THE UNIVERSITY of TENNESSEE KNOXVILLE

Day 5: Decisions, Bandits, Search, and Reinforcement Learning

Instructor: Sergei V. Kalinin

Definitions:

- **Objective:** overall goal that we aim to achieve. Not available during or immediately after experiment.

- **Reward:** the measure of success available at the end of experiment

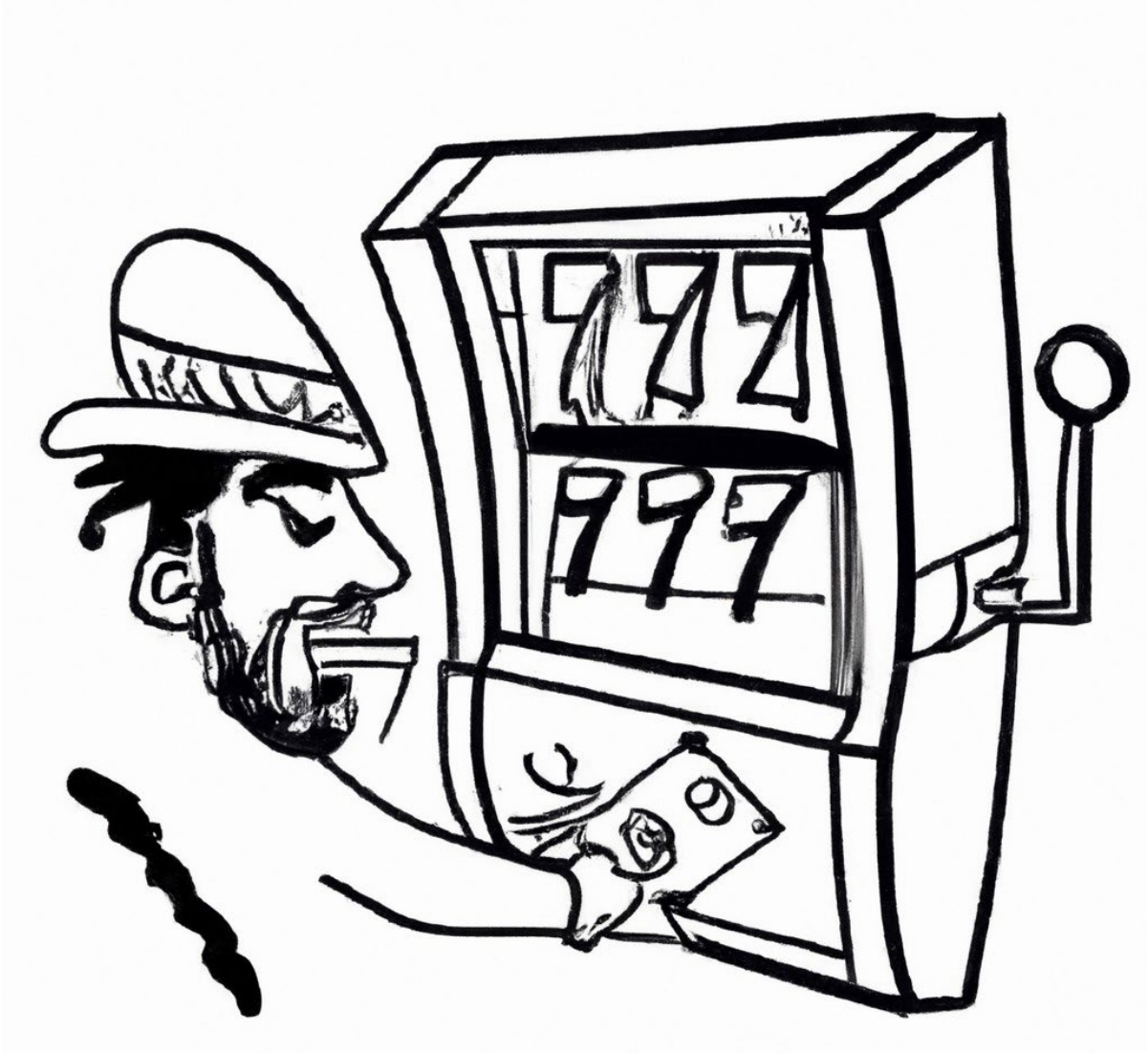
- **Value:** expected reward. Difference between reward and value is a feedback signal for multiple types of active learning

- **Action:** how can ML agent interact with the system

- **State:** information about the system available to ML agent

- **Policy:** rulebook that defines actions given the observed state

Bandit problem



- Imagine that we have a number of slot machines with different probabilities of win…
- Or different web-sites to places ads on…
- Or groups of patients for specific medical protocol….
- Or team members to synthesize certain material…
- Or reaction pathways to choose

How do we optimize this problem and maximize our reward?

Bandits

- **Objective:** get rich!
- **Reward:** pay-off from specific hand/click-rate of ad/effectiveness of drug
- **Value:** expected reward
- **Action:** playing a hand/placing ad/administering drug
- **State:** no state
- **Policy:** gameplan given the values of specific actions

A/B Testing

- The most common exploration strategy is **A/B testing**, a method to determine which one of the two alternatives (of online products, pages, ads etc.) performs better.
- The users are randomly split into two groups to try different alternatives. At the end of the testing period, the results are compared to choose the best alternative, which is then used in production for the rest of the problem horizon.
- This approach can be applied for more then two alternatives - **A/B/n testing**.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Definitions: reward and value

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

- First, we denote the reward (i.e. 1 for click, 0 for no click) received after selecting the action a for the n^{th} time by R_i.

- Q_n estimates the expected value of the reward that this action yields, R, after n-1 observations.

- Q_n is also called the action value of a. Here, Q_n estimates of the action value after selecting this action n - 1 times.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Updating value

$$Q_{n+1} = \frac{R_1 + R_2 + \cdots + R_n}{n} = Q_n + \frac{1}{n} \cdot (R_n - Q_n)$$

- Q_n is the estimate for the action value of a before we take it for the n^{th} time.
- When we observe the reward R_n, it gives us another signal for the action value.
- We adjust our current estimate, Q_n, in the direction of the **error** that we calculate based on the latest observed reward, R_n, with a **step size** $1/n$ and obtain a new estimate Q_{n+1}
- For convenience, $Q_0 = 0$ (But - Human heuristics!)

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Updating value: generalization

$$Q_{n+1}(a) = Q_n(a) + \alpha(R_n(\alpha) - Q_n(a))$$

- The rate at which we adjust our estimate will get smaller as we make more observations
- The step size must be smaller than 1 for the estimate to converge (and larger than 0 for a proper update).
- Using a fixed α will make the weights of the older observations to decrease exponentially as we take action a more and more.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Limitations of A/B testing

- **A/B/n testing is inefficient as it does not modify the experiment dynamically by learning from the observations.** It fails to benefit from the early observations in the test by writing off/promoting an alternative even if it is obviously under- or outperforming the others.

- **It is unable to correct a decision once it's made.** There is no way to correct the decision for the rest of the deployment horizon.

- **It is unable to adapt to the changes in a dynamic environment.** If the underlying reward distributions change over time, plain A/B/n testing has no way of detecting such changes after the selection is fixed.

- **The length of the test period is a hyperparameter to tune, affecting the efficiency of the test.** If this period is chosen to be shorter than needed, an incorrect alternative could be declared the best because of the noise in the observations. If the test period is chosen to be too long, too much money gets wasted in exploration.

- **A/B/n testing is simple.** Despite all these shortcomings, it is intuitive and easy to implement, therefore widely used in practice

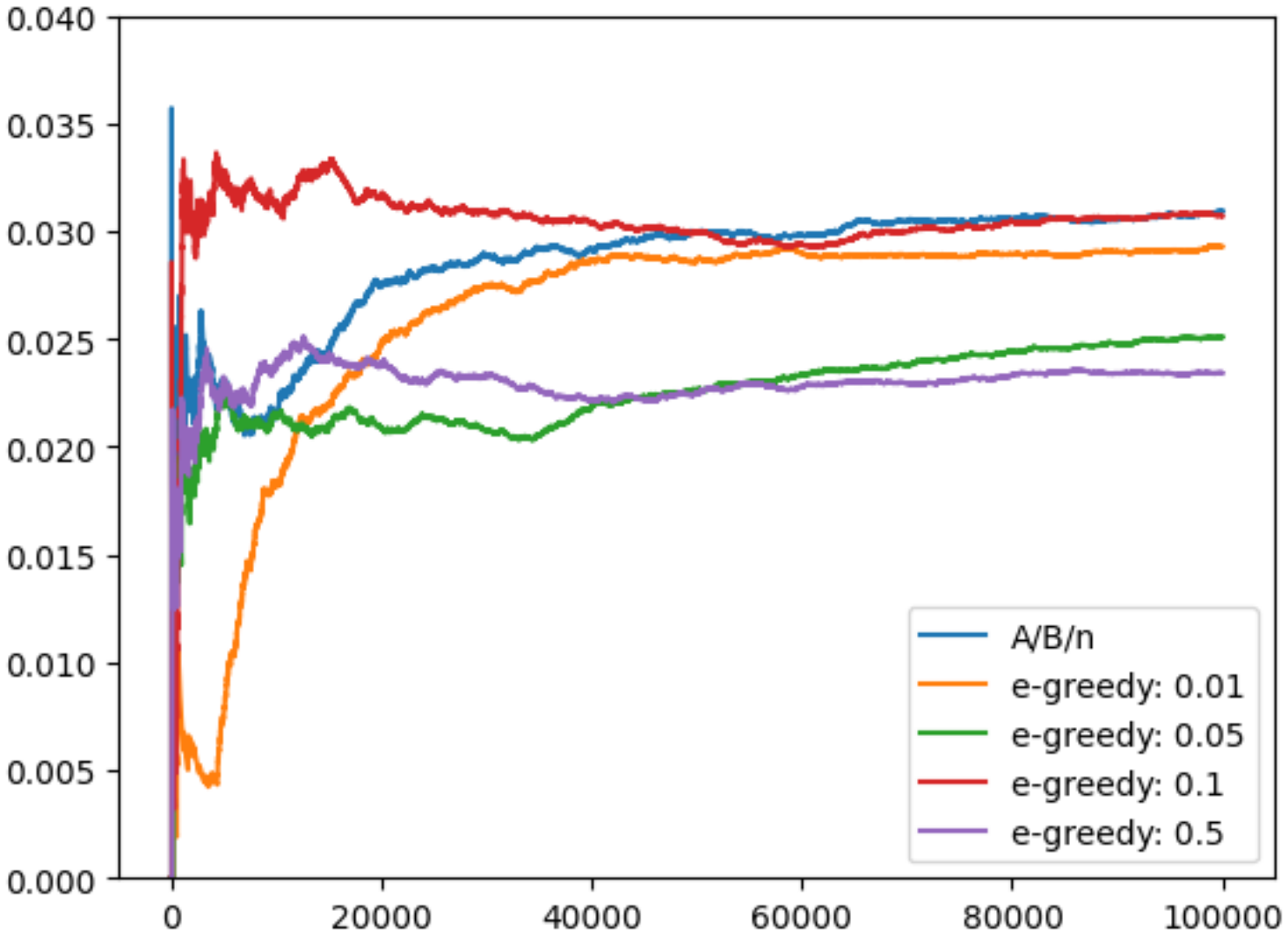
From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

ε-greedy policies

- Most of the time, greedily taking the action that is the best according to the rewards observed that far in the experiment (i.e. with 1-ε probability)
- Once in a while (i.e. with ε probability) take a random action regardless of the action performances.
- Here ε is a number between 0 and 1, usually closer to zero (e.g. 0.1) to "exploit" in most decisions.
- Obviously, the number of alternatives has to be fairly small
- Parameter ε can change during the experiment

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

ε-greedy policies



A/B vs. ε-greedy policies

- ε-greedy actions and A/B/n tests are similarly inefficient and static in allocating the exploration budget. The ε-greedy approach, too, fails to write off actions that are clearly bad and continues to allocate the same exploration budget to each alternative. Similarly, if a particular action is under-explored/over-explored at any point, the exploration budget is not adjusted accordingly.

- With ε-greedy actions, exploration is continuous, unlike in A/B/n testing. This means if the environment is not stationary, the ε-greedy approach has the potential to pick up the changes and modify its selection of the best alternative.

- The ε-greedy actions approach could be made more efficient by dynamically changing the ε. For example, one could start with a high ε to explore more at the beginning and gradually decrease it to exploit more later. This way, there is still continuous exploration, but not as much as at the beginning when there was no knowledge of the environment.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

A/B vs. ε-greedy policies

The ε-greedy actions approach could be made more dynamic by increasing the importance of the more recent observations:

$$Q_{n+1}(a) = Q_n(a) + \alpha(R_n(a) - Q_n(a))$$

- **Modifying the ε-greedy actions approach introduces new hyperparameters, which need to be tuned.**
- Both gradually diminishing ε and using exponential smoothing for *Q*, come with additional hyperparameters, and it may not be obvious what values to set these to.
- Incorrect selection of these hyperparameters may lead to worse results than what the standard version would yield.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Upper Confidence Bound (UCB)

$$A_t \triangleq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

- At each step, we select the action that has the highest potential for reward.
- The potential of the action is calculated as the sum of the action value estimate and a measure of the uncertainty of this estimate. This sum is what we call the upper confidence bound.
- **Overall:** the action is selected either because our estimate for the action value is high, or the action has not been explored enough (i.e. as many times as the other ones) and there is high uncertainty about its value, or both.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Digging deeper: UCB

$$A_t \triangleq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

- $Q_t(a)$ and $N_t(a)$ have the same meanings as before. This formula looks at the variable values, which may have been updated a while ago, at the time of decision making a, whereas the earlier formula described how to update them.
- In this equation, the square root term is a measure of the uncertainty for the estimate of the action value of a.
- The more we select a, the less we are uncertain about its value, and so is the $N_t(a)$ term in the denominator.
- As the time passes, however, the uncertainty grows due to the $\ln t$ term (which makes sense especially if the environment is not stationary), and more exploration is encouraged.
- The emphasis on uncertainty during decision making is controlled by a hyperparameter, c. This obviously requires tuning and a bad selection could diminish the value in the method.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Advantages and disadvantages of UCB

• **UCB is a set-and-forget approach.** It systematically and dynamically allocates the budget to alternatives that need exploration. If there are changes in the environment, for example, if the reward structure changes because one of the ads gets more popular for some reason, the method will adapt its selection of the actions accordingly.

