(A_i) = \Sigma_{O_j} Pr(O_j \mid A_j) U(O_j) \rightarrow$
    $Q(s,a) = \Sigma_{s'} Pr(s' \mid s,a)[R(s,a,s') + V(s')]$
- **Model**
  - **Decision epoch**: Points at which decisions are made
    - **Finite horizon**: Policy depends on what you did until now. Policies are non-stationary and deterministic.
    - **Infinite horizon**: Policy only depends on current state. Policies are stationary and deterministic.
  - **Absorbing state**: Do not allow any transitions out. Typically goal states.
  - **Reward**: $R(s), R(s,a), R(s,a,s')$
  - **Markov assumption**: The Markov property states that the future state depends only on the current state and action, and not on the past history of states and actions.
- **Policies**

- **Decision Rule**: Rule to choose action in each state for a given decision epoch.
- **Policy**: set of decision rules to be used at all decision epochs
- **Deterministic vs Stochastic policies**:
  - **Deterministic Policy**: only one action to take in each state. for MDP all policies are deterministic.
  - **Stochastic Policy**: $\pi(a|s)$ gives the probability of taking action a in state s.
- **Stationary vs Non-stationary**:
  - **Stationary Policy**: for each time step, decisions are same.
  - **Non-stationary Policy**: for each time step, decisions can be different.

- **Discount Factor** Introducing γ to future rewards, so system will give priority to immediate rewards.

$$Q(s,a) = \sum_{s'} Pr(s' \mid s,a)[R(s,a,s') + \gamma V(s')]$$

3.2.4 Solving MDPs

Common algorithms for finding optimal policies:

- **Value Iteration**: Iteratively updates the value function until convergence to the optimal value function, then extracts the optimal policy. Algorithm as follows,
  - Initialize $V^0(s) = 0$ for all states s
  - Then,
    - $Q^1(s,a) = \sum_{s'} Pr(s' \mid s,a)[R(s,a,s') + \gamma V^0(s')], \forall a$
    - $V^1(s) = \underset{a}{max} Q^1(s,a)$
  - Repeat until $V^t(s)$ and $V^{t+1}(s)$ are close (or t= horizon).

  Guaranteed to yield optimal policy,

$$\pi^*(s) = \arg\max_a Q^{t+1}(s,a)$$

  But it's time consuming.

- **Policy Iteration**: Iterate over policy instead of value function in value iteration.
  - Start with random policy
  - Loop until convergence
    - Evaluate how good the policy is
    - Improve policy

$V(s) \leftarrow 0, \forall s$;
$\pi \leftarrow random$;
repeat
  $\Delta \leftarrow 0$;
  for $s \in \forall s$ **do**
    $v \leftarrow V(s)$;
    $V(s) \leftarrow \sum_{s'} Pr(s'|s, \pi(s)) [r + \gamma V(s')]$; `// Bellman update`
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
  end
until $\Delta < \theta$;

Algorithm 2: Policy Evaluation

for $s \in \forall s$ **do**
  $old\_action \leftarrow \pi(s)$;
  if $Q(s, a) > Q(s, \pi(s))$ **then**
    $\pi(s) \leftarrow \arg\max_a \sum_{s'} Pr(s'|s, a)[R(s, a, s') + \gamma V(s')]$;
  end
end
return π;

Algorithm 3: Policy Improvement

This is faster than value iteration because this starts from a random policy.

- **Linear Programming**: MDP can be also formulated as linear program and solves it using linear programming techniques.
  - Primal LP, Minimize $\sum_s V(s)$

$$V(s) \geq \sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma V(s')], \forall a$$

- Dual LP, Maximize $\sum s, ax(s,a)R(s,a)$

$$\sum_a x(s,a) - \gamma \sum_{s'} x(s',a)P(s \mid s',a) = \delta(s)$$

3.3 Reinforcement Learning

3.3.1 Bandit Problems (Multi-Arm Bandits)

Subset of RL problems with a simpler architecture of learning. Either no states (MAB), states with no transitions (C-MAB) or states with small action space (R-MAB).

