Agent

An agent is defined by:

1. **Action Space (A) :** The set of all action the agent can do
2. **Percept Space (P):** The set of all things that the agent can perceive in the world
3. **State space (S) :**The set of all states that the agent can be in.
4. **Word dynamics (T: SXA→S):** the function mapping a state and a action to a next state
5. **Percept Function(Z: S→P):** The function defining all the objects the agent can perceive at S.
6. **Utility Function (S→ R):** Function defining the utility (desirability) of a certain state (mapping to a real number).

An agent is trying to optimize the actions it will take to maximise the utility function (i.e reach the more desirable states.)

An agent will be designed differently depending on its environment.

An Environment is defined by

7. **Discrete/Continuous:** Are the state/Action/percept space discrete (i.e is there a finite number of states ?)
8. **Deterministic / Stochastic(Non deterministic):** Does the agent always know exactly what the next state will be after performing an action.
   Is the world dynamics a one-to-one relation.
9. **Fully observable vs Partially observable:** does the agent know the exact state of the world/itself or only has partial information.
10. **Static/Dynamic:** Can the world change while the agent is thinking?

Searches

To formulate a problem as a search problem, we define the Agent (A,P,S,T,Z,S) + a initial and goal state.

We start by generating a **State graph** defined by:

11. Vertex V: Represent states
12. Edges E: Represents world dynamics, each edge is labelled by its cost.

The solution is a path: within the graph from initial to goal vertices.
The cost associated with the solution is the sum of the cost of all edges in the path.
The optimal solution is the path with the minimum cost.

We measure the performance of a search with:

13. **Completeness:** The algorithm will find a solution whenever one exists.
14. **Optimality:** The algorithm will return the minimum cost path whenever there is one.
15. **Complexity:** Amount of time and memory needed by the algorithm to solve the problem.

We can represent the visited nodes in a graph with a **Search Tree**, with root = initial state and the tree expanding everytime we visit a new node.

Deterministic search in discrete space

Environment is Discrete, Fully observable, Deterministic and static.

Blind searches

Blind searches do not estimate the cost from the current vertex to the goal vertex

Breadth first search(BFS)

Explore all nodes off the current level of search tree before exploring deeper nodes.
Uses FIFO queue to keep fringe nodes
Cost: #steps
Algorithm:

1. Set the initial vertex I as root of the search tree.
2. Push I to the queue.
3. Loop
    a. t = queue.dequeue(), mark t as expanded
    b. If t is the goal vertex
        return.
    c. For each v in successor(t)
        i. If v is not in the tree yeT
            1. Queue.push(v)
            2. Put v as a child of t in the search tree

BFS is complete if b is **finite**

It will generate the **optimal** solution.

Complexity: Time&Space = O(b^d) (where b = branching factor d – shallowest goal node.)

Depth first search(DFS)

Explores ones full SubGraph before exploring the other.
Uses a stack (LIFO) to keep track of the fringe nodes.
Cost: #steps
Algorithm:

Set the initial vertex I as root of the search tree
Push I to the stack.
Loop
- t = front of the stack.
- Remove t from the stack and mark t as expanded
- If t is the goal vertex, then return.
- For each v in successor(t)
    - If v is not in the path to t yet
        - Push v to the stack.
        - Put v as a child of t in the search tree.

DFS is **complete** if b&m are finite (b = branching factor, m= max depth)
DFS is **NOT optimal**
DFS has complexity: time: O(b^m) ;space: O(bm).

Iterative deepening DFS

Performing multiple DFS with a specified Depth limit that increments at every iteration until goal is found.
Iterative DFS is **complete** if b is finite
It is **optimal** in terms of #steps.
Complexity: Time O(b^d), Space: O(bd).

Uniform cost search

Expand fringe node with lowest cost from root, uses a Priority Queue (PQ) to keep fringe nodes.
g(n) = Cost from root to node n, expand the n that minimises g(n).
Algorithm:

Set the initial vertex I as root of the search tree.
Push I to the PQ.
Loop
- t = retrieve a node from PQ.
- Remove t from PQ and mark t as expanded
- If t is the goal vertex, then return.
- For each v in successor(t)
  - If v has not been expanded yet
    - Insert v to the PQ.
    - Put v as a child of t in the search tree.

Uniform cost search is **Complete** if b is finite and all edges have cost> e (where e is the min cost of a ste)
It is **Optimal** if all edges have a positive cost
Complexity : Time&Space : O(b^(1+floor(C/e)) where C is the cost of the optimal solution.

Informed searches

Informed search algorithms try and estimate the cost from the current node to the goal.
g(n) = Cost from root to node n
h(n) = Estimated cost from n to goal (based on heuristics)
in an informed search **the next node selection is based on a function f(n)** which includes h(n).

Greedy Best first search

Expand fringe nodes with lowest estimated cost form current node to goal node.
f(n) = h(n).
g(n) is ignored.
Expand node with lowest f(n).
Uses PQ to store fringe node, but the PQ is based on f(n) the node with the lowest f(n) value will have higher priority.
Algorithm:

- Set the initial vertex I as root of the search tree.
- Push I to the PQ.
- Loop
  - t = retrieve a node from PQ.
  - Remove t from PQ and mark t as expanded
  - If t is the goal vertex, then return.
  - For each v in successor(t)
    - Insert v to the PQ.
    - Put v as a child of t in the search tree.

Not Complete
Not optimal
Complexity: Worst case O(b^m) (b = branching factor, m = maximum depth).