• **UCB can be further optimized for dynamic environments, potentially at the expense of introducing additional hyperparameters.** The formula we provided for UCB is a common one, but it can be improved, for example, by using exponential smoothing. There are also more effective estimations of the uncertainty component in literature. These modifications, though, could potentially make the method more complicated.

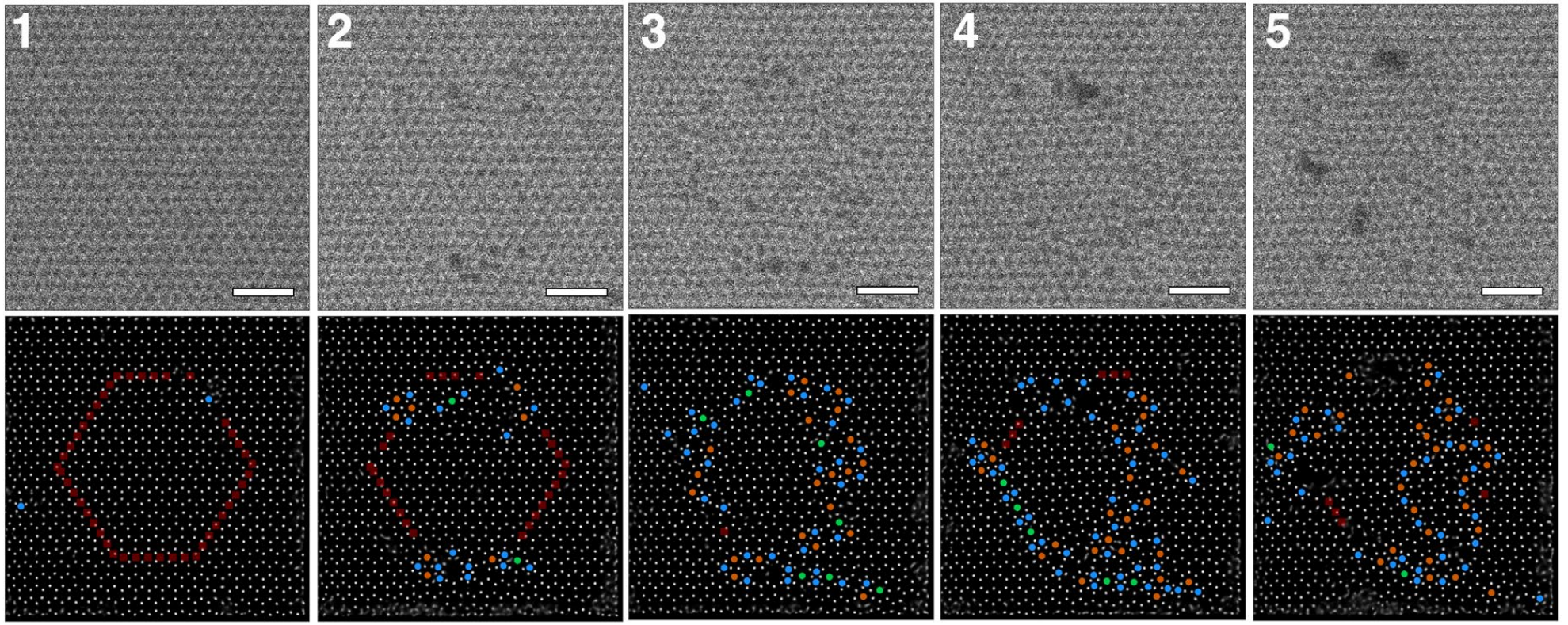
• **UCB could be hard to tune.** It is somewhat easier make the call and say "I want to explore 10% of the time, and exploit for the rest" for the ε-greedy approach than saying "I want my uncertainty to be 0.729" for the UCB approach, especially if you are trying these methods on a brand-new problem. When not tuned, an UCB implementation could give unexpectedly bad results.

From Enes Bilgin, Mastering Reinforcement Learning with Python, Packt 2020

Fixed policy experiments

Defect patterning in graphene

- Locate atomic coordinates (and defects)
- Direct beam in predetermined path to generate defects (vacancies) – unclear if they form / remain in place
- Repeat scan path, but avoid formed defects
- **Hexagon pattern**



What were we doing?

- **Objective(s):**
  - Understanding electronic and vibrational properties of defects
  - Building structures on the atomic level for biological sequencing and quantum sensing
  - … and so on. We will find out later!

- **Reward:**
  - The number of discovered defects (not used as feedback)
  - Atom moved in desired location

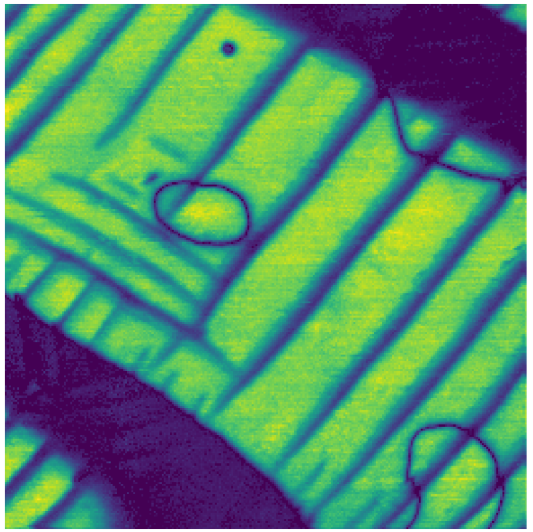
- **Value:** expected reward

- **Action:** position electron beam at given location, take EELS spectrum

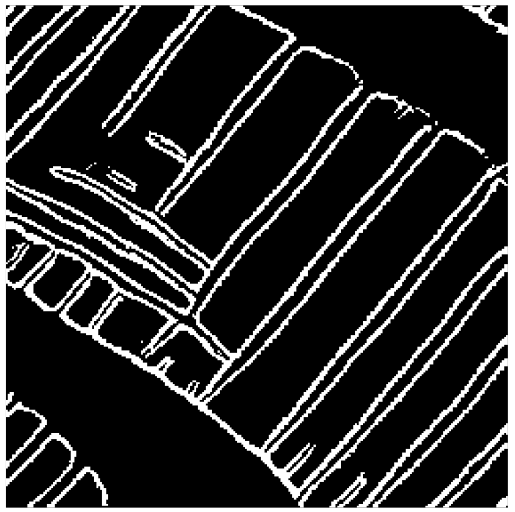
- **State:** image

- **Policy:** fixed action table (if detect defect, take EELS)

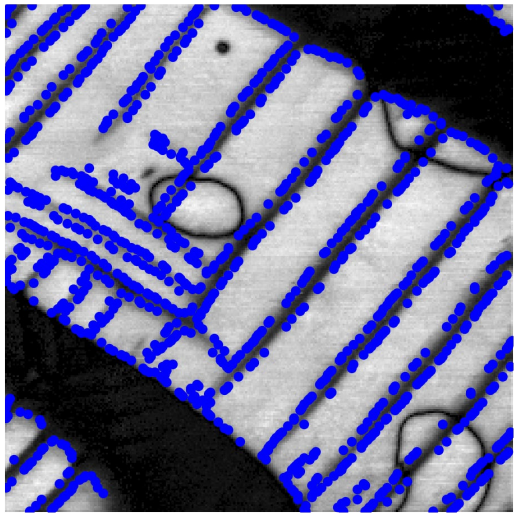
Mapping Activity of Domain Walls



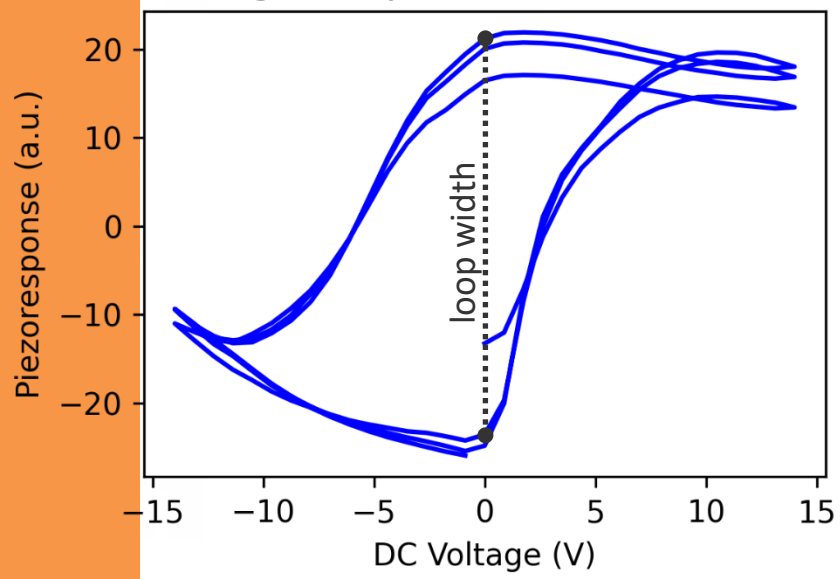
ResHedNet Prediction



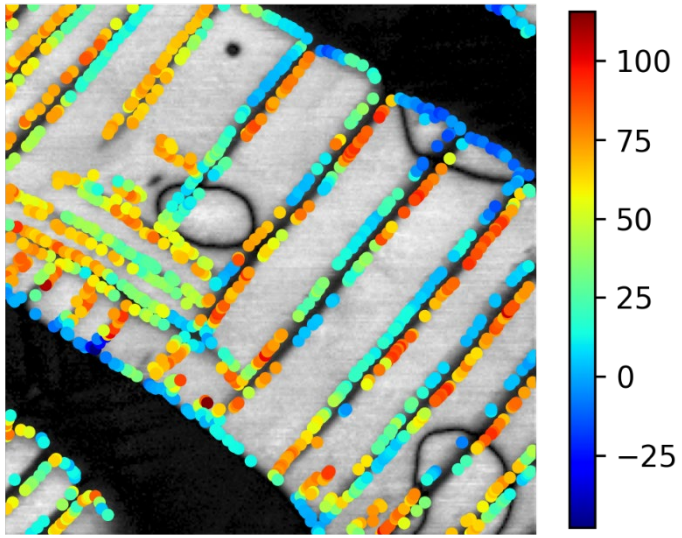
BEPS Measurement Points

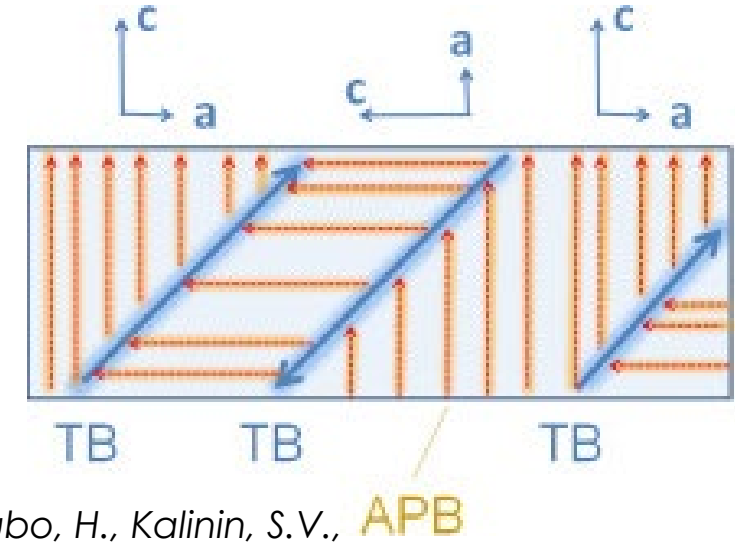


Averaged loop over ferroelastic walls



Loop height at ferroelastic walls





Liu, Y., Kelley, K.P., Funakubo, H., Kalinin, S.V., Ziatdinov, M., Arxiv, submitted

What were we doing?

- **Objective(s):**
  - Understanding the role of domain walls on polarization switching
  - Discover what makes these materials good piezoelectrics
  - … and so on. For fundamental research, vey often impact is clear later!

