Lecture 2

RFID communication:

- passive tags
- no power source
- transmission through back-scattering

In a RFID system, the tags reflect the high-power constant signal generated by the reader to send their unique ID.
Tags cannot transmit spontaneously: the reader queries tags and they respond with their ID by back-scattering the received signal. Tags cannot hear each other (NO Carrier Sense, NO Collision Detection).
Several sequential MAC protocols have been proposed to identify tags in a RFID system:

- Tree based
    - Binary Splitting
    - Query tree
- Aloha based
    - Framed Slotted Aloha

Binary splitting

BS recursively splits answering tags into two subgroups until obtaining single tag groups.
Tags answer to reader's queries according to the generation of a binary random number.
Given a set of tags to identify:

- Each tag has a counter initially set to zero
- The reader sends a query
- All tags reply → collision
- Each tag generates a random binary number (0,1) and sums it to the counter
- The process repeats
    - The reader sends a query
    - All tags with C=0 replies
    - If collision → each replying tag generates a random binary number and sums it to its counter
    - Each other tag (silent) → C=C+1
    - If none or one tag replies → all tags: counter = counter - 1

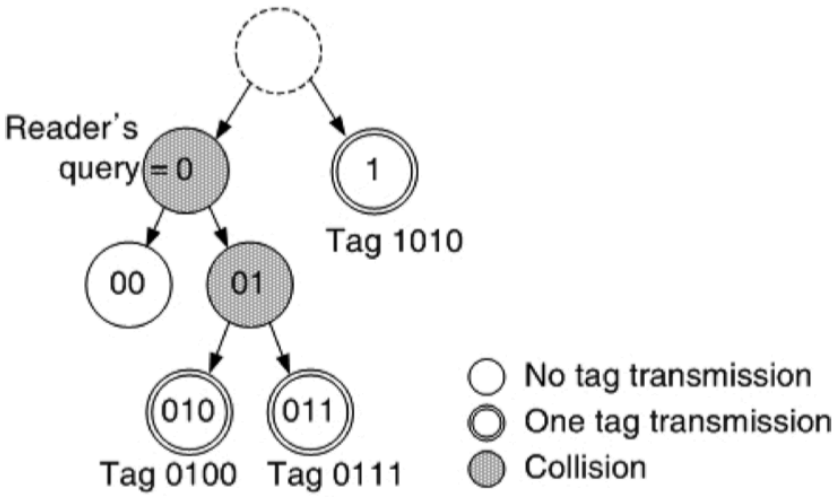
Low performance in terms of [single responses / total number of queries].

Query Tree

QT queries tags according to the binary structure of their ID. Each tag has an ID of typically 96 bits. The reader interrogates tags by sending them a string, and only those tags whose IDs have a prefix matching that string respond to the query. At the beginning, the reader queries all tags: this is implemented by including a NULL string in the query. If a collision occurs, then the string length is increased by one bit until the collision is solved and a tag is identified.

Example:

- Suppose we have 3 tags whose IDs are:
- 0100
- 0111
- 1010



Framed Slotted Aloha

Time is divided in slots, which are then grouped into frames.
Each tag randomly picks a slot to respond. When a reader issues a start of frame, it includes the number of slots in a frame. Each tag then randomly picks a slot in which to reply.
Collisions occur if two or more tags pick the same slot.
The process repeats itself until all tags are identified.
Once a tag is identified, it no longer responds to the start of frame.

Lecture 3

Two types of networks:

- RFID Networks -> RFID tags transmit their unique ID
- Wireless Sensor Networks (WSN) -> made by sensors equipped with batteries that can sense the environment, listen and also transmit. They can also make more complex computations.

Applications of a WSN
WSN can be used everywhere there is a need for monitoring a physical space OR using sensors for controlling a procedure (smart homes, healthcare ecc.).

Structural health monitoring (SHM) allows to detect deterioration and potential damages of a structural system by observing the changes of its material and geometric properties over long periods of time.
Usually there are 3 main risks in a lifetime of a structure:

- During or after the construction
- Due to outer impact
- When it gets old

WSNs can help monitoring with a quality level similar to the conventional (wired) ones and they are both non-obtrusive and non-disruptive.
There are also underwater WSNs useful to interconnect underwater sensors for real-time data and information exchange in marine environments and Aerial WSN (Dronets) to monitor large areas.

Participants in WSNs have 3 roles:

- Sources of data: they can measure data (report measurements and save them somewhere)
- Sinks of data: they can listen for receiving data from the WSN
- Actors/actuators: they can control some devices based on data

Deployment options of a WSN

- Random deployment: uniform random distribution for nodes over a finite area
- Regular deployment: fixed structure
- Mobile sensor nodes: Can be modified by external forces (water, wind)
- Can move itself to compensate

Characteristics of a WSN:

- Scalability: support large number of nodes and performance should not degrade (good performance with increasing of nodes)
- Wide range of densities: it should work both with vast or small number of nodes per unit area
- Limited resources for each device: it should use low amount of energy, low cost, low size, low weight
- Mostly static topology
- Service in WSN: communication is triggered by queries or events and there is asymmetric flow of information
- Quality of service: must apply non-traditional metrics
- Fault tolerance: it must be robust against node failure (battery run out, destruction etc.)
- Lifetime: it should be durable as long as possible without relying on single nodes lifetime
- Programmability: reprogramming of nodes could be necessary to improve flexibility
- Maintainability: WSN has to adapt to changes

Typical adopted mechanisms:

- multi-hop wireless communication
- energy-efficient operation (communication, computation, etc.)
- Self-configuration
- Collaboration and in-network data preprocessing (to improve efficiency)

Mechanisms used to meet the requirements of a WSN:

- Data centric networking: focus network design on data first, not on node identifiers
- Locality: do things locally on the nodes as much as possible
- Exploit tradeoffs: take into account tradeoffs (eg. Energy vs accuracy)

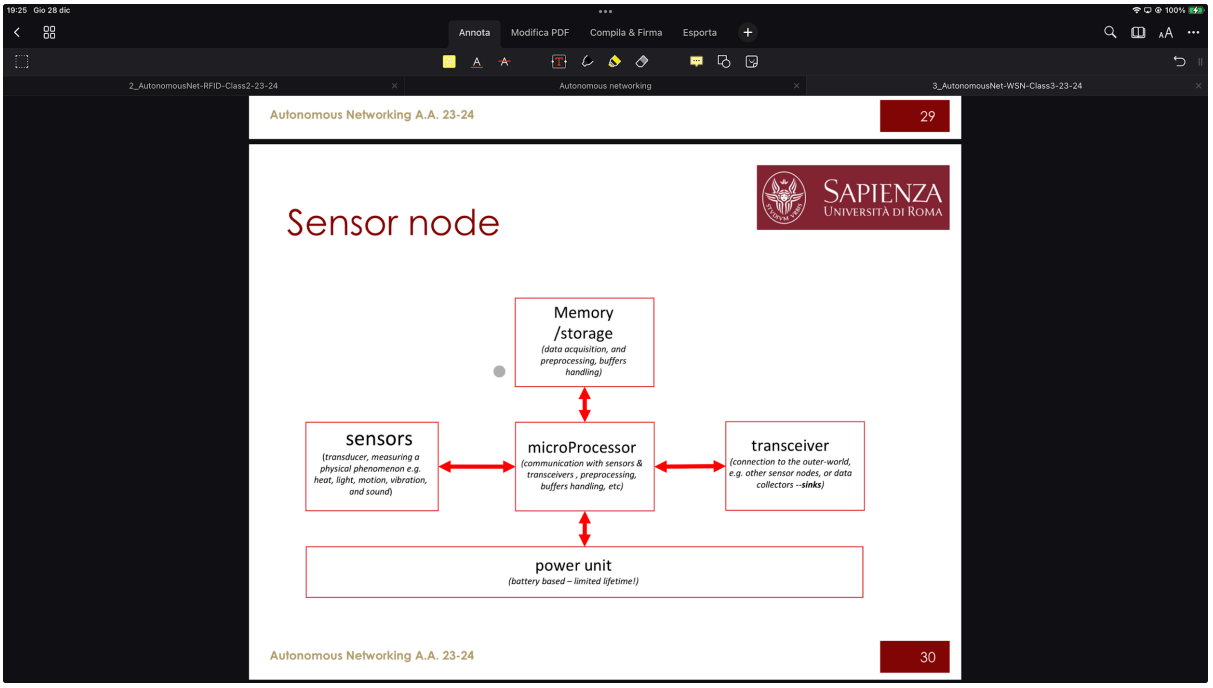
WSN usages:

- Collect: collect information from the environment
- Couple: couple the end-user directly to some measurements
- Provide: provide some precisely localized information just when needed
- Establish: establish a bidirectional link with the physical space

Components of a sensor node

A sensor of a WSN consists of:

- An antenna and radio (RF) transceiver
- A memory unit
- A CPU
- The sensor unit
- The power source (battery usually)
- OS (TinyOS, which is designed to run on platforms with limited computational power and memory space)



How sensing is performed

1. Sensors capture a signal corresponding to a physical phenomenon
2. Then the signal is prepared for further operations (amplification, filtering etc.)
3. Finally Analog-to-digital conversion (ADC) translates the (analog) signal into a digital signal

WSNs vs conventional networks

Conventional Networks	WSN
General purpose design (many applications)	Serving a single application or a bouquet of applications
Network Performance and Latency	Energy is the primary challenge
Devices and networks operate in controlled / mild environments (or over an appropriate infrastructure)	Unattended, harsh conditions & hostile environments
Easily accessible	Physical access is difficult / undesirable
Global knowledge is feasible and centralized management is possible	Localized decisions –no support by central entity

WSNs main characteristics:

- Wireless signal:
  - Attenuation: the strength of the electromagnetic signals decrease as the distance from the transmitter increases
  - Multi-path propagation: encountering an obstacle causes the reflection of the radio wave, losing power. A source signal can reach a station through multiple paths
  - Interference: can happen both from the same single source (due to multipath for example) and from multiple sources (multiple transmitters using the same frequency band to communicate)
- Errors:
  - Signal to Noise Ratio (SNR) measures the ratio of good to bad signal
    - High: signal is stronger than the noise, so it can be converted to data
    - Low: signal has been damage, data cannot be recovered

Packet collision: when a node receives more than one packet at the same time, these packets are termed collided. All packets that cause the collision have to be discarded and retransmissions of these packets are required, which increase the energy consumption.