A search*

The priority queue uses f(n) = g(n) +h(n) as priority, the lowest f(n) has the highest priority.
g(n) = cost from root to node n
h(n) = estimated cost from node to goal
f(n) = g(n)+h(n).
Algorithm:

Set the initial vertex I as root of the search tree.
Push I to the PQ.
Loop

- t = retrieve a node from PQ.
- Remove t from PQ and mark t as expanded
- If t is the goal vertex, then return.
- For each v in successor(t)
  - Insert v to the PQ.
  - Put v as a child of t in the search tree.

An heuristic is admissible if it never overestimates the cost from n to goal.
Complete if all edges have cost >e
Optimal if Complete and heuristic is admissible.
Complexity depends on heuristics.

Deterministic search in continuous space/Motion Planning

Motion planning

Study of computational methods enabling agents to choose its own motions to get to a goal state from an initial state.

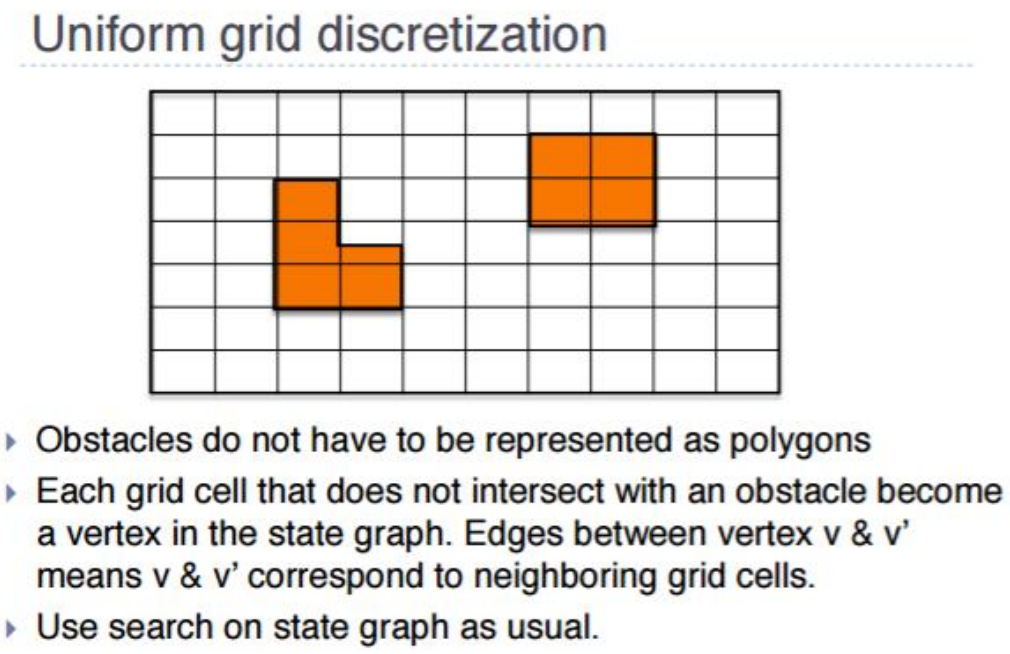
We need to discretise our infinite state graph.

2D motion planning

Visibility graph

undirected graph : nodes are vertices of obstacles, an edge between 2 vertices represents an edge for the obstacle or a collision free straight line between 2 vertices.
Given an initial and goal state: find the vertex Vi nearest to to init where the straight line segment between qi and init is collision free, same for the goal.
Complexity: Construction time = O(n^3); space = O(n^2)

Uniform grid discretization

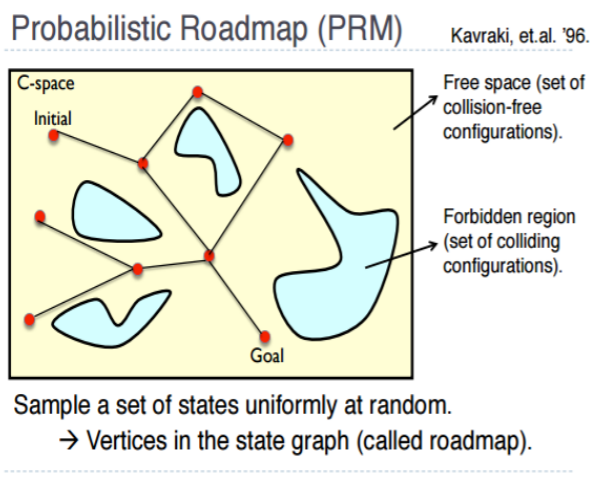


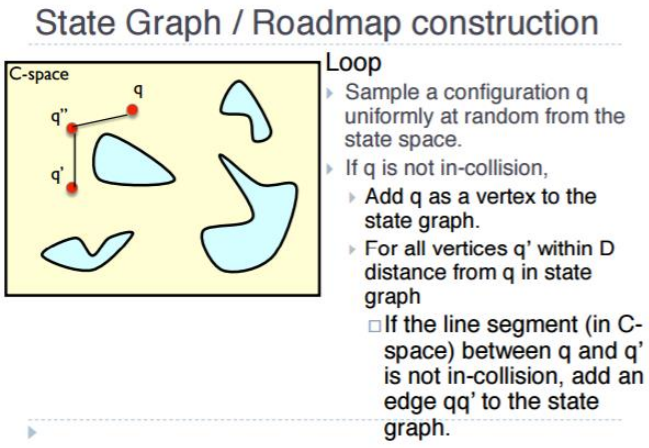
Higher dimensions.