- **Reward:**
  - The number of explored domain walls (not used as feedback)

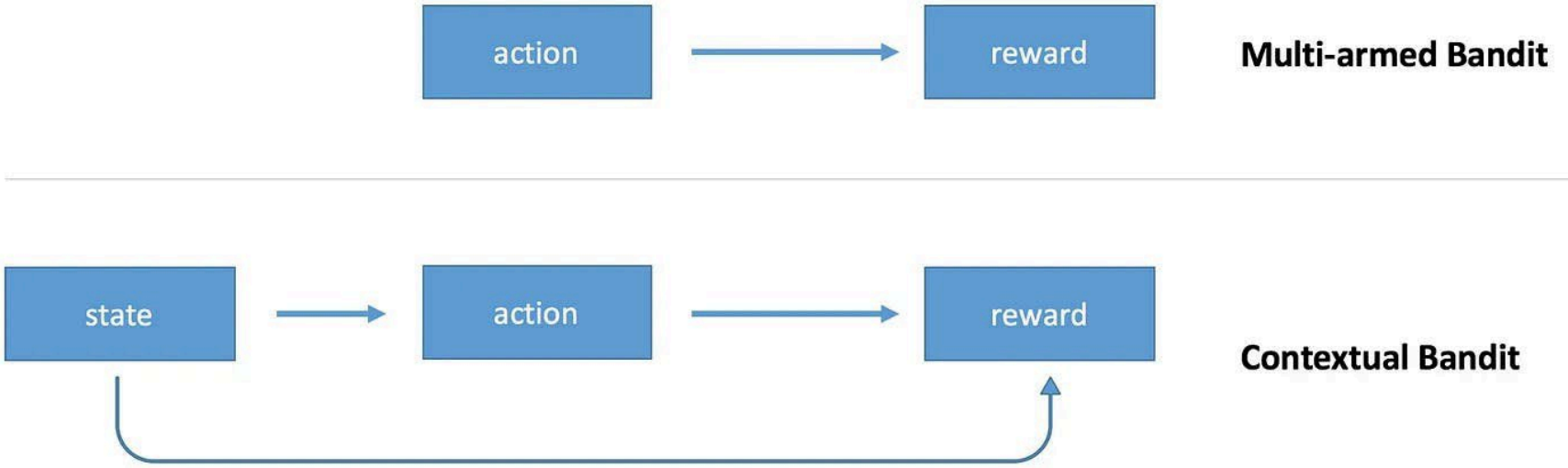
- **Value:** expected reward

- **Action:** position SPM probe at a given location, take PFM spectrum

- **State:** image

- **Policy:** fixed action table (if detect wall, take spectrum)

Contextual Bandits

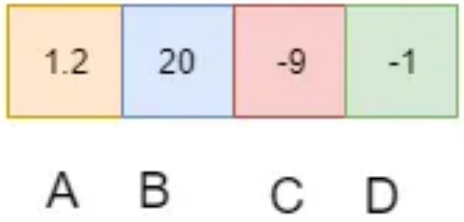


- In contextual bandits the decision-making process is informed by the context.
- The context, in this case, refers to a set of observable variables that can impact the result of the action.
- This addition makes the bandit problem closer to real-world applications, such as personalized recommendations, clinical trials, or ad placement, where the decision depends on specific circumstances.

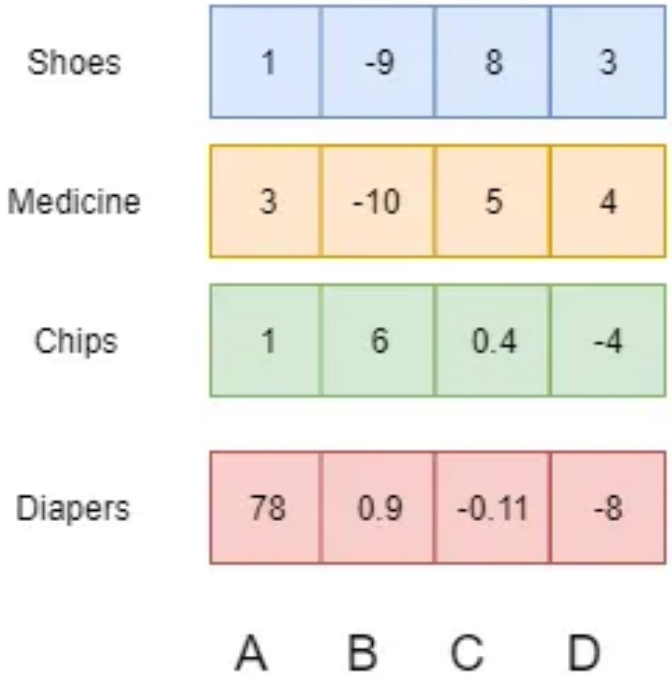
https://hackernoon.com/contextual-multi-armed-bandit-problems-in-reinforcement-learning

Simple example

Bandit



Contextual Bandit



https://medium.com/data-science-in-your-pocket/contextual-bandits-in-reinforcement-learning-explained-with-example-and-codes-3c707142437b

Contextual Bandits

Algorithm 1 Contextual Bandit

Require: Number of actions A
Require: Context feature dimensions d
Require: Context generator
Require: Reward function $\rho(a_t, x_t)$
Require: Learning rate α
Require: Bandit model (e.g., LinUCB, Neural Bandit, Decision Tree Bandit)
  Initialize the bandit model
  for $t = 1$ to T **do**
    Receive the context x_t from the context generator
    Predict the expected reward for each action $\hat{r}(a_t|x_t)$ using the bandit model
    Select the action $a_t = \arg\max_a \hat{r}(a_t|x_t)$
    Receive the reward $r_t = \rho(a_t, x_t)$
    Update the bandit model using the pair (a_t, r_t)
  end for
Ensure: The trained bandit model, The cumulative reward

https://hackernoon.com/contextual-multi-armed-bandit-problems-in-reinforcement-learning

Contextual Bandits

At each time step t, the environment presents a context x_t to the algorithm (often called the agent). Based on this context, the agent chooses an action a_t from a set of possible actions A.

In response to the action, the agent receives a reward r_t. The goal of the agent is to learn a policy π, a mapping from contexts to actions, that maximizes the cumulative reward over time:

$$\max_{\pi} \mathbb{E}\left[\sum_{t=1}^{T} r_t | \pi, D\right] (1)$$

where $D = (x_1, a_1, r_1), ..., (x_T, a_T, r_T)$ is the data (context, action, reward) collected until time T. The expectation is taken over the randomness in the context and rewards.

- The goal of the algorithm is to maximize the cumulative reward over some time horizon.
- In each round, the reward for only the chosen action is observed.
- This is known as partial feedback, which differentiates bandit problems from supervised learning

https://hackernoon.com/contextual-multi-armed-bandit-problems-in-reinforcement-learning

Linear UCB

$$\mathbb{E}[r|x,a] = x^T\theta_a(1)$$

$\mathbb{E}[r|x,a] = x^T\theta_a$ is the expected reward of action a given context x under the linear model
θ_a is the parameter vector for action a.
The objective is to learn θ_a for all actions based on the data.

$\Rightarrow$ At each time step t, given context x_t, LinUCB selects the action a_t that has the maximum UCB:

$$a_t = \arg\max_{a \in A} \left(x_t^T\theta_a + \alpha\sqrt{x_t^T A_a^{-1} x_t} \right)$$

A_a is a matrix that keeps track of the features seen so far for action a,
α is a parameter controlling the trade-off between exploration and exploitation,
the second term is the UCB, which is proportional to the standard deviation of the estimated reward.

Linear UCB Update

Chosen action:

$$a_t = \arg\max_{a \in A} \left(x_t^T \theta_a + \alpha \sqrt{x_t^T A_a^{-1} x_t} \right)$$

Model updates:

$$A_a = A_a + x_t x_t^T \qquad b_a = b_a + r_t x_t$$

$$\theta_a = A_a^{-1} b_a$$

- LinUCB algorithm is a contextual bandit algorithm that models the expected reward of an action given a context as a linear function
- LinUCB it selects actions based on the Upper Confidence Bound (UCB) principle to balance exploration and exploitation.
- It exploits the best option available according to the linear, but it also explores options that could potentially provide higher rewards, considering the uncertainty in the model's estimates.

Decision Tree cMAB

Formally, a decision tree T defines a partition of the context space X into K disjoint subsets $X_1, X_2, ..., X_K$, each associated with a recommended action a_k.
The reward of action a_k in context x under the decision tree T is:

$$\mathbb{E}[r|x, a_k, T] = \mathbb{E}[r|x \in X_k]$$

$\Rightarrow$ Given a context x, the decision tree selects the action associated with the subset that x falls into.

The objective is to learn the best decision tree that maximizes the expected cumulative reward:

$$\max_T \mathbb{E}\left[\sum_{t=1}^{T} r_t | T, D\right]$$

Where
$D = (x_1, a_1, r_1), ..., (x_T, a_T, r_T)$ is the data (context, action, reward) collected until time T.

Decision Tree Bandit models the reward function as a decision tree, where each leaf node corresponds to an action and each path from the root to a leaf node represents a decision rule based on the context. It performs exploration and exploitation through a statistical framework, making splits and merges in the decision tree based on statistical significance tests.

Neural Network cMAB

- Deep learning models can be used to approximate the reward function in high-dimensional or non-linear cases. The policy is typically a neural network that takes the context and available actions as input and outputs the probabilities of taking each action.

- A popular deep learning approach is to use an actor-critic architecture, where one network (the actor) decides which action to take, and the other network (the critic) evaluates the action taken by the actor.

- For more complex scenarios where the relationship between the context and the reward is not linear, we can use a neural network to model the reward function. One popular method is to use a policy gradient method, such as REINFORCE or actor-critic.

Neural Network cMAB

Let $\pi_\theta(a|x)$ be the policy parameterized by θ, i.e., the probability of selecting action a given context x. The objective is to find θ that maximizes the expected cumulative reward:

$$\max_\theta \mathbb{E}\left[\sum_{t=1}^{T} r_t|\pi_\theta, D\right]$$

$\Rightarrow$ Using the policy gradient theorem, we can update θ iteratively:

$$\theta_{t+1} = \theta_t + \alpha(r_t - b(x_t))\nabla \log \pi_\theta(a_t|x_t)$$

where α is the learning rate, $b(x_t)$ is a baseline function (often the average reward), and the expectation is approximated by sampling actions according to π_θ.

- Neural Bandit uses neural networks to model the reward function, taking into account the uncertainty in the parameters of the neural network.
- It further introduces exploration through policy gradients where it updates the policy in the direction of higher rewards.
- This form of exploration is more directed, which can be beneficial in large action spaces.

Multistage decisions: agent

Autonomous car (or robot, or even Roomba):

- **Percepts:** inputs from sensors, cameras, GPS system, etc
- **Actions:** steering, acceleration, controls
- **Goals:** reach destination, safety, comfort, maximize profit, etc…
- **Environment:** street, highway, mountain trail, ….
- Goals specifiable by performance measure defining a numerical value for any environment history (reward)
- **Rational action:** whichever action maximizes the expecte dvalue of the performance measure given the percept sequence to date
- **Rational** does not mean omniscient, clairvoyant, or successful

Multistage decisions

	Solitaire	Backgammon	Internet shopping	Taxi
Accessible??	Yes	Yes	No	No
Deterministic??	Yes	No	Partly	No
Episodic??	No	No	No	No
Static??	Yes	Semi	Semi	No
Discrete??	Yes	Yes	Yes	No

- The environment type determines the agent design
- Real world is inaccessible, stochastic, sequential, dynamic, continuous

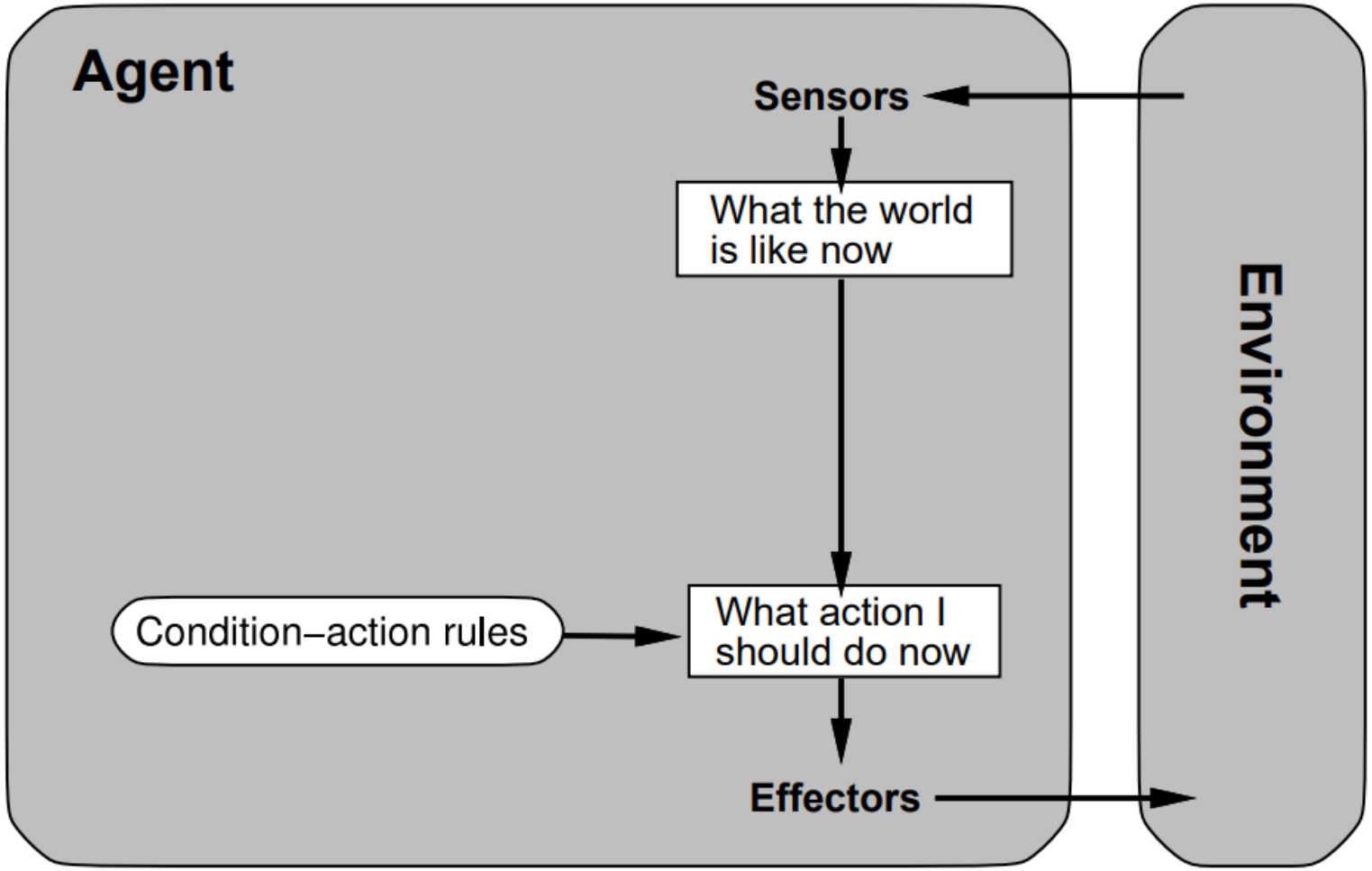
Agent functions

- Agent function is completely specified by policy (agent function) mapping percept sequence to actions
- Most primitive policy will be look-up table (i.e. simple enumeration)