- **A/B Testing**
  - **Traditional A/B Testing**: Distribution doesn't change
  - **MAB (Multi Arm Bandit) Testing**: Distribution changes based on users' interactions
  - **Contextual MAB**: Each type of user have their own distribution (which also changes with users' interaction)
- **Analogy**: Set of slot-machines with hidden probability to give out money, probability doesn't depend on previous outcomes.
- **Applications**
  - Online advertisements
  - Clinical trials
  - Netflix artwork
- **Generic Bandit Framework**:
  - for t in range(1 to T) do,
    - SELECT: Use the bandit algorithm to pick the arm to pull
    - OBSERVE: Pull the selected arm and observe the reward on that pull
    - UPDATE: Update the estimated reward
- **Bandit Strategies**: Trick is to find the correct balance of explore to exploit ratio.

- **Greedy**: Always pull the best arm based on results up-to "now". Problem - doesn't do much of exploring.

$$Q_t(a) = \frac{R_1(a) + R_2(a) + ... + R_{k_t(a)}(a)}{k_t(a)}$$

$$a_t^* = \arg\max_a Q_t(a)$$

- ϵ**-greedy**: explore arms with ϵ probability. exploit (choose the "best" arm) with $1 - \epsilon$ probability. For the most of times this is the best strategy.

$$P_{r_a} = \begin{cases} 1 - \epsilon & \text{if } a = a_t^* \\ \dfrac{\epsilon}{n-1} & \text{if } a \neq a_t^* \end{cases}$$

- **Soft-max**: Pick an arm with a probability based on $Q_t(a)$

$$Pr_a = \frac{e^{Q_t(a)}}{\sum_b e^{Q_t(b)}}$$

- **Upper Confidence-bound (UCB)**: Calculate a "UCB" for each arm and pick the arm with highest. Give more chance to least explored arm.

$$a_t^* = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\log(t)}{k_t(a)}} \right]$$

- **Thompson Sampling**: Idea here is considering only the average is not good enough.
  - **Beta distribution**:
    - Parameters (when reward =1): α : number of successes, β : number of failure
    - Mean: $\dfrac{\alpha}{\alpha + \beta}$
  - **Algorithm**:
    - Initialize $\alpha = \beta = 1$ and reward between 0 and 1.

    - * Sample values from beta distribution and pick the arm with highest value, then update the α and β values.
    - * For the picked arm, $\alpha = \alpha + reward; \beta = \beta - (1 - reward)$

- **Contextual MAB**:
  - Compared to MAB, C-MAB have states addition to actions. So it's more powerful and learning process can be longer.
  - Compared to MDP/RL C-MAB don't have transitions between states.
  - Usage in ad recomendation:
    - * State $\rightarrow$ user profile; action $\rightarrow$ which product to recommend
    - * Learn reward for each (s, a) combination. neural network can be used for this. prior user activities used as the training data.
    - * Once the reward values are learnt, MAB strategies (greedy, ϵ-greedy, etc) can be employed.

- **Restless Multi-Arm Bandits (RMAB)**:
  - **Compared to MAB**, with R-MAB,
    - * Arms are always(irrespective of being frozen/activated) evolving (can accrue reward and change state). eg: health status. In MAB only activated arms evolve.
    - * Multiple arms can be activated at each time step, whereas in MAB only one arm is activated
    - * Optimal policy no longer depend on values - Q_i of individual arms.
    - * Similar to MAB, objective is to maximize long-run reward
  - **Motivation**:
    - * Preventive interventions in public health
    - * Target tracking for security
    - * Music recommendation
  - Suitable when there are constraints on resources.
  - **structure**:

    - ∗ "n" arms
    - ∗ Each arm is an MDP with,
        - · 2 states: "good" and "bad" for each arm.
        - · Different transition probabilities for different actions
        - · Reward for arm "i" is given by $r_i(s_i, a_i)$
    - ∗ Pick "m" arms to activate each time step to maximize expected reward
  - **Mathematical Representation**

$$\text{Maximize } \frac{1}{T}\mathbb{E}\left[\sum_{t=1}^{T}\gamma^t\sum_{i=1}^{n}r_i(s_i^t, a_i^t) \mid s^0\right]$$

$$\text{s.t. } \sum_{i=1}^{n}a_i^t = m, \forall t$$

    Explanation,

    - ∗ **Objective**: Maximize overall reward over all "T" time steps
        - · Inner sum: overall reward at "t" for all arms
        - · Outer sum: overall discounted reward
        - · Expectation over transitions
    - ∗ **Constraint**: Pick a maximum of m arms

- **Solving a R-MAB**: Two ways to solve a R-MAB,

  1. **As a one big MDP**: States will be all the state combinations for all the arms. Action space is all possible actions for combined arm.

     example: consider a system with 3 arms - A1, A2, A3 and each arm have 2 possible actions - 0,1. action space - 000, 001, 010,... and action space - A1A2, A2A3, A1,A3

     Brute Force algorithm. Not feasible for big systems.

  2. **As multiple small MDPs**: This is more intuitive and faster than solving it as a one big MDP.

- **Whittle Index**:

- **Algorithm**
  - Each arm has an index that needs to be calculated.
  - Rank the arms based on index (once calculated)
  - Choose the arms with top indices
- **Definition**: For a fixed arm and a given subsidy W,
  - Whenever an arm is not selected, we give a subsidy W

$$V(s,W) = \max \begin{cases} r(s,1) + \beta \sum_{s'} P^1_{ss'}[V(s',W)], & Q(s,1,W) \\ r(s,0) + W + \beta \sum_{s'} P^0_{ss'}[V(s',W)], & Q(s,0,W) \end{cases}$$

  - W $\rightarrow$ too small, active action is better. W $\rightarrow$ too large, passive action is better.
  - Whittle index is the minimum value possible such that both actions have equal values.
- **Calculating whittle index**:
  - initialize W
  - repeat
    - evaluate $V_1(s_i, W)$
    - calculate $\Delta = Q_i(S_i, 0, W) - Q_i(s_i, 1, W)$
    - if $\Delta > 0$: reduce W
    - if $\Delta < 0$: increase W
    - if $\Delta = 0$: break

3.3.2 Reinforcement Learning (RL)

This is how typically humans/animal learn. It also works well for agents. It works so well, today the best game players use RL. Environment can be modeled as a MDP with unknown P (transition probability) and R (reward).

Analogy to MAB - Each room has multiple arms. Once arm is pulled, agent get teleported to another room (according to an unknown probability distribution) and need to figure out which arm to pull next. Each arm is an action.

2 types of learning,

1. **Model based learning**: Estimate R and P from data. solve MDP for optimal policy. This can take long time and gets difficult when environment is complex.

2. **Model free learning**: Learn from experience.

- **Formulating RL**:
  - **World**: Discrete, finite set of states and actions
  - **Experience**: (s_t, a_t, r_t, s_{t+1}) - When in state s_t, take an action a_t, receive reward r_t and moved to new state s_{t+1}
  - **Episode**: Group of experiences from start state to goal state.
- **(Exact) Q-Learning**: Learning algorithm for RL. We can't apply value/policy iteration here because we don't know the $Pr(s' \mid s, a)$ or $R(s, a, s')$
  - **Goal**: maximize $V^*(s) = \underset{a}{max} Q(s, a)$
  - Improve the estimate of $Q(s, a)$ with every experience using formula,

    $$\text{New Estimate} = \text{Old Estimate} + \text{Step Size}[\text{Target-Old Estimate}]$$

    $$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha * [r_t + \gamma * \underset{a}{max} Q(s_{t+1}, a) - Q(s_t, a_t)]$$