Visibility graph and uniform grid still hold

To build the state graph we can use

Probabilistic Road Map (PRM)





Once the graph is constructed usual search can be applied to find a path.

Interleaving: Doing graph construction and searching simultaneously, every n vertices added, search the graph.

Sampling strategies: Random, near obstacle, in between 2 obstacles….

Multi-arm bandit problem: Technique for trading off exploration vs exploitation, the strategy that would be the most useful in solving the problem is chosen with higher probability.

Collision checks.

Line collision check: having 2 linep1q1, p2q2; if p1q1p2 & p1q1q2 have different orientations AND p2q2p1 &p2q2q1 have different orientation

Hierarchical bounding volume (HBV): Tree of bounding volume for each object to check, start by building the bounding volumes of the primitive lines and build up from there.
Then when checking for collision use the tree to quickly check for collision before going into **Line collision checks** (if one is found at a leaf node).

Distance calculation with HBV:

- Suppose T (previous slide) is a hierarchical bounding sphere of O.
- Distance between a point p and O can be computed using T as follows:
  1. Initialize the current known distance $d_c(p, O)$. If no additional information, $d_c(p, O) = inf$.
  2. Add t_1 (the root of T) to stack S.
  3. **Dist(p, O) = DFSDistance**(T, S, $d_c(p, O)$)

DFSDistance(T, S, $d_c(p, O)$):

If (S is not empty)
  Let t be the element at the top of S, and remove this element from stack S.
  If dist(p, t) < $d_c(p, O)$
    If t is a leaf node
      Compute dist(p, prim(t)), where prim(t) is the geometric primitive encapsulated in t.
      If $d_c(p, O)$ > dist(p, prim(t)) then $d_c(p, O)$ = dist(p, prim(t))
    Else
      Expand t (i.e., add the children of t the stack S).
    Return DFSDistance(T, S, $d_c(p, O)$)
Return $d_c(p, O)$

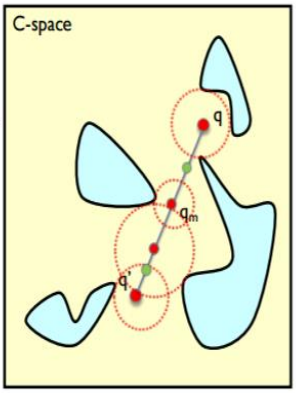
Collision check for a line segment
A better method using distance computation

If q or q' is in collision, return collision.

Compute distance d_q between q and its nearest forbidden region. Create an empty ball B_q with center q and radius d_q. Similarly for q'.

Let $q_m = (q+ q')/2$

If q_m is inside B_q and $B_{q'}$, the entire segment qq' is collision free. Otherwise, repeat from #1, but for segments qq_m and q_mq'.



Logic, representation, validity, satisfiability

Inference rules

- **Transformation for logical expressions**
- **Modus ponens**

$$\frac{\alpha \rightarrow \beta \quad \alpha}{\beta}$$

- **Modus tollens**

$$\frac{\alpha \rightarrow \beta \quad \sim\beta}{\sim\alpha}$$

- **And-elimination**

$$\frac{\alpha \wedge \beta}{\alpha}$$

α, β: Sentence (atomic or complex).

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{De Morgan}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{De Morgan}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

Propositional logic

Atomic sentences that can be either **true or false.**

Logic operators: Not, AND, OR, implication, biconditional implication

Valid sentence: All possible interpretations of the sentence are true. (Eg P V ~P)

Satisfiable sentence: There exists an interpretation of the sentence such that it Is true. (Eg ~P) (everything that is valid is also satisfiable).

Unsatisfiable sentence: All possible interpretations of the sentence are false (Eg: P ^ ~P)

Propositional Logic Problem

To formulate propositional logic problems, we create a **Knowledge Base (KB).**

KB: a set of sentences such that KB is false in models that contradict what the agents knows.

Validity problems

Is the given sentence valid?

Model checking:

Create truth table and check that the sentence is true given that all premises in KB are true.

Theorem proving.

Using Inference rules to reduce the sentence by using the KB.

- State space:
  - All possible set of sentences.
- Action space:
  - All inference rules.
- World dynamics:
  - Apply the inference rule to all sentences that match the above the line part of the inference rule. Become the sentence that lies below the line of the inference rule.
- Initial state:
  - Initial knowledge base.
- Goal state:
  - The state contains the sentence we're trying to prove.

Theorem proving can also be done using searches in **Natural deduction.**

Conjunctive Normal Form(CNF): (AVB)^(CVD)^(EVF)… conjunction of disjunctions.

When in CNF used we can use **Resolution.**

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

α, β: Sentence (atomic or complex).

RESOLUTION REFUTATION

Theorem proving -- Resolution refutation

- Three steps:
  - Convert all sentences into CNF.
  - Negate the desired conclusion.
  - Apply resolution rule until
    - Derive false (a contradiction).
    - Can't apply the rule anymore.

Satisfiability problems

There are several ways of solving Satisfiability problems

Model checking: *Davis Putnam Logeman Loveland*

DPLL(Sentence S)

- If S is empty, return True
- If S has an empty clause, return False
- If S has a unit clause U, return DPLL(S(U))
  - Unit clause: Consists of only 1 literal
  - S(U) means a simplified S after a value is assigned to U
- If S has a pure clause U, DPLL(S(U))
  - Pure clause: Appear as positive only or negative only in S
- Pick a variable v,
  - If DPLL(S(v)) then return true Else return DPLL(S(~v))
  - Heuristic to pick the variable: Max #occurences, Min size clauses. Basically, pick the most constrained variable first (if it's going to fail, better fail early).