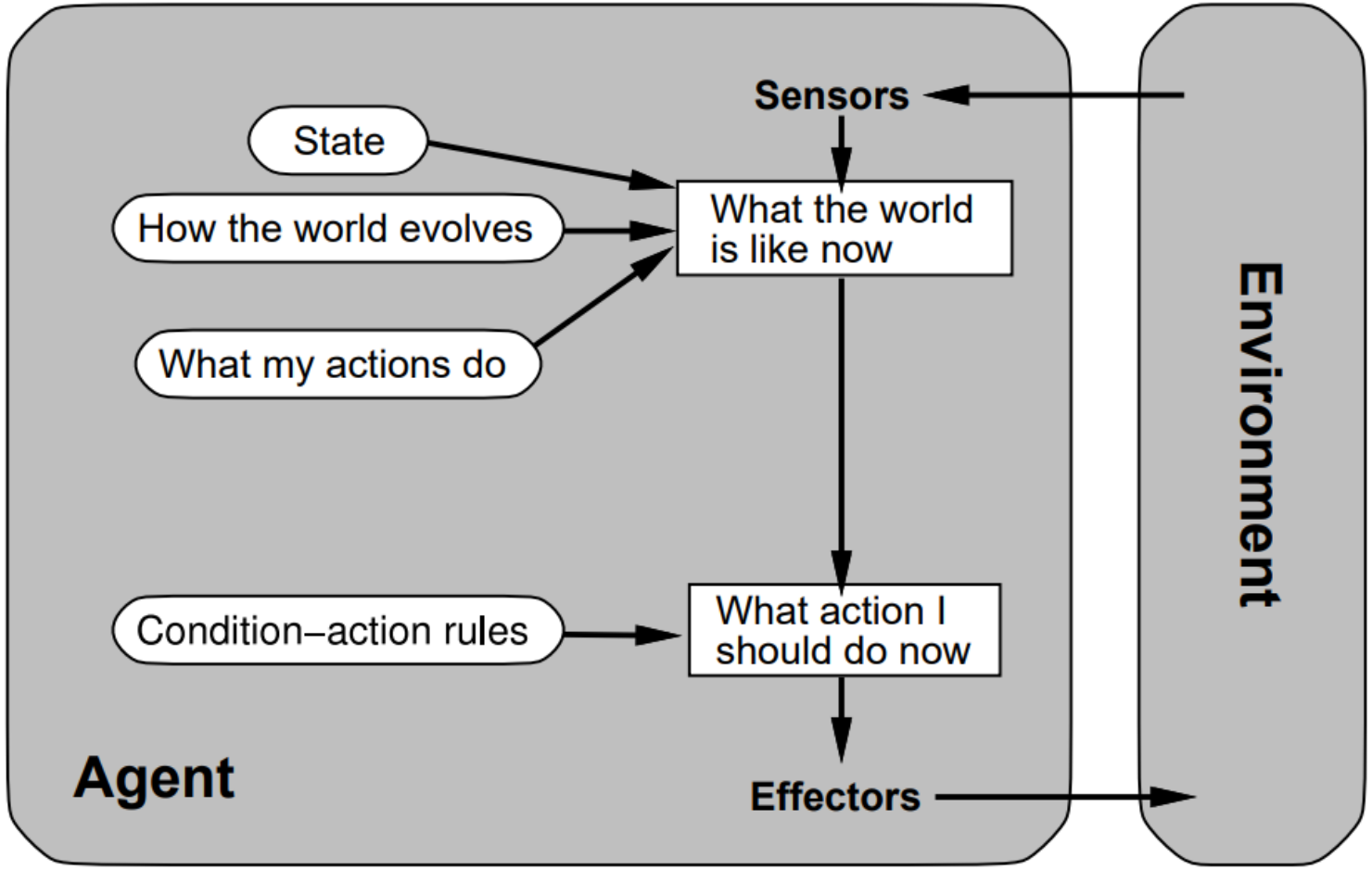
Agent types:

- Simple reflex agents
- Reflex agents with state
- Goal-based agents
- Utility-based agents

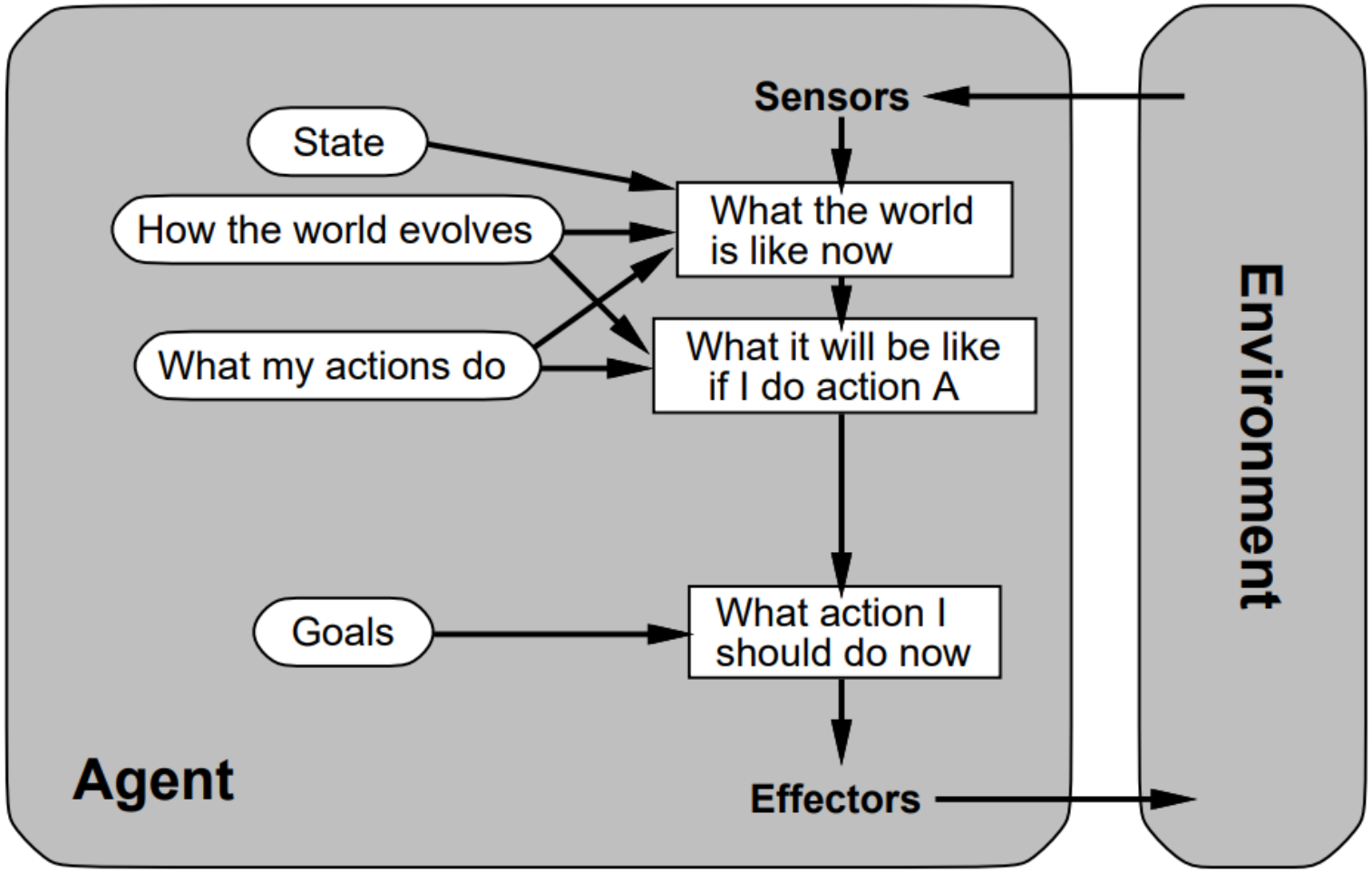
Simple reflex agents



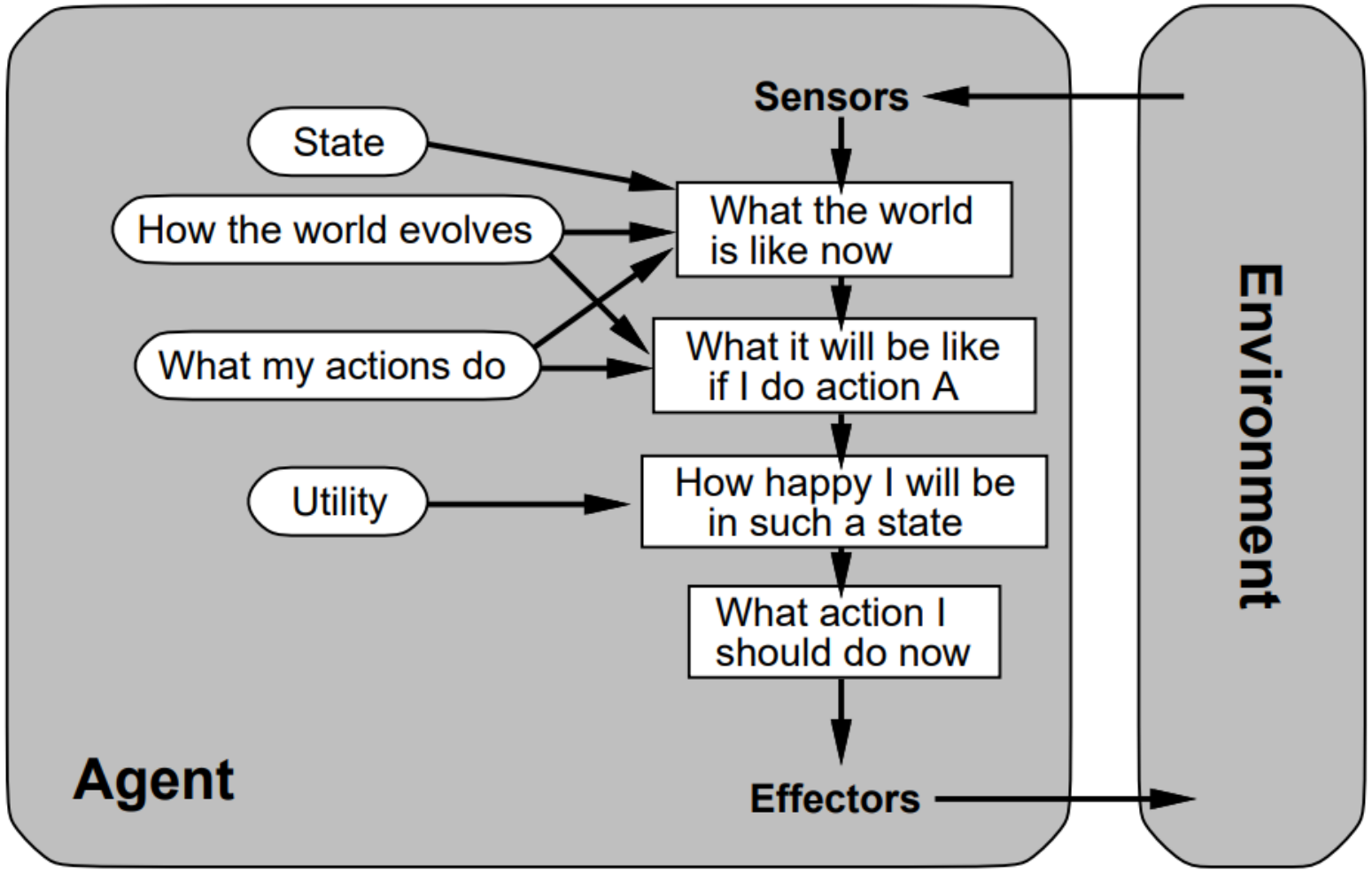
Reflex agents with state



Goal based agents



Utility based agents



From agents to search

- **Given:**
  - An initial state of the world
  - A set of possible actions or operators that can be performed.
  - A goal test that can be applied to a single state of the world to determine if it is a goal state.

- **Find:** A solution stated as a path of states and operators that shows how to transform the initial state into one that satisfies the goal test.

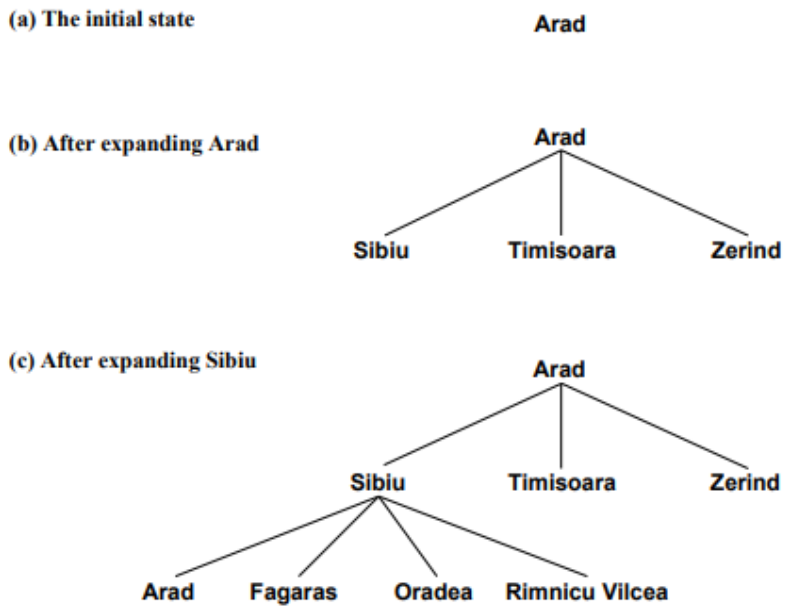
- The initial state and set of operators implicitly define a state space of states of the world and operator transitions between them. May be infinite.

