- States
    - Initial state
    - Successor state - state that can be reached by taking one action
    - Expanding the state - listing all successor states of some state s
- Search problem
    - Search problems solve for a sequence of optimal actions to minimize the total cost of these actions
    - Reinforcement learning is similar but the cost is unknown so the agent has learn the cost (usually in the form of reward) while finding the optimal actions
    - State spaces are usually huge so we do not want to search every state
- State space search
    - V - set of nodes - represents the states
    - E - set of edges - possible action at each state
    - c - cost/weights associated with each edge, assumed to be positive
    - Search strategy is the order in which the states are expanded
    - Frontier - leafs of a search tree
- Search performance
    - Search strategy is complete if it finds at least one solution
    - Search strategy is optimal if it finds the optimal solution
    - Compare time complexity (worst case)
    - Space complexity (worst case) maximum number of states stored in the frontier at a single time
    - ⇨ The depth of the goal state is denoted by d.
      ⇨ The maximum depth of a search tree is denoted by D.
      ⇨ The maximum number of actions associated with a state is called the branching factor, denoted by b.
- BFS
    - Convention - stop the search when the goal state is expanded, not when the goal is put in the frontier
    - T = 1+ b + b^2 + b^3 + … + b^d
    - S = b^(d+1)
    - Bidirectional search does BFS from the initial and goal states at the same time and stops when they meet
    - Complete
    - Optimal when the cost of every action is 1
- DFS
    - Incomplete if D = infinity
    - DFS is not optimal
    - Time complexity is T = 1 + b + b^2 + … + b^D - (b + b^2 + … + b^(D-d)) = 1+ b^(D-d+1) + b^(D-d+2) + … + b^D
    - S = (b-1) * D + 1

- What we want: optimal like BFS but low space complexity like DFS
- Iterative deepen search
  - IDS
  - Performs multiple DFS with increasing depth limits
  - DFS and stops if path length is larger than 1, larger than 2, …
  - IDS is complete
  - IDS is optimal when the cost of all action is 1
  - Time complexity is $T = (d+1) + db + (d-1)b^2 + ... + 3b^{d-2} + 2b^{d-1} + 1b^d$.
    ⇨ This count includes the root and the goal, and the first DFS step is only on the root.
    Space complexity is $S = (b-1)d + 1$.
- Heuristic
  - Additional information can be given in the form of a heuristic cost from any state to the goal state: this is an estimate or guess of the minimum cost from s to a goal state
  - The cost from the initial state to a state s is denoted by g(s)
  - The cost from the state s to the goal state is denoted by h*(s), which is unknown during the search
  - The estimated value of h*(s) is called the heuristic cost, denoted by h(s)
- Uniform cost search
  - Which is Dijkstra's algorithm
  - Expand the state with the lowest current cost g(s)
  - It is BFS with a priority queue based on g(s), and it is equivalent to BFS if the cost of every action is 1
  - Complete
  - Optimal
  - For this course, we remove repeated state with lower priority (higher cost)
    - But multiple expanding of one node is ok
  - Stop when the goal is expanded, not when it is in the queue
- Best first greedy search
  - Same as dijkstra except we use h(s) instead of g(s)
  - Expand the state with lowest heuristic cost h(s)
  - BFGS is not an abbreviation of greedy search, since it usually stands for Broyden Fletcher Goldfarb Shanno algorithm (a version of a gradient descent algorithm)
  - Greedy uses a priority queue based on h(s)
  - Greedy is incomplete
  - Greedy is not optimal
- A search
  - A search expands the state with the lowest total cost g(s) + h(s)
  - A search uses a priority queue on g(s) + h(s)

- A is complete
- A is not optimal

- Admissive heuristic
  - A heuristic is admissible if it never overestimates the true cost: 0<=h(s)<=h*(s)
  - A search with an admissible heuristic is called A* search
  - A* is complete
  - A* is optimal
  - Heuristic can be defined according to the scenario to be admissive
- Dominated heuristic
  - h2 dominates h1 (or h1 dominated by h2) if h1(s) <= h2(s) <= h*(s) for every state s
  - A* with a dominated heuristic is less informed and worse for A*
- Some variants
  - Iterative Deepening A* (IDA*) expands states with limit on $g(s) + h(s)$ instead of depth in IDS: Link, Wikipedia.
    Beam Search keeps a priority queue with a limited size: Wikipedia.
- Adversarial search
  - The search problem for games (one or more players searching at the same time) is adversarial search
  - [1 points] 5 pirate got 100 gold coins. Each pirate takes a turn to propose how to divide the coins, and all pirates who are still alive will vote whether to (1) accept the proposal or (2) reject the proposal, kill the pirate who is making the proposal, and continue to the next round. Use strict majority rule for the vote, and use the assumption that if a pirate is indifferent, they will vote reject with probability 50 percent. How will the first pirate propose? Enter a vector of length 5, all integers, sum up to 100.