Model Checking: *GSAT*

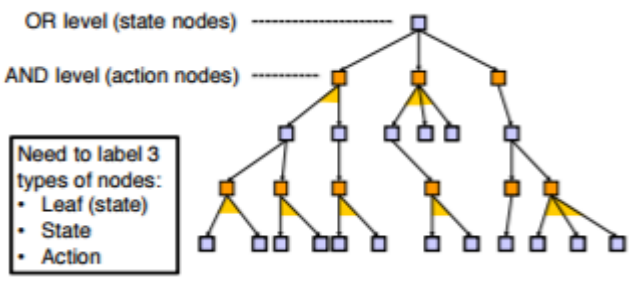
- **GSAT(Sentence S)**
  - Loop n times
    - Randomly choose assignment for all variables, say A
    - Loop m times
      - Flip the variable that results in lowest cost
        Cost(assignment) = number of unsatisfied clauses
      - Return True if cost is zero
- **Sound but not complete**
- **Cannot be used to generate all satisfiable assignments**

- Weakly constrained problems (large proportion of the assignments is satisfiable):
  - Easy for DPLL ☺ & Easy for GSAT ☺
- Highly constrained problems (very few satisfiable assignments, sometimes only 1):
  - Easy for DPLL ☺ , but Hard for GSAT ☹
- Problems in the middle:
  - Hard for DPLL ☹ & Hard for GSAT ☹

AND/OR Tree

Fully observable, Non-deterministic, static, discrete Environment.

An and or tree is a tree interleaving And/Or levels.
Each Or level branching is introduced by the agents choice.
Each And level branching is introduced by the environments



Labeling

Node/Label	Solved	Closed
Leaf(state)	It is a goal state.	It is not a goal.
AND node (action)	All its children are solved.	At least one of its children is closed.
OR node (state, non-leaf)	At least one of its children is closed.	All its children are closed.

Label from leaf nodes to root nodes if the root node is solved the problem is solved, if the root node is closed the problem is impossible to solve.

Solution

The subtree that establishes that the root is solved is the solution to the AND/OR tree.

it defines a conditional plan to pick the next action.

AND/OR Tree search.

Search an AND-OR Tree

- Start from a state node (OR level).
  - Fringe nodes are state nodes.
- Use any search algorithms we have studied,
  - Select a fringe node to expand.
  - Select an action to use.
  - Insert the corresponding action node.
  - Insert all possible outcome of the action, as the child of the action node.
  - Backup to (re-)label the ancestor nodes.
- Cost calculation at AND level:
  - Weighted sum (when uncertainty is quantified using probability, expectation).
  - Take the minimum.

Min-Max

A min max tree is similar to an AND/OR tree but is designed to model adversarial situation.

OR levels are agents moves, called **MAX**

AND levels are Opponents move, called **MIN.**

Evaluation function

e: S→R

e>0 favourable to max, e<0 favourable to min, e=0 neutral.

- Usually a weighted sum of "features":

$$e(s) = \sum_{i=1}^{n} w_i f_i(s)$$

w: weight.

f(s): features.

- Features may include
  - Number of pieces of each type
  - Number of possible moves
  - Number of squares controlled

MINIMAX ALGORTIHM

Minimax Algorithm

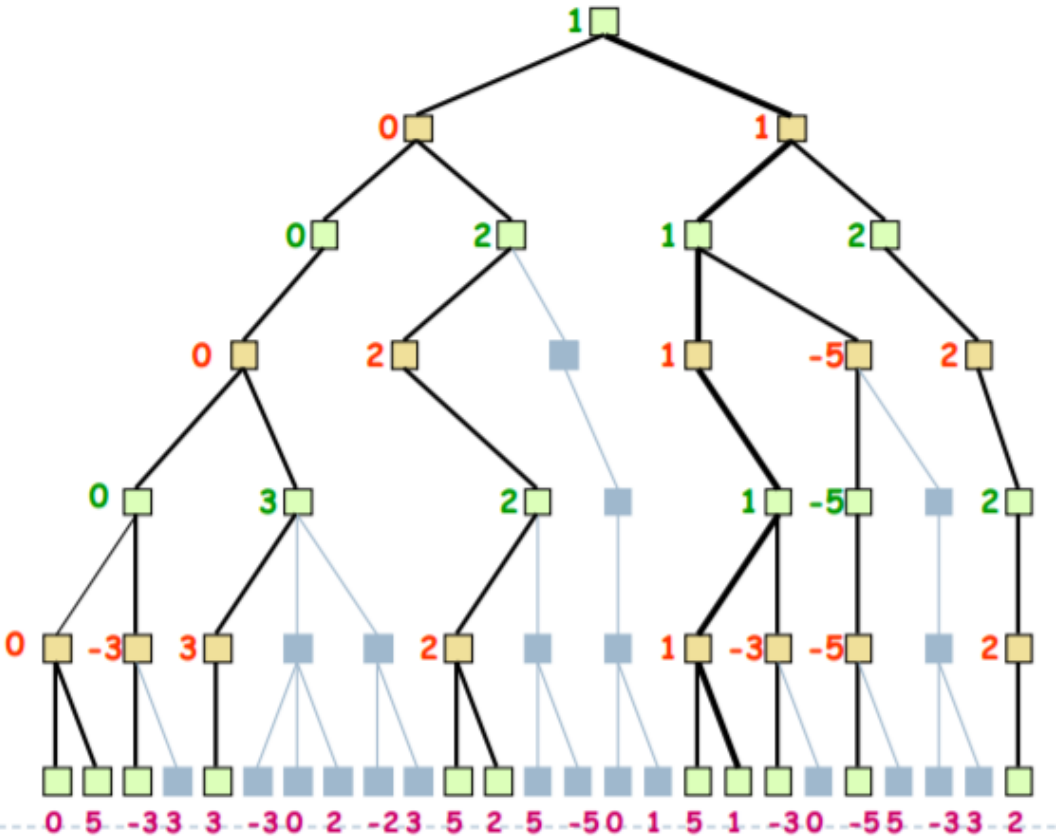
1. Expand the game tree from the current state (where it is MAX's turn to play) to depth h
2. Compute the evaluation function at every leaf of the tree
3. Back-up the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