  - **Algorithm**
    1. $Q(s, a) = 0, \forall s, a$
    2. Start in some initial state, s_0
    3. Do forever,
       (a) Select an action a_t (from greedy, ϵ-greedy, softmax, UCS, etc) and execute it.
       (b) Receive (immediate) reward, r_t
       (c) Observe new state, s_{t+1}
       (d) Update $Q(s, a)$ with,

       $$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha * [r_t + \gamma * \underset{a}{max} Q(s_{t+1}, a) - Q(s_t, a_t)]$$

       (e) Set current state to s_{t+1}

(f) If end up in an absorbing/terminal state, restart from step 2.

- **Exploration/Exploitation**:
  - By employing ϵ- greedy or softmax when choosing the action.
  - Modify the update function based on number of visits.
  - Vary the learning parameters during learning process.
- **Advantages**:
  - Guaranteed to converge in stationary (transitions and rewards remain same) environment.
- **Disadvantages**:
  - Not scalable for large or continuous state spaces.
  - Doesn't make efficient use of experience. Meaning if rewards are sparse, q-learning can run many iterations without learning.
  - Slow convergence.

- **Approximate Q-Learning**
  - **Feature based representations**: Describe a state using features. Then represent Q value in term of features,

$$Q(s,a) = w_1 * f_1(s,a) + w_2 * f_2(s,a)...$$

Advantages: Learning is faster cause exprience summarized concisely.
Disadvantages: States may share features but differ in values

- **Q-learning with Features**
  - Experience: (s_t, a_t, r_t, s_{t+1})
  - Exact Q-learning:

$$Q^{new}(s_t,a_t) = Q(s_t,a_t) + \alpha * [r_t + \gamma * \max_a Q(s_{t+1},a) - Q(s_t,a_t)]$$

  - Approximate Q-learning:

$$w^i = w^i + \alpha * [r_t + \gamma * \max_a Q(s_{t+1},a) - Q(s_t,a_t)] * f^i(s_t,s_t)]$$

$$Q^{new}(s_t,a_t) = \sum_i w^i f^i(s_t,a_t)$$

- **Issues**: Where to get the features? Sometimes it's possible to get from experts. This used to be the case in old days. But today the prefered way is learning the features with Deep Learning.

4 Decision Learning - Deep Reinforcement Learning

4.1 Introduction to Deep Learning

4.2 Value based Deep Reinforcement Learning

4.2.1 Deep Q-Learning

- Represent value function with weights: $Q_w(s_t, a_t)$
- Given experience (s_t, a_t, r_t, s_{t+1}),
  - Input: s_t, a_t or just s_t
  - Output: $r_t, \gamma * \underset{a}{max} Q_w(s_{t+1}, a)$
  - Loss: $[r_t + \gamma * \underset{a}{max} Q_w(s_{t+1}, a) - Q_w(s_t, a)]^2$
- Train over all experiences.
- **Expectation**: At convergence, $Q(s_t, a_t) \leftarrow r_t + \gamma * \underset{a}{max} Q(s_{t+1}, a)$ like it does in exact Q-Learning. (But this doesn't happen)
- **Why convergence fail ?**
  - Correlation between samples (Deep learning doesn't like this)
  - Non-stationary targets - after each update, $r_t + \gamma * \underset{a}{max} Q_w(s_{t+1}, a)$ chages.
- **How to fix**:
  - Remove correlations: Build data set from agent's own experiences and when training take a random sample from it.
  - Fix the target parameters

4.2.2 Deep Q-Learning with Experience Replay and Target Network

- **Target Network**: Target weights updated less frequently.
- **Experience Replay**: Agent's own experiences recorded in a buffer and when training take a random sample from it.
- **High level outline**:
  While True,
  1. Get Experience
  2. Put in the buffer
  3. Randomly sample experience Ex_{random} from the buffer
  4. Learn from Ex_{random}
  5. Move to the new state
  6. Go to step 1
- **Algorithm**:

Initialize replay memory D to capacity N;
Initialize action-value function Q with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for *episode = 1 to M* **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
  for $t = 1$ *to* T **do**
    With probability ϵ, select a random action a_t;
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$;
    Execute action a_t in emulator and observe reward r_t and image x_{t+1};
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$;
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D;
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D;
    Set $y_j = \begin{cases} r_j, \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-), \text{otherwise} \end{cases}$;
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ;
    if *every C steps* **then**
      reset $\hat{Q} = Q$;
    end
  end
end

Algorithm 4: Deep Q-learning with experience replay

4.3 Policy based Deep Reinforcement Learning

Policy is the neural network (classification problem), where in DQN Q-Network is the neural network (regression problem).

4.3.1 Policy Gradient

- **Intuition**: Similar to playing a tennis game. Learn from the winning trajectory (replicate them) and learn from the losing trajectory (avoid them)

- **REINFORCE Algorithm**:
  - Generate trajectories τ^i using policy $\pi_\theta(s, a)$. 1 game = 1 trajectory
  - Calculate the average reward $J(\pi_\theta)$ by averaging over rewards
  - Update policy parameters using gradients of $J(\pi_\theta)$:

$$\theta_{(k+1)} = \theta_k + \alpha \nabla_\theta J(\pi_\theta) \mid_{\theta_k}$$

    .
    $\nabla_\theta J(\pi_\theta) \mid_{\theta_k}$ is the gradient of the function $J(\pi_\theta)$with respect to θ, but specifically at the current parameter value, $\theta = \theta_k$.

- **Practical Scenario - Pong Game**:
  - Input: image
  - Output: UP/DOWN action
  - Rewards: +1 - ball goes past opponent (This will be rare at the beginning of the training); -1 - we miss the ball; 0 - otherwise
  - Goal: maximize reward
  - Credit Assignment: all the action in the trajectory will get the reward

- **Code - Pong Game**:
  - W1, W2 weights to and from hidden layer. W1 - Figure out game scenario (e.g. the ball is in the top, and our paddle is in the middle). W2 - Decide to go UP or DOWN
  - $h = np.dot(W1, x)$ # hidden layer neuron activations
  - $h[h < 0] = 0$ # ReLU nonlinearity: threshold at zero
  - $logp = np.dot(W2, h)$ # log probability of going up
  - $p = 1.0/(1.0 + np.exp(-logp))$ # sigmoid function (gives probability of going up)

- **Compared to Supervised Learning**,
  Quite similar to supervised learning problem. Difference here is the "ground truth" gets available only after sampling the action.

- **Issues with PG**:
  - Need the whole trajectory to learn. Hence learning is slow.
  - High variance in $J(\pi_\theta)$ value due to the credit assignment method used. And sudden update of reward for all the actions (in the trajectory).

5 Multi Agent Systems (MAS)

5.1 Game Theory

5.1.1 Assumptions

- Players are rational - They try to maximize their own reward
- Each player assumes other players are rational. And others know that they are rational

5.1.2 Popular Games

- Prisoner's Dilemma
- Rock Paper Scissors
- Chicken or Dare?
- Cake Cutting

5.1.3 Representations

- **Extensive Form**: Graph representation. Each player actions is one level of the graph. Number of edges = (player 1 number of possible actions) * (player 2 number of possible actions) * ... At the edge it's the reward values for each player if that branch of actions were taken.
- **Normal Form**: Table representation. Column player and row player. Number of columns = Number of actions available for column player. Number of rows = Number of actions available for row player. Each cell contain the reward for all the players for that action combination.

5.1.4 Types of Games

- Simultaneous and Sequential
- One-shot or Repeated
- Zero-sum and Constant-sum
- Perfect information and Imperfect information

- Population based games

5.1.5 Solution Concepts and Algorithms

- **Nash Equilibrium**: State where no player has incentive to deviate from their chosen actions. If they deviate their reward will be less. Every finite game has at-least one such state.
  - **Pure vs Mixed Strategy**: Player always picking the same action vs picking an action based on some probability. (Mixed is more sensible)
  - **Algorithms for Finding Nash Equilibrium**:
    - **Iterative Best Response**
      - Iterative algorithm
      - Each agent starts with a random policy
      - At every iteration (in a round robin manner), one agent finds the best response given fixed policies of other agents
      - If it converges, it converges to a Nash Equilibrium
      - Does not converge in many cases - ex: rock - paper - scissors
    - **Fictitious play**
      - Iterative algorithm
      - Each agent starts with a random policy
      - At every iteration, each player computes best response against aggregated (over previous iterations) belief of other player actions
      - If it converges, it converges to a Nash Equilibrium
      - Converges for, potential games, identical interest games, zero sum games.
- **Correlated Equilibrium**:
  - Third party coordinates players
  - Probability distribution on all joint strategies
  - Given the distribution no players have incentive to deviate (other players keep to there given action)

- $\sum_{s:si=si\#} p(s)\text{payoff}_i(s) \geq \sum_{s:si=si'} p(s)\text{payoff}_i(s_1, ...s'_i, ...s_k) \forall s'_i$

- **Notes:**
  - All mixed Nashes are correlated equilibrium
  - All convex combinations of mixed Nashes are correlated
- **More solution concepts** (Not discussed in details)
  - Sub-game perfect equilibrium
  - Trembling hand perfect equilibrium
  - Coarse correlated equilibrium - Generalized correlated equilibrium.
  - Epsilon equilibrium - haven't reached an equilibrium but close enough.

5.2 Interesting games

5.2.1 Stackelberg games

- Sequential game
- Leader goes first
- Follower responds to the leader strategy
- **Assumption:** Follower observes leader's action
- **Examples:**
  - Modeling advertisement campaigns of competing products
  - Patrolling for infrastructure security
    - **Problem**: Limited security resources: Selective checking and Adversary monitors defenses, exploits patterns
    - **Model**: Security forces commit first, adversary follows
    - **Goal**: payoff maximizing strategy for police with adversary maximizing corresponding pay-off
    - **Example strategy**: Terminal 1 - 70%, Terminal 2 - 30%

- **Problem and Approach**
  - Followers are independent of each others (equivalent)
  - Compute target coverage probabilities instead of policies
  - Output will be a vector with target patrol probabilities of each target
- **Notation**
  - $x \rightarrow$ policy of leader (defender):

$$\text{ex} : [t1 = 0.6, t2 = 0.3, t3 = 0.1] \leftarrow \text{random}$$

  - $q \rightarrow$ policy of follower (attacker):

$$\text{ex} : [t1 = 0, t2 = 0, t3 = 1] \leftarrow \text{deterministic}$$

  - $X \rightarrow$ set of defender strategies
  - $Q \rightarrow$ set of attacker strategies
  - Direct attempts at solving Stackelberg would yield a quadratic program. But with some tricks it can be modeled as a linear program.
  - Computing the strategy for the follower (attacker), given the strategy of leader (defender) x.
    - **Primal**

$$max_q \sum_{j \in Q} \sum_{i \in X} C_{ij} x_i q_j$$

      $C_{ij} \leftarrow$ adversary utility, $x_i \leftarrow$ patrol probability, $q_j \leftarrow$what we need to find

$$s.t. \sum_{j \in Q} q_j = 1$$

$$q_j \geq 0$$

    - **Dual**

$$min_a \; a$$

$$s.t. \; a \geq \sum_{i \in X} C_{ij} x_i \quad j \in Q$$

– Computing the strategy for the leader (defender), given the follower (adversary) going to respond.

$$max_x \sum_{i \in X} \sum_{j \in Q} R_{ij} q(x)_j x_i$$

$$s.t. \sum_{i \in X} x_i = 1$$

$$x_i \in [0...1]$$

5.2.2 Potential Function

- Potential games = well behaved games. Players' objectives are aligned with each other. If a game has a potential function it's a potential game.
- **Definition**:

$$\phi_i(a_i^{'}, a_{-i}) - \phi_i(a_i^{''}, a_{-i}) = u_i(a_i^{'}, a_{-i}) - u_i(a_i^{''}, a_{-i})$$

5.2.3 Congestion games

- Selfish agents competing for common resources. Ex: drivers in roads, users for internet bandwidth
- Every congestion game is a potential game. Potential function is sum of delays on all resources.

$$P(s) := \sum_{j \in s_1 \cup ... \cup s_n} \sum_{k=1}^{u_j(s)} d_j(k)$$

- N players, E of facilities
- $d_j(k) \leftarrow$ delay on resource j when k users are using the facility
- $c_i(s) \leftarrow$ cost for agent i given joint strategy s. Depends on number of users using facility used by agent i
  - $c_i(s) = \sum_{j \in s_i} d_j(u_j(s))$
  - $u_j(s)$ is number of users using facility j given joint strategy s
- **Braess's Paradox**: Adding an extra node to a map make it worse for everyone (though by intuition it should make it easier)

5.2.4 Identical interest games

- All agents have a common utility function
- Typically, represents a coordination problem thus is a potential game
- Ex: Coordinating companies to organize humanitarian missions

5.3 Auction Theory

Application of game theory to economics with well defined set of rules which leads to theoretical predictions.

5.3.1 Single item auctions

- English Auction: Each bid must be higher than previous. Last bid wins, pay the last bid.
- Japanese Auction: Price rises, bidders drop out of bidding. Last bidder wins.
- Dutch Auction: Price drops until someone takes it.
- Sealed bid Auction: Each bidder submit a bid in an envelope.
  - First Price Auction: Highest bidder wins. And pays the highest bid.
  - Second Price (Vickery) Auction: Highest bidder wins. But pays the second highest bid. This forces the participants to bid the true price.

5.3.2 Incentive Compatibility

Notion to decide the auction is fair. Two types of Equilibrium. Forces people to bid the true value of a item

- Stronger Notion: (Strategy proof-ness) Doesn't depend on what others do. Truth telling is the dominant strategy. Hard to archive.
- Weaker Notion: Bayes Nash equilibrium. If others are telling the truth, its better to tell the truth.

5.3.3 Combinatorial Auctions

If the multi-items biding follows the single item bidding strategy, that won't be efficient. Because the value of one item may depend on whether the bidder own another item. It's inefficient to do the multi-item bidding sequentially. Because it can prevent the items from going to the bidder who values them the most as a combination, leading to an inefficient allocation of resources.

5.3.4 Integer Programming

Integer/Linear programming can be used to maximize the overall value when bidders submit complementary valuations.

$$\text{Maximize} \sum_b v(b) * X_b; \text{ s.t.} \forall j, \sum_b : \text{j in b } X_b \leq 1$$

If X_b can take only integers this can be solved as a integer program. If it can take fractional values, it can be solved in polynomial time.

Example: **Spectrum Auctions:**

Imagine a government auctioning licenses for 5G spectrum in three regions (A, B, C). Instead of separate bids for each item, companies submit bids for combinations of licenses they want.

- Company X might bid:
  - $200 million for A & B together
  - $100 million for A alone
  - $80 million for B alone
- Company Y might bid:
  - $180 million for B & C together
  - $90 million for B alone
  - $90 million for A alone, etc.

The government (auctioneer) gathers all these bids. They then use integer programming to determine the winning combination that maximizes total revenue (or potentially other objectives like promoting competition).

The End.