Hidden terminal problem: A can “see” just B, B can “see” both A and C, C can “see” just B. In order to address this kind of problem, we can use the Medium Access Control (MAC) protocols.
The main objective of MAC is to control how the shared medium (transmission channel) should interact with different devices by controlling when to send a packet and when to listen for a packet, that are the two most important operations in a wireless network, especially because idle waiting means large amounts of energy wasted.
So, MAC objectives are:

- Collision avoidance (and so reducing retransmissions)
- Energy efficiency:
  - Avoid collisions
  - Avoid idle listening
  - Sensors are typically battery powered and battery replacement has high costs, must be avoided as much as possible, low power communication is required

        - Limit overhearing (a node that receives packets destined to other nodes)
        - Avoid overemitting (transmission of a message to a destination node that is not yet ready)
    - Scalability
    - Efficiency
    - Latency
    - Throughput
    - Bandwidth usage

Communication patterns:
    - Broadcast (1 to all)
        - A base station (called sink) transmits some information to all the sensor nodes (called receivers) in the network. Broadcasted information may include queries, program updates for sensor nodes, or control packets for the whole system.
    - Convergecast (all/many to 1)
        - All (or a group) of sensors communicate to the sink. It's typically used to collect sensed data

Properties of a well-defined MAC protocol
To design a good MAC protocol for wireless sensor networks, the following attributes must be considered:
    - Energy efficiency (to prolong network lifetime)
    - Scalability and adaptability to changes (size, node density and topology should be handled)
    - (Optionally) latency throughput and bandwidth utilization efficiency,

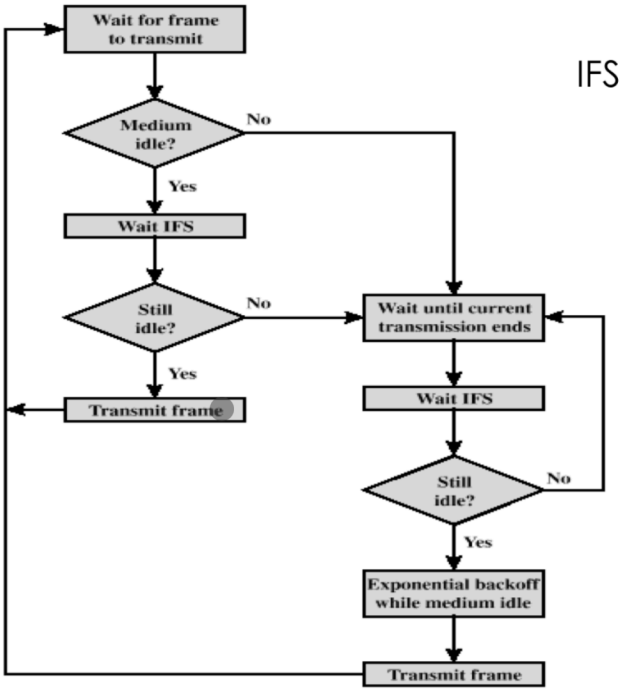
Carrier sense: before transmitting, a node first listens to the shared medium to determine whether another node is transmitting or not

Energy-efficient WSN MAC protocols:
    - Contention based: on-demand allocation for nodes that have data to transmit based on carrier sense mechanism (before attempting a transmission, nodes perform carrier sensing)
        - PRO: Robust, scalable, no central authority needed
        - CONTRO: Idle listening, interference, collisions (collision probability increases with increasing node density), energy consumption
    - Scheduled based: define a schedule for when and how long a node may transmit over the shared medium
        - PRO: energy efficient, no interference, no collision
        - CONTRO: Synchronization and central authorities needed

Collisions
Collisions can't be detected, so they must be avoided as much as possible.

Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)

IEEE 802.11 Distributed Coordination Function



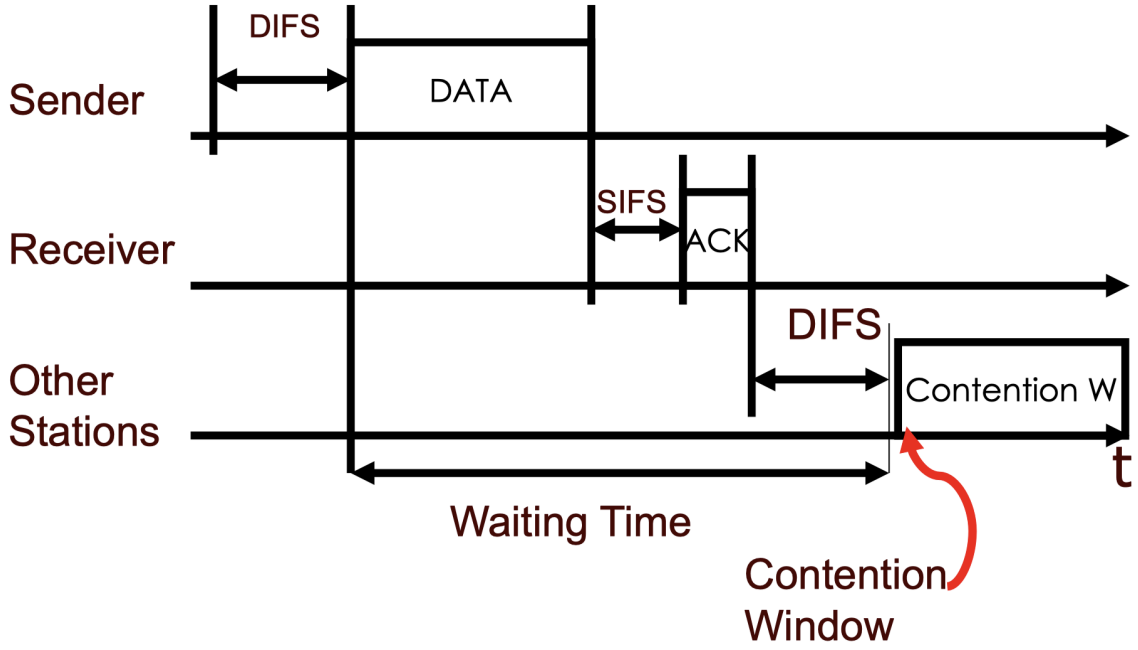
IFS: interframe space

IFS time can be different based on priorities: priorities are defined through different inter frame spaces.

- SIFS (Short inter-frame space): used for high priority packets (ACK, CTS, polling response)
- DIFS (DCF, Distributed Coordination Function IFS): lowest priority, used for async data service

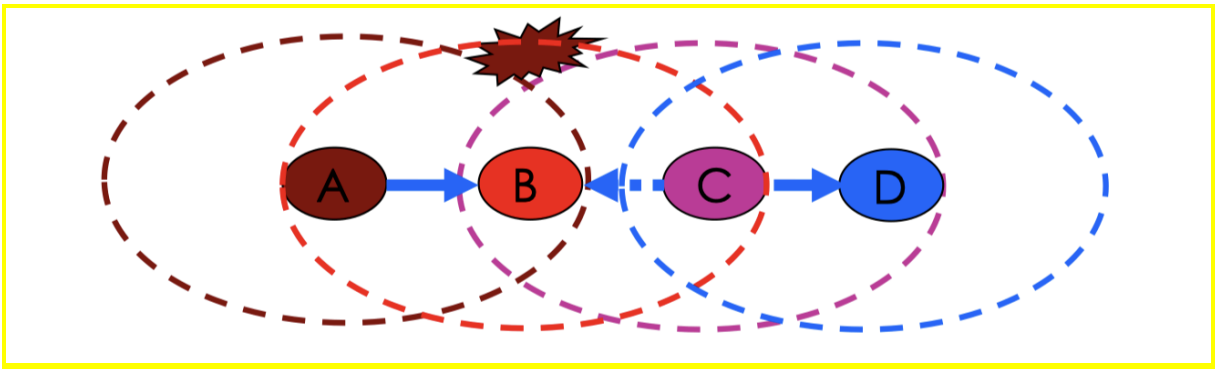
DCF CSMA/CA with ACK

- IF receiver is idle, sender waits for DIFS and then sends the data
- Receiver waits for SIFS and then sends back the ACK to the sender (without sensing the sender)
- If ACK is lost, retransmission is done

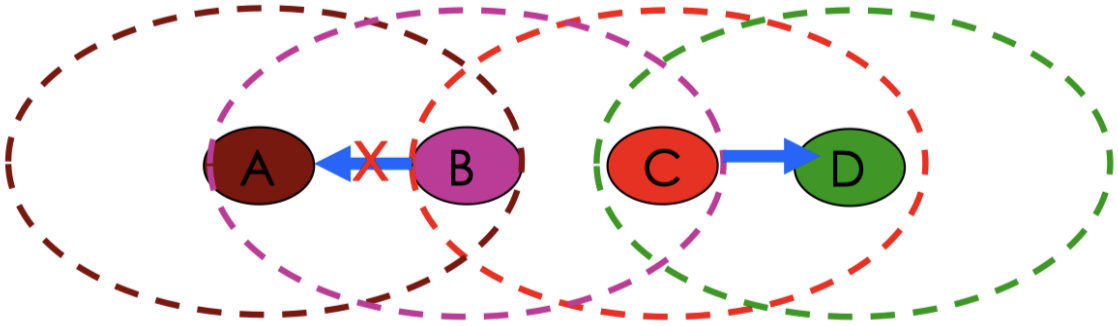


Problems of CSMA/CA

- Hidden Terminal Problem: given A,B,C,D where B can communicate with both A and C and C can communicate just with B and D (so A and C cannot hear each other), when A transmits to B, C cannot hear using carrier sense mechanism, so if C transmits to D there will be a collision at B