Alpha-Beta pruning

- α: Best already explored option along path to the root for maximizer.
- β: Best already explored option along path to the root for minimizer.
- Explore the game tree to depth h in depth-first manner
- Back up α and β values whenever possible
- Prune branches that can't lead to changing the final decision

Alpha-Beta Algorithm: When to Prune?

- Update the alpha/beta value of the parent of a node N when the search below N has been completed or discontinued
- Discontinue the search below a MAX node N if its alpha value is ≥ the beta value of a MIN ancestor of N
- Discontinue the search below a MIN node N if its beta value is ≤ the alpha value of a MAX ancestor of N



Decision/Utility theory

Decision theory

A framework for an agent to choose the best decision in a non-deterministic manner.
(Calculus for decision making)

Using this framework, a problem is represented as:

- Set of states (primitive outcomes).
- Preference. Which outcome is preferred.

$A \succ B$: The agent prefers A over B.

$A \sim B$: The agent is indifferent between A and B.

- Lotteries: A set of possible outcomes with their probability of occurring, e.g.: $L = [p_A, A; p_B, B]$.

Represents probability of non-determinism of an action.

Maximum Expected Utility(MEU): assigns a utility value to each outcome representing preference

"Best decision" maximises the MEU (utility theory)

Axioms of Utility Theory

16. **Orderability:**
    $$A \succ B \text{ OR } B \succ A \text{ OR } A \sim B$$
17. **Transitivity:**
    $$\text{IF } (A \succ B \text{ AND } B \succ C) \text{ THEN } A \succ C$$
18. **Continuity:**
    $$\text{IF } (A \succ B \succ C) \text{ THEN } \exists p [p, A; 1-p, C] \sim B$$
19. **Substitutability:**
    $$\text{IF } (A \sim B) \text{ THEN } [p, A; 1-p, C] \sim [p, B; 1-p, C]$$
20. **Monotonicity:**
    $$\text{IF } A \succ B \text{ THEN}$$
    $$(p \geq q \Leftrightarrow [p, A; 1-p, B] \succ [q, A; 1-q, B])$$
21. **Decomposability:**
    $$[p, A; 1-p, [q, B; 1-q, C]] \sim$$
    $$[p, A; (1-p)q, B; (1-p)(1-q)C]$$
22. **Main theorem (if the axioms are respected).**
    $$A \succ B \Leftrightarrow U(A) > U(B)$$
    $$A \sim B \Leftrightarrow U(A) = U(B)$$

RISK in Utility theory

1. Risk Neutral : the utility is the Expected utility
2. Risk Averse: we prefer a smaller reward for sure then a larger reward with low chance. The utility function will need to be redefined
3. Risk seeker, we prefer a uncertain outcome to a certain (not necessarily advantageous) outcome, same here

Decision Tree

The decision tree is represented as an AND/OR tree where:

- OR levels are Agents decisions
- AND levels are Lottery corresponding to the outcome of the decision made at the parent node.

Markov Decision Process (MDP)

Framework to find the best course of actions to perform when the outcome of each action is non-deterministic.

Representation

A Markov decision process is framed as follows:

- **State space : S**
- **Action Space: A**
- **Transition function: T(s,a,s')** = $P(S_{t+1} = s' | S_t = s, A_t = a)$
- **Reward function: R(s)** || R(s,a) || R(s,a,s')

Can be represented by an AND/OR tree when the root is the initial state.

Solution

OFFLINE SOLUTIONS

Finding a solution ↔ finding a working strategy (called **policy**)

Optimal policy denoted as π' is mapping from states to actions, for each state S π' will tell us what is the optimal action to perform.

After performing each action the agent will receive a reward depending on what state he ends up in.

Therefore, the "best action" is trying to maximize the expected reward, called the value function.
Value function:

$$V_\pi(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') V_\pi(s')$$

Immediate reward

Expected total future reward

Q(s, a)

Optimal value function:

$$V^*(s) = \max_a \left(R(s) + \gamma \sum_{s'} T(s, a, s') V^*(s') \right)$$

Bellman equation

VALUE ITERATION

- Iterate calculating the optimal value of a state until convergence.
- Algorithm:

Initialize $V^0(s) = R(s)$ for all s.

Loop

For all s {

$$V^{t+1}(s) = \max_a \left(R(s) + \gamma \sum_{s'} T(s,a,s') V^t(s') \right)$$

}

t = t + 1

Until $V^{t+1}(s) = V^t(s)$ for all s (impl: $\max_s |V^{t+1}(s) - V^t(s)| < 1e\text{-}7$).

Often called value update or Bellman update or Bellman backup.