    Answer (comma separated vector): .

    when there is only 1 player left: 0, 0, 0, 0, 100
    when there are two players left (need 2 votes): [0, 0, 0, 0, 100], 50% survive
    when there are three players (need 2 votes): [0, 0, 99, 1, 0]. will get 2 votes, survive
    when there are four players (need to get 3 votes): [0, 97, 0, 2, 1]. will get 3 votes from players 2, 4, 5, survive
    at the beginning of the game (need 3 votes): [97, 0, 1, 0, 2] will get 3 votes from players 1, 3, 5
  - Indifferent means that if the rewards for each choice are the same, choose any of them
  - The initial state is the beginning of the game.
  - Each successor represents an action or a move in the game.
  - The goal states are called terminal states, the search problem tries to find the terminal state with the lowest cost (highest reward).
  - The opponents search at the same time and try to minimize their costs (or maximize their rewards).
    - For zero-sum games, the sum of the rewards and costs for the two players is 0 at every terminal state. The opponent minimizes the reward (or maximizes the cost) of the first agent
- Minimax algorithms
  - Dfs on the game tree

- The game is solved backwards starting from the terminal states
    - Each player chooses the best action given the optimal action of all players in the subtrees
    - For zero-sum games, the optimal value at an internal state for the max player is called alpha(s) and for the min player is called beta(s)
    - The time and space complexity is the same as DFS with D = d
- Non-deterministic games
    - Some interval states represent movies by chance (can be viewed as another player), for example, dice roll or coin flip. For those states, expect castor reward are used instead of max or min
    - Dfs on games with chance is called expectiminimax
- Pruning
    - When a branch will not lead the current agent to win, the branch can be pruned (both players will not need to search the branch)
    - Alpha-beta pruning - dfs with pruning
    - ⇨ Here, the $\alpha(s)$ and $\beta(s)$ values are the current best value of an internal state so far (based only on the successor states that are expanded), which are not necessarily the final optimal values.
      ⇨ Prune the subtree after s if $\alpha(s) \geq \beta(s)$.
    - If we know that the branch will choose the current option (which is worse than an existing good option) or another one that is even worse, prune the branch
    - Alpha is for the max player and beta is for the min player
- Static evaluation function
    - The heuristic to estimate the value at an internal state for games is called a static (board) evaluation (SBE) function
        - For zero-sum games, SBE for one player should be the negative of the SBE for the other player
        - At terminal states, SBE should agree with the cost or reward at that state
    - For chess, the SBE can be computed by a neural network based some features such as material, mobility, king center control, or a convolutional neural network treating the board as an image
    - IDS can be used with SBE
        - In iteration d, the depth s limited to d and SBE of the internal states at depth d are used as their cost or reward
- Monte carlo tree search
    - Random subgames ca be simulated by selecting random moves for both players
    - Win rate is calculated by large number of simulation, and among them calculate win/total games

    - The move corresponding to the highest expected reward (win rates) can be picked
        - The move corresponding to the highest optimistic estimate of the reward can be also picked
- Rationalizability
    - Unlike sequential games, for simultaneous move games, one player (agent) does not know the action taken by the other player.
    - Given the actions of the other players, the optimal action is called the best response.
    - An action is dominated if it is worse than another action given all actions of the other players.
        - For finite games (finite number players and finite number of actions), an action is dominated if and only if it is never the best response.
        - An action is strictly dominated if it is strictly worse than another action given all actions of the other players. A dominated action is weakly dominated if it is not strictly dominated.
    - Rationalizability (IESDS, Iterative Elimination of Strictly Dominated Strategies): iteratively remove the actions are that dominated (or never best responses for finite games)
    - Write down an integer between 0 and 100 that is the closest to two thirds $\frac{2}{3}$ of the average of everyone's (including yours) integers.
        - 1-rationalizable: (actions that may be optimal, given other players are choosing valid actions)
            - 68, 69, 70, … 100 are not optimal no matter what the other players are choosing
            - 1-rationalizable actions: {0, 1, 2,.., 67}
        - 2-rationalizable: actions that may be optimal, given other players are choosing 1-rationalizable actions
            - 2-rationalizable actions: {0, 1, 2, …, 43, 44}
        - 3-rationalizable actions: {0, 1, 2, …, 30}
        - Infinity-rationalizable actions are called rationalizable actions: {0, 1}