From agents to search

- **Path cost:** a function that assigns a cost to a path, typically by summing the cost of the individual operators in the path. May want to find minimum cost solution.
- **Search cost:** The computational time and space (memory) required to find the solution.
- Generally there is a trade-off between path cost and search cost and one must satisfice and find the best solution in the time that is available

Search concepts

- A state can be expanded by generating all states that can be reached by applying a legal operator to the state
- State space can also be defined by a successor function that returns all states produced by applying a single legal operator
- A search tree is generated by generating search nodes by successively expanding states starting from the initial state as the root
- A search node in the tree can contain
  - Corresponding state
  - Parent node
  - Operator applied to reach this node
  - Length of path from root to node (depth)
  - Path cost of path from initial state to node



Search strategies

Properties of search strategies

- Completeness: does it always find solution if one exists?
- Time Complexity: number of nodes generated/expanded
- Space Complexity: maximum number of nodes in memory
- Optimality: does it always find least cost solution?

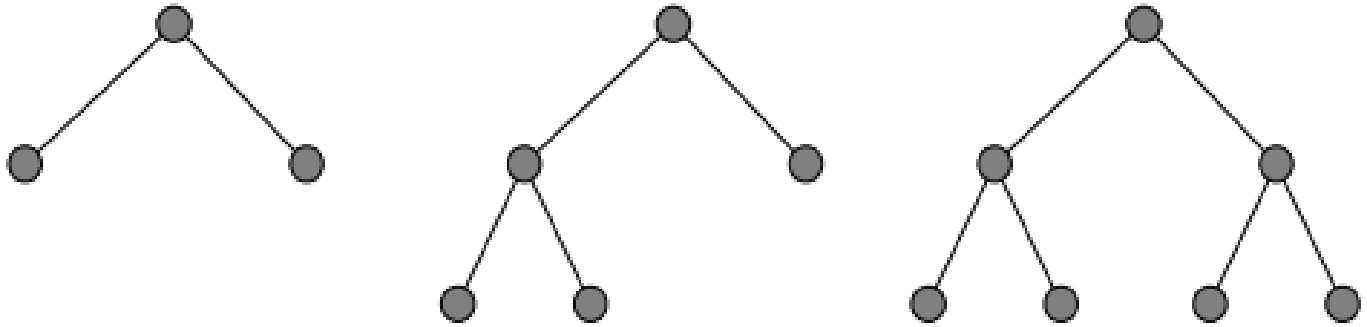
Informed vs. uninformed:

• Uninformed search strategies (blind, exhaustive, brute force) do not guide the search with any additional information about the problem.

• Informed search strategies (heuristic, intelligent) use information about the problem (estimated distance from a state to the goal) to guide the search

Breadth-first search

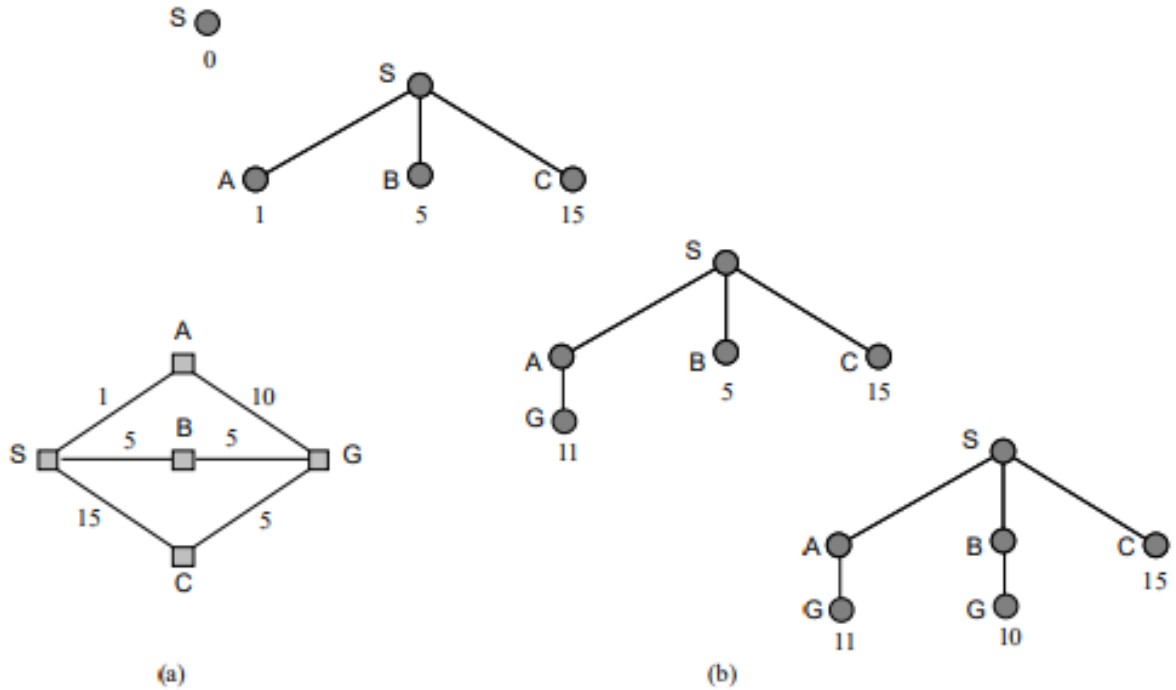
- Expands search nodes level by level, all nodes at level *d* are expanded before expanding nodes at level *d*+1



- Implemented by adding new nodes to the end of the queue
- Since eventually visits every node to a given depth, guaranteed to be complete
- Optimal provided path cost is a nondecreasing function of the depth of the node (e.g. all operators of equal cost) since nodes explored in depth order

Uniform cost search

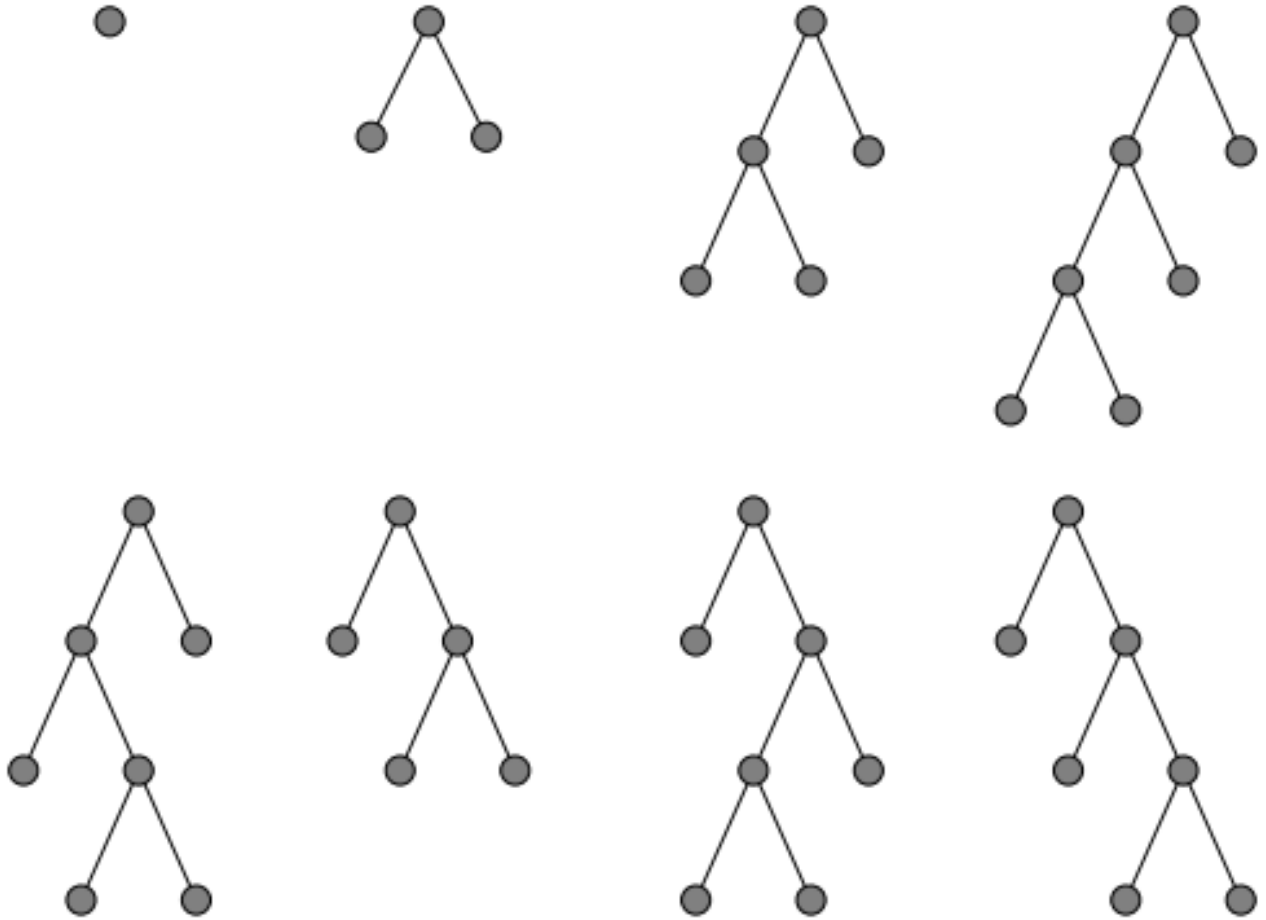
- Like breadth-first except always expand node of least cost instead of least depth (i.e. sort new queue by path cost)



- Do not recognize goal until it is the least cost node on the queue and removed for goal testing
- Therefore, guarantees optimality as long as path cost never decreases as a path increases (non-negative operator costs)

Depth-first search

- Always expand node at deepest level of the tree, i.e. one of the most recently generated nodes. When hit a dead-end, backtrack to last choice



Heuristic Search

- Heuristic or informed search exploits additional knowledge about the problem that helps direct search to more promising paths.

- A **heuristic function**, $h(n)$, provides an estimate of the cost of the path from a given node to the closest goal state

- Must be zero if node represents a goal state.
  - Example: Straight-line distance from current location to the goal location in a road navigation problem

- Many search problems are NP-complete so in the worst case still have exponential time complexity; however a good heuristic can:
  - Find a solution for an average problem efficiently.
  - Find a reasonably good but not optimal solution efficiently.

Minimizing total cost: A* Search

- A* combines features of uniform cost search (complete, optimal, inefficient) with best-first (incomplete, non-optimal, efficient).
- Sort queue by estimated total cost of the completion of a path:

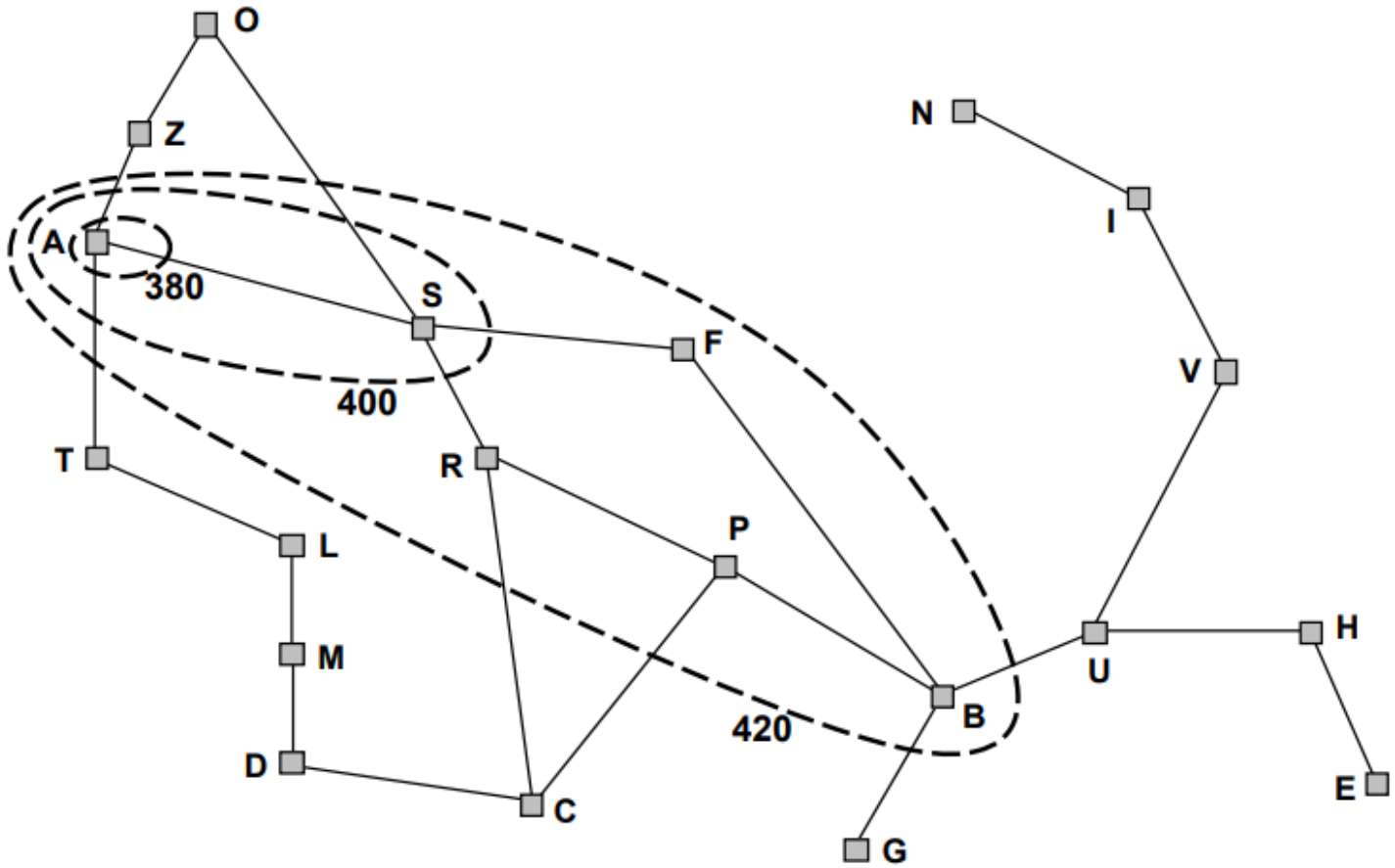
$$f(n) = g(n) + h(n)$$

Cost to this moment Estimated cost to the goal

- If the heuristic function always underestimates the distance to the goal, it is said to be admissible.
- If h is admissible, then $f(n)$ never overestimates the actual cost of the best solution through n