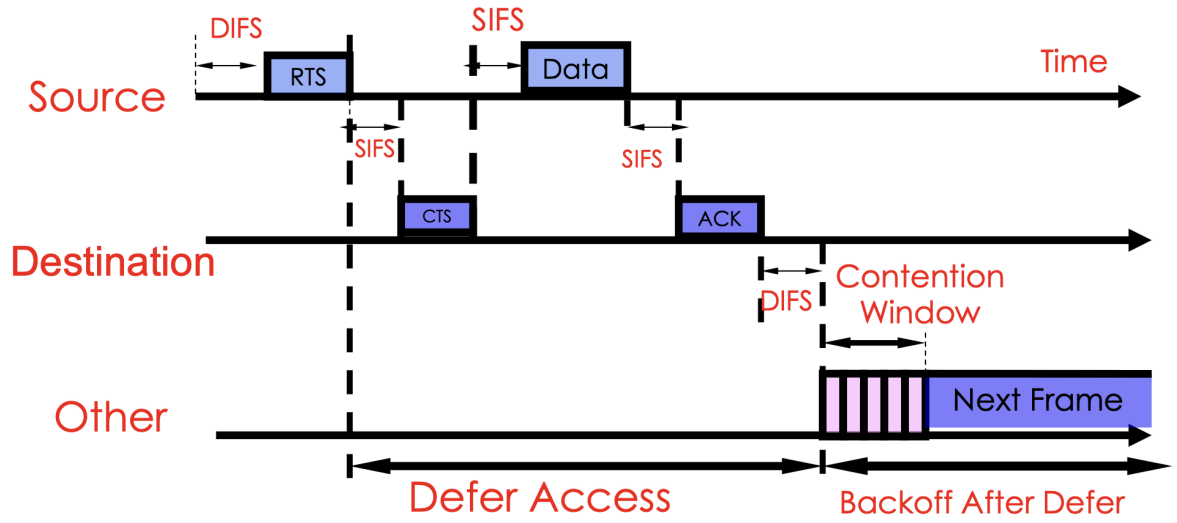
- Exposed Terminal Problem: given A,B,C,D where B can communicate with A and C and C can communicate with B and D, if C transmits to D then B would detect it and it would postpone any transmission to A even if there would be no collision since A cannot hear C



The solution to these problems is RTS/CTS:

- transmitter sends an RTS (request to send) message after the channel has been idle for time>DIFS
- Receiver responds with CTS (Clear to send) after channel has been idle for time>SIFS
- Then, data is transmitted

RTS/CTS is used for reserving channel for data transmission so that the collision can only occur in control message.
Recap of DCF CSMA/CA with RTS/CTS:



- Why is Hidden Terminal Problem solved: considering the example above, when A sends RTS then B sends back the CTS, which is detected by C too, so C will inhibit its own transmitter (avoiding the collision)
- Why is Exposed Terminal Problem solved: B sends RTS to A (detected by C), then A sends back the CTS (which is not detected by C). Since C didn't hear the CTS of A, it assumes A is either down or out of range and doesn't inhibit it's transmission to D

Anyway, collisions are still possible, but just for RTS packets, that are not as bad as data collisions in CSMA since they're much smaller typically.

Virtual Carrier Sensing -> provided by the NAV (Network Allocation Vector), adds a duration field to each frame (both RTS and CTS) which is used to reserve the medium (cannel) for a fixed amount of time. Transmitter sets the NAV to the expected time of transmission, other stations detecting that packet will start a countdown from NAV to 0. When the channel is virtually available, then MAC checks for the physical condition of the channel.

Lecture 4

S-MAC
Sleep-MAC is another contention-based Mac protocol. It's based on the idea of nodes not listening all the time. Aims at reducing energy waste and overhearing. It implements a

periodic listen and sleep cycle. Each node goes into periodic sleep mode during which it switches the radio off and sets a timer to awake later. When the timer expires it wakes up and listens to see if any other node wants to talk to it.
All nodes are free to choose their own listen/sleep schedules. To reduce control overhead, neighboring nodes are synchronized together (they listen at the same time and go to sleep at the same time). Each node maintains a table with neighbors' schedules, SYNC packets are exchanged periodically to maintain schedule synchronization. These packets contain the sender node ID and the next sleep time.
This protocol uses RTS/CTS to avoid collisions and performs carrier sense before transmitting and, using the duration field, every node knows how much it has to sleep before retrying.

Routing -> technique needed for sending data between sensor nodes and base stations in order to establish multi-hop communication.

Flat routing protocols for WSNs -> 3 main categories based on the routing strategy:

- Proactive protocols: each node always tries to keep its routing data updated, each node maintains routes to all reachable destinations at all times
    - Fast but involves overhead, meaning high power consumption
    - Destination Sequence Distance Vector (DSDV)
- Reactive protocols: route determined only when it's actually needed, when a route from a source to a destination is needed, a kind of global search procedure is started. How it works: node S sends the packet with dest=D and source=S, all the intermediate nodes store the information of how to get to S and rebroadcast the original packet until D is reached, so every node also knows how to get to S.
    - It's slower, but it generates less overhead
    - Some reactive protocols:
        - Flooding: copies of incoming packets are sent by every link except the one by which the packets arrived.
            - Enormous amount of traffic
        - Gossiping: nodes send the incoming packets to a randomly selected neighbor
            - Slow propagation of the message across the network
        - Dynamic Source Routing (DSR): every node that wants to send data packets includes in the packet header the complete, ordered list of nodes through which the packet will pass, so that other nodes forwarding or overhearing any of these broadcasted packets can also easily cache this routing information for future use
- Hybrid protocols: combination of the two behaviors
- Geographic routing -> infer the information of which is the next hop by physical placement (position of current node, position of neighbors etc.). Send it to that neighbor that is closest to the destination (or to the current node).
    - Saves much energy
    - Problem: could send a packet to a dead end
    - Geographic routing protocols in sensor networks leverage location information to determine the next hop for forwarding data from a source node to a destination.
    - Geographic routing does not use routing tables but position information.

- Main operation of a geographic routing protocol:
    - Localization: Sensor nodes determine their geographical coordinates using localization techniques, such as GPS.
    - Neighbor Discovery: Nodes identify their neighboring nodes and exchange location information. This process creates a network map that the routing protocol can use to establish efficient communication paths.
    - Route Discovery: When a source node wants to transmit data to a destination, it consults its location information and calculates the direction toward the destination. The source then selects a neighbor that is closer to the destination and forwards the data.
    - Hop-by-Hop Routing: Each intermediate node, based on its own location information, repeats the process of selecting the next-hop neighbor that is closer to the destination. This hop-by-hop process continues until the data reaches the destination node.
    - Routing decision: Greedy protocols make local decisions at each hop based on proximity to the destination (most forward)

What if the network topology is continuously changing, like for drones?

Lecture 5

Smart object -> everyday physical objects having:

- a name
- Communication capabilities
- Computing capabilities
- Physical phenomena sense capability - this in particular is the difference with entities (routers, terminals etc.) -> direct interaction with the environment

IoT pillars (related to smart objects):

- Be identifiable
- Can communicate
- Can interact (among themselves in a network or with end-users)

RFIDs and Sensor/actor networks (SANETs) are key devices in IoT.

IoT is a highly dynamic and radically distributed networked system, composed of a very large number of smart objects producing and consuming information. Its ability to interface with the physical environment is achieved through:

- the presence of sensors: devices able to sense physical phenomena and translate them into a stream of data
- The presence of actuators: devices able to trigger actions having an impact on the physical environment

Main issues:

- Scalability
- Self-management

Quick recap
Sensors -> passively interface with the physical environment (perform sensing operation), low-cost and low-power
Actors -> actively interface with physical environment (performing actions), resource-rich devices equipped with high processing capabilities

SANETs
SANETs are distributed wireless systems of heterogeneous devices referred to as sensors and actors (Sensor/actor networks).

Backscattering
It's the reflection of signals back to the direction from which they come. It makes it possible to have smart objects without batteries. It's possible to use the radio frequency (RF) signals as a power source and use them to sense, compute and transmit via reflection of the RF signal.
Two backscattering techniques:

- Ambient backscattering: devices harvest power from signals available in the environment
    - Pro: use of existing RF signals, no further RF emitter required
    - Contro: low data rate, possible weak signal (outdoor and mostly indoor)
- RFID backscattering: power is harvested from the signal emitted by RFID reader
    - Pro: signal is always available (as the reader is always present in that system)
    - Contro: there must be the reader

Sensor Augmented RFID Tags
Limited power (no battery), limited range, limited time, limited storage and limited datarate, but still can run sensors.
Use the EPC Global Standard protocol for communication.
Many problems appear when working with 2+ SA RFID Tags.

Infrastructure-based Wireless Networks
Based on infrastructure (GSM etc.), consist of base stations that are connected to a wired backbone network. Mobile devices communicate wirelessly with the base stations. Traffic between different mobile devices is managed by base stations and the wired backbone network. Mobility is supported by switching from one base station to another.
Contro: requires infrastructure, expensive, long setup time

Infrastructureless-based Wireless Networks

- Wireless Ad Hoc Networks: network without central infrastructure, relies on the participants' networking abilities. Requires self-organizations. Challenges are:
    - Lack of central entity
        - Discovering devices presence
        - MAC (transmission resources must be decided in a distributed way)
        - Find a route from a participant to another
    - Limited range
        - Can be addressed with multi-hop wireless networks
    - Mobility of participants

Lecture 6

Unmanned Aerial Vehicle (UAV)
It's the common drone, an aircraft without a human pilot abroad. Can operate autonomously or under remote human control. Can have Wifi, cellular or other radio modules.

Dronet (aka UAVNET)-> network of drones that cooperate in a mission, can exchange a lot of data among themselves and/or the control station (depot). Have higher mobility.
It can use reactive protocols but not proactive protocols because these would generate slow reaction to topology changes causing delays (topology changes very often in Dronets).
Hybrid protocols are suitable too and also geographical protocols: every drone has a GPS device embedded so routing doesn't need to know the entire network, just the neighbor UAVs and destination UAV information are required.