[4 points] Perform iterated elimination of strictly dominated strategies (i.e. find rationalizable actions). Player A's strategies are the rows. The two numbers are (A, B)'s payoffs, respectively. Recall each player wants to maximize their own payoff. Enter the payoff pair that survives the process. If there are more than one rationalizable action, enter the pair that leads to the largest payoff for player A.

A \ B	I ☐	II ☑	III ☐
I ☑		3, 7	
II ☑		5, 4	
III ☑		2, 8	
IV ☐			

Answer (comma separated vector): .

- row player is player 1 (selecting one of the rows), column player is player 2 (choose one of the columns)
  if column plaher picks column 1: (4, 2, 1, 6)
  if col player picks column 3: (5, 3, 3, 9)
  column player's action 2 is strictly demonated, can be removed
  1-rationalizable action for col is {II, III}
  row player's action IV is strictly dominated by II, can be removed
  2-rationalizable action for row is {I, II, III}
  3-ratioinalizable action for column player is {II}
  the rationalizable actiion is {II} for row player and {II} for col player

- Nash equilibrium
  - If the actions are mutual best responses, the action forms a nash equilibrium
  - [4 points] What is the row player's value in a Nash equilibrium of the following zero-sum normal form game? A (row) is the max player, B (col) is the min player. If there are multiple Nash equilibria, use the one with the largest value (to the max player).

A \ B	I	II	III	IV
I	8 ☐☐	-2 ☑☐	10 ☑☐	-4 ☐☑
II	-3 ☐☐	-5 ☐☐	-1 ☐☐	-8 ☐☑
III	5 ☐☐	-2 ☑☑	0 ☐☐	9 ☐☐
IV	9 ☑☐	-10 ☐☑	6 ☐☐	10 ☑☐

Answer: .

  - for zero sum games, the valuess in the game matrix is for the max player (row player). either reward for max player or cost for min player.
    best response: fix other player's action and find my optimal response action.
    (III, II) is a pair of mutual best responses, therefore, it is (the) nash equilibrium
- Prisoner's dilemma
  - A symmetric simultaneous move game is a prisoner's dilemma game if the Nash equilibrium (using strictly dominant actions) is strictly worse for all players than another outcome
  - ⇨ For two players, the game can be represented by a game matrix: $\begin{bmatrix} - & C & D \\ C & (x,x) & (0,y) \\ D & (y,0) & (1,1) \end{bmatrix}$, where C stands for Cooperate (or Deny) and D stands for Defect (or Confess), and $y > x > 1$. Here, (D, D) is the only Nash equilibrium (using strictly dominant actions) but (C, C) is strictly preferred by both players.
  - (C, C) is preferred for both players, but (D, D) is the only Nash (rationalizable) actions

- Mixed strategy
    - When a player randomized between multiple actions
    - A pure strategy is when a player only use one action with probability 1
    - A mixed strategy nash equilibrium is a nash equilibrium for the game in which mixed strategies are allowed (called mixed extension of the original game)

$a_1 \setminus a_2$	Rock	Paper	Scissors
Rock	0	-1	1
Paper	1	0	-1
Scissors	-1	1	0

no pure strategy nsah
guess and check: guess (1/3, 1/3, 1/3)
given column is player (1/3, 1/3, 1/3), row player will get:
if rock: 0(1/3)+(-1)(1/3)+1(1/3) = 0
if paper, expected value: 1(1/3) + 0(1/3) + (-1)(1/3) = 0
if scissors, expected value: (-1)(1/3) + 1(1/3) + 0(1/3) = 0
in particular, row player can randomize they like, for example (1/3, 1/3, 1/3)
For rock paper scissors, both players choosing (1/3, 1/3, 1/3) is the only nash equilibrium

for example (1/2, 1/2, 0), R will give -1/2, P will give 1/2, S will give 0, so P is the unique best response

[4 points] Given the following game payoff table, suppose the row player uses a pure strategy, and column player uses a mixed strategy playing L with probability q. What is the smallest and largest value of q in a mixed strategy Nash equilibrium?