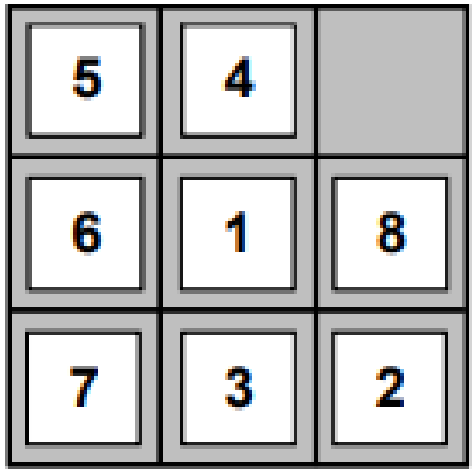
Lemma: A* expands nodes in order of increasing f value

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$

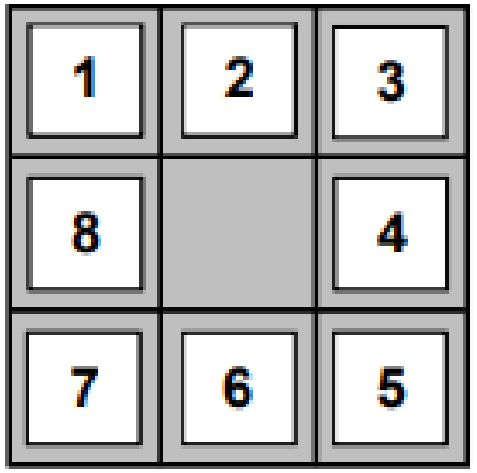


But what about heuristic functions?

- 8-puzzle search space
  - Typical solution length: 20 steps
  - Average branching factor: 3
  - Exhaustive search: $3^{20}=3.5 \times 10^9$
  - Bound on unique states:
    9! = 362,880



Start State



Goal State

- Admissible Heuristics:
  - Number of tiles out of place (h1): 7
  - City-block (Manhattan) distance (h2): 2+3+3+2+4+2+0+2=18

Inventing Heuristics

- Many good heuristics can be invented by considering relaxed versions of the problem (abstractions)

- **For 8-puzzle:** A tile can move from square A to B if A is adjacent to B and B is blank
  - (a) A tile can move from square A to B if A is adjacent to B
  - (b) A tile can move from square A to B if B is blank
  - (c) A tile can move from square A to B

- If there are a number of features that indicate a promising or unpromising state, a weighted sum of these features can be useful. Learning methods can be used to set weights.

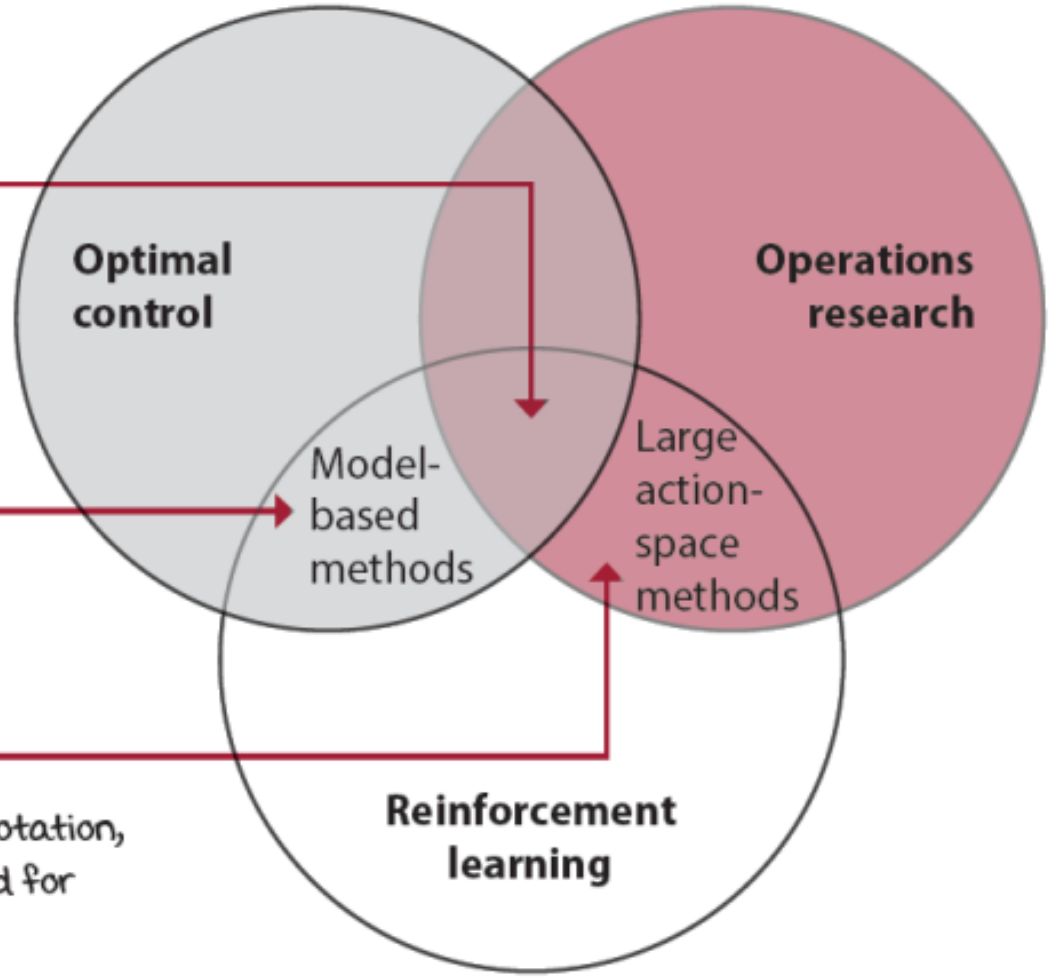
Where is reinforcement learning in ML world?

(1) All of these fields (and many more) study complex sequential decision-making under uncertainty.

(2) As a result, there's a synergy between these fields. For instance, reinforcement learning and optimal control both contribute to the research of model-based methods.

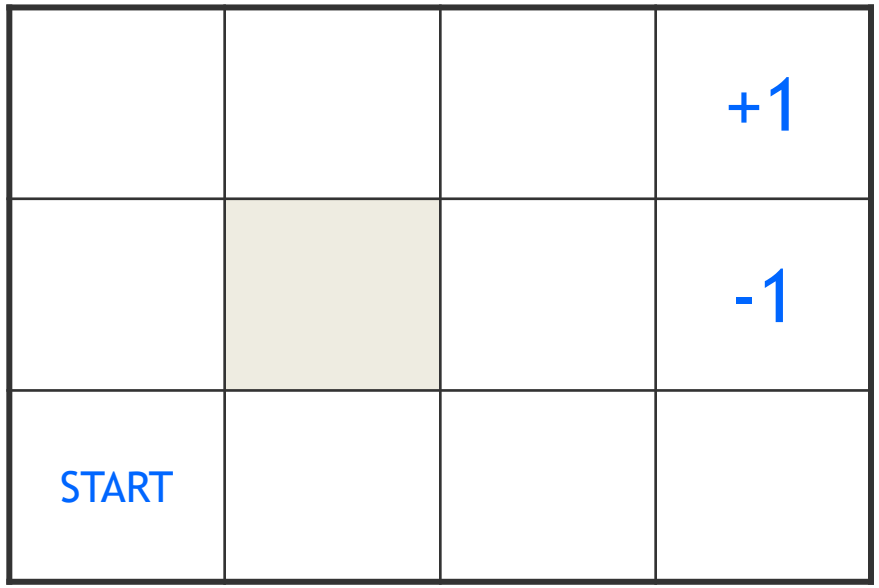
(3) Or reinforcement learning and operations research both contribute to the study of problems with large action spaces.

(4) The downside is an inconsistency in notation, definitions, and so on, that makes it hard for newcomers to find their way around.



https://livebook.manning.com/book/grokking-deep-reinforcement-learning/chapter-1/31

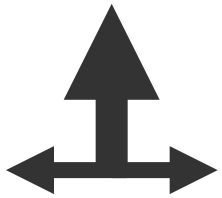
Robot in the room



actions: UP, DOWN, LEFT, RIGHT

UP

80% move UP
10% move LEFT
10% move RIGHT

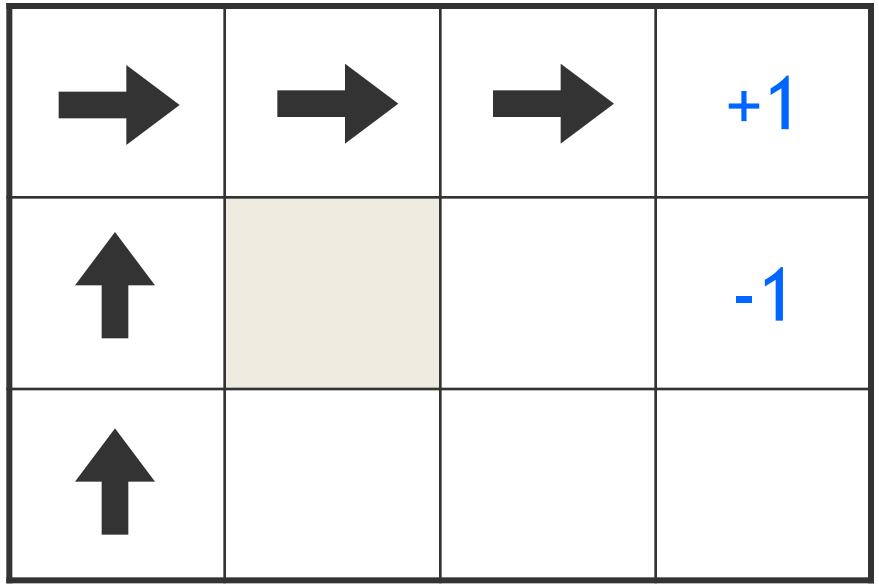


- Reward +1 at [4,3], -1 at [4,2]
- Reward -0.04 for each step

- What's the strategy to achieve max reward?
- What if the actions were deterministic?

From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Is this a solution?

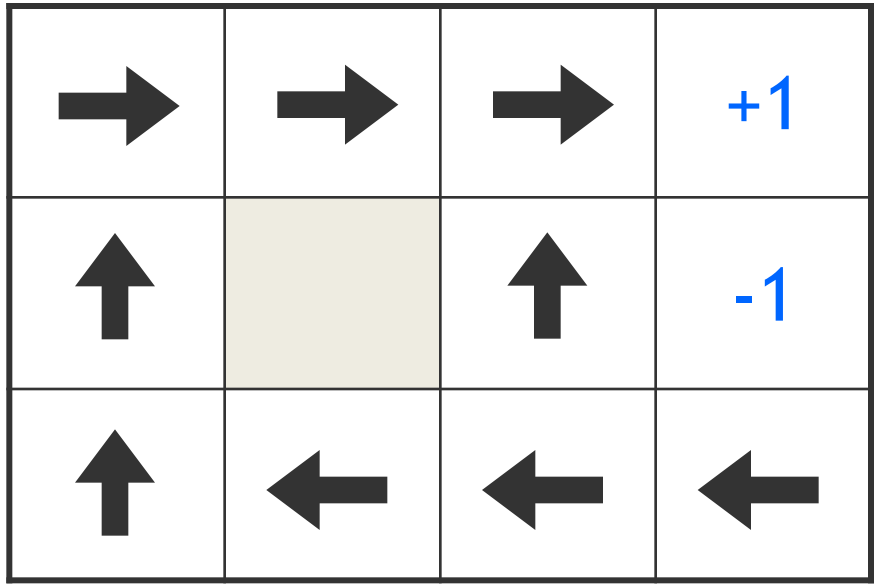


- only if actions deterministic
- not in this case (actions are stochastic)