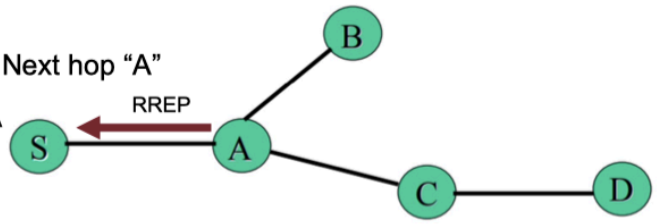
Different network types comparison

	WSN	IoT	Dronet
Mobility	None or partial	None or Low	**High (even 3D)**
Topology	Random, ad-hoc Node failure	Star or ad hoc	Mesh **Jeopardized**
Infrastructure	Absent (sink)	Partial (central point)	**Absent (depot)**
Energy source	battery	Wired, battery, backscattering	**battery**
Typical use	Environmental monitoring	Smart environments	Rescue, monitoring, Surveillance

AODV – Ad Hoc On demand Distance Vector

DRONETS are similar to Mobile Ad-Hoc Networks (MANETs) and WSN, but typically have much higher mobility. AODV is a reactive protocol used for Dronets:

- Node S wants to communicate with D
- S broadcasts RREQ packets with Destination “D” and Source “S”
- A stores route reversal: Dest “S” – Next hop “S”
- A rebroadcasts RREQ
- C stores reversal route: Dest “S” – Next hop “A”
- C creates RREP and unicast it to A
- A stores: Dest “D” – Next hop “C”
- A sends RREP in unicast to S
- **S stores the route: Dest “D” – Next hop “A”**



Lecture 7

Reinforcement Learning (RL) -> how an intelligent agent can learn to make a good sequence of decisions, where the “goodness” is given by some utility measure over the decisions.
RL is learning what to do - how to map situations to actions so as to maximize a numerical reward signal.
Two main characteristics:

- Trial-and-error search: learner must discover which action yield the most reward by doing it
- Delayed reward: in many cases an action can affect multiple next situations and, so, the subsequent rewards, not just the immediate one

A learning agent must be able to:

- Sense the state of the environment
- Take actions that affect the state

It must also have a goal related to the state of the environment.

RL is different from other ML paradigms in:

- No supervisor
- Feedback can be delayed
- Time matters (sequential flow)
- A good sequence generates a “reinforcement” in the process
- Online learning (learning while interacting) rather than just learning from data (training)

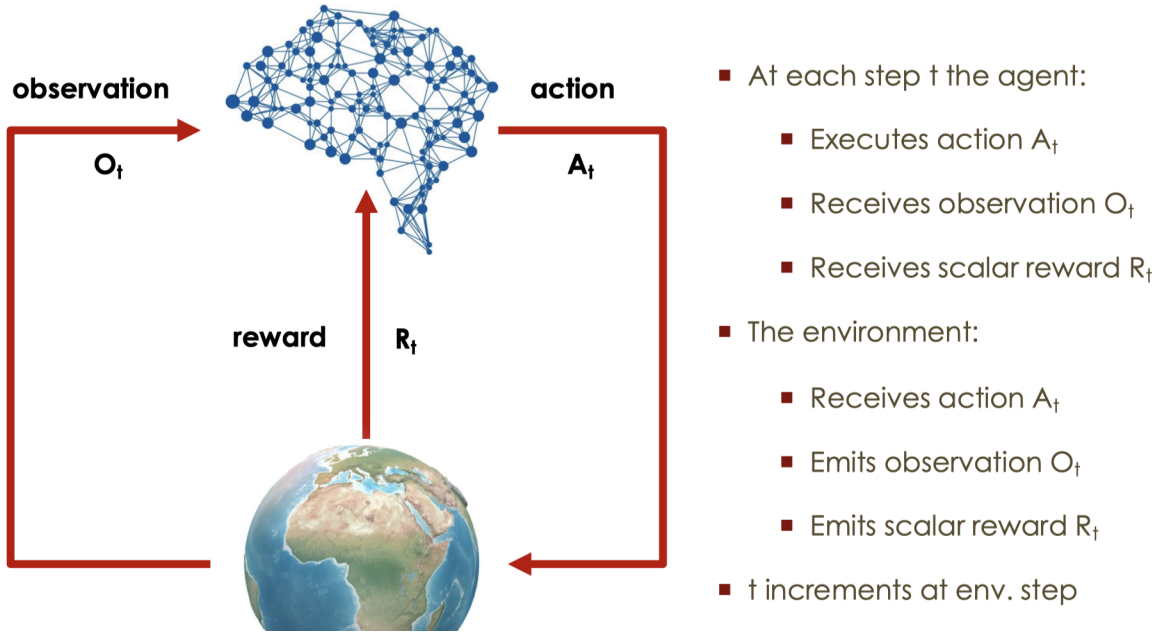
Applications: self-driving cars, industry automation

Reward -> a reward Rt is a scalar feedback signal that indicates how well the agent is doing at step t. Agent’s job is to maximize it.

Reward hypothesis: all goals can be described by the maximization of expected cumulative reward.

Exploration vs Exploitation trade-off: retrying actions that it found to be effective in the past producing reward (Exploit) is good, but in order to discover such actions it has to try new actions not selected before (Explore), so the agent has to exploit what it has already experienced in order to obtain reward, but it has to explore in order to make better action selections in the future. The problem arises from incomplete information: we need enough info to make the best decisions, but how much is enough?
With exploitation, we take advantage of the best option we know. With exploration, we take some risk to collect information about unknown options.
The agent must try a variety of actions and progressively favor those that appear to be best.

General RL framework



History (Ht) -> sequence of observations, actions and rewards up to t

State (St)-> a function of the history: St=f(Ht)

Environment state (Ste) -> environment's private representation (not visible to the agent usually, i.e. whatever data the environment uses to pick the next observation/reward)

Agent state (Sat) -> agent's internal representation (i.e. whatever info is used by agent/RL algorithm to pick next action), can be any function of history: Sat = f(Ht)
Agent must construct its own state representation
If elements are fully observable, then agent state = environment state

RL Agent -> may include one or more:

- Policy: agent's behavior function, map from state to action (core of the RL agent). It's the method we use to choose the action. The reward signal specifies what is good in an immediate sense and it can lead to alteration of the policy: if an action selected by the policy leads to low reward then policy might be changed to select some other action. Can be:
  - Deterministic: simple function of the state

    - Stochastic: specify probabilities for each action
- Value function: how good is each state and/or action, specifies what is good in the long run. It's a prediction of future reward (total amount of reward the agent can expect to accumulate starting from the current state). Values are predictions of reward
    - Example: a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards
- Model: agent's representation of the environment, predicts what the environment will do next (and optionally the next reward)

Lecture 8

K-armed bandit problem -> you're faced repeatedly with a choice among K different options (action) and after each choice you receive a reward chosen from a stationary probability distribution (that depends on the action you selected). The objective is to maximize the expected total reward over some time period.
Formalization of k-armed bandit:

- S -> one single state
- A -> set of actions (or "arms")
- R -> reward function (that follows some **unknown** probability distribution)

At each step, the agent selects an action from A and the environment generates a reward.
The goal is to maximize the cumulative reward.
Action value -> it's the value of selecting action a, defined by the expected reward we receive by selecting that action

Action value function is then defined as follows:

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$$

where Rt is the reward at timestep t and At is the action selected at timestep t.
The goal is to maximize the expected reward, but the reward distributions and their expected values are unknown at the beginning.

Action-values estimation (methods to evaluate action values)

There are 2 methods for action-values estimation (i.e. how to estimate action-value), basically it's the same method implemented in two different ways:

- Sample-average method
- Incremental method
    - Stationary problems
    - Non-stationary problems

Sample-average method

Since the reward distribution is not known in advance, we maintain, for each action a, a record of all the rewards that have followed the selection of that action. Then, when the estimate of the value of action a is needed at time t, it can be computed as the sum of the rewards observed when taking that action divided by the total number of times that action has been taken:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}},$$

where denominator 1 predicate denotes some random variable that is 1 if the predicate is true and 0 otherwise. If the denominator is 0, then Qt(a) = 0.

Incremental method

A problem with the sample-average method is that its memory and computational requirements grow over time without bound. So, we can devise incremental update formulas for computing averages with small, constant computation required to process each new reward. For some action a, given Ri= reward received at the i-th selection of a and Qn=the estimate of the action value after a has been selected n-1 times such that:

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}$$

Then, given Qn and the n-th reward Rn, the new average of all the n rewards can be computed as follows:

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i = Q_n + \frac{1}{n}\Big[R_n - Q_n\Big]$$

General form:

$$NewEstimate \leftarrow OldEstimate + StepSize\Big[Target - OldEstimate\Big]$$

Useful explanation: http://incompleteideas.net/book/ebook/node19.html

Pseudocode for a bandit algorithm using incrementally computed sample average and ε-greedy action selection:

A simple bandit algorithm

Initialize, for $a = 1$ to k:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:
$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

The function bandit(a) is assumed to take an action and return a corresponding reward.

The averaging methods seen so far are suitable for stationary bandit problems (i.e. bandit problems in which the reward probabilities do not change over time).
For non-stationary problems (i.e. problems with reward probabilities changing over time), one option is to use a fixed rewards step size:

$$Q_{n+1} \doteq Q_n + \alpha\left[R_n - Q_n\right] = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_n$$

Where $\alpha \in (0,1]$

Action selection techniques

Action selection techniques (i.e. how to select an action)

- Random -> take actions randomly
- Greedy
- ε-greedy
- Optimistic initial values

Greedy

The greedy action is the action that has the largest estimated value (the one that can give the most reward right now). Selecting the greedy action means the agent is exploiting its current knowledge to maximize immediate reward. Greedy actions can lead to non optimal actions since the agent doesn't consider apparently inferior actions. The greedy action is computed as the argmax of the estimated values:

$$A_t \doteq \arg\max_a Q_t(a)$$

ε-greedy

Behave greedy most of the time (exploitation), but sometimes (with a small probability ε) select randomly among all the actions with equal probability without considering action value estimates (exploration)

$$A_t \leftarrow \begin{cases} \underset{a}{\text{argmax}}\ Q_t(a) & \textbf{with probability } 1-\epsilon \\ a \sim Uniform(\{a_1 \dots a_k\}) & \textbf{with probability } \epsilon \end{cases}$$

As the number of step increases Qt(a) converges to q*(a).

Optimistic initial values

All the other action selection methods are based on the initial action-value estimates. The key of this method instead is to use initial action values as a simple way to improve exploration, so the system will do a good amount of initial exploration even selecting greedy actions [praticamente ogni azione viene considerata ottima finché non viene verificato il contrario, questo ovviamente incentiva l'esplorazione]. All actions are tried many times before the value estimates would converge.

Lecture 9

To assess the performance of action selection methods we use a 10-armed bandit problem [non l'ho capito]

Optimism in the face of uncertainty principle: do not take the arm you think it's the best, take the one that has the most potential to be the best.

How to estimate/reduce uncertainty? We use a confidence interval, that is the region of values representing the uncertainty (i.e. the real value will be somewhere in that region):

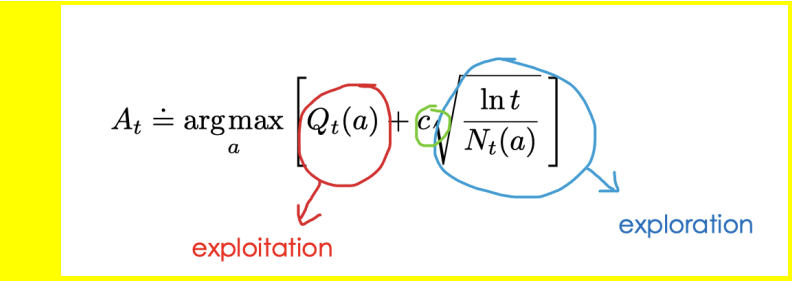
- small confidence interval -> very certain about the estimated value
- large confidence interval -> uncertain about the estimated value

The more uncertain we are about an action-value, the more important it is to explore that action.

Upper Confidence Bound

UCB is an action selection strategy that follows the optimism principle: if we are uncertain about something, we should optimistically assume that it is good, so for example given 3 actions with associated uncertainties, the agent will (optimistically) pick the action that has the highest upper bound: the result is that either we get a good reward or we learnt more information about that unknown action (so both positives).
UCB uses uncertainty in the estimates to drive exploration.

UCB formally:



where:

- t is the timesteps
- Qt(a) here represents the current estimate for action a at time t
- Nt(a) is the number of times action a is taken
- c is a user-specified parameter that controls the amount of exploration

We select the action that has the highest estimated action-value (exploitation) plus the upper-confidence bound exploration term (i.e. uncertainty term).
UCB explores more to systematically reduce uncertainty but its exploration reduces over time. Thus we can say that UCB obtains greater reward on average than other algorithms such as Epsilon-greedy, Optimistic Initial Values, etc. but has difficulty in dealing with nonstationary problems.

Lecture 9b

The first step in applying reinforcement learning is to formulate the problem as an MDP.
Given a Markov process, if we add rewards we get a Markov Reward Process and then adding actions we get a Markov Decision Process.
MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations (states). MDPs also involve delayed rewards concept and the related need to trade off between immediate and delayed rewards.

Bandit -> **no state concept** in the estimation process, we just estimated q*(a) of each action
MDPs -> we estimate the value q*(a,s) of each action a in each state s, or we estimate the value v*(s) of each state s given optimal action selection.

The agent interacts with the environment at each timestep, receiving a representation of the environment state and, on that basis, it selects an action. One time step later (in part as a consequence of the selected action), it receives a numerical reward and it finds itself in a new state.

An MDP formally describes the environment for RL where the environment is fully observable. For example, bandits are MDPs with one state.

Markov property -> “the future is independent of the past given the present”, i.e. the state is a sufficient statistic of the future because the state captures all the relevant information from the history, which could be thrown away once the state is known.
Formally: a state St is Markov if and only if $\mathbb{P}[S_{t+1} | S_t] = \mathbb{P}[S_{t+1} | S_1, ..., S_t]$

Given a Markov state s and successor state s’, the state transition probability is defined by:

$$P_{ss'} = \mathbb{P}[S_{t+1}=s' \mid S_t=s]$$

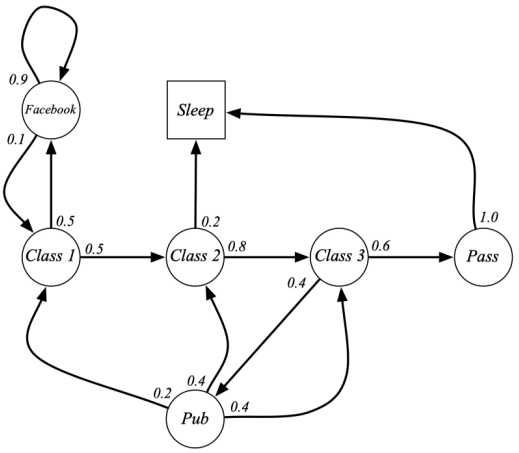
The state transition matrix P defines the transition probabilities from each state s to all successor states s’ (each row of the matrix sums to 1):

$$\mathcal{P} = \text{from} \begin{matrix} \\ \\ \\ \end{matrix} \overset{\text{to}}{\begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix}}$$

A **Markov process** is a memoryless random process (or a sequence of random states S1, S2… that respects the Markov property). Formally, a Markov Process (or Markov Chain) is a tuple $\langle S, P \rangle$ where:

- S is a (finite) set of states
- P is a state transition probability matrix, $P_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$

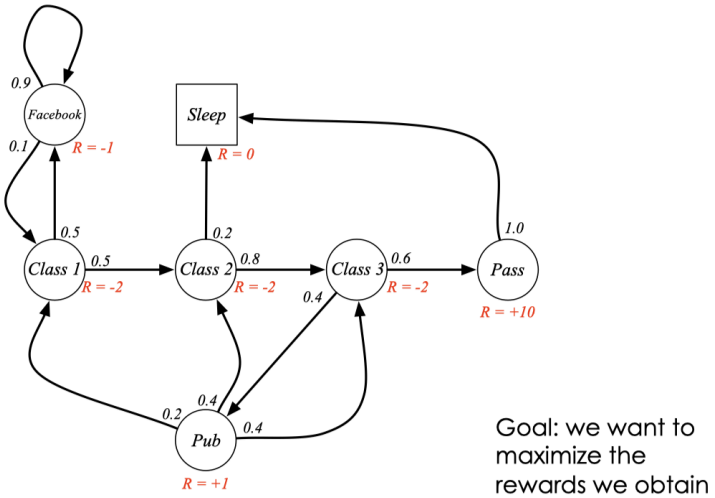
Example of a MP:



A **Markov reward process** is a Markov process with values (rewards). Formally:
A Markov Reward Process is a tuple $\langle S, P, R, \gamma \rangle$

- S is a (finite) set of states
- P is a state transition probability matrix, $P_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$
- R is a reward function, $R_s = \mathbb{E}[R_{t+1} \mid S_t = s]$
- γ is a discount factor, $\gamma \in [0, 1]$

Example of a MRP:



Return ->the return Gt is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where:

- The discount γ ∈ [0, 1] is the present value of future rewards
- The value of receiving reward R after k + 1 time-steps is γk R

This values immediate reward above delayed reward

- γ close to 0 leads to ”short-sighted” evaluation
- γ close to 1 leads to ”far-sighted” evaluation

In practice, we discount rewards into the future by the discount rate γ ∈ [0, 1].

Most Markov reward and decision processes are discounted because:

- It’s mathematically convenient to discount rewards
- Avoids infinite returns in cyclic Markov processes
- Uncertainty about the future may not be fully represented

It is sometimes possible to use undiscounted Markov reward processes (i.e. γ = 1), for example if all sequences terminate.

episode -> a path of states

State value function -> the state value function v(s) gives the long-term value of (being in) state s (i.e. it’s the expected return starting from state s). Formally:

$V = \mathbb{E}\ [\ Gt \mid St = s\ ]$

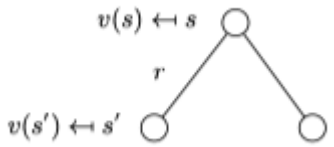
In order to solve the state value function of a state S, we do a lot of episode samples for that state, then for each episode we compute the discounted return and we do the average dividing by the total number of episodes for that state (so that it can converge to the true value function). By doing this for all the states we can compute the value of being in every state.

Bellman Equation for MRPs

The value function can be decomposed in 2 parts:

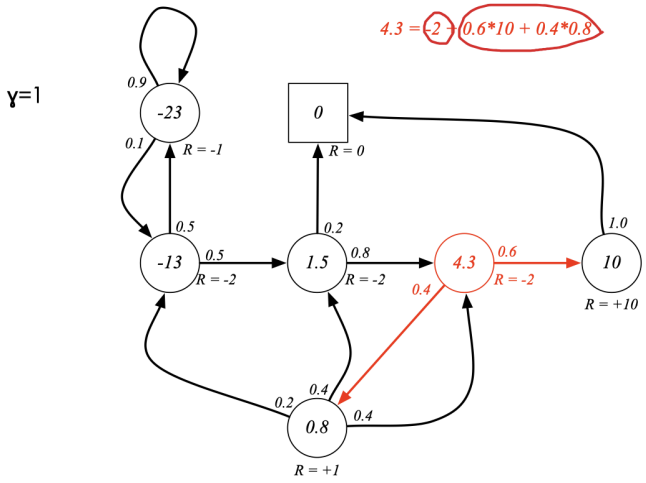
- immediate reward: Rt+1
- discounted value of successor state: γ (S□+1)

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$



The Bellman equation expresses a relationship between the value of a state and the values of its successor states. It averages over all the possibilities, weighting each by its probability of occurring. The value of the start state must be equal to the (discounted) value of the expected next state, plus the reward expected along the way. Computational complexity complexity is O(n^3) for n states. Direct solution is only possible for small MRPs, while there are many iterative methods for large MRPs (dynamic programming, Monte-Carlo evaluation etc.).

Example:



A **Markov decision process** (MDP) is a Markov reward process with decisions. It is an environment where all the states are Markov. Formally:

A Markov Decision Process is a tuple ⟨ S, A, P, R, γ ⟩

- S is a (finite) set of states
- A is a finite set of actions
- P is a state transition probability matrix, P ss′ - = ℙ [St+1 =s′ | St =s, At =a] (one matrix for each action)

- R is a reward function, R s - = $\mathbb{E}$ [R t+1 | S t = s, At =a]
- γ is a discount factor, γ ∈ [0, 1]

Now we can choose the actions, so the goal is to find the best path to maximize rewards, so the concept of policies comes back.