Row \ Col	L	R
U	5, 9	0, 9
D	0, 9	10, 0

if row player choose U: get 5 with prob p and 0 with prob 1-p, expected value is 5p
if row player chooses D, get 0 with prob p and 10 with prob 1-p, expected value is 10-10p
* best response for row player is: U with 5p>10-10p, whcih means p>2/3 ...... [horizontal blue linie at the top]
* D when 5p<10-10p, which means p<2/3 ...... [horizontal blue lnie at the bottom]
* U or D or any mix between U and D (q is anything) if p = 2/3 ...... [vertical blue line]

if col player choose L: get 9
if col player choose: R: get 9 with prob q, 0 with prob 1-q, expecetd value is 9q
best response for col player is
* L when q<1 ......[vertical redline]
* L or R or any mix between L and R (means p is anything) when q = 1 ....[horizontal red line]

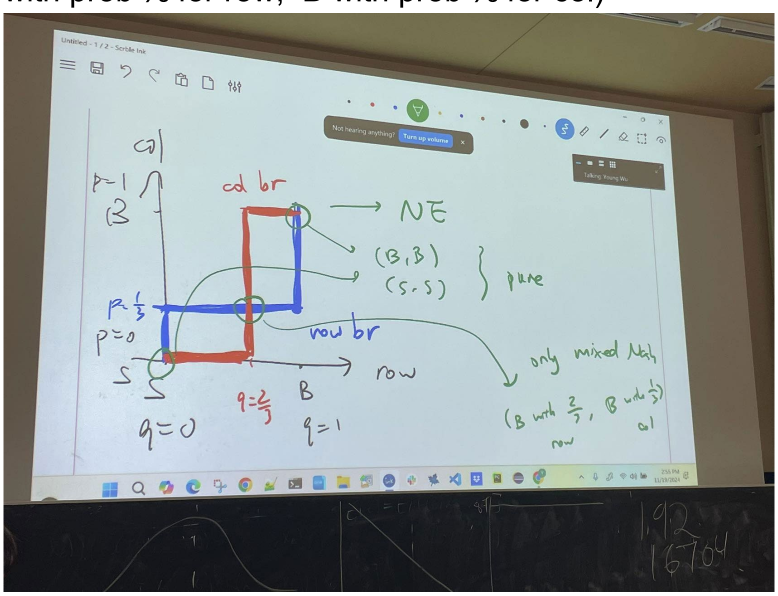
(U, L) is a pure Nash.
(U, L with prob 2/3, R with prob 1/3) is a mixed Nash
by looking at the best response plot: (U, L with any prob large than or equal to 2/3) are ALL Nash equilibria
(U, L with prob 1.3, R with prob 2/3) is not a nash

Romeo \ Juliet	Bach	Stravinsky
Bach	6, 3	0, 0
Stravinsky	0, 0	3, 6

Answer: Calculate

indifference conditions:
suppose Juliet is choosing B with prob p and S with prob 1-p
if Row chooses B, will get 6p
if Row chooses S, will get 3(1-p)
row will mix with 6p = 3-3p, meaning p = 1/3

juliet will mix with 3q = 6(1-q). meaning q = 2/3
things are nash in this case

- ○
- If col is choosing B: expected reward is 3q + 0(1-q) = 3q
- If col is choosing S: expected reward is 0q + 6(1-q) = 6-6q
- Best response for col player is:
- If 3q>6-6q or q>⅔, then best response is B
- If q = ⅔, then best response is B or S or any mix
- If q = ⅔, then best response is S
- There are two pure Nash: (B, B), (S, S), there is only one mixed Nash, (B with prob ⅔ for row, B with prob ⅓ for col)
- 

● Nash theorem - Every finite game has a (possibly mixed) Nash equilibrium

● Reinforcement learning
  - An agent interacts with an environment and receives a reward (or incurs a cost) based on the state of the environment and the agent's action

- The goal of reinforcement learning is to maximize the cumulative reward by learning the optimal action in every state
- Unlike search problems, the agent needs to learn the reward or cost function.