- solution/policy
- mapping from each state to an action

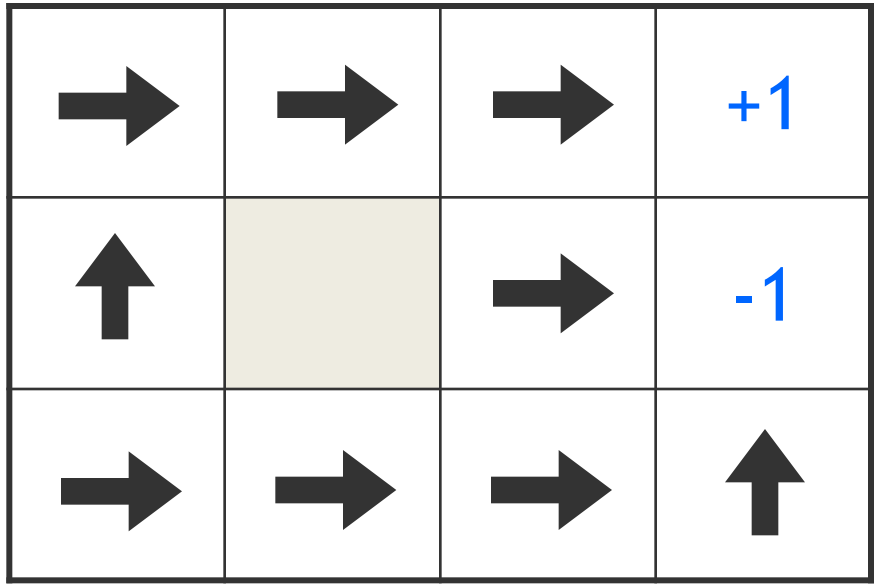
From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Optimal policy



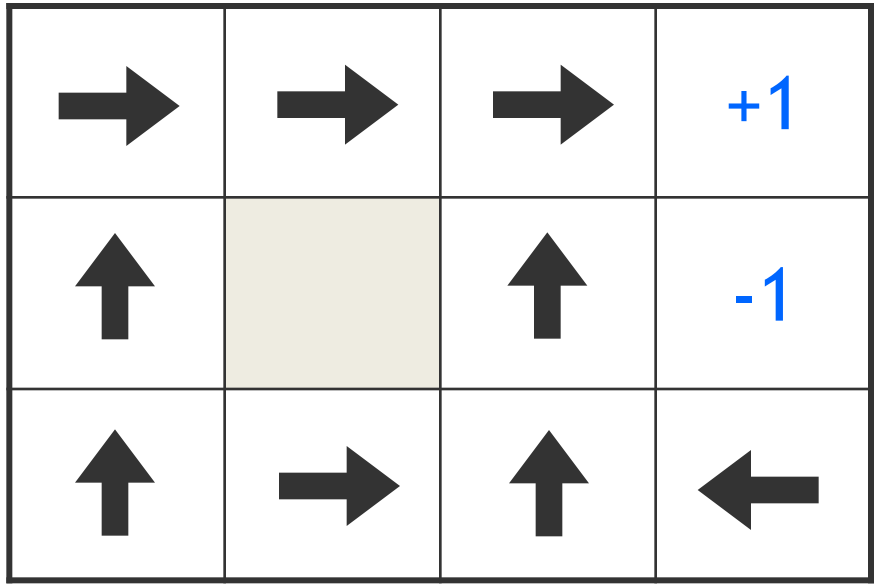
From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

What if the reward for each step is -2?



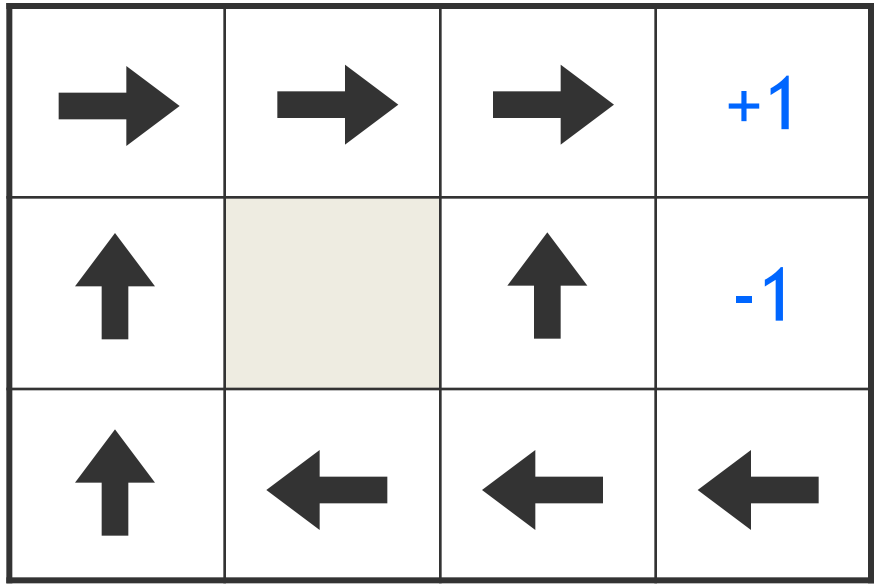
From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Reward for each step is -0.1



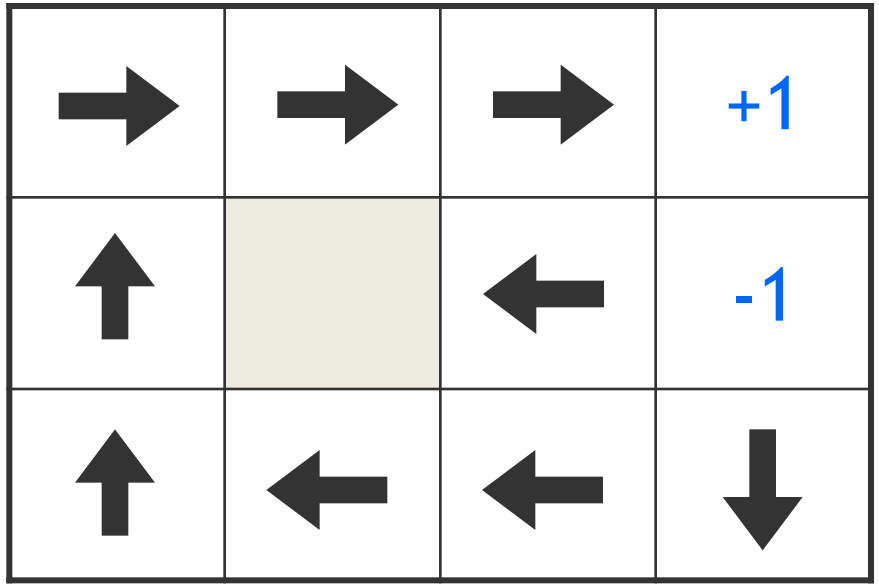
From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Reward for each step is -0.04



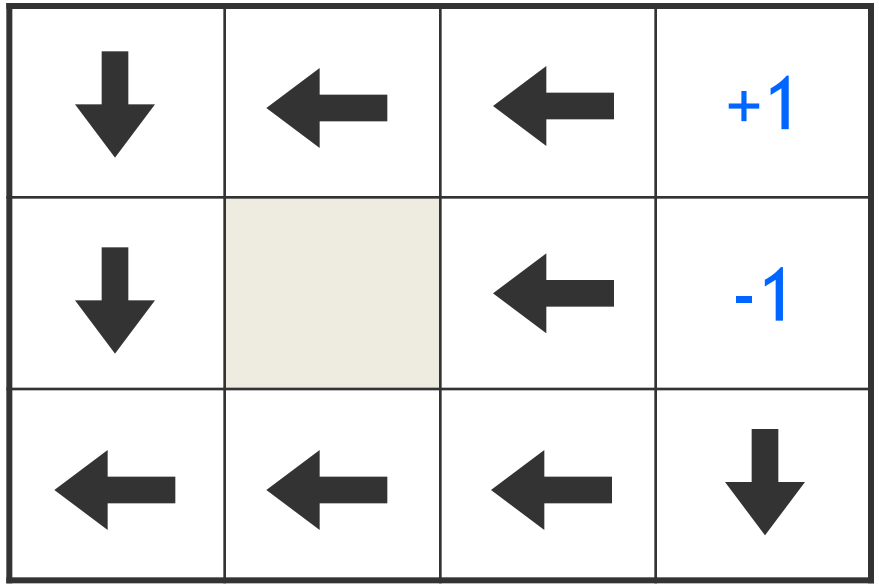
From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Reward for each step is -0.01



From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

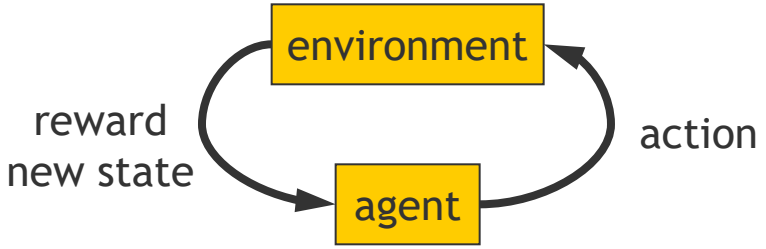
Reward for each step is +0.01



From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Markov Decision Processes (MDP)

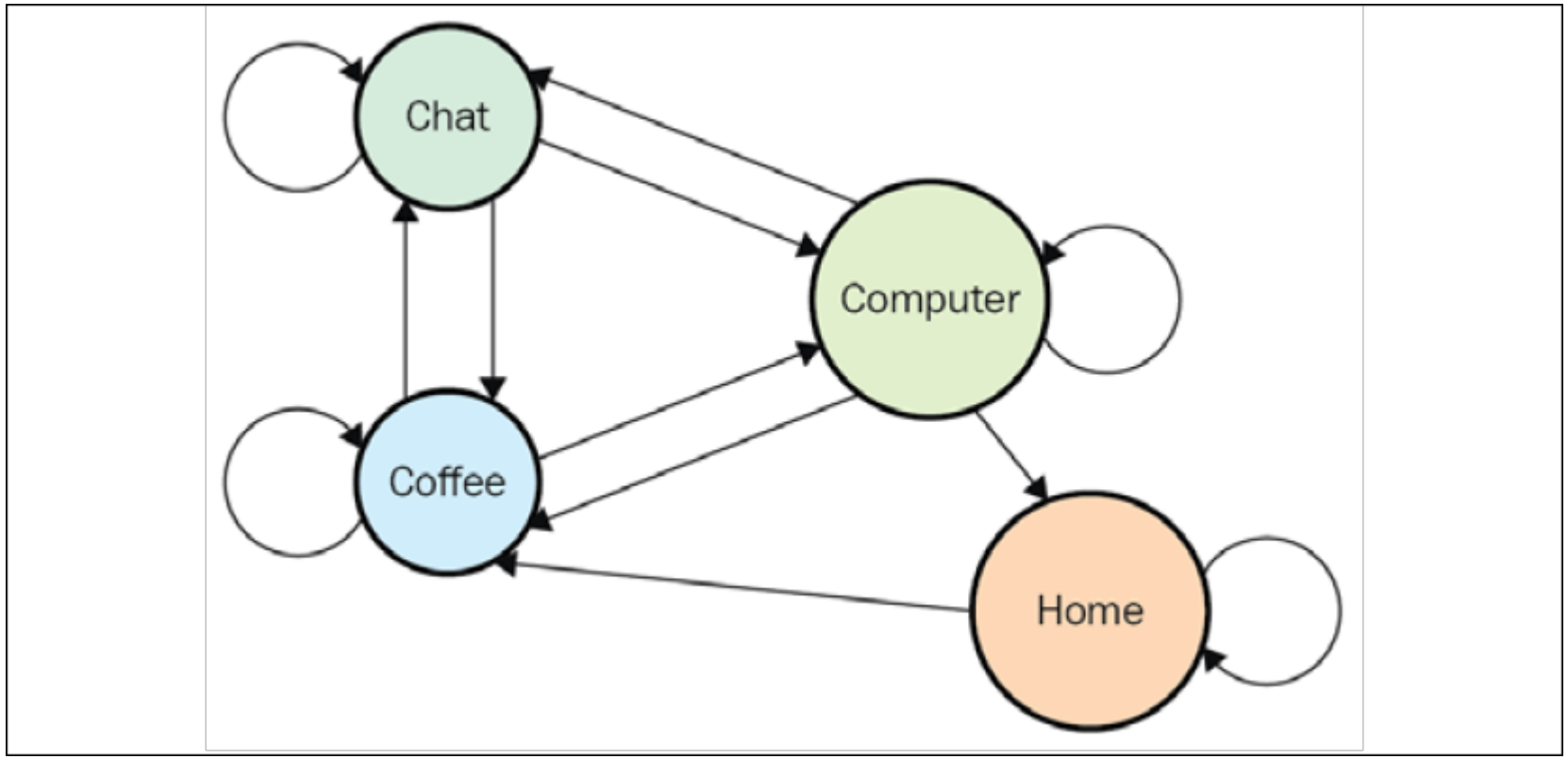
- Set of states S, set of actions A, initial state S_0
- Transition model P(s,a,s')
  - P([1,1], up, [1,2]) = 0.8
- Reward function r(s)
  - r([4,3]) = +1



- Goal: maximize cumulative reward in the long run
- Policy: mapping from S to A
  - $\pi(s)$ or $\pi(s,a)$ (deterministic vs. stochastic)
- Reinforcement learning
  - transitions and rewards usually not available
  - how to change the policy based on experience
  - how to explore the environment

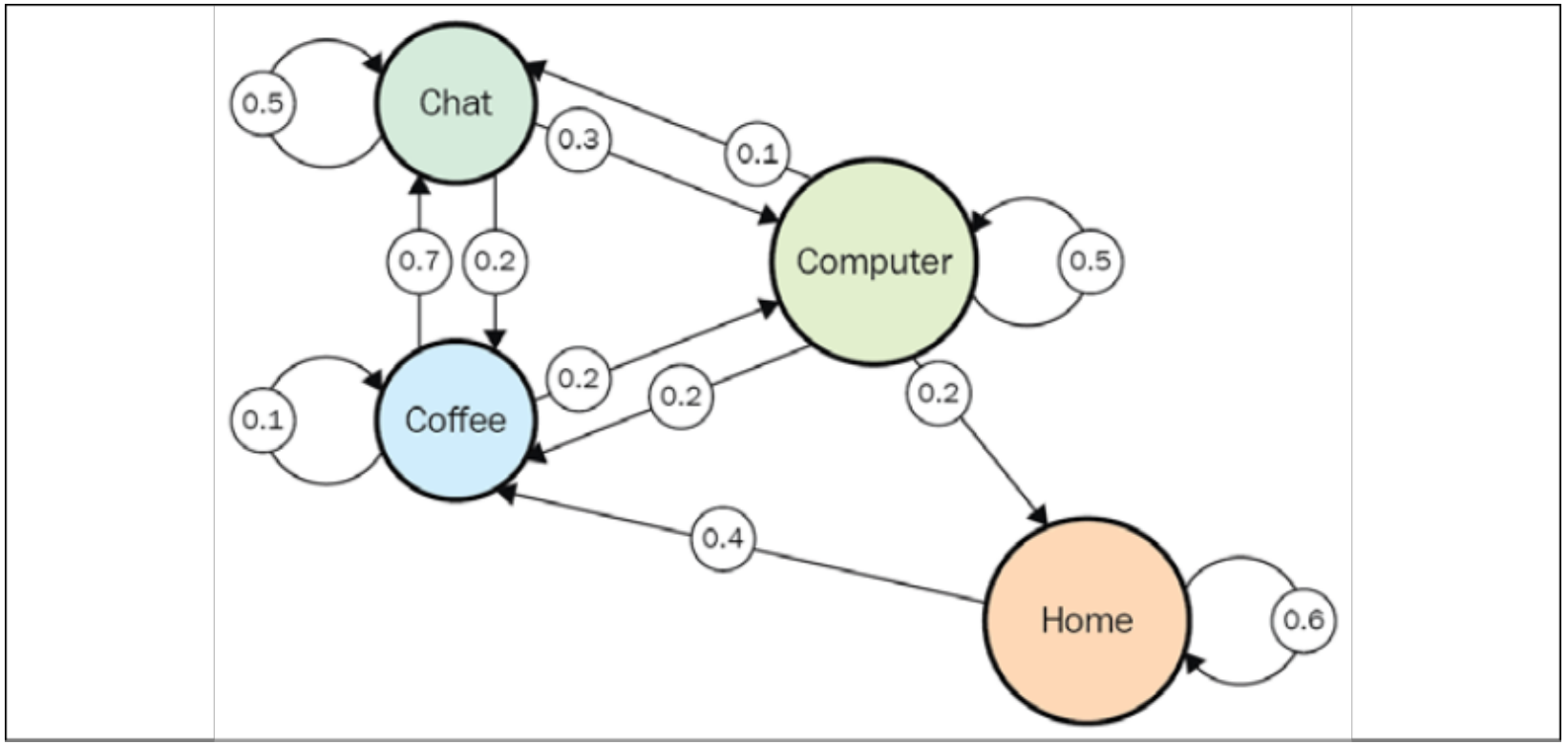
From Reinforcement Learning Tutorial,
Peter Bodík, RAD Lab, UC Berkeley

Let's define states

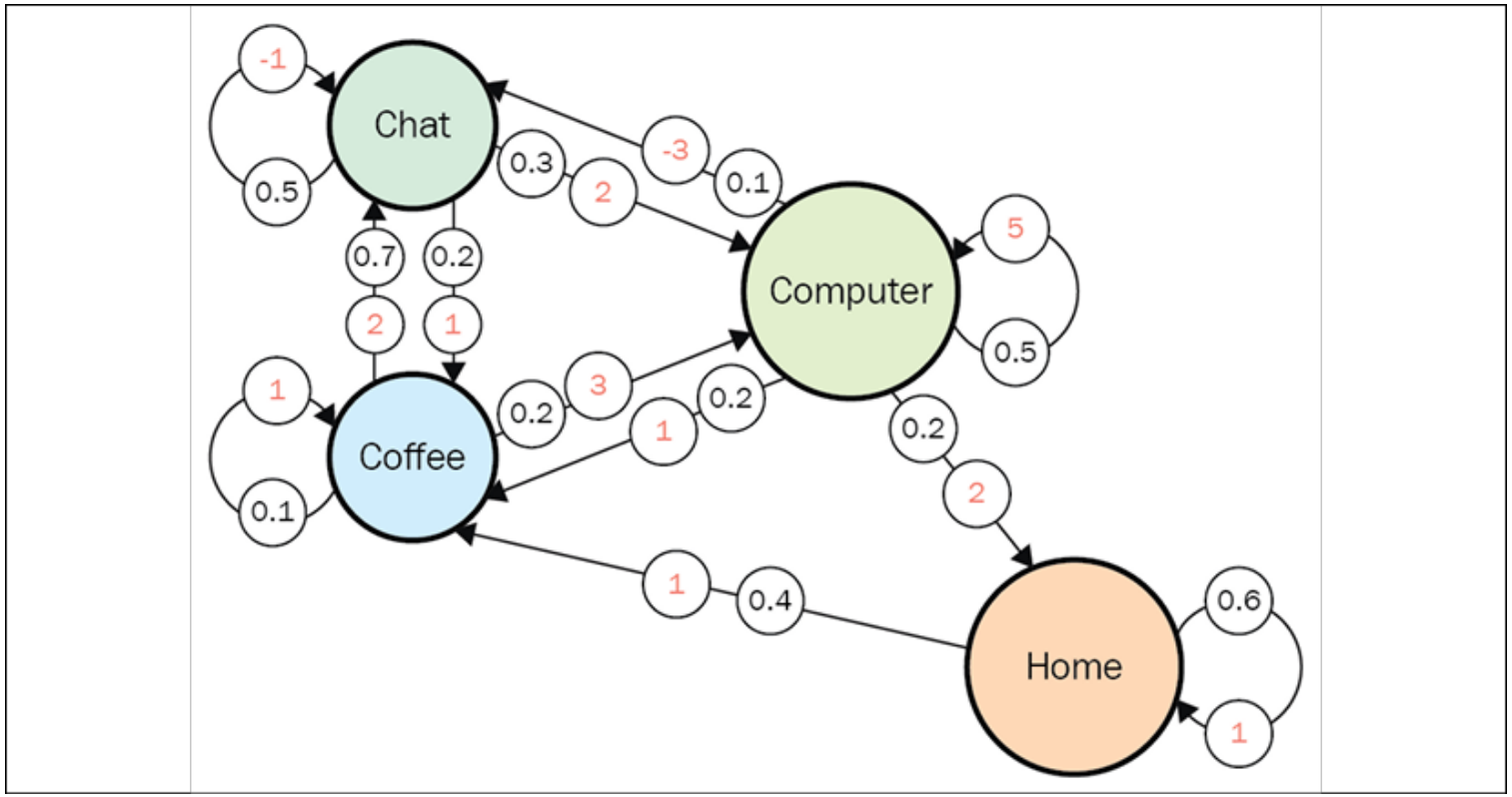


https://subscription.packtpub.com/book/data/9781838826994/1/ch01lvl1sec07/the-theoretical-foundations-of-rl

And transition probabilities



And rewards


Chat
Computer
Coffee
Home
-1
0.5
0.3
2
-3
0.1
5
0.5
0.7
2
0.2
1
1
0.1
0.2
3
1
0.2
0.2
2
1
0.4
0.6
1

Dynamic Programming

State Transition Probabilities: $P(s'|s,a)$— the probability of transitioning to state s' when taking action a from state s.

Rewards: $R(s,a)$— the expected reward for taking action a in state s.

Policy: A mapping $\pi(s)$ that specifies the action to take in each state s.

State Value Function: $V(s)$ — the expected total reward starting from state s, following a policy.

Action Value Function: $Q(s,a)$ — the expected total reward starting from state s, taking action a, and then following a policy.

Computing Return From Rewards

- Episodic (vs. continuing) tasks
  - "game over" after N steps
  - optimal policy depends on N; harder to analyze

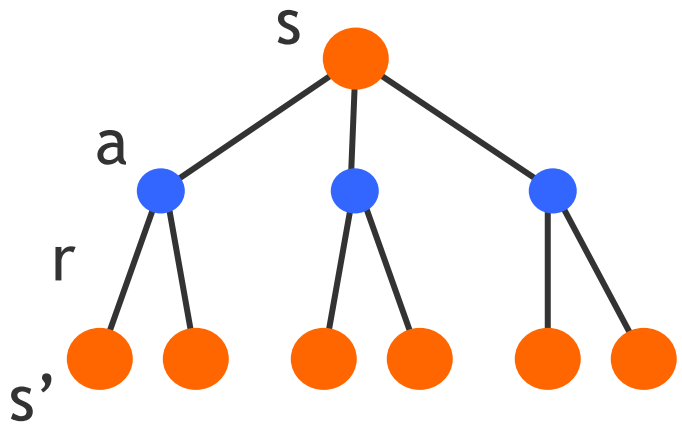
- Additive rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + r(s_1) + r(s_2) + \ldots$
  - infinite value for continuing tasks

- Discounted rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + \gamma * r(s_1) + \gamma^2 * r(s_2) + \ldots$
  - value bounded if rewards bounded

From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Value Function

- State value function: $V^{\pi}(s)$
  - expected return when starting in s and following π
- State-action value function: $Q^{\pi}(s,a)$
  - expected return when starting in s, performing a, and following π
- Useful for finding the optimal policy
  - can estimate from experience
  - pick the best action using $Q^{\pi}(s,a)$
- Bellman equation



$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[r^{a}_{ss'} + \gamma V^{\pi}(s') \right] = \sum_{a} \pi(s,a) Q^{\pi}(s,a)$$

From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Optimal Value Function

- There's a set of *optimal* policies
  - V^{π} defines partial ordering on policies
  - they share the same optimal value function

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

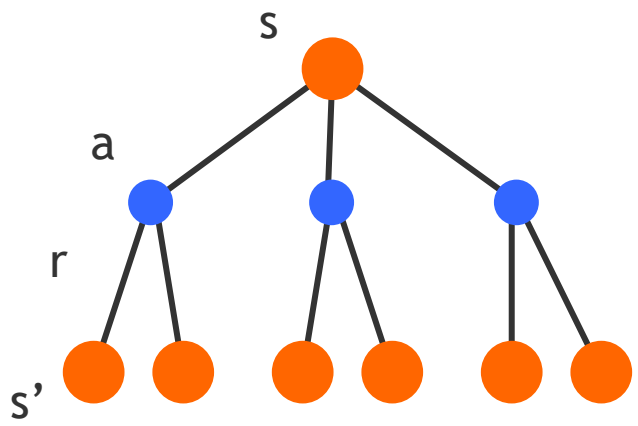
- Bellman optimality equation

$$V^*(s) = \max_{a} \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V^*(s') \right]$$

  - system of *n* non-linear equations
  - solve for V*(s)
  - easy to extract the optimal policy

- Having Q*(*s*,*a*) makes it even simpler

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



From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Value Iteration

Value Iteration finds the optimal value function by iteratively improving the estimates of $V(s)$ using the Bellman Optimality Equation. Once the optimal value function $V^*(s)$ is found, the optimal policy $\pi^*(s)$ can be derived by choosing the action that maximizes the expected return at each state.

Initialize: Start with an arbitrary value function $V(s)=0$ for all states.

Update: Iteratively update the value for each state using the Bellman equation:

$$V_{k+1}(s)=\max_a \sum_{s'} P(s'|s,a)[R(s,a)+\gamma V_k(s')]$$

Where:

- γ is the discount factor (between 0 and 1)
- $P(s'|s,a)$ is the transition probability.
- $R(s,a)$ is the immediate reward
- $V_k(s)$ is the value function at iteration k

Value Iteration

Stop Condition: Continue until $V(s)$ converges

Extract Policy: Once $V^*(s)$ converges, derive the optimal policy $\pi^*(s)$ by selecting the action that maximizes the expected return:

$$\pi^*(s)=\underset{a}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)[R(s,a)+\gamma V^*(s')]$$

Policy Iteration

Policy Iteration alternates between policy evaluation and policy improvement steps to iteratively converge to the optimal policy. Unlike value iteration, where values are updated for all actions, policy iteration evaluates the current policy and then improves it.

Policy Evaluation: Given a policy $\pi(s)$, compute the value function $V^{\pi}(s)$ by solving the Bellman equation for the fixed policy:

$$V(s)=\sum_{s'}P(s'|s,\pi(s))[R(s,\pi(s))+\gamma V(s')]$$

This can be solved iteratively or by solving a system of linear equations.

Policy Iteration

Policy Improvement: Update the policy by choosing the action that maximizes the expected return at each state:

$$\pi'(s) = \arg\max_{a} \sum_{s'} P(s'|s,a)[R(s,a) + \gamma V(s')]$$

Stop Condition: If the policy doesn't change after an improvement step, stop. Otherwise, repeat the evaluation and improvement steps.

Value Iteration vs. Policy Iteration

Value Iteration iteratively updates the value function using the Bellman Optimality Equation until convergence, then derives the optimal policy.

Policy Iteration alternates between evaluating the current policy and improving it until the policy stops changing.

Both methods work when the MDP's transition probabilities and rewards are known, and they converge to the optimal policy $\pi^*(s)$ and optimal value function $V^*(s)$. Value Iteration is more efficient when fewer iterations are needed, while Policy Iteration can be more stable in some environments.

Using Dynamic Programming

- Need complete model of the environment and rewards
  - robot in a room
    - state space, action space, transition model

- can we use DP to solve
  - robot in a room?
  - back gammon?
  - helicopter?

From Reinforcement Learning Tutorial, Peter Bodík, RAD Lab, UC Berkeley

Reinforcement Learning

- No knowledge of environment
  - Can only act in the world and observe states and reward
- Many factors make RL difficult:
  - Actions have non-deterministic effects
    - Which are initially unknown
  - Rewards / punishments are infrequent
    - Often at the end of long sequences of actions
    - How do we determine what action(s) were really responsible for reward or punishment? (credit assignment)
  - World is large and complex
- Nevertheless learner must decide what actions to take
  - We will assume the world behaves as an MDP