Policy -> a policy is a distribution over actions given states. It fully defines the behavior of the agent and it depends just on the current state, not the history. It is stationary (time-independent, does not depend on the time step, but only on the state).

$$\pi(a \mid s) = \mathbb{P}\left[A_t = a \mid S_t = s \right]$$

So, the state value function $v_\pi(s)$ of an MDP becomes the expected return starting from state s and then following the policy π, formally:

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s \right]$$

And then the action-value function $q_\pi(s,a)$ is the expected return starting from state s, taking action a and then following policy π, formally:

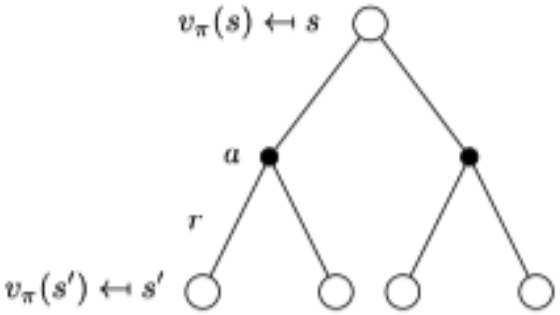
$$q_\pi(a \mid s) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a \right]$$

Bellman Expectation Equation [policy]

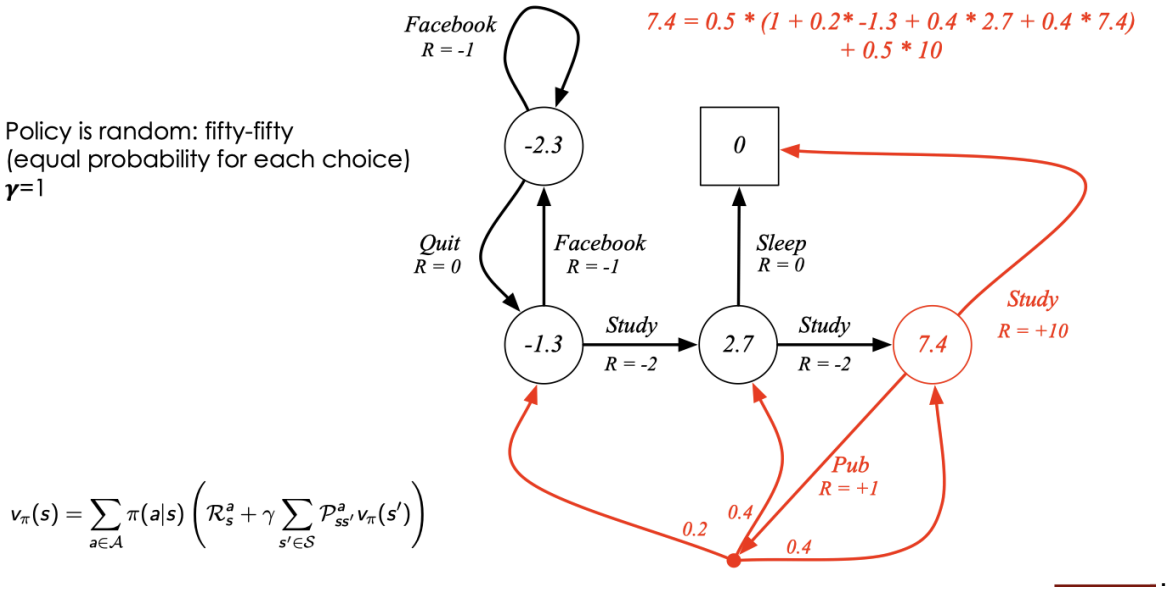
The BEE tells us that the value of a particular state is determined by the immediate reward plus the value of successor states when we are following a certain policy(π) with a discount factor(γ).
The difference between Bellman Equation and Bellman Expectation Equation is that now we are finding the value of a particular state subjected to some policy(π).
Given that the probability of taking one action or the other depends on the policy, for each action I might take there is a Q-value. We average the Q-values which tells us how good it is to be in a particular state. So, the value functions can also be written in the form of a Bellman Expectation Equation as follows:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$



Example:



Lecture 10

Good explanation:
https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3

So, this is how we can formulate Bellman Expectation Equation for a given MDP to find it's State-Value Function and State-Action Value Function. But, it does not tell us the best way to behave in an MDP. For that let's talk about what is meant by Optimal Value and Optimal Policy Function.
So far, policies were just given, but the goal of RL is not just to evaluate policies but also to find a policy that obtains as much reward as possible in the long run. Finding the best solution to MDP becomes finding the optimal policy.
In order to define an optimal policy, we must understand how to compare policies first.

Optimal state-value function v*(s) -> it's the maximum value function over all policies, it tells us the maximum possible reward you can extract from the system. It says how good is to be in each state, but it does not say how to behave. Formally:

$$V_*(s) = \max_\pi V_\pi(s)$$

Optimal action-value function q∗(s,a) -> it's the maximum action-value function over all policies, it tells which actions to take to behave optimally (i.e. iit tells us what is the maximum possible reward you can extract from the system starting at state s and taking action a).
Formally:

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

If you know q∗ then you know the right action to take and behave optimally in the MDP and therefore solving the MDP.

Ordering over policies

$$\pi \geq \pi' \text{ if } V_\pi(s) \geq V_{\pi'}(s),\ \forall s$$

Theorem: For any Markov Decision Process:

- There exists an optimal policy $\pi*$ that is better than or equal to all other policies such that $\pi* \geq \pi$, $\forall \pi$
- All optimal policies achieve the optimal value function, $V\pi*(s) = V*(s)$
- All optimal policies achieve the optimal action-value function, $q\pi*(s,a) = q*(s,a)$

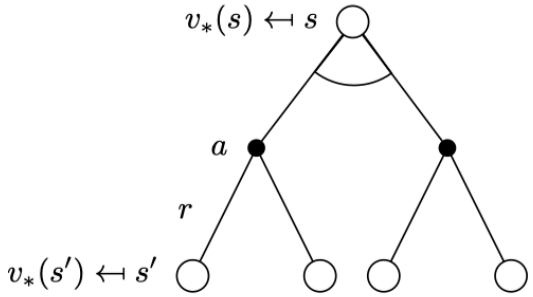
Optimal policy
An optimal policy can be found by maximizing over q∗(s, a):

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ q_*(s,a) \\ 0 & otherwise \end{cases}$$

This says that for a state s we pick the action a with probability 1, if it gives us the maximum q*(s,a). So, if we know q*(s,a) we can get an optimal policy from it.
There is always a deterministic optimal policy for any MDP.

Bellman Optimality Equation

Bellman Optimality equation is the same as Bellman Expectation Equation but the only difference is instead of taking the average of the actions our agent can take the action with the max value.
Given the following Diagram for State-Action Value Function(Q-Function):



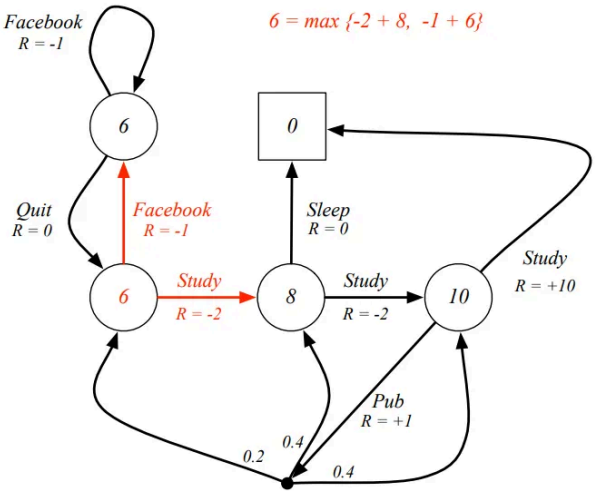
Suppose our agent is in state S and from that state it can take two actions (a). So, we look at the action-values for each of the actions and unlike, Bellman Expectation Equation, instead of taking the average our agent takes the action with greater q* value. This gives us the value of being in the state S.
Formally, the Bellman Optimality Equation for State-value Function is:

$$v_*(s) = \max_a q_*(s, a)$$

And the Bellman Optimality Equation for State-Action Value Function is:

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a')$$

Example:



Bellman Optimality Equation is non-linear, so it has no closed form solution. Anyway, there are many iterative solution methods, one of which is the Q-learning.

RECAP
Bellman Equation

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

Bellman Expectation Equation

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Bellman Optimality Equation

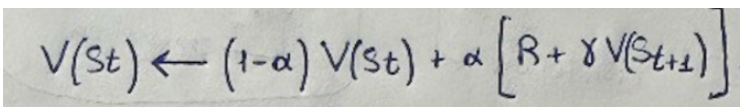
$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Temporal Difference (TD) Learning

TD Learning aims at solving the prediction problem: estimating the value function v_π for a given policy π

Rather than attempting to calculate the total future reward, temporal difference learning just attempts to predict the combination of immediate reward and its own reward prediction at the next moment in time. Now when the next moment comes and brings fresh information with it, the new prediction is compared with the expected prediction. If these two predictions are different from each other, the Temporal Difference Learning algorithm will calculate how different the predictions are from each other (TD-error) and make use of this temporal difference to adjust the old prediction toward the new prediction. TD methods learn directly from experience: at each step the state value function is updated:

Key point: sample updates are based on the single successor state rather than on the complete distribution of all the successor states.

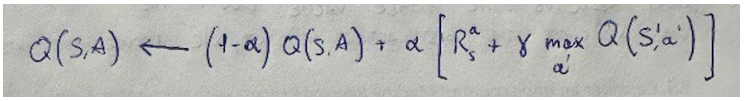


This is the simplest version of TD, called TD(0), where the zero represents the fact that one has to wait for a single step to perform the update, yet the idea is extendable to more complex algorithms like n-step TD and TD(λ). The update rule of a state makes use of the estimates of another one: this method is called bootstrapping.