- Multi armed bandit problem
  - A simple reinforcement learning problem where the state does not change is called the multi-armed bandit

    🟩 There is a set of actions $1, 2, \ldots, K$, reward from action k follows some distribution with mean μ_k, for example normal distribution with mean μ_k and fixed variance σ^2, or $r \sim N\left(\mu_k, \sigma^2\right)$.

    🟩 The agent's goal is to maximize the total reward from repeatedly taking an action in T rounds.

    ⇨ Reward maximization: $\max_{a_1, a_2, \ldots, a_T} \left(r_1\left(a_1\right) + r_2\left(a_2\right) + \ldots + r_T\left(a_T\right)\right)$.

    ⇨ Regret minimization: $\min_{a_1, a_2, \ldots, a_T} \left(\max_k \mu_k - \frac{1}{T}\left(r_1\left(a_1\right) + r_2\left(a_2\right) + \ldots + r_T\left(a_T\right)\right)\right)$.
  - 🟩 An algorithm is called no-regret if as $T \to \infty$, the regret approaches 0 with probability 1: Wikipedia.

- Exploration vs exploitation
  - There is a trade-off between taking actions for exploration vs exploitation
    - Exploration: taking actions to get more information (e.g. figure out the expected reward from each action).
    - Exploitation: taking actions to get the highest rewards based on existing information (e.g. take the best action based on the current estimates of rewards).

    🟩 Epsilon-first strategy: εT rounds of pure exploration, then use the empirically best in the remaining $(1 - \varepsilon) T$ rounds.

    ⇨ Empirically best action is $\text{argmax}_k \hat{\mu}_k$, where $\hat{\mu}_k$ is the average reward from rounds where action k is used.
  - 🟩 Epsilon-greedy strategy: in every round, use the empirically best action with probability $1 - \varepsilon$, and use a random action with probability ε.

- Upper confidence bound

  🟩 The "best" action (based on current information) can also be defined based on the principle of optimism under uncertainty.

  🟩 An optimistic guess of the average reward (adjusted for uncertainty) is $\hat{\mu}_k + c\sqrt{\frac{2\log(T)}{n_k}}$, where n_k is the number of rounds action k is used.

  ⇨ This term $\sqrt{\frac{2\log(T)}{n_k}}$ represents the amount of uncertainty in the estimate $\hat{\mu}_k$, the more action k is explored (higher n_k), the smaller the value of this term.
  - 🟩 The algorithm that always uses the action with the highest UCB is called the UCB1 algorithm: Wikipedia.

- Markov decision process
  - States: s1, s2, … st (in the car example, it represents which square the car is on)
  - Action: a1, a2, …, at (in the car example, U, D, L, R, S)
  - Reward: R(st, at) is the reward you get from taking action at when you are in state st sometimes it depends on st+1 as well, realized random rewards are r1, … rt
  - Transition: s’ = st+1 = T(st, at) is which state you will go to in next period if you start at st and used action at
  - A markov decision process is a set of states, actions, reward and transition function
  - Goal: we are trying to find a policy function: pi(s) it takes in state s and output an optimal action a
- Value function