- Guarantee to converge to V^*

POLICY ITERATION

- Starts with an initial policy π (any random policy will do).
- Loop until convegence
  - Evaluate the value of the policy:

$$V_\pi^0(s) = R(s)$$

$$V_\pi^{t+1}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') V_\pi^t(s')$$

  - Improve the policy using one step look ahead.

$$\pi'(s) = \operatorname*{argmax}_a \left(R(s) + \gamma \sum_{s'} T(s,a,s') V_\pi(s') \right)$$

  - $\pi = \pi'$.
- Guarantee to converge.

Ideally, until no more changes in the values. Usually not practical

ONLINE SOLUTIONS

Real Time Dynamic Programming (RTDP)

- Repeat until planning time ran out
  - Simulate greedy policy from starting state until a goal state is reached
  - Perform Bellman backup on visited states
- RTDP(Initial state s_0, Goal set G)
  - Repeat until time ran out
    - $s = s_0$
    - While (s is not in G)
      - $a_{greedy} = argmax_{a \in A}\, Q(s,a)$
      - $V(s) = Q(s, a_{greedy})$
      - s' = sampleFrom(T(s, a, s')) and set s = s'

Labelled RTDP (LRTDP)

- Check if the value of a state can no longer improve (isSolved function in the following algorithm.)
  - If the value of a state s & all its descendent (that have been visited) change less than a small value ε, label s as solved
- RTDP(Initial state s_0, Goal set G)
  - Repeat until s_0 is solved
    - $s = s_0$
    - While (s is not solved)
      - $a_{greedy} = argmax_{a \in A}\, Q(s,a)$
      - $V(s) = Q(s, a_{greedy})$
      - s' = sampleFrom(T(s, a, s')), add s' to list L, and set s = s'
    - For all state s in L, if (isSolved(s)) Label s as solved

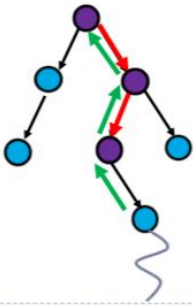
Monte Carlo Tree Search (MCTS)

Sampling based method to approximate the value of complicated functions, in MDP used to approximate the expected total reward.

$$V_{\pi^*}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') V_{\pi^*}(s')$$

Monte Carlo Tree Search (MCTS)

- Build the search tree based on the outcomes of the simulated playouts.
- Iterate over 4 main components:
  - Selection: Choose the best path
  - Expansion: When a terminal node is reached, add a child node n
  - Simulation: Simulate from the newly added node n, to estimate its value
  - Backpropagation: Update the value of the nodes visited in this iteration
- Many variants for each component



SELECTION: Multi arm bandit is used to select an action

Choose an action a to perform at s as:

$$\pi_{UCT}(s) = \arg\max_{a \in A} Q(s,a) + c\sqrt{\frac{\ln(n(s))}{n(s,a)}}$$

Exploitation | Exploration

Where c is a variable indicating how to balance exploration vs exploitation
n(s) = number of time the node has been visited
n(s,a) = number of times the out-edge of s with label a has been visited.

SIMULATION: often called rollout**, use heuristic** (eg greedy, solution of deterministic case..)

Partially Observable MDP (POMDP)

Partially observable, non-deterministic, static, discrete environment.

Representation

- **State Space**: S
- **Action Space**: A
- **Observable Space:** O
- **Transition function:** T
- **Observation function:** Z
- **Reward function:** R

Belief: mapping from a state to a probability, can be represented as a parametric distribution.

Strategy/policy: Mapping from beliefs to actions.

Value function: $V_\pi(b) = R(b) + \gamma \sum_o P(o|b,a)V_\pi(b')$

Expected immediate reward

Expected total future reward

$R(b) = \sum_{\forall s \in S} R(s)b(s)$

b': subsequent belief after the agent at belief b performs a and perceives o

Computing belief: using Bayes rule

How to implement? Particle Filter

Recall 2 steps:

- Effect of action a: For each particle s, compute next particle by sampling from transition function T (i.e., P(s' I s, a))
- Effect of observation o:
  - Assign weight to the samples, based on Bayes rule on Z (i.e., P(o I s, a)P(s))
  - Resample based on the weight

Solutions

Any solution applied to MDP can be applied to MDP but states become beliefs.

Machine learning- Supervised and Unsupervised

Definition: Algorithms to enabling agents to improve their behaviour with experience.

3 main types of machine learning:

- **Supervised Learning:** Learn from examples (sometimes we are not sure if the examples are correct
- **Unsupervised learning:** Learn by finding structure in data

- **Reinforcement learning:** Learn by doing.

Boolean Classifier Example

The answer (y component) is a binary value.

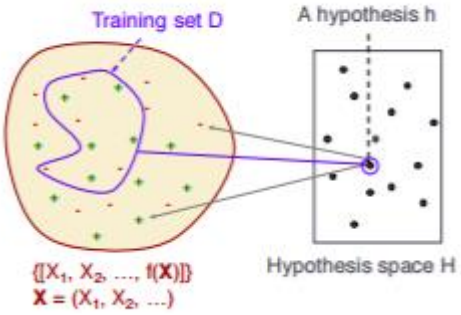
Decide whether a customer will wait for a table at a restaurant, based on the following attributes:

1. Alternate: is there an alternative restaurant nearby?
2. Bar: is there a comfortable bar area to wait in?
3. Fri/Sat: is today Friday or Saturday?
4. WaitEstimate: estimated waiting time (0-10, 10-30, 30-60, >60)

Based on previous experience the restaurant owner have collected some data:

X1	X2	X3	X4	Wait/No
Yes	Yes	Fri	0-10	Wait
Yes	No	Fri	0-10	Wait
Yes	No	Sat	10-30	No

Supervised learning: Boolean Classifier

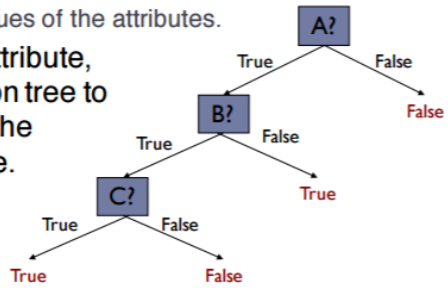


DECISION TREE

Decision tree is a tree where

- Non-leaf nodes: Attributes (i.e., the Xs)
- Leaf nodes: The results (i.e., the Y).
- Edges: The values of the attributes.

Given a new attribute, use the decision tree to "predict" what the outcome will be.



How to build a Decision Tree?

- Given: a set of training data.
- Goal: Build the smallest possible decision tree consistent with the training data.
- A simple method: Greedy
- Assuming that we will only include one attribute predicate in the decision tree, which attribute should we test to minimize the probability of error (i.e., the # of misclassified examples in the training set)?
  - Greedy w.r.t. probability of misclassification
  - Greedy w.r.t. information gain

Information gain:

- Entropy (measure of uncertainty)
- Trying to reduce the uncertainty.

$$H(X) = -\sum_{x \in X} P(x) \log_2 P(x)$$

Greedy Decision tree learning:

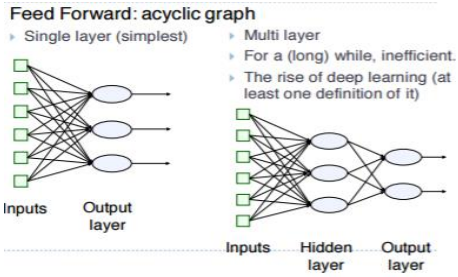
If all Ys in the training data are positive then return True
If all Ys in the training data are negative then return False
If atribute is empty then return *failure*
A ← error-minimizing attribute
Return the tree whose:
- root is A,
- left branch is DTL(D^{+A}),
- right branch is DTL(D^{-A})

Subset of examples that satisfy A

NEURAL NETWORK

A network of artificial neurons

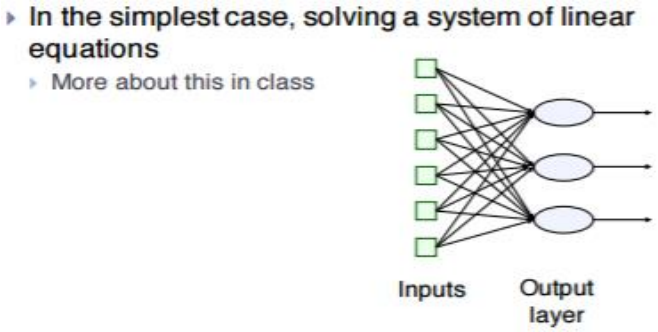
Neuron: a function applied to a linear combination of the input attributes.

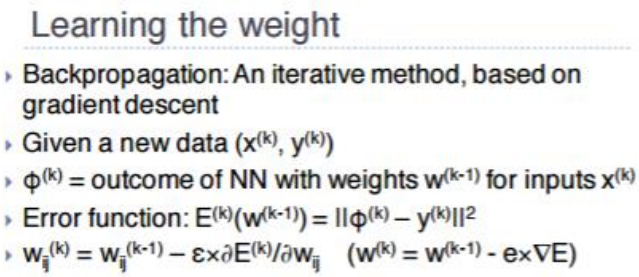


Recurrent: recurrent neural networks have at least one cycle to represent fe edback connection.

Learning a neural network:

- Finding a suitable Linear combination
- Finding suitable structure, size and activation function.





BAYESIAN NETWORKS

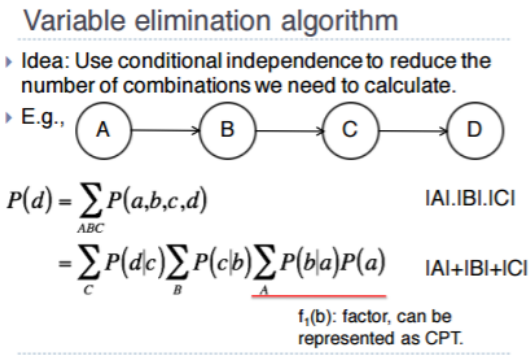
A Graph related to conditional probability tables.

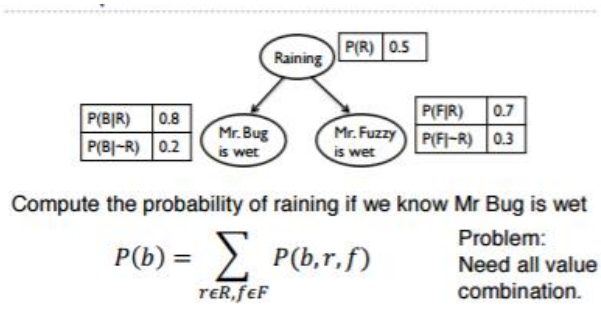
The Graph: Set of nodes representing a set of random variables (each with a finite set of values).

The Conditional Probability Table (CPT): conditional probability of a node given its parent.

Markov Blanket: The Markov blanket of a node X is the parent of X the children of X and the Parents of Children of X

Inference: Compute the probability of an event knowing one or more have happened, answer queries by summing over the variables not involved in query.