It includes a learning rate parameter (α) that defines how quickly Q-values are adjusted

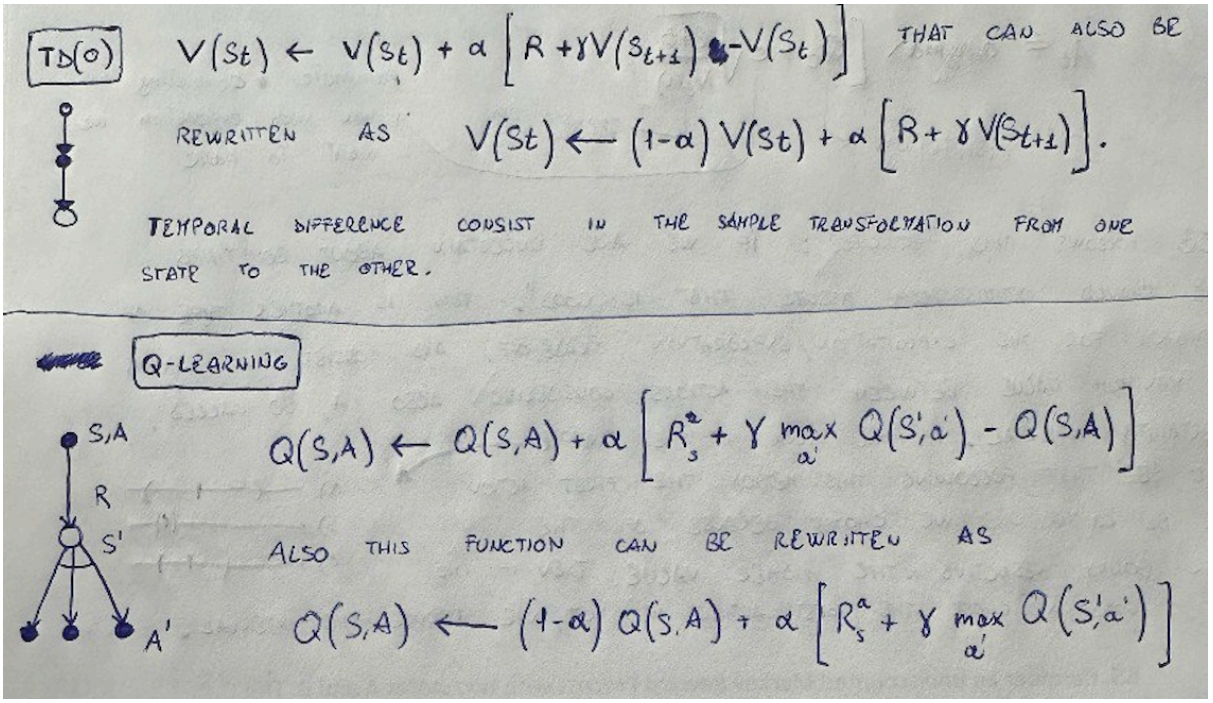
Q-learning

Q-Learning is an off-policy TD method to solve the Control problem (estimates the Q value function Q(s,a) i.e. finding an optimal policy). In off-policy learning, the optimal value function is learned from actions taken independently from the current policy $\pi(a|s)$. It aims to determine the optimal action based on its current state.



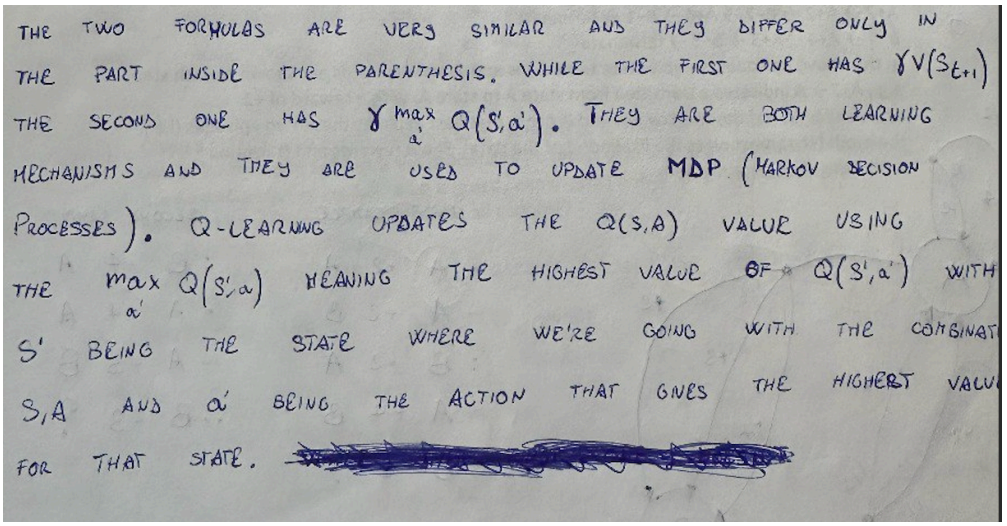
The Q-values are the expected future values for action (somma delle future ricompense scontate).

Temporal difference vs Q-learning



Where S' is the state we reach by taking action a from s.

First difference: they solve different problems, TD solves the prediction problem (tries to estimate the state value function $V\pi(s)$) while Q-learning solves the control problem (estimates the Q value function Q(s,a)).
Second difference:

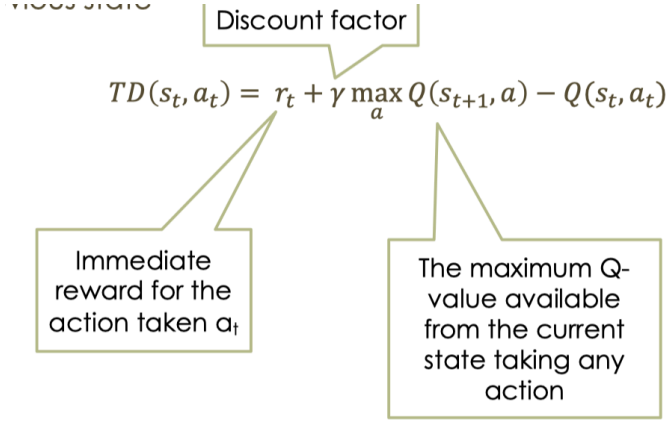


Q-learning

Q-learning aims at solving the control problem: finding an optimal policy.
Q-learning is a machine learning approach that enables a model to iteratively learn and improve over time by taking the correct action. Q-learning is a type of reinforcement learning. It also takes an off-policy approach to reinforcement learning: a Q-learning approach aims to determine the optimal action based on its current state and it can accomplish this by either developing its own set of rules or deviating from the prescribed policy, so a defined policy is not needed.
Off-policy approach in Q-learning is achieved using Q-values -- also known as action values. The Q-values are the expected future values for action (somma delle future ricompense scontate).

There are two methods to determine the Q-value:

- **Temporal difference**: this formula calculates the Q-value by incorporating the value of the current state and action by comparing the differences with the previous state and action.
  Q-learning uses TD prediction for the control problem by applying TD to the Q-value and the TD error becomes how much should you adjust the Q-value for the previous state.



- **Q-learning**: it tells us what new value to use as the Q-value for the action taken in the previous state. It relies on both the old Q-value for the action taken in the previous state and what has been learned after moving to the next state. Includes a learning rate parameter (α) that defines how quickly Q-values are adjusted

$$Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha\, TD(s_t, a_t)$$

The Q-learning process involves modeling optimal behavior by learning an optimal action value function or q-function. This function represents the optimal long-term value of action a in state s and subsequently follows optimal behavior in every subsequent state.

Pro:

- Model-free: rather than requiring prior knowledge about an environment, the Q-learning agent can learn about the environment as it trains
- Off-policy optimization: the model can optimize to get the best possible result without being strictly tethered to a policy that might not enable the same degree of optimization.

Contro:

- Exploration vs. exploitation tradeoff: it can be hard for a Q-learning model to find the right balance between the two

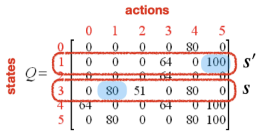
Lecture 12 - Deep Reinforcement Learning

RL -> an agent wants to learn a policy (how to act) in an environment to maximize the cumulative reward obtained from the environment
Optimize policy -> discounted reward
Given an MDP, the transition probability function is known only in model-based RL value/policy iteration algorithms
The objective is to learn π* that maximizes cumulative discounted reward

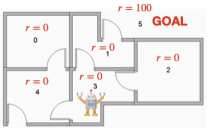
Q-learning

Optimal policy $\pi^*(s) = \arg\max_{a \in A} Q^*(s,a)$

- **set** $Q(s,a) = 0 \; \forall s \in S, a \in A$
- **while** true:
  - from state s (random at the start of the epoch) let the process evolve, until s is end, as:
    - choose action *initially random* with decaying probability, then $a = \arg\max_{a' \in A} Q(s,a')$
    - take action and observe s', r
    - $Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a' \in A} Q(s',a'))$
    - **set** $s = s'$
  - **exit** when converged



$$Q(3,1) = 0.9 \cdot 80 + 0.1(0 + 0.9 \cdot 100) = 81$$



Q-learning limitations -> Q-table doesn't scale

Lecture 13 - application of RL to routing

RL can be applied to the networking field to address issues like:

- Energy optimization
- Congestion control
- Next hop selection

In particular, RL can automatically learn the dynamics of the network like:

- Topology changes
- Congestion points
- Energy consumption

In general, it improves the service quality offered to end-users while optimizing networks resources (and providers' revenues)
One major concern in routing is the optimization of routes while considering dynamic topology changes

(RL RECAP)
The learner, also called Agent, interacts with its environment and selects its actions to be applied to the environment according to its current state and the reinforcement it collects from the environment. The role of the reward is to provide feedback to the learning algorithm about the effect of the recently taken action.
Reward function -> what is good (or bad) in an immediate sense
Value function -> what is good (or bad) in the long term
The objective of the agent is to take actions in order to maximize the global discounted reward, denoted by Gt.
There are different models to address optimality and to define Gt:

- Finite-horizon model: the agent should optimize the reward for the next h steps (agent lifetime is known)
- Infinite-horizon model: the agent should optimize the reward for the long-term run. A discount factor is used to "weigh" future rewards.
    - γ =0 -> the agent is called `myopic' and it is only concerned by maximizing the immediate reward.
    - As γ approaches 1, the awareness to the future rewards is stronger

Environment model -> it's described by the transition matrix and the reward function
Two approaches to model the environment:

- Model-based: the agent learns the environment model (computes the transition probability matrix) and it improves its policy to reach optimality
- Model-free: the agent improves its policy without a prior knowledge of the environmental model (without requiring a transition probability matrix)

In the field of routing in networks, most of proposed solutions are model-free and apply Q-learning.

The learner tries to improve the current solution while switching between exploration and exploitation of the solution space.
The choice of exploration/exploitation technique has a great impact on the speed of convergence to optimality of the learning process.

Applications of RL to routing

In RL-based routing a node consists of an agent and optional components:

- Local reward: the cost of communication as seen by packet sender i.e. cost to send packets in terms of:
    - Delivery delay
    - Loss rate

    - Energy consumption
- Remote reward: feedback sent by the next hop (or the destination node)
- Link-state information maintenance: useful link state information (i.e. location and residual energy of neighboring nodes or the quality of links) through periodic on-demand Hello beacon packets

The state space may be:

- Set of nodes: the current node is the index of the node holding the packet It's the most popular in RL-based routing protocols.
- Set of grids: the current node is the number of the grid holding the packet (in grid-organized networks)

The action space may be:

- "Select node j as next hop and forward packet'" (most popular), in this case, the action space is the set of node Ids
- "Select a subset of neighbors s and broadcast packet", in this case, the action space is the set of partitions of Node Id set
- …

Solution space exploration techniques:

- Greedy
- epsilon-Greedy
- Probability-based

Basic RL vs RL applied to Routing

In basic RL, each agent is independent and interacts with its environment. Using its link-state information, the agent applies an action to the environment, it receives a reward, and then it changes its state.

In applications of RL to routing, almost all proposed protocols are based on collaborating agents, which involves not only reward but also exchanges of link-state information without actions undertaken from RL point of view. Indeed, in addition to reward, agents exchange link-state information with their neighbors to select actions. From an RL point of view, selecting a next hop is an RL action, while receiving periodic Hello packets is not.

3 types of agent collaboration among agents:

- No collaboration: either there is only one agent or, if there are multiple agents, they make decisions only based on their local view
- Reactive collaboration: when a node receives a data packet, either it returns its feedback (i.e. its Q-value or reward, depending on the adopted protocol) directly inside the ACK packet or it includes its link-state information in each data packet when it forwards it (thus providing feedback to previous sender)
- Proactive collaboration: nodes periodically broadcast their link-state information to the neighbors in addition to sending (directly or indirectly) their feedback upon reception of a packet. This link-state information may include Q-values, distances, locations, residual energy and so on. This enables agents to update their information

used in metrics calculation and/or update their routing tables (i.e. their Q-value table) without taking RL actions.

2 main classes of reward functions:

- Test-based reward function: reward value takes a constant value depending on a simple test. The most common test is “is the packet delivered to its destination?” (e.g. reward equals 1 when a packet is delivered to destination and 0 otherwise)
- Linear/non linear reward function: rewards are a function of a metric or a combination of metrics (e.g., a nonlinear reward can depend on the remaining number of hops and the cost of each hop).

Paper Q-routing

This algorithm aims at learning a routing policy which balances minimizing the number of "hops" a packet will take with the possibility of congestion along popular routes. It does this by experimenting with different routing policies and gathering statistics about which decisions minimize total delivery time. The learning is continual and online, it uses only local information, and is robust in the face of irregular and dynamically changing network connection patterns and load.
The question is: to which adjacent node should the current node send its packet to get it as quickly as possible to its eventual destination?
Since the policy's performance is measured by the total time taken to deliver a packet, there is no "training signal" for directly evaluating or improving the policy until a packet finally reaches its destination. However, using reinforcement learning, the policy can be updated more quickly and using only local information.

Given:

- x -> sender node
- y -> neighbor node of x
- d -> destination node
- Qx(d, y) -> the time (estimated by x) it takes to deliver a packet P bound for node d by way of x's neighbor node y

Upon sending P to y, x immediately gets back y's estimate for the time remaining in the trip, namely:

$$t = \min_{z \in \text{neighbors of } y} Q_y(d, z)$$

Given also:

- q -> units of time spent by the packet in x’s queue
- s -> units of time in transmission between x and y

Then x can revise its estimate as follows:

$$\Delta Q_x(d,y) = \eta\,(\ \overbrace{q+s+t}^{\text{new estimate}} - \overbrace{Q_x(d,y)}^{\text{old estimate}}\ )$$

where eta is a "learning rate" parameter.
The algorithm can be characterized as a version of the Bellman-Ford shortest paths algorithm that performs its path relaxation steps asynchronously and online and (2) measures path length not merely by number of hops but rather by total delivery time. The Q-function Qx(d, y) can be represented by a large table.

Paper Aloha-Q protocol

The ALOHA-Q protocol applies Q-Learning to frame based ALOHA as an intelligent slot selection strategy. A frame comprises a fixed number of slots, each node has individual Q values for every slot in the frame which are updated by transmission outcomes (success or failure). The largest value determines which slot is selected for the next transmission. Nodes only wake up when they need to transmit in the slots with the highest Q values and to receive the associated acknowledgements (ACKs). Idle listening is not used in ALOHA-Q. Time references for synchronisation are embedded in the ACK packets sent from the sink node, so that the transmitting nodes are able to maintain synchronisation with the sink node as long as they transmit data packets to the sink and receive ACK packets.

Nodes in the network all start with random access (all Q values are 0), learn through transmission, and finally reach their optimal transmission strategy in which nodes have found unique slots for contention free transmission. Q values are denoted Q(i,k), indicating the preference of node i to transmit a packet in slot k. The previous Q values and current reward contribute to the Q value update after every data packet transmission.

$$Q_{t+1}(i,k) = Q_t(i,k) + \alpha(r - Q_t(i,k))$$

where α is the learning rate and r is the current reward. One of the Q values of a node is updated after each data packet transmission.

If a transmission succeeds, a reward of +1 is returned otherwise the reward is -1. Slots with higher Q values are preferred but if multiple slots have the same higher Q value, one (or more) will be randomly selected from the set.

Paper IotJournal (Apt-Mac)

Context: using RFID backscattering to query battery-less sensor augmented RFID devices, for example a joystick.
RFID devices are passive, they can't spontaneously transmit data to the reader, instead the reader must execute the query and it has to decide which device to query at a given time slot.
Devices can be of different kinds, based on their transmitting requirements (event based, periodic, real-time), therefore a real time device must be queried much more often than a periodic device.
APT-MAC is an epsilon greedy, time slotted, zero configuration protocol as it learns the priorities of each device in the network through reinforcement learning.

APT-MAC is a bandit protocol
Agent: the reader
Actions: the sensors (action a(i) is: "query sensor i")
State: one state ("ready to query")
Reward vector: Q, where Q(i) is the expected reward of action a(i)
Reward function: the function to compute the reward of an action

ALGORITHM

At each timeslot the agent chooses the next action in a greedy manner, so the one with the highest expected reward (or at random if epsilon occurs)

The reader queries the sensor, then

- if the sensor has new fresh data then the reward is positive, it is computed (bonus-malus),
- if the sensor doesn't have new fresh data then the reward is negative and it is only the malus
- then the expected reward for that actions is updated accordingly

Algorithm 1 APT-MAC: Pseudocode for Slot Assignment

```
1:  Master M;                                   ▷ The reader
2:  Set D;                                      ▷ All devices
3:  Map R:(d∈D)→double;                         ▷ Rewards Map
4:  /* Initialization of the Rewards map */
5:  for d ∈ D do
6:      R[d] = 1.0;
7:  end for
8:  R = softmax(R);
9:  /* Each cycle corresponds to a time slot */
10: while true do
11:     Device next = chooseNext(R);
12:     Bool goodQuery = M.query(next);
13:     if goodQuery then
14:         R[next]=updateReward(next,bonus);
15:     else
16:         R[next]=updateReward(next,malus);
17:     end if
18:     R = softmax(R);
19: end while
```

At the end of each timeslot a softmax is applied to the reward vector, which ensures that the sum of all the values in it add up to 1.

To avoid any starvation problem, in which a tag is never queried because the others have always higher reward values, we also set a maximum query delay (**MaxQD**), that is the interval of time at which tags have to be queried regardless the expected reward, meaning that each tag will not wait longer than this time between two consecutive queries. For example if a sensor has a MaxQD of 600ms, then the reader must query it within 600ms since the last query.

MaxQD is individual for each sensor and is set at 2000ms for all the devices during startup, then it's updated dynamically during the process. Specifically the reader keeps track of the data loss for each sensor, which is computed as the number of data samples received divided by the total number of data samples produced by the sensor (the sensor itself has to keep track of it and communicate it to the reader).

If the data loss is too high (higher than 15%) then the reader must query the sensor more often and therefore reduces its MaxQD by 150ms. The minimum possible MaxQD is 50ms.

Also no sensor is so fast to generate new data every slot, therefore a tag cannot be queried at each slot, and thus we fixed a minimum query delay (**MinQD**) global, that is the minimum interval of time at which tags can be queried, this value is empirically fixed as 50ms, therefore the reader cannot query the same sensor again before 50ms.

Paper QGeo

QGeo is a Q-Learning based Geographic routing protocol for autonomous robotic networks. The basic idea of QGeo is that mobile nodes make geographic routing decisions distributively, utilizing a reinforcement machine learning algorithm without knowledge of the entire network. The QGeo consists of:

- Location estimation: updates current mobile node location information that has been reported by GPS or other localization methods. Local information is exchanged by periodic HELLO messages that include location, current Q-value, link condition, and location error.
- Neighbor table: manages the location of neighbors, their mobility pattern, link condition, location error and reported Q-value from received periodic HELLO messages. Neighbor location, Q-value, and link information are utilized in the Q-learning algorithm to generate a reward value.
- Q-learning: is the key component of routing decisions. In the exploration phase of learning, QGeo periodically exchanges state information. Each information is stored in the neighbor table of current state. In the exploitation phase, QGeo utilizes stored information to calculate reward function for deciding next state. For supporting rapid and reliable transmission of unmanned robotic networks, we utilize a new concept of packet travel speed in the reward function, which is based on not only distance but also link status. The packet travel speed (f pts) can be calculated as below:

$$f_{pts}(diff_{i,j}, P) = \frac{diff_{i,j}}{T_{i,j}(P)}$$