    - Optimal means we want to policy that leads to the highest (total, discounted, expected) value
    - V(s) = E[r1 + beta r2 + beta^2 r3 + …], where beta is the discount factor
    - If the rewards are bounded by 1, then value is bounded by 1+beta+beta^2+... = 1/(1-beta), assumes beta<1
    - In the 2 state 2 action example, beta = 0.8
        - States: A, B
        - Actions: stay, move
        - Rewards: R(A, stay) = 1, R(B, stay) = 2, R(A, move) = R(B, move) = 0
        - Transition function: T(A, stay) = A, T(B, stay) = B, T(A, move) = B, T(B, move) = A
        - Guess optimal policy is pi(A) = move, pi(B) = stay
        - V^{pi}(A) = 0 + 0.8*2 + 0.8^2*2 + 0.8^3*2 + … = 2(1 + 0.8 + 0.8^2 + …) - 2 = 2/(1-0.8) - 2
        - V^{pi}(B) = 2+0.8*2 + 0.8^2*2 + … = 2/(1-0.8) = 10
        - Try another policy pi'(A) = pi'(B) = stay
        - V$^{pi'}$(A) = 1 + 0.8*1 + 0.8^2*1 + … = 1/(1-0.8) = 5 < 8, this policy is worse at state A compared to the previous policy
        - pi''(A) = stay, pi''(B) = move
- Q function
    - Value function given a specific action in the current period (and follows pi in future periods).
    - Q^{pi}(s, a) = R(s, a) + beta * V^{pi}(s') = R(s$_t$, a$_t$) + beta*R(s$_{t+1}$, pi*(s$_{t+1}$)) + beta^{2}R(s$_{t+2}$, pi*(s$_{t+2}$)) + …
    - Q function is value you get from taking action a in the current period, and the action specified by pi starting from the next period
    - Then pi * (s) = argmax$_a$ Q^{pi}(s, a)
- Q learning
    - The Q function can be learned by iteratively update the Q function using the Bellman's equation: Link, Wikipedia.
      ⇨ $\hat{Q}(s_t, a_t) = (1-\alpha)\hat{Q}(s_t, a_t) + \alpha\left(r_t + \beta \max_a \hat{Q}(s_{t+1}, a)\right)$, where α is the learning rate and is sometimes set to $\frac{1}{1+n(s_t, a_t)}$, $n(s_t, a_t)$ is the number of visits to state s_t and action a_t in the past.
      Under certain assumptions, Q learning converges to the correct (optimal) Q function, and the optimal policy can be obtained by: $\pi(s_t) = \text{argmax}_a Q^\star(s_t, a)$ for every state.
    - Q learning is not learning the value function Q of our epsilon greedy policy, it is just converging to the optimal policy, therefore it is an off-policy RL algorithm
    - SARSA is an on-policy RL algorithm, it actually learns the Q of our epsilon greedy policy
- SARSA

- An alternative to Q learning is SARSA (State Action Reward State Action). It uses a pre-specified action for the next period instead of the optimal action based on the current Q estimate: Wikipedia.
  ⇨ $\hat{Q}(s_t, a_t) = (1-\alpha)\hat{Q}(s_t, a_t) + \alpha\left(r_t + \beta\hat{Q}(s_{t+1}, a_{t+1})\right)$.
  The main difference is the action used in state s_{t+1}.
  ⇨ Q learning is an off-policy learning algorithm since a_{t+1} is the optimal policy in the next period, not a pre-specified policy (the Q function during learning does not correspond to any policy).
  ⇨ SARSA is an on-policy learning algorithm since it computes the Q function based on a fixed policy.
- Exploration vs exploitation
  - The policy used to generate the data for Q learning or SARSA can be Epsilon Greedy or UCB (requires some modification for MDPs).
    ⇨ Epsilon greedy: with probability ε, a_t is chosen uniformly randomly among all actions; with probability $1-\varepsilon$, $a_t = \text{argmax}_a \hat{Q}(s_t, a)$.
    The choice of action can be randomized too: $\mathbb{P}\{a_t | s_t\} = \dfrac{c^{\hat{Q}(s_t, a_t)}}{c^{\hat{Q}(s_t,1)} + c^{\hat{Q}(s_t,2)} + \ldots + c^{\hat{Q}(s_t,K)}}$, where c is a parameter controlling the trade-off between exploration and exploitation.
- Deep Q learning
  - In practice, Q function stored as a table is too large if the number of states is large or infinite (the action space is usually finite): Link, Link, Wikipedia.
    If there are m binary features that represent the state, then the Q table contains $2^m |A|$, which can be intractable.
    In this case, a neural network can be used to store the Q function, and if there is a single layer with m units, then only $m^2 + m|A|$ weights are needed.
    ⇨ The input of the network $\hat{Q}$ is the features of the state s, and the outputs of the network are Q values associated with the actions a or $Q(s, a)$ (the output layer does not need to be softmax since the Q values for different actions do not need to sum up to 1).
    ⇨ After every iteration, the network can be updated based on an item $\left(s_t, (1-\alpha)\hat{Q}(s_t, a_t) + \alpha\left(r_t + \beta \max_a \hat{Q}(s_{t+1}, a)\right)\right)$.
- Local search
  - For some problems, every state is a solution, only some states are better than other states specified by a cost function (sometimes score or reward)
  - The search strategy will go from state to state, but the path between states is not important
  - Local search assumes similar states have similar costs, and search through the state space by iteratively improving the cost to find an optimal state
  - The successor states are called neighbors (or move sets)
- Hill climbing
  - Discrete version of gradient descent
    - Starts at random state
    - Move to the best neighbor (successor) state
    - Stop when all neighbors are no better than the current state (local minimum)
  - Random restarts can be used to pick multiple random initial states and find the best local minimum (similar to neural network training)
  - If there are too many neighbors, first choice hill climbing randomly generates neighbors until a better neighbor is found
- Simulated annealing
  - Simulated annealing uses a process similar to heating solids (heating and slow cooling to toughen and reduce brittleness)
    - Each time, a random neighbor is generated
    - If the neighbor has a lower cost, move to the neighbor

⇨ If the neighbor has a higher cost, move to the neighbor with a small probability: $p = e^{-\dfrac{\left| f(s') - f(s) \right|}{T(t)}}$, where f is the cost and $T(t)$ is the temperature and decreasing in t.
⇨ Stop until bored.

- Simulated annealing is a version of Metropolis-Hastings algorithm: Wikipedia.

- Temperature
  - The temperature function should be decreasing over time. They can change arithmetically or geometrically.
    ⇨ Arithmetic sequence: for example, $T(t+1) = \max\left\{T(t) - 1, 1\right\}$.
    ⇨ Geometric sequence: for example, $T(t+1) = 0.9T(t)$.
    When the temperature is high: almost always accept any state.
    When the temperature is low: first choice hill climbing.

- Genetic algorithm
  - Genetic algorithm starts with a fixed population of initial states, and the successors are found through cross-over and mutation: Wikipedia.
    Each state in the population with N states has probability of reproduction proportional to the fitness (or negatively proportional to the costs): $p_i = \dfrac{f(s_i)}{f(s_1) + f(s_2) + \ldots + f(s_N)}$.
    If the states are encoded by strings, cross-over means swapping substrings at a fixed point: for example, abcde and ABCDE cross-over at position 2 results in abCDE and ABcde: Wikipedia.
    If the states are encoded by strings, mutation means randomly updating substrings with a small probability called the mutation rate: for example, abcde can be updated to abCde or aBcDe or ... with small probabilities: Link
    Genetic algorithm: in each generation, the reproduction process is:
    ⇨ Randomly sample two states based on the reproduction probabilities.
    ⇨ Cross-over these two states to produce two children states.
    ⇨ Mutate these two states with small probabilities.
    ⇨ Repeat the process until the same population size is reached, and continue to the next generation.
  - [4 points] When using the Genetic Algorithm, suppose the states are $\begin{bmatrix} x_1 & x_2 & \ldots & x_T \end{bmatrix}$ = $\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$, $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$, $\begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$, $\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$. Let T = 6, the fitness function (not the cost) is $\text{argmax}_{t \in \{0, \ldots, T\}} x_t = 1$ with $x_0 = 1$ (i.e. the index of the last feature that is 1). What is the reproduction probability of the first state: $\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$?

    Answer: . Calculate

    the fitness for states are: note [0, 0, 0, 0, 0, 0] has fitness of 0
    (1): f(s1) = 3, f(s2) = 6, f(s3) = 4, f(s4) = 4
    prob(s1) = 3/(3+6+4+4) = 0.1765
    prob(s2) = 6/17 = 0.3529
    prob(s3) = prob(s4) = 4/17 = 0.2353
    randomly sample two states based on this distriibution [0.1765, 0.3529, 0.2353, 0.2353]
    suppose for exampke, we sampked s2 and s3, cross over after position 3
    "001001 and "101100", the children are "001 100" and "101 001"
    suppose we sampled randomly again, s2 and s1
    "001001" and "101000" , the children are "001 000" and "101 001"
    then mutation on these four children states, repeat for next generation
  - Variants
    The parents do not survive in the standard genetic algorithm, but if reproduction between two copies of the same states is allowed, the parents can survive.
    The fitness or cost functions can be replaced by the ranking.
    ⇨ If state s_i has the k-th lowest fitness value among all states, the reproduction probability can be computed by $p_i = \dfrac{k}{1 + 2 + \ldots + N}$.
    In theory, cross-over is much more efficient than mutation.
    ▼Example
    - Many problems can be solved by genetic algorithm (but in practice, reinforcement learning techniques are more efficient and produce better policies).

- State representation of neural networks
  - A neural network can be represented by a sequence of weights (a signal state)
  - Two neural networks can swap a subset of weights (cross-over).

- One neural network can randomly update a subset of weights with small probability (mutation).
- Genetic algorithms can be used to train neural networks to perform reinforcement learning tasks.