Reinforcement learning

A Reinforcement learning problem is a **MDP where the transition and/or rewards functions are not known.**

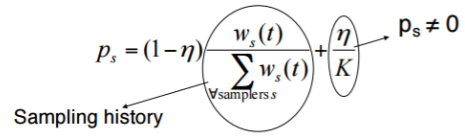
Solving: Computing the best action to perform even though the transition& reward function might not be defined. In general it boils down to **Exploitation vs Exploration.**

Exploration vs Exploitation

Epsilon Greedy

- Assign weight to each strategy (starting with equal weight)
- Strategy with highest weight is selected with P = 1-e, the rest a selected with P = e/N
- Increment weight of strategy if it is successful.

Exp3



$$w_i(t+1) = w_i(t)\exp\left(\frac{\eta r / p_i}{K}\right);$$

$r = 1$ if # connected components in the roadmap increases/decreases.

K : # samplers

Upper Confidence Bound(UCB)

- Choose an action a to perform at s as:

$$\pi(s) = \arg\max_{a \in A} Q(s,a) + c\sqrt{\frac{\ln(n(s))}{n(s,a)}}$$

Exploitation Exploration

c: A constant indicating how to balance exploration & exploitation, need to be decided by trial & error.

n(s): #times node s has been visited.

n(s,a): #times the out-edge of s with label a has been visited.

Approaches for solving

Model Based: Data is used to learn the missing components of the MDP, Once we know T&R we solve MDP, indirect learning but most efficient use of data.

Model Free: Data is used to learn the value function and policy directly, direct learn but not the most efficient use of data.

Passive: The agent is given a data set to learn from, agent does not need to decide what action to perform.

Active: The classical reinforcement learning, the agent select what action to perform then learns from the outcome and converges to the correct MDP

Model Based: Simple Frequentist

Learning the T & R steps:

- Use counting to estimate T & R
- P'(s, a, s') = #data where the agent ends up in s' after performing a from s / #data where the agent perform a from s
- R'(s, a) has a finite & known range, we can do counting, same as learning P(s, a, s')
- R'(s, a) infinite / unknown range, we can fit a function to the date (can use methods from regression/supervised learning)
- This simple strategy will converge to the true values

Model Based, Passive

1. Use a supervised learning method to compute the T&R,
2. Solve the MDP with computed T&R, compute difference between Data and the expected trajectory.
3. If the difference is large: improve the MDP model repeat from 2

Bayesian Reinforcement Learning (model based active)

- Bayesian view:
  - The parameters (T & R) we want to estimate are represented as a random variable
  - Start with a prior over models
  - Compute posterior based on data
- Quite useful when the agent actively gather data
- Can decide how to balance exploration & exploitation or how to improve the model & solve the problem optimally
  - Often represented as Partially Observable Markov Decision Processes (POMDPs)

- The problem of finding solving MDP with unknown T & R can be represented as a POMDP with partially observed MDP model
- POMDP model:
  - S: MDP states X T X R
  - A: MDP action
  - T(s, a, s'): The transition assuming the MDP model is as described by POMDP state s
  - Ω: The resulting next state and reward of the MDP
  - Z(s, a, o): Perceived next state & reward assuming the MDP model is as described by POMDP state s
  - R(s, a): The reward assuming the MDP model is as described by POMDP state s

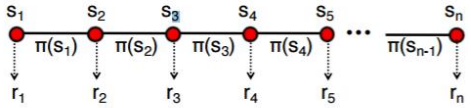
Model Free

Monte Carlo

- Goal: Given a policy, learn the value of the policy when T & R are unknown
- Assumption: Episodic MDP
  - Each episode (i.e., each run) is guaranteed to terminate within a finite amount of time.
- Loop over:
  - Generate an episode
  - Compute the total discounted reward for the episode
  - Update the value

Monte Carlo update

- Suppose we have the following episode



- For each s_i, $R(s_i) = r_i + \gamma r_{i+1} + \ldots + \gamma^{n-1} r_n$
- Value update: $V(s_i) = V(s_i) + \alpha\left(R(s_i) - V(s_i)\right)$

Temporal Difference (TD)learning

Iteratively reduce the difference between the value or the Q-value estimates.

$$Q(s_i, a_i) = Q(s_i, a_i) + \alpha[r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i)]$$

$$V(s_i) = V(s_i) + \alpha[r_i + \gamma V(s_{i+1}) - V(s_i)]$$

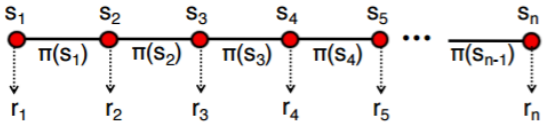
where α is a constant in [0, 1], representing the learning rate.

In some implementations, it decreases as #data increases, e.g., set to 1/(#visit + 1).

Values are updated after each step (instead of each run for monte carlo)

Temporal Difference (TD) Learning

- Given a policy π, suppose the reinforcement learning agent traverse the following episode



- Value update:

$$V(s_i) = V(s_i) + \alpha\left(r_i + \gamma V(s_{i+1}) - V(s_i)\right)$$

Q-Learning: Off-Policy TD Control

- Off-policy: Update the Q-value based on the (estimated) best next actions, even though it's not the action performed
  - The policy being followed is not the same as the policy being evaluated.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal

SARSA: On-policy TD Control

- Consider the actual action that the agent will take at the next state
- Data is (s, a, r, s', a